

Scott Krig

Computer Vision Metrics

Textbook Edition

Survey, Taxonomy and Analysis of Computer
Vision, Visual Neuroscience, and Deep Learning

Computer Vision Metrics

Scott Krig

Computer Vision Metrics

Textbook Edition

Survey, Taxonomy and Analysis
of Computer Vision, Visual
Neuroscience, and Deep Learning



Scott Krig
Krig Research, USA

ISBN 978-3-319-33761-6 ISBN 978-3-319-33762-3 (eBook)
DOI 10.1007/978-3-319-33762-3

Library of Congress Control Number: 2016938637

© Springer International Publishing Switzerland 2016

This Springer imprint is published by Springer NatureThis work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

Foreword to the Second Edition

The goal of this second version is to add new materials on deep learning, neuroscience applied to computer vision, historical developments in neural networks, and feature learning architectures, particularly neural network methods. In addition, this second edition cleans up some typos and other items from the first version. In total, three new chapters are added to survey the latest feature learning, and hierarchical deep learning methods and architectures. Overall, this book provides a wide survey of computer vision methods including local feature descriptors, regional and global features, and feature learning methods, with a taxonomy for organizational purposes. *Analysis* is distributed through the book to provide *intuition* behind the various approaches, encouraging the reader to think for themselves about the motivations for each approach, why different methods are created, how each method is designed and architected, and why it works. Nearly 1000 references to the literature and other materials are provided, making computer vision and imaging resources accessible at many levels.

My expectation for the reader is this: *if you want to learn about 90 % of computer vision, read this book. To learn the other 10 %, read the references provided and spend at least 20 years creating real systems.* Reading this book will take a matter of hours, and reading the references and creating real systems will take a lifetime to only scratch the surface. We follow the axiom of the eminent Dr. Jack Sparrow, who has no time for extraneous details, and here we endeavor to present computer vision materials in a fashion that makes the fundamentals accessible to many outside the inner circles of academia:

"I like it. Simple, easy to remember".
Jack Sparrow, Pirates of the Caribbean

This book is suitable for independent study, reference, or coursework at the university level and beyond for experienced engineers and scientists. The chapters are divided in such a way that various courses can be devised to incorporate a subset of chapters to accommodate course requirements. For example, typical course titles include "Image Sensors and Image Processing," "Computer Vision And Image Processing," "Applied Computer Vision And Imaging Optimizations," "Feature Learning, Deep Learning, and Neural Network Architectures," "Computer Vision Architectures," "Computer Vision Survey." Questions are available for coursework at the end of each chapter. It is recommended that this book be used as a

complement to other fine books, open source code, and hands-on materials for study in computer vision and related scientific disciplines, or possibly used by itself for a higher-level survey course.

This book may be used as required reading to provide a survey component to academic coursework for science and engineering disciplines, to complement other texts that contain hands-on and how-to materials.

This book DOES NOT PROVIDE extensive how-to coding examples, worked out examples, mathematical proofs, experimental results and comparisons, or detailed performance data, which are already very well covered in the bibliography references. The goal is to provide an analysis across a representative survey of methods, rather than repeating what is already found in the references. This is not a workbook with open source code (only a little source code is provided), since there are many fine open source materials available already, which are referenced for the interested reader.

Instead, this book DOES PROVIDE an extensive survey, taxonomy, and analysis of computer vision methods. The goal is to find the *intuition* behind the methods surveyed. *The book is meant to be read*, rather than worked through. This is not a workbook, but is intended to provide sufficient background for the reader to find pathways forward into basic or applied research for a variety of scientific and engineering applications. *In some respects, this work is a museum of computer vision, containing concepts, observations, oddments, and relics which fascinate me.*

The book is designed to complement existing texts and fill a niche in the literature. The book takes a complete path through computer vision, beginning with image sensors, image processing, global-regional-local feature descriptor methods, feature learning and deep learning, neural networks for computer vision, ground truth data and training, applied engineering optimizations across CPU, GPU, and software optimization methods. The author could not find a similar book, otherwise I would not have begun this work.

This book aims at a survey, taxonomy, and analysis of computer vision methods from the perspective of the features used—the feature descriptors themselves, how they are designed and how they are organized. Learning methods and architectures are necessary and supporting factors, and are included here for completeness. However, I am personally fascinated by the feature descriptor methods themselves, and I regard them as an art-form for mathematically arranging pixel patterns, shapes, and spectra to reveal how images are created. I regard each feature descriptor as a work of art, like a painting or mathematical sculpture prepared by an artist, and thus the perspective of this work is to survey feature descriptor and feature learning methods and appreciate each one.

As shown in this book over and over again, researchers are finding that a wide range of feature descriptors are effective, and that one of the keys to best results seems to be the sheer number of features used in feature hierarchies, rather than the choice of SIFT vs. pixel patches vs. CNN features. In the surveys herein we see that many methods for learning and

training are used, many architectures are used, and the consensus seems to be that hierarchical feature learning is now the mainstay of computer vision, following on from the pioneering work in convolutional neural networks and deep learning methods applied to computer vision, which has accelerated since the new millennium. The older computer vision methods are being combined with the newer ones, and now applications are beginning to appear in consumer devices, rather than exotic military and intelligence systems of the past.

Special thanks to Courtney Clarke at Springer for commissioning this second version, and providing support and guidance to make the second version better.

Special thanks to all the wonderful feedback on the first version, which helped to shape this second version. Vin Ratford and Jeff Bier of the Embedded Vision Alliance (EVA) arranged to provide copies of the first version to all EVA members, both hardcopy and e-book versions, and maintained a feedback web page for review comments—much appreciated. Thanks to Mike Schmidt and Vadim Pisarevsky for excellent review comments over of the entire book. Juergen Schmidhuber provided links to historical information on neural networks and other useful information, Kunihiko Fukushima provided copies of some of his early neural network research papers, Rahul Suthankar provided updates on key trends in computer vision, and Hugo LaRochelle provided information and references on CNN topics and Patrick Cox on HMAX topics. Interesting information was also provided by Robert Gens, Andrej Karpathy. And I would like to re-thank those who contributed to the first version, including Paul Rosin regarding synthetic interest points, Yann LeCun for providing key references into deep learning and convolutional networks, Shree Nayar for permission to use a few images, Luciano Oviedo for blue sky discussions, and many others who have influenced my thinking including Alexandre Alahi, Steve Seitz, Bryan Russel, Liefeng Bo, Xiaofeng Ren, Gutemberg Guerra-filho, Harsha Viswana, Dale Hitt, Joshua Gleason, Noah Snavely, Daniel Scharstein, Thomas Salmon, Richard Baraniuk, Carl Vodrick, Hervé Jégou, Andrew Richardson, Ofri Weschler, Hong Jiang, Andy Kuzma, Michael Jeronimo, Eli Turiel, and many others whom I have failed to mention.

As usual, thanks to my wife for patience with my research, and also for providing the “governor” switch to pace my work, without which I would likely burn out more completely. And most of all, special thanks to the great inventor who inspires us all, Anno Domini 2016.

Scott Krig

Contents

1	Image Capture and Representation	1
	Image Sensor Technology	1
	Sensor Materials	2
	Sensor Photodiode Cells	2
	Sensor Configurations: Mosaic, Foveon, BSI	4
	Dynamic Range, Noise, Super Resolution	5
	Sensor Processing	5
	De-Mosaicking	6
	Dead Pixel Correction	6
	Color and Lighting Corrections	6
	Geometric Corrections	6
	Cameras and Computational Imaging	7
	Overview of Computational Imaging	7
	Single-Pixel Computational Cameras	8
	2D Computational Cameras	9
	3D Depth Camera Systems	10
	3D Depth Processing	21
	Overview of Methods	22
	Problems in Depth Sensing and Processing	22
	Monocular Depth Processing	27
	3D Representations: Voxels, Depth Maps, Meshes, and Point Clouds	30
	Summary	31
	Chapter 1: Learning Assignments	33
2	Image Pre-Processing	35
	Perspectives on Image Processing	35
	Problems to Solve During Image Preprocessing	36
	Vision Pipelines and Image Preprocessing	36
	Corrections	38
	Enhancements	38
	Preparing Images for Feature Extraction	39
	The Taxonomy of Image Processing Methods	43
	Point	44
	Line	44
	Area	44
	Algorithmic	45

Data Conversions	45
Colorimetry	45
Overview of Color Management Systems	46
Illuminants, White Point, Black Point, and Neutral Axis	47
Device Color Models	47
Color Spaces and Color Perception	48
Gamut Mapping and Rendering Intent	48
Practical Considerations for Color Enhancements	49
Color Accuracy and Precision	50
Spatial Filtering	50
Convolutional Filtering and Detection	50
Kernel Filtering and Shape Selection	52
Point Filtering	53
Noise and Artifact Filtering	54
Integral Images and Box Filters	55
Edge Detectors	56
Kernel Sets: Sobel, Scharr, Prewitt, Roberts, Kirsch, Robinson, and Frei–Chen	56
Canny Detector	57
Transform Filtering, Fourier, and Others	58
Fourier Transform Family	58
Morphology and Segmentation	61
Binary Morphology	62
Gray Scale and Color Morphology	63
Morphology Optimizations and Refinements	63
Euclidean Distance Maps	63
Super-pixel Segmentation	64
Depth Segmentation	65
Color Segmentation	66
Thresholding	66
Global Thresholding	67
Local Thresholding	70
Summary	72
Chapter 2: Learning Assignments	73
3 Global and Regional Features	75
Historical Survey of Features	75
Key Ideas: Global, Regional, and Local Metrics	76
Textural Analysis	78
Statistical Methods	80
Texture Region Metrics	81
Edge Metrics	82
Cross-Correlation and Auto-correlation	83
Fourier Spectrum, Wavelets, and Basis Signatures	84
Co-occurrence Matrix, Haralick Features	85
Laws Texture Metrics	93
LBP Local Binary Patterns	94
Dynamic Textures	95

Statistical Region Metrics	96
Image Moment Features	96
Point Metric Features	97
Global Histograms	98
Local Region Histograms	99
Scatter Diagrams, 3D Histograms	99
Multi-resolution, Multi-scale Histograms	102
Radial Histograms	103
Contour or Edge Histograms	104
Basis Space Metrics	104
Fourier Description	107
Walsh–Hadamard Transform	108
HAAR Transform	108
Slant Transform	108
Zernike Polynomials	109
Steerable Filters	109
Karhunen–Loeve Transform and Hotelling Transform	110
Wavelet Transform and Gabor Filters	110
Hough Transform and Radon Transform	112
Summary	113
Chapter 3: Learning Assignments	114
4 Local Feature Design Concepts	115
Local Features	115
Detectors, Interest Points, Keypoints, Anchor Points, Landmarks	116
Descriptors, Feature Description, Feature Extraction	116
Sparse Local Pattern Methods	117
Local Feature Attributes	117
Choosing Feature Descriptors and Interest Points	117
Feature Descriptors and Feature Matching	118
Criteria for Goodness	119
Repeatability, Easy vs. Hard to Find	120
Distinctive vs. Indistinctive	120
Relative and Absolute Position	120
Matching Cost and Correspondence	120
Distance Functions	121
Early Work on Distance Functions	121
Euclidean or Cartesian Distance Metrics	122
Grid Distance Metrics	124
Statistical Difference Metrics	124
Binary or Boolean Distance Metrics	125
Descriptor Representation	126
Coordinate Spaces, Complex Spaces	126
Cartesian Coordinates	127
Polar and Log Polar Coordinates	127
Radial Coordinates	127

Spherical Coordinates	128
Gauge Coordinates	128
Multivariate Spaces, Multimodal Data	128
Feature Pyramids	129
Descriptor Density	129
Interest Point and Descriptor Culling	129
Dense vs. Sparse Feature Description	130
Descriptor Shape Topologies	130
Correlation Templates	131
Patches and Shape	131
Object Polygon Shapes	133
Local Binary Descriptor Point-Pair Patterns	134
FREAK Retinal Patterns	135
Brisk Patterns	136
ORB and BRIEF Patterns	137
Descriptor Discrimination	138
Spectra Discrimination	138
Region, Shapes, and Pattern Discrimination	139
Geometric Discrimination Factors	140
Feature Visualization to Evaluate Discrimination	140
Accuracy, Trackability	143
Accuracy Optimizations, Subregion Overlap, Gaussian Weighting, and Pooling	145
Sub-pixel Accuracy	145
Search Strategies and Optimizations	146
Dense Search	146
Grid Search	146
Multi-scale Pyramid Search	147
Scale Space and Image Pyramids	147
Feature Pyramids	149
Sparse Predictive Search and Tracking	150
Tracking Region-Limited Search	150
Segmentation Limited Search	150
Depth or Z Limited Search	151
Computer Vision, Models, Organization	151
Feature Space	152
Object Models	152
Constraints	154
Selection of Detectors and Features	154
Overview of Training	155
Classification of Features and Objects	156
Feature Learning, Sparse Coding, Convolutional Networks	161
Summary	164
Chapter 4: Learning Assignments	165

5	Taxonomy of Feature Description Attributes	167
	General Robustness Taxonomy	170
	General Vision Metrics Taxonomy	173
	Feature Metric Evaluation	182
	SIFT Example	183
	LBP Example	184
	Shape Factors Example	184
	Summary	185
	Chapter 5: Learning Assignments	186
6	Interest Point Detector and Feature Descriptor Survey	187
	Interest Point Tuning	188
	Interest Point Concepts	189
	Interest Point Method Survey	191
	Laplacian and Laplacian of Gaussian	192
	Moravec Corner Detector	192
	Harris Methods, Harris–Stephens, Shi–Tomasi, and Hessian Type Detectors	192
	Hessian Matrix Detector and Hessian–Laplace	193
	Difference of Gaussians	193
	Salient Regions	193
	SUSAN, and Trajkovic and Hedy	194
	Fast, Faster, AGHAST	194
	Local Curvature Methods	195
	Morphological Interest Regions	196
	Feature Descriptor Survey	196
	Local Binary Descriptors	197
	Census	205
	Modified Census Transform	205
	BRIEF	206
	ORB	206
	BRISK	207
	FREAK	208
	Spectra Descriptors	208
	SIFT	209
	SIFT-PCA	213
	SIFT-GLOH	214
	SIFT-SIFER Retrofit	214
	SIFT CS-LBP Retrofit	214
	RootSIFT Retrofit	215
	CenSurE and STAR	216
	Correlation Templates	217
	HAAR Features	219
	Viola–Jones with HAAR-Like Features	220
	SURF	221
	Variations on SURF	222
	Histogram of Gradients (HOG) and Variants	223
	PHOG and Related Methods	224

Daisy and O-Daisy	225
CARD	226
Robust Fast Feature Matching	228
RIFF, CHOG	229
Chain Code Histograms	230
D-NETS	231
Local Gradient Pattern	232
Local Phase Quantization	232
Basis Space Descriptors	233
Fourier Descriptors	234
Other Basis Functions for Descriptor Building	235
Sparse Coding Methods	235
Polygon Shape Descriptors	235
MSER Method	236
Object Shape Metrics for Blobs and Polygons	237
Shape Context	239
3D, 4D, Volumetric and Multimodal Descriptors	241
3D HOG	241
HON 4D	242
3D SIFT	243
Summary	244
Chapter 6: Learning Assignments	245
7 Ground Truth Data, Content, Metrics, and Analysis	247
What Is Ground Truth Data?	247
Previous Work on Ground Truth Data: Art vs. Science	249
General Measures of Quality Performance	249
Measures of Algorithm Performance	250
Rosin’s Work on Corners	251
Key Questions for Constructing Ground Truth Data	252
Content: Adopt, Modify, or Create	252
Survey of Available Ground Truth Data	252
Fitting Ground Truth Data to Algorithms	253
Scene Composition and Labeling	254
Defining the Goals and Expectations	256
Mikolajczyk and Schmid Methodology	256
Open Rating Systems	256
Corner Cases and Limits	257
Interest Points and Features	257
Robustness Criteria for Ground Truth Data	258
Illustrated Robustness Criteria	258
Using Robustness Criteria for Real Applications	259
Pairing Metrics with Ground Truth	261
Pairing and Tuning Interest Points, Features, and Ground Truth	261
Examples Using the General Vision Taxonomy	261
Synthetic Feature Alphabets	262
Goals for the Synthetic Dataset	263

	Synthetic Interest Point Alphabet	266
	Hybrid Synthetic Overlays on Real Images	268
	Summary	269
	Chapter 7: Learning Assignments	271
8	Vision Pipelines and Optimizations	273
	Stages, Operations, and Resources	274
	Compute Resource Budgets	275
	Compute Units, ALUs, and Accelerators	277
	Power Use	278
	Memory Use	278
	I/O Performance	282
	The Vision Pipeline Examples	282
	Automobile Recognition	282
	Face, Emotion, and Age Recognition	289
	Image Classification	296
	Augmented Reality	299
	Acceleration Alternatives	304
	Memory Optimizations	304
	Coarse-Grain Parallelism	307
	Fine-Grain Data Parallelism	308
	Advanced Instruction Sets and Accelerators	310
	Vision Algorithm Optimizations and Tuning	311
	Compiler and Manual Optimizations	312
	Tuning	312
	Feature Descriptor Retrofit, Detectors,	
	Distance Functions	313
	Boxlets and Convolution Acceleration	313
	Data-Type Optimizations, Integer vs. Float	314
	Optimization Resources	314
	Summary	315
	Chapter 8: Learning Assignments	316
9	Feature Learning Architecture Taxonomy and Neuroscience Background	319
	Neuroscience Inspirations for Computer Vision	320
	Feature Generation vs. Feature Learning	321
	Terminology of Neuroscience Applied to Computer Vision	322
	Classes of Feature Learning	327
	Convolutional Feature Weight Learning	328
	Local Feature Descriptor Learning	328
	Basis Feature Composition and Dictionary Learning	329
	Summary Perspective on Feature Learning Methods	329
	Machine Learning Models for Computer Vision	329
	Expert Systems	330
	Statistical and Mathematical Analysis Methods	331
	Neural Science Inspired Methods	331
	Deep Learning	331
	DNN Hacking and Misclassification	333

History of Machine Learning (ML) and Feature Learning	333
Historical Survey, 1940s–2010s	334
Artificial Neural Network (ANN) Taxonomy Overview	338
Feature Learning Overview	339
Learned Feature Descriptor Types	339
Hierarchical Feature Learning	340
How Many Features to Learn?	340
The Power Of DNNs	341
Encoding Efficiency	341
Handcrafted Features vs. Handcrafted Deep Learning	341
Invariance and Robustness Attributes for Feature Learning	343
What Are the Best Features and Learning Architectures?	343
Merger of Big Data, Analytics, and Computer Vision	345
Key Technology Enablers	347
Neuroscience Concepts	348
Biology and Blueprint	349
The Elusive Unified Learning Theory	350
Human Visual System Architecture	351
Taxonomy of Feature Learning Architectures	356
Architecture Topologies	357
ANNs (Artificial Neural Networks)	358
FNN (Feed Forward Neural Network)	358
RNN (Recurrent Neural Network)	358
BFN (Basis Function Network)	359
Ensembles, Hybrids	359
Architecture Components and Layers	360
Layer Totals	360
Layer Connection Topology	361
Memory Model	362
Training Protocols	362
Input Sampling Methods	363
Dropout, Reconfiguration, Regularization	363
Preprocessing, Numeric Conditioning	365
Features Set Dimensions	365
Feature Initialization	366
Features, Filters	366
Activation, Transfer functions	367
Post-processing, Numeric Conditioning	368
Pooling, Subsampling, Downsampling, Upsampling	369
Classifiers	371
Summary	371
Chapter 9: Learning Assignments	373

10 Feature Learning and Deep Learning Architecture Survey	375
Architecture Survey	376
FNN Architecture Survey	377
P—Perceptron	377
MLP, Multilayer Perceptron, Cognitron, Neocognitron	383
Concepts for CNNs, Convnets, Deep MLPs	387
LeNet	417
AlexNet, ZFNet	419
VGGNet and Variants MSRA-22, Baidu Deep Image, Deep Residual Learning	422
Half-CNN	425
NiN, Maxout	426
GoogLeNet, InceptionNet	431
MSRA-22, SPP-Net, R-CNN, MSSNN, Fast-R-CNN	434
Baidu, Deep Image, MINWA	437
SYMNETS—Deep Symmetry Networks	438
RNN Architecture Survey	442
Concepts for Recurrent Neural Networks	443
LSTM, GRU	451
NTM, RNN-NTM, RL-NTM	454
Multidimensional RNNs, MDRNN	457
C-RNN, QDRNN	460
RCL-RCNN	461
dasNET	463
NAP—Neural Abstraction Pyramid	465
BNF Architecture Survey	469
Concepts for Machine Learning and Basis Feature Networks	469
PNN—Polynomial Neural Network, GMDH	486
HKD—Kernel Descriptor Learning	488
HMP—Sparse Feature Learning	490
HMAX and Neurological Models	495
HMO—Hierarchical Model Optimization	506
Ensemble Methods	506
Deep Neural Network Futures	508
Increasing Depth to the Max—Deep Residual Learning (DRL)	509
Approximating Complex Models Using A Simpler MLP (Model Compression)	510
Classifier Decomposition and Recombination	511
Summary	511
Chapter 10: Learning Assignments	513
Appendix A: Synthetic Feature Analysis	515
Appendix B: Survey of Ground Truth Datasets	547
Appendix C: Imaging and Computer Vision Resources	555

Appendix D: Extended SDM Metrics	563
Appendix E: The Visual Genome Model (VGM)	575
References	601
Index	627

"The changing of bodies into light, and light into bodies, is very conformable to the course of Nature, which seems delighted with transmutations."

—Isaac Newton

Computer vision starts with images. This chapter surveys a range of topics dealing with capturing, processing, and representing images, including computational imaging, 2D imaging, and 3D depth imaging methods, sensor processing, depth-field processing for stereo and monocular multi-view stereo, and surface reconstruction. A high-level overview of selected topics is provided, with references for the interested reader to dig deeper. Readers with a strong background in the area of 2D and 3D imaging may benefit from a light reading of this chapter.

Image Sensor Technology

This section provides a basic overview of image sensor technology as a basis for understanding how images are formed and for developing effective strategies for image sensor processing to optimize the image quality for computer vision.

Typical image sensors are created from either CCD cells (charge-coupled device) or standard CMOS cells (complementary metal-oxide semiconductor). The CCD and CMOS sensors share similar characteristics and both are widely used in commercial cameras. The majority of sensors today use CMOS cells, though, mostly due to manufacturing considerations. Sensors and optics are often integrated to create *wafer-scale cameras* for applications like biology or microscopy, as shown in Fig. 1.1.

Image sensors are designed to reach specific design goals with different applications in mind, providing varying levels of sensitivity and quality. Consult the manufacturer's information to get familiar with each sensor. For example, the size and material composition of each photodiode sensor cell element is optimized for a given semiconductor manufacturing process so as to achieve the best trade-off between silicon die area and dynamic response for light intensity and color detection.

For computer vision, the effects of sampling theory are relevant—for example, the Nyquist frequency applied to pixel coverage of the target scene. The sensor resolution and optics together must provide adequate resolution for each pixel to image the features of interest, so it follows that a feature of interest should be imaged or sampled at least two times greater than the minimum size of the smallest pixels of importance to the feature. Of course, 2 \times oversampling is just a minimum target for accuracy; in practice, single pixel wide features are not easily resolved.



Figure 1.1 Common integrated image sensor arrangement with optics and color filters

For best results, the camera system should be calibrated for a given application to determine the sensor noise and dynamic range for pixel bit depth under different lighting and distance situations. Appropriate sensor processing methods should be developed to deal with the noise and nonlinear response of the sensor for any color channel, to detect and correct dead pixels, and to handle modeling of geometric distortion. If you devise a simple calibration method using a test pattern with fine and coarse gradations of gray scale, color, and different scales of pixel features, appropriate sensor processing methods can be devised. In Chap. 2, we survey a range of image processing methods applicable to sensor processing. But let us begin by surveying the sensor materials.

Sensor Materials

Silicon-based image sensors are most common, although other materials such as gallium (Ga) are used in industrial and military applications to cover longer IR wavelengths than silicon can reach. Image sensors range in resolution, depending upon the camera used, from a single pixel phototransistor camera, through 1D line scan arrays for industrial applications, to 2D rectangular arrays for common cameras, all the way to spherical arrays for high-resolution imaging. (Sensor configurations and camera configurations are covered later in this chapter.)

Common imaging sensors are made using silicon as CCD, CMOS, BSI, and Foveon methods, as discussed a bit later in this chapter. Silicon image sensors have a nonlinear spectral response curve; the near infrared part of the spectrum is sensed well, while blue, violet, and near UV are sensed less well, as shown in Fig. 1.2. Note that the silicon spectral response must be accounted for when reading the raw sensor data and quantizing the data into a digital pixel. Sensor manufacturers make design compensations in this area; however, sensor color response should also be considered when calibrating your camera system and devising the sensor processing methods for your application.

Sensor Photodiode Cells

One key consideration for image sensors is the photodiode size or cell size. A sensor cell using small photodiodes will not be able to capture as many photons as a large photodiode. If the cell size is near the wavelength of the visible light to be captured, such as blue light at 400 nm, then additional problems must be overcome in the sensor design to correct the image color. Sensor manufacturers take great care to design cells at the optimal size to image all colors equally well (Fig. 1.3). In the extreme, small sensors may be more sensitive to noise, owing to a lack of accumulated photons and sensor readout noise. If the photodiode sensor cells are too large, there is no benefit either, and the die size and cost for silicon go up, providing no advantage. Common commercial sensor devices may have sensor cell sizes of around 1 square micron and larger; each manufacturer is different, however, and trade-offs are made to reach specific requirements.

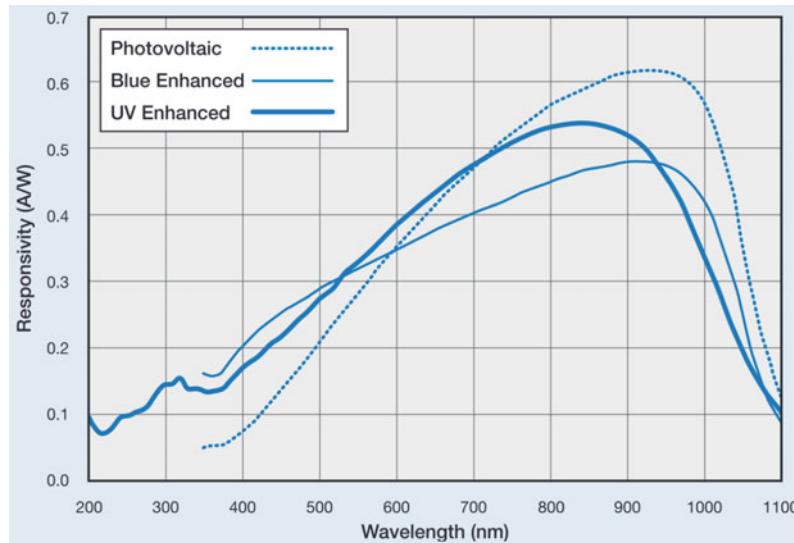


Figure 1.2 Typical spectral response of a few types of silicon photodiodes. Note the highest sensitivity in the near-infrared range around 900 nm and nonlinear sensitivity across the visible spectrum of 400–700 nm. Removing the IR filter from a camera increases the near-infrared sensitivity due to the normal silicon response. (Spectral data image © OSI Optoelectronics Inc. and used by permission)

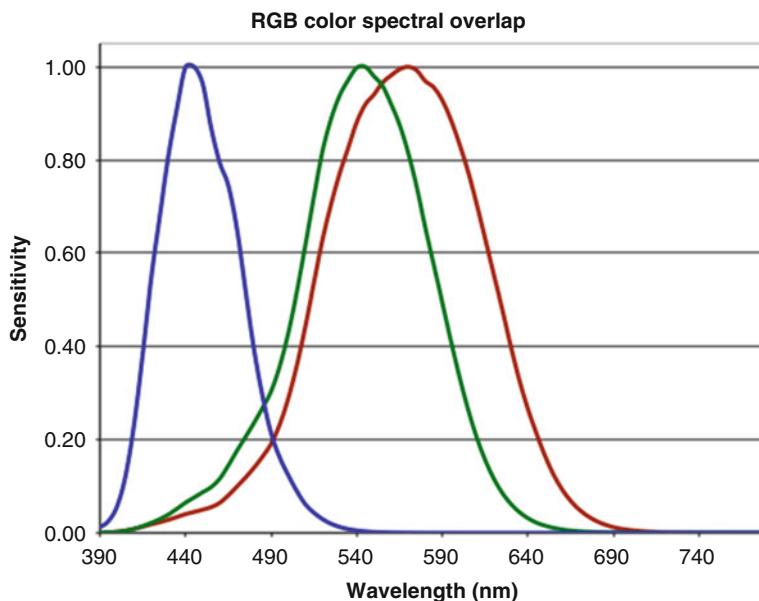


Figure 1.3 Primary color assignment to wavelengths. Note that the primary color regions overlap, with green being a good monochrome proxy for all colors

Sensor Configurations: Mosaic, Foveon, BSI

There are various on-chip configurations for multispectral sensor design, including mosaics and stacked methods, as shown in Fig. 1.4. In a *mosaic method*, the color filters are arranged in a mosaic pattern above each cell. The *Foveon¹* *sensor stacking method* relies on the physics of depth penetration of the color wavelengths into the semiconductor material, where each color penetrates the silicon to a different depth, thereby imaging the separate colors. The overall cell size accommodates all colors, and so separate cells are not needed for each color.

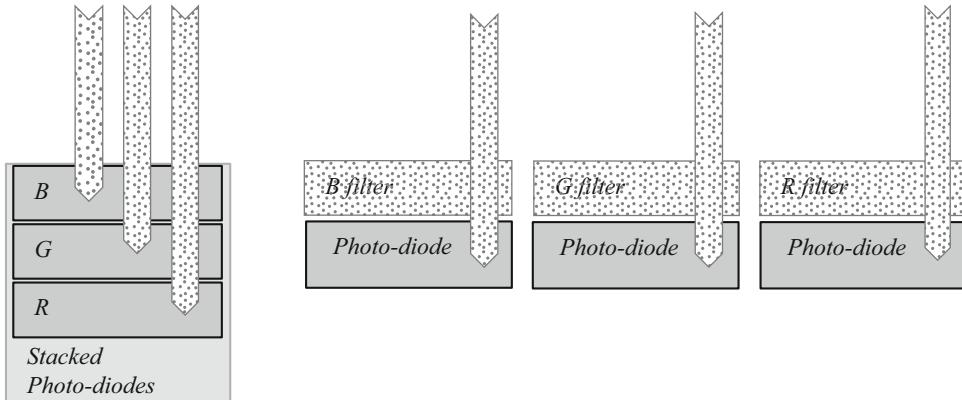


Figure 1.4 (Left) The Foveon method of stacking RGB cells to absorb different wavelengths at different depths, with all RGB colors at each cell location. (Right) A standard mosaic cell placement with RGB filters above each photodiode, with filters only allowing the specific wavelengths to pass into each photodiode

Back-side-illuminated (BSI) sensor configurations rearrange the sensor wiring on the die to allow for a larger cell area and more photons to be accumulated in each cell. See the Aptina [392] white paper for a comparison of front-side and back-side die circuit arrangement.

The arrangement of sensor cells also affects the color response. For example, Fig. 1.5 shows various arrangements of primary color (R,G,B) sensors as well as white (W) sensors together, where W sensors have a clear or neutral color filter. The sensor cell arrangements allow for a range of pixel processing options—for example, combining selected pixels in various configurations of neighboring cells during sensor processing for a pixel formation that optimizes color response or spatial color resolution. In fact, some applications just use the raw sensor data and perform custom processing to increase the resolution or develop alternative color mixes.

The overall sensor size and format determines the lens size as well. In general, a larger lens lets in more light, so larger sensors are typically better suited to digital cameras for photography applications. In addition, the cell placement aspect ratio on the die determines pixel geometry—for example, a 4:3 aspect ratio is common for digital cameras while 3:2 is standard for 35 mm film. The sensor configuration details are worth understanding in order to devise the best sensor processing and image preprocessing pipelines.

¹ Foveon is a registered trademark of Foveon Inc.

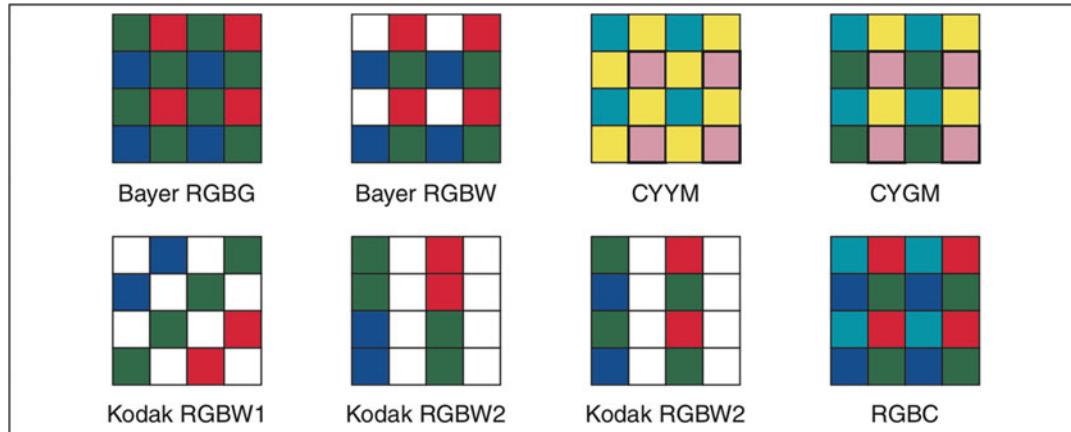


Figure 1.5 Several different mosaic configurations of cell colors, including white, primary RGB colors, and secondary CYM cells. Each configuration provides different options for sensor processing to optimize for color or spatial resolution. (Image used by permission, © Intel Press, from Building Intelligent Systems)

Dynamic Range, Noise, Super Resolution

Current state-of-the-art sensors provide at least 8 bits per color cell, and usually are 12–14 bits. Sensor cells require area and time to accumulate photons, so smaller cells must be designed carefully to avoid problems. Noise may come from optics, color filters, sensor cells, gain and A/D converters, post-processing, or the compression methods, if used. Sensor readout noise also affects effective resolution, as each pixel cell is read out of the sensor, sent to an A/D converter, and formed into digital lines and columns for conversion into pixels. Better sensors will provide less noise and higher effective bit resolution, however effective resolution can be increased using super resolution methods, by taking several images in rapid succession averaged together to reduce noise [886], or alternatively the sensor position can be micro-MEMS-dithered to create image sequences to average together to increase resolution. A good survey of de-noising is found in the work by Ibenthal [391].

In addition, sensor photon absorption is different for each color, and may be problematic for blue, which can be the hardest color for smaller sensors to image. In some cases, the manufacturer may attempt to provide a simple gamma-curve correction method built into the sensor for each color, which is not recommended. For demanding color applications, consider colorimetric device models and color management (as will be discussed in Chap. 2), or even by characterizing the nonlinearity for each color channel of the sensor and developing a set of simple corrective LUT transforms. (Noise-filtering methods applicable to depth sensing are also covered in Chap. 2.)

Sensor Processing

Sensor processing is required to de-mosaic and assemble the pixels from the sensor array, and also to correct sensing defects. We discuss the basics of sensor processing in this section.

Typically, a dedicated sensor processor is provided in each imaging system, including a fast HW sensor interface, optimized VLIW and SIMD instructions, and dedicated fixed-function hardware blocks to deal with the massively parallel pixel-processing workloads for sensor processing. Usually, sensor processing is transparent, automatic, and set up by the manufacturer of the imaging system,

and all images from the sensor are processed the same way. A bypass may exist to provide the raw data that can allow custom sensor processing for applications like digital photography.

De-Mosaicking

Depending on the sensor cell configuration, as shown in Fig. 1.5, various de-mosaicking algorithms are employed to create a final RGB pixel from the raw sensor data. A good survey by Losson et al. [388] and another by Li et al. [389] provide some background on the challenges involved and the various methods employed.

One of the central challenges of de-mosaicking is pixel interpolation to combine the color channels from nearby cells into a single pixel. Given the geometry of sensor cell placement and the aspect ratio of the cell layout, this is not a trivial problem. A related issue is color cell weighting—for example, how much of each color should be integrated into each RGB pixel. Since the spatial cell resolution in a mosaicked sensor is greater than the final combined RGB pixel resolution, some applications require the raw sensor data to take advantage of all the accuracy and resolution possible, or to perform special processing to either increase the effective pixel resolution or do a better job of spatially accurate color processing and de-mosaicking.

Dead Pixel Correction

A sensor, like an LCD display, may have dead pixels. A vendor may calibrate the sensor at the factory and provide a sensor defect map for the known defects, providing coordinates of those dead pixels for use in corrections in the camera module or driver software. In some cases, adaptive defect correction methods [390] are used on the sensor to monitor the adjacent pixels to actively look for defects and then to correct a range of defect types, such as single pixel defects, column or line defects, and defects such as 2×2 or 3×3 clusters. A camera driver can also provide adaptive defect analysis to look for flaws in real time, and perhaps provide special compensation controls in a camera setup menu.

Color and Lighting Corrections

Color corrections are required to balance the overall color accuracy as well as the white balance. As shown in Fig. 1.2, color sensitivity is usually very good in silicon sensors for red and green, but less good for blue, so the opportunity for providing the most accurate color starts with understanding and calibrating the sensor.

Most image sensor processors contain a geometric processor for vignette correction, which manifests as darker illumination at the edges of the image, as discussed in Chap. 7 Table 7.1 on robustness criteria. The corrections are based on a geometric warp function, which is calibrated at the factory to match the optics vignette pattern, allowing for a programmable illumination function to increase illumination toward the edges. For a discussion of image warping methods applicable to vignetting, see Ref. [472].

Geometric Corrections

A lens may have geometric aberrations or may warp toward the edges, producing images with radial distortion, a problem that is related to the vignetting discussed above and shown in Chap. 7 (Fig. 7.6). To deal with lens distortion, most imaging systems have a dedicated sensor processor with a

hardware-accelerated digital warp unit similar to the texture sampler in a GPU. The geometric corrections are calibrated and programmed in the factory for the optics. See Ref. [472] for a discussion of image warping methods.

Cameras and Computational Imaging

Many novel camera configurations are making their way into commercial applications using *computational imaging* methods to synthesize new images from raw sensor data—for example, depth cameras and high dynamic range cameras. As shown in Fig. 1.6, a conventional camera system uses a single sensor, lens, and illuminator to create 2D images. However, a computational imaging camera may provide multiple optics, multiple programmable illumination patterns, and multiple sensors, enabling novel applications such as 3D depth sensing and image relighting, taking advantage of the depth information, mapping the image as a texture onto the depth map, and introducing new light sources and then re-rendering the image in a graphics pipeline. Since computational cameras are beginning to emerge in consumer devices and will become the front end of computer vision pipelines, we survey some of the methods used.

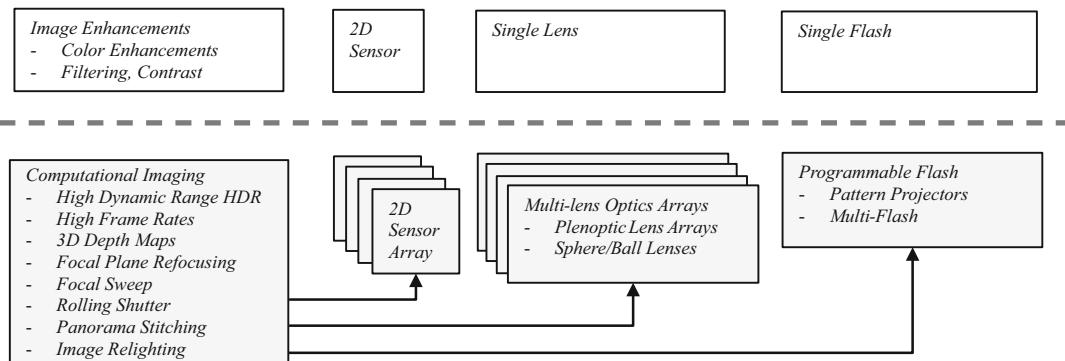


Figure 1.6 Comparison of computational imaging systems with conventional cameras. (*Top*) Simple camera model with flash, lens, and imaging device followed by image enhancements like sharpening and color corrections. (*Bottom*) Computational imaging using programmable flash, optics arrays, and sensor arrays, followed by computational imaging applications. NOT SHOWN: super resolution [886] discussed earlier

Overview of Computational Imaging

Computational imaging [396, 429] provides options for synthesizing new images from the raw image data. A computational camera may control a programmable flash pattern projector, a lens array, and multiple image sensors, as well as synthesize new images from the raw data, as illustrated in Fig. 1.6. To dig deeper into computational imaging and explore the current research, see the CAVE Computer Vision Laboratory at Columbia University and the Rochester Institute of Technology Imaging Research. Here are some of the methods and applications in use.

Single-Pixel Computational Cameras

Single-pixel computational cameras can reconstruct images from a sequence of single photo detector pixel images of the same scene. The field of single-pixel cameras [95, 96] falls into the domain of compressed sensing research, which also has applications outside image processing extending into areas such as analog-to-digital conversion.

As shown in Fig. 1.7, a *single-pixel camera* may use a *micro-mirror array* or a *digital mirror device* (DMD), similar to a diffraction grating. The gratings are arranged in a rectangular micro-mirror grid array, allowing the grid regions to be switched on or off to produce binary grid patterns. The binary patterns are designed as a pseudo-random binary basis set. The resolution of the grid patterns is adjusted by combining patterns from adjacent regions—for example, a grid of 2×2 or 3×3 micro-mirror regions.

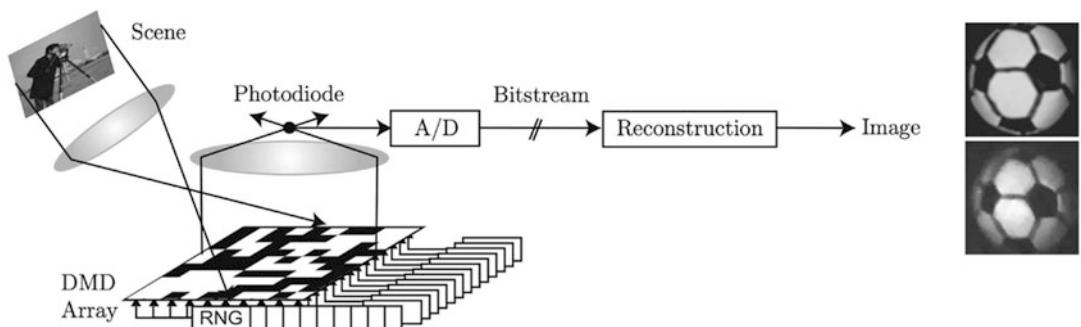


Figure 1.7 A single-pixel imaging system where incoming light is reflected through a DMD array of micro-mirrors onto a single photodiode. The grid locations within the micro-mirror array can be opened or closed to light, as shown here, to create binary patterns, where the white grid squares are reflective and *open*, and the *black grid squares* are closed. (Image used by permission, © R.G. Baraniuk, Compressive Sensing Lecture Notes)

A sequence of single-pixel images is taken through a set of pseudo-random micro lens array patterns, then an image is reconstructed from the set. In fact, the number of pattern samples required to reconstruct the image is lower than the Nyquist frequency, since a sparse random sampling approach is used and the random sampling approach has been proven in the research to be mathematically sufficient [95, 96]. The grid basis-set sampling method is directly amenable to image compression, since only a relatively sparse set of patterns and samples are taken. Since the micro-mirror array uses rectangular shapes, the patterns are analogous to a set of HAAR basis functions. (For more information, see Figs. 3.21, 6.21 and 6.22.)

The DMD method is remarkable, in that an image can be reconstructed from a fairly small set of images taken from a single photo detector, rather than a 2D array of photo detectors as in a CMOS or CCD image sensor. Since only a single sensor is used, the method is promising for applications with wavelengths outside the near IR and visible spectrum imaged by CMOS and CCD sensors. The DMD method can be used, for example, to detect emissions from concealed weapons or substances at invisible wavelengths using non-silicon sensors sensitive to nonvisible wavelengths.

2D Computational Cameras

Novel configurations of programmable 2D sensor arrays, lenses, and illuminators are being developed into camera systems as *computational cameras* [406–408], with applications ranging from digital photography to military and industrial uses, employing computational imaging methods to enhance the images after the fact. Computational cameras borrow many computational imaging methods from confocal imaging [401] and confocal microscopy [402, 403]—for example, using multiple illumination patterns and multiple focal plane images. They also draw on research from synthetic aperture radar systems [404] developed after World War II to create high-resolution images and 3D depth maps using wide baseline data from a single moving-camera platform. Synthetic apertures using multiple image sensors and optics for overlapping fields of view using wafer-scale integration are also topics of research [401]. We survey here a few computational 2D sensor methods, including *high resolution* (HR), *high dynamic range* (HDR), and *high frame rate* (HF) cameras.

The current wave of commercial digital megapixel cameras, ranging from around 10 megapixels on up, provide resolution matching or exceeding high-end film used in a 35 mm camera [394], so a pixel from an image sensor is comparable in size to a grain of silver on the best resolution film. On the surface, there appears to be little incentive to go for higher resolution for commercial use, since current digital methods have replaced most film applications and film printers already exceed the resolution of the human eye.

However, very high resolution gigapixel imaging devices are being devised and constructed as an array of image sensors and lenses, providing advantages for computational imaging after the image is taken. One configuration is the *2D array camera*, composed of an orthogonal 2D array of image sensors and corresponding optics; another configuration is the *spherical camera* as shown in Fig. 1.8 [393, 397], developed as a DARPA research project at Columbia University CAVE.

High dynamic range (HDR) cameras [398–400] can produce deeper pixels with higher bit resolution and better color channel resolution by taking multiple images of the scene bracketed with different exposure settings and then combining the images. This combination uses a suitable weighting scheme to produce a new image with deeper pixels of a higher bit depth, such as 32 pixels per color channel, providing images that go beyond the capabilities of common commercial CMOS and CCD sensors. HDR methods allow faint light and strong light to be imaged equally well, and can combine faint light and bright light using adaptive local methods to eliminate glare and create more uniform and pleasing image contrast.

High frame rate (HF) cameras [407] are capable of capturing a rapid succession of images of the scene into a set and combining the set of images using bracketing techniques to change the exposure, flash, focus, white balance, and depth of field.

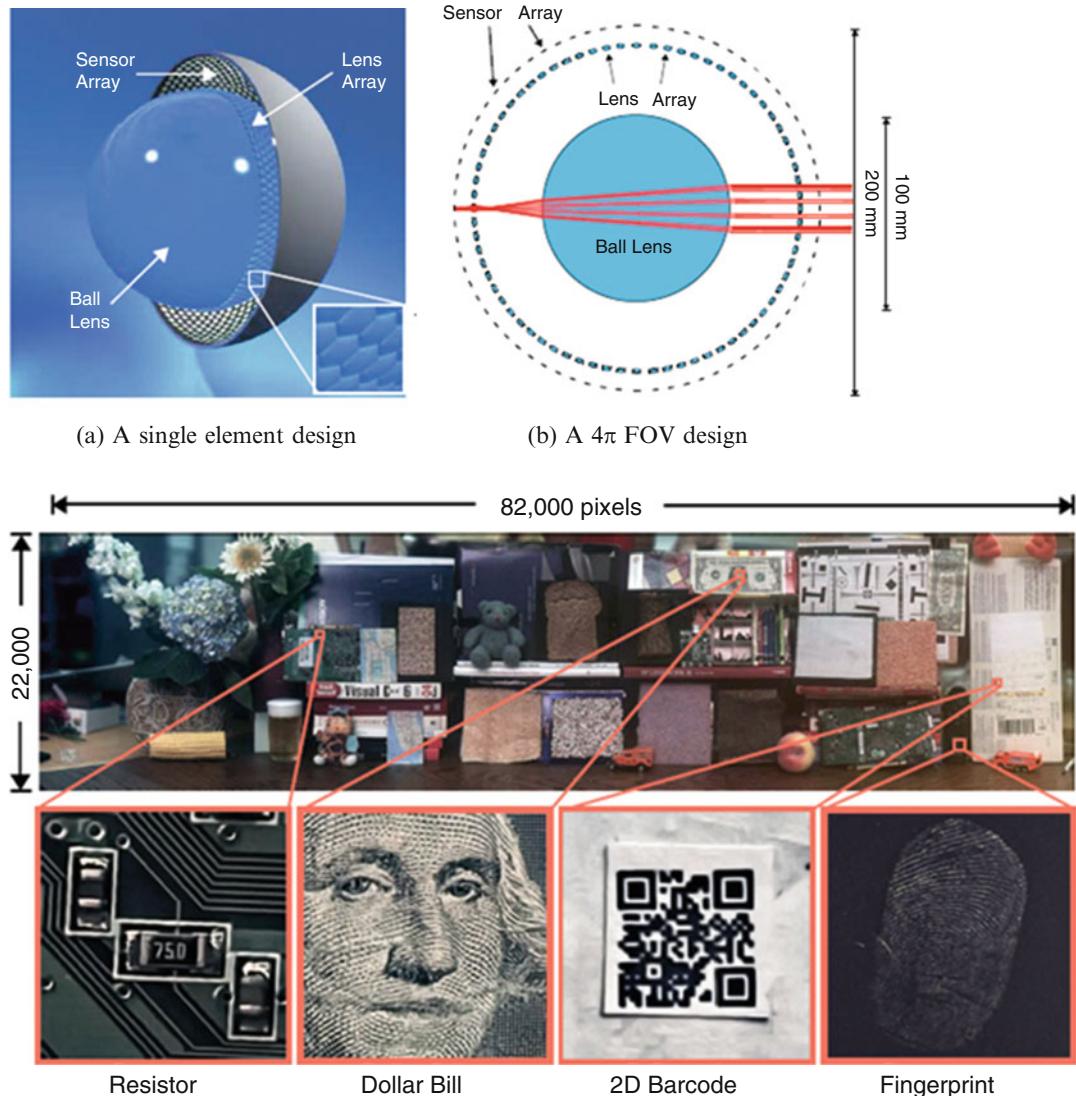


Figure 1.8 (Top) Components of a very high resolution gigapixel camera, using a novel spherical lens and sensor arrangement. (Bottom) The resulting high-resolution images shown at $82,000 \times 22,000 = 1.7$ gigapixels. (All figures and images used by permission © Shree Nayar Columbia University CAVE research projects)

3D Depth Camera Systems

Using a 3D depth field for computer vision provides an understated advantage for many applications, since computer vision has been concerned in large part with extracting 3D information from 2D images, resulting in a wide range of accuracy and invariance problems. Novel 3D descriptors are being devised for 3D depth field computer vision, and are discussed in Chap. 6.

With depth maps, the scene can easily be segmented into foreground and background to identify and track simple objects. Digital photography applications are incorporating various computer vision

methods in 3-space and thereby becoming richer. Using selected regions of a 3D depth map as a mask enables localized image enhancements such as depth-based contrast, sharpening, or other preprocessing methods.

Table 1.1 Selected Methods for Capturing Depth Information

Depth Sensing Technique	# of Sensors	Illumination Method	Characteristics
Parallax and Hybrid Parallax	2/1/array	Passive – Normal lighting	Positional shift measurement in FOV between two camera positions, such as stereo, multi-view stereo, or array cameras.
Size Mapping	1	Passive – Normal lighting	Utilizes color tags of specific size to determine range and position
Depth of Focus	1	Passive – Normal lighting	Multi-frame with scanned focus
Differential Magnification	1	Passive – Normal lighting	Two-frame image capture at different magnifications, creating a distance-based offset
Structured light	1	Active – Projected lighting	Multi-frame pattern projection
Shape from Shading	1	Passive – Normal lighting	Based on surface normals, surface reflectivity function, and light source positions
Time of Flight	1	Active – Pulsed lighting	High-speed light pulse with special pixels measuring return time of reflected light
Shading shift	1	Active – Alternating lighting	Two-frame shadow differential measurement between two light sources as different positions
Pattern spreading	1	Active – Multi-beam lighting	Projected 2D spot pattern expanding at different rate from camera lens field spread
Beam tracking	1	Active – Lighting on object(s)	Two-point light sources mounted on objects in FOV to be tracked
Spectral Focal Sweep	1	Passive – Normal Lighting	Focal length varies for each color wavelength, with focal sweep to focus on each color and compute depth [416]
Diffraction Gratings	1	Passive – Normal Lighting	Light passing through sets of gratings or light guides provides depth information [418]
Conical Radial Mirror	1	Passive – Normal Lighting	Light from a conical mirror is imaged at different depths as a toroid shape, depth is extracted from the toroid [411]

Source: Courtesy of Ken Salsmann Aptina [409], with a few other methods added by the author.

As shown in Table 1.1, there are many ways to extract depth from images. In some cases, only a single camera lens and sensor are required, and software does the rest. Note that the *illumination method* is a key component of many depth-sensing methods, such as structured light methods. Combinations of sensors, lenses, and illumination are used for depth imaging and computational imaging, as shown in Fig. 1.9. We survey a few selected depth-sensing methods in this section.

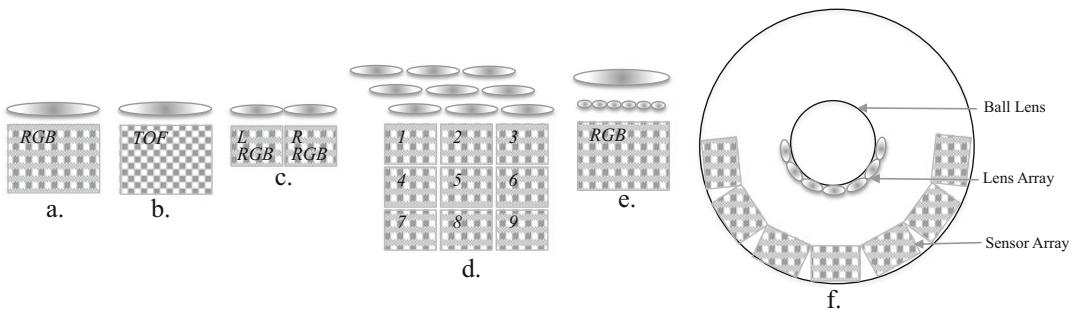


Figure 1.9 A variety of lens and sensor configurations for common cameras: (a) conventional, (b) time-of-flight, (c) stereo, (d) array, (e) plenoptic, (f) spherical with ball lens

Depth sensing is not a new field, and is covered very well in several related disciplines with huge industrial applications and financial resources, such as satellite imaging, remote sensing, photogrammetry, and medical imaging. However, the topics involving depth sensing are of growing interest in computer vision with the advent of commercial depth-sensing cameras such as Kinect, enabling graduate students on a budget to experiment with 3D depth maps and point clouds using a mobile phone or PC.

Multi-view stereo (MVS) depth sensing has been used for decades to compute digital elevation maps or DEMs, and digital terrain maps or DTMs, from satellite images using RADAR and LIDAR imaging, and from regional aerial surveys using specially equipped airplanes with high-resolution cameras and stable camera platforms, including digital terrain maps overlaid with photos of adjacent regions stitched together. *Photo mosaicking* is a related topic in computer vision that is gaining attention. The literature on *digital terrain mapping* is rich with information on proper geometry models and disparity computation methods. In addition, *3D medical imaging* via CAT and MRI modalities is backed by a rich research community, uses excellent depth-sensing methods, and offers depth-based rendering and visualization. However, it is always interesting to observe the “reinvention” in one field, such as computer vision, of well-known methods used in other fields. As Solomon said, “There is nothing new under the sun.” In this section we approach depth sensing in the context of computer vision, citing relevant research, and leave the interesting journey into other related disciplines to the interested reader.

Binocular Stereo

Stereo [414, 415, 419] may be the most basic and familiar approach for capturing 3D depth maps, as many methods and algorithms are in use, so we provide a high-level overview here with selected standard references. The first step in stereo algorithms is to parameterize the projective transformation from world coordinate points to their corresponding image coordinates by determining the *stereo calibration* parameters of the camera system. Open-source software is available for stereo calibration.² Note that the L/R image pair is rectified prior to searching for features for disparity computation. Stereo depth r is computed, as shown in Fig. 1.10.

² <http://opencv.org>, Camera Calibration and 3D Reconstruction.

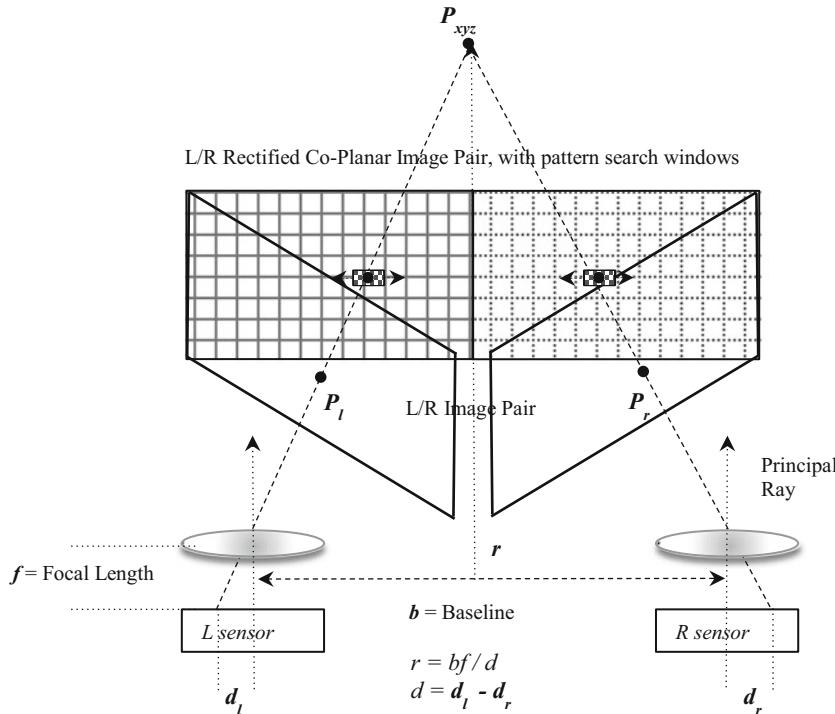


Figure 1.10 Simplified schematic of basic binocular stereo principles

An excellent survey of stereo algorithms and methods is found in the work of Scharstein and Szeliski [422] and also Lazaros [423]. The stereo geometry is a combination of *projective* and *Euclidean* [419]; we discuss some of the geometric problems affecting their accuracy later in this section. The standard online resource for comparing stereo algorithms is provided by Middlebury College,³ where many new algorithms are benchmarked and comparative results provided, including the extensive ground truth datasets discussed in Appendix B.

The fundamental geometric calibration information needed for stereo depth includes the following basics.

- **Camera Calibration Parameters.** Camera calibration is outside the scope of this work, however the parameters are defined as 11 free parameters [414, 417]—three for rotation, three for translation, and five intrinsic—plus one or more lens distortion parameters to reconstruct 3D points in world coordinates from the pixels in 2D camera space. The camera calibration may be performed using several methods, including a known calibration image pattern or one of many self-calibration methods [418]. *Extrinsic* parameters define the location of the camera in world coordinates, and *intrinsic* parameters define the relationships between pixel coordinates in camera image coordinates. Key variables include the calibrated baseline distance between two cameras at the principal point or center point of the image under the optics; the focal length of the optics; their pixel size and aspect ratio, which is computed from the sensor size divided by pixel resolution in each axis; and the position and orientation of the cameras.

³ <http://vision.middlebury.edu/~schar/stereo/web/results.php>.

- **Fundamental Matrix or Essential Matrix.** These two matrices are related, defining the popular geometry of the stereo camera system for projective reconstruction [418–420]. Their derivation is beyond the scope of this work. Either matrix may be used, depending on the algorithms employed. The *essential matrix* uses only the extrinsic camera parameters and camera coordinates, and the *fundamental matrix* depends on both the extrinsic and intrinsic parameters, and reveals pixel relationships between the stereo image pairs on epipolar lines.

In either case, we end up with projective transformations to reconstruct the 3D points from the 2D camera points in the stereo image pair.

Stereo processing steps are typically as follows:

1. **Capture:** Photograph the left/right image pair simultaneously.
2. **Rectification:** Rectify left/right image pair onto the same plane, so that pixel rows x coordinates and lines are aligned. Several projective warping methods may be used for rectification [419]. Rectification reduces the pattern match problem to a 1D search along the x -axis between images by aligning the images along the x -axis. Rectification may also include radial distortion corrections for the optics as a separate step; however, many cameras include a built-in factory-calibrated radial distortion correction.
3. **Feature Description:** For each pixel in the image pairs, isolate a small region surrounding each pixel as a *target feature descriptor*. Various methods are used for stereo feature description [112, 206].
4. **Correspondence:** Search for each target feature in the opposite image pair. The search operation is typically done twice, first searching for left-pair target features in the right image and then right-pair target features in the left image. Subpixel accuracy is required for correspondence to increase depth field accuracy.
5. **Triangulation:** Compute the disparity or distance between matched points using triangulation [421]. Sort all L/R target feature matches to find the best quality matches, using one of many methods [422].
6. **Hole Filling:** For pixels and associated target features with no corresponding good match, there is a hole in the depth map at that location. Holes may be caused by occlusion of the feature in either of the L/R image pairs, or simply by poor features to begin with. Holes are filled using local region nearest-neighbor pixel interpolation methods.

Stereo depth-range resolution is an exponential function of distance from the viewpoint: in general, the wider the baseline, the better the long-range depth resolution. A shorter baseline is better for close-range depth (see Figs. 1.10 and 1.20). Human-eye baseline or inter-pupillary distance has been measured as between 50 and 75 mm, averaging about 70 mm for males and 65 mm for females.

Multi-view stereo (MVS) is a related method to compute depth from several views using different baselines of the same subject, such as from a single or monocular camera, or an array of cameras. Monocular, MVS, and array camera depth sensing are covered later in this section.

Structured and Coded Light

Structured or coded light uses specific patterns projected into the scene and imaged back, then measured to determine depth; see Fig. 1.11. We define the following approaches for using structured light for this discussion [427]:

- **Spatial single-pattern methods**, requiring only a single illumination pattern in a single image.
- **Timed multiplexing multi-pattern methods**, requiring a sequence of pattern illuminations and images, typically using binary or n -array codes, sometimes involving phase shifting or dithering the patterns in subsequent frames to increase resolution. Common pattern sequences include gray codes, binary codes, sinusoidal codes, and other unique codes.

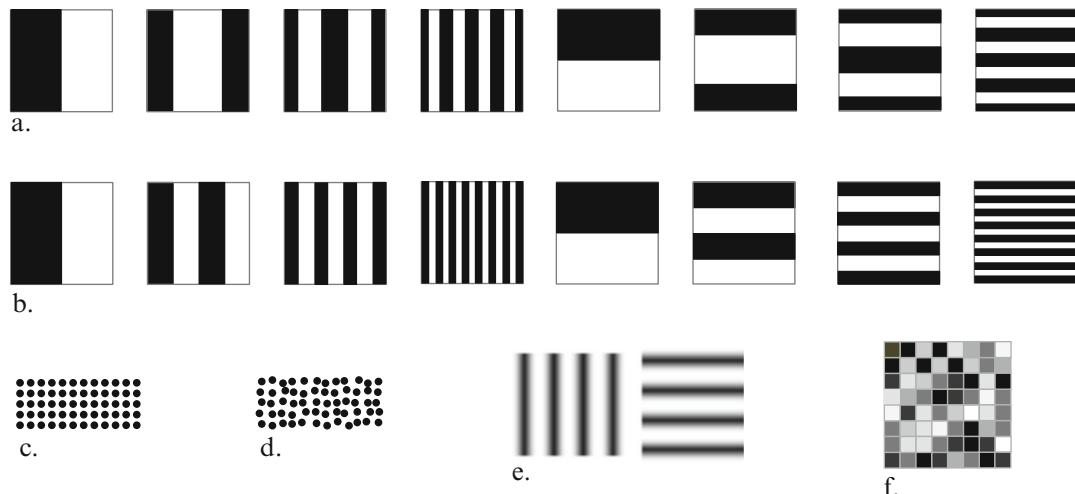


Figure 1.11 Selected structured light patterns and methods: (a) gray codes, (b) binary codes, (c) regular spot grid, (d) randomized spot grid (as used in original Kinect), (e) sinusoidal phase shift patters, (f) randomized pattern for compressive structured light [428]

For example, in the original Microsoft Kinect 3D depth camera, structured light consisting of several slightly different micro-grid patterns or pseudo-random points of infrared light are projected into the scene, then a single image is taken to capture the spots as they appear in the scene. Based on analysis of actual systems and patent applications, the original Kinect computes the depth using several methods, including (1) the size of the infrared spot—larger dots and low blurring mean the location is nearer, while smaller dots and more blurring mean the location is farther away; (2) the shape of the spot—a circle indicates a parallel surface, an ellipse indicates an oblique surface; and (3) by using small regions or a micro pattern of spots together so that the resolution is not very fine—however, noise sensitivity is good. Depth is computed from a *single image* using this method, rather than requiring several sequential patterns and images.

Multi-image methods are used for structured light, including projecting sets of time-sequential structured and coded patterns, as shown in Fig. 1.11. In multi-image methods, each pattern is sent sequentially into the scene and imaged, then the combination of depth measurements from all the patterns is used to create the final depth map.

Industrial, scientific, and medical applications of depth measurements from structured light can reach high accuracy, imaging objects up to a few meters in size with precision that extends to micrometer range. Pattern projection methods are used, as well as laser-stripe pattern methods using multiple illumination beams to create wavelength interference; the interference is measured to compute the distance. For example, common dental equipment uses small, hand-held laser range finders inserted into the mouth to create highly accurate depth images of tooth regions with missing

pieces, and the images are then used to create new, practically perfectly fitting crowns or fillings using CAD/CAM micro-milling machines.

Of course, infrared light patterns do not work well outdoors in daylight; they become washed out by natural light. Also, the strength of the infrared emitters that can be used is limited by practicality and safety. The distance for effectively using structured light indoors is restricted by the amount of power that can be used for the IR emitters; perhaps 5 m is a realistic limit for indoor infrared light. Kinect claims a range of about 4 m for the current TOF (time of flight) method using uniform constant infrared illumination, while the first-generation Kinect sensor had similar depth range using structured light.

In addition to creating depth maps, structured or coded light is used for measurements employing optical encoders, as in robotics and process control systems. The encoders measure radial or linear position. They provide IR illumination patterns and measure the response on a scale or reticle, which is useful for single-axis positioning devices like linear motors and rotary lead screws. For example, patterns such as the binary position code and the reflected binary gray code [426] can be converted easily into binary numbers (see Fig. 1.11). The gray code set elements each have a Hamming distance of 1 between successive elements.

Structured light methods suffer problems when handling high-specular reflections and shadows; however, these problems can be mitigated by using an optical diffuser between the pattern projector and the scene using the diffuse structured light methods [425] designed to preserve illumination coding. In addition, multiple-pattern structured light methods cannot deal with fast-moving scenes; however, the single-pattern methods can deal well with frame motion, since only one frame is required.

Optical Coding: Diffraction Gratings

Diffraction gratings are one of many methods of optical coding [429] to create a set of patterns for depth-field imaging, where a light structuring element, such as a mirror, grating, light guide, or special lens, is placed close to the detector or the lens. The original Kinect system is reported to use a diffraction grating method to create the randomized infrared spot illumination pattern. Diffraction gratings [412, 413] above the sensor, as shown in Fig. 1.12, can provide angle-sensitive pixel sensing. In this case, the light is refracted into surrounding cells at various angles, as determined by the placement of the diffraction gratings or other beam-forming elements, such as light guides. This allows the same sensor data to be processed in different ways with respect to a given angle of view, yielding different images.

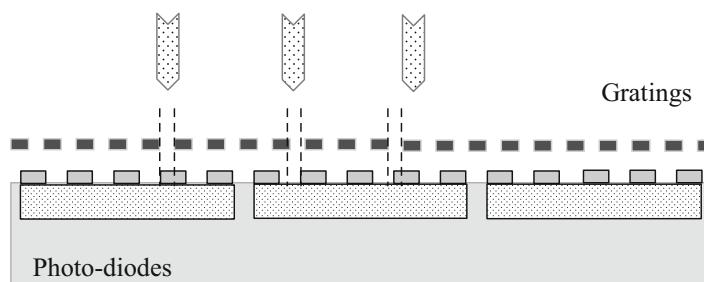


Figure 1.12 Diffraction gratings above silicon used to create the Talbot Effect (first observed around 1836) for depth imaging. (For more information, see Ref. [412].) Diffraction gratings are a type of light-structuring element

This method allows the detector size to be reduced while providing higher resolution images using a combined series of low-resolution images captured in parallel from narrow aperture diffraction gratings. Diffraction gratings make it possible to produce a wide range of information from the same sensor data, including depth information, increased pixel resolution, perspective displacements, and focus on multiple focal planes after the image is taken. A diffraction grating is a type of illumination coding device.

As shown in Fig. 1.13, the light-structuring or coding element may be placed in several configurations, including (See [429]):

- Object side coding: close to the subjects
- Pupil plane coding: close to the lens on the object side
- Focal plane coding: close to the detector
- Illumination coding: close to the illuminator

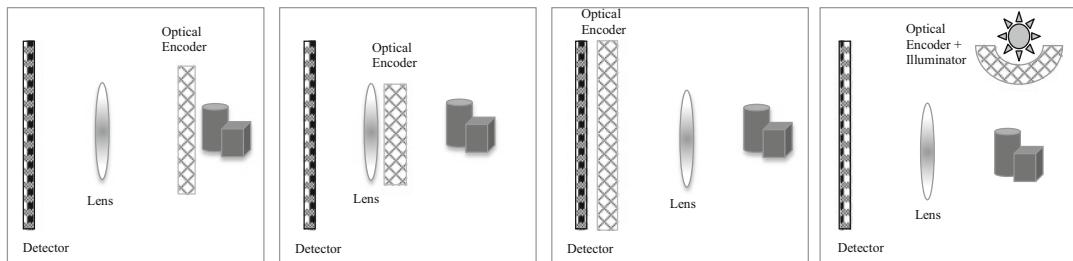


Figure 1.13 Various methods for optical structuring and coding of patterns [429]: (Left to right): Object side coding, pupil plane coding, focal plane coding, illumination coding or structured light. The illumination patterns are determined in the optical encoder

Note that illumination coding is shown as structured light patterns in Fig. 1.11, while a variant of illumination coding is shown in Fig. 1.7, using a set of mirrors that are opened or closed to create patterns.

Time-of-Flight Sensors

By measuring the amount of time taken for infrared light to travel and reflect, a *time-of-flight* (TOF) sensor is created [432]. A TOF sensor is a type of range finder or laser radar [431]. Several single-chip TOF sensor arrays and depth camera solutions are available, such as the second version of the Kinect depth camera. The basic concept involves broadcasting infrared light at a known time into the scene, such as by a pulsed IR laser, and then measuring the time taken for the light to return at each pixel. Submillimeter accuracy at ranges up to several hundred meters is reported for high-end systems [431], depending on the conditions under which the TOF sensor is used, the particular methods employed in the design, and the amount of power given to the IR laser.

Each pixel in the TOF sensor has several active components, as shown in Fig. 1.14, including the IR sensor well, timing logic to measure the round-trip time from illumination to detection of IR light, and optical gates for synchronization of the electronic shutter and the pulsed IR laser. TOF sensors provide laser range-finding capabilities. For example, by gating the electronic shutter to eliminate short round-trip responses, environmental conditions such as fog or smoke reflections can be reduced. In addition, specific depth ranges, such as long ranges, can be measured by opening and closing the shutter at desired time intervals.

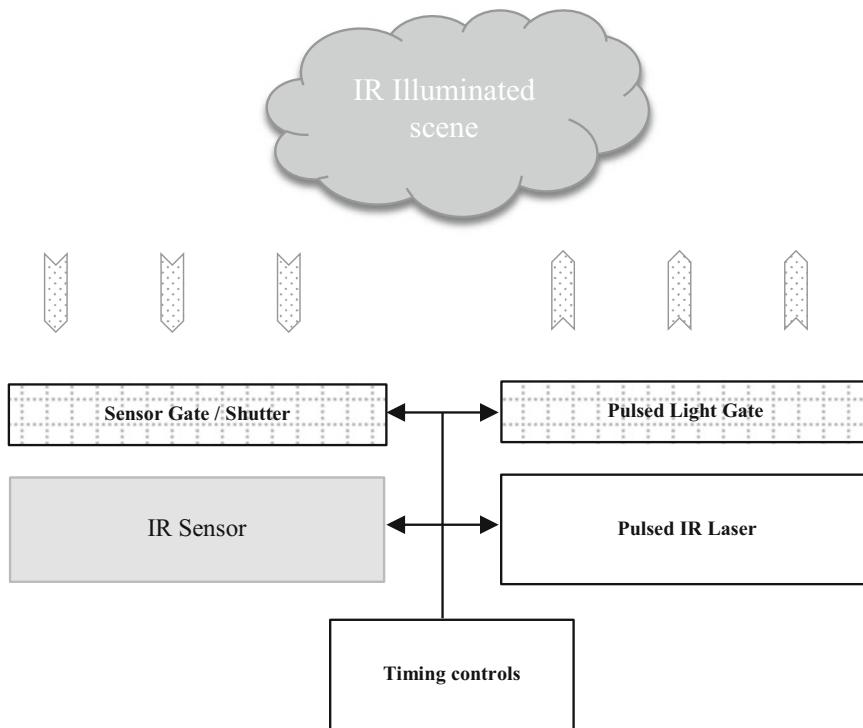


Figure 1.14 A hypothetical TOF sensor configuration. Note that the light pulse length and sensor can be gated together to target specific distance ranges

Illumination methods for TOF sensors may use very short IR laser pulses for a first image, acquire a second image with no laser pulse, and then take the difference between the images to eliminate ambient IR light contributions. By modulating the IR beam with an RF carrier signal using a photonic mixer device (PMD), the phase shift of the returning IR signal can be measured to increase accuracy—which is common among many laser range-finding methods [432]. Rapid optical gating combined with intensified CCD sensors can be used to increase accuracy to the submillimeter range in limited conditions, even at ranges above 100 m. However, multiple IR reflections can contribute errors to the range image, since a single IR pulse is sent out over the entire scene and may reflect off of several surfaces before being imaged.

Since the depth-sensing method of a TOF sensor is integrated with the sensor electronics, there is very low processing overhead required compared to stereo and other methods. However, the limitations of IR light for outdoor situations still remain [430], which can affect the depth accuracy.

Array Cameras

As shown earlier in Fig. 1.9, an *array camera* contains several cameras, typically arranged in a 2D array, such as a 3×3 array, providing several key options for computational imaging. Commercial array cameras for portable devices are beginning to appear. They may use the multi-view stereo method to compute disparity, utilizing a combination of sensors in the array, as discussed earlier. Some of the key advantages of an array camera include a wide baseline image set to compute a 3D

depth map that can see through and around occlusions, higher-resolution images interpolated from the lower-resolution images of each sensor, all-in-focus images, and specific image refocusing at one or more locations. The maximum aperture of an array camera is equal to the widest baseline between the sensors.

Radial Cameras

A conical, or radial, mirror surrounding the lens and a 2D image sensor create a radial camera [395], which combines both 2D and 3D imaging. As shown in Fig. 1.15, the radial mirror allows a 2D image to form in the center of the sensor and a radial toroidal image containing reflected 3D information forms around the sensor perimeter. By processing the toroidal information into a point cloud based on the geometry of the conical mirror, the depth is extracted and the 2D information in the center of the image can be overlaid as a texture map for full 3D reconstruction.

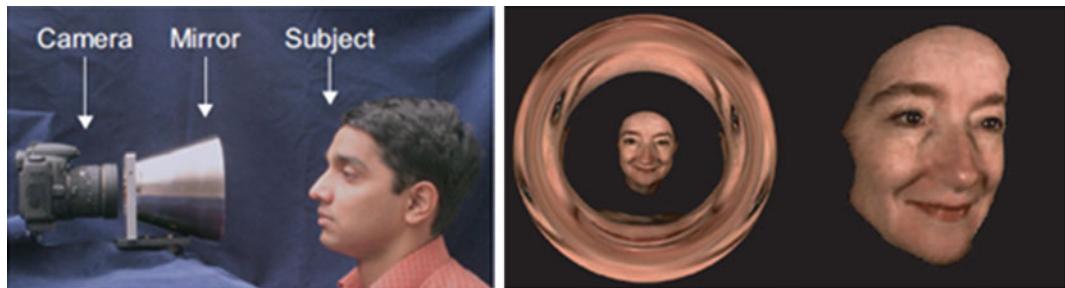


Figure 1.15 (Left) Radial camera system with conical mirror to capture 3D reflections. (Center) Captured 3D reflections around the edges and 2D information of the face in the center. (Right) 3D image reconstructed from the radial image 3D information and the 2D face as a texture map. (Images used by permission © Shree Nayar Columbia University CAVE)

Plenoptics: Light Field Cameras

Plenoptic methods create a 3D space defined as a *light field*, created by multiple optics. Plenoptic systems use a set of micro-optics and main optics to image a 4D light field and extract images from the light field during post-processing [405, 433, 434]. Plenoptic cameras require only a single image sensor, as shown in Fig. 1.16. The 4D light field contains information on each point in the space, and can be represented as a volume dataset, treating each point as a *voxel*, or 3D pixel with a 3D oriented surface, with color and opacity. Volume data can be processed to yield different views and perspective displacements, allowing focus at multiple focal planes after the image is taken. Slices of the volume can be taken to isolate perspectives and render 2D images. Rendering a light field can be done by using ray tracing and volume rendering methods [435, 436].

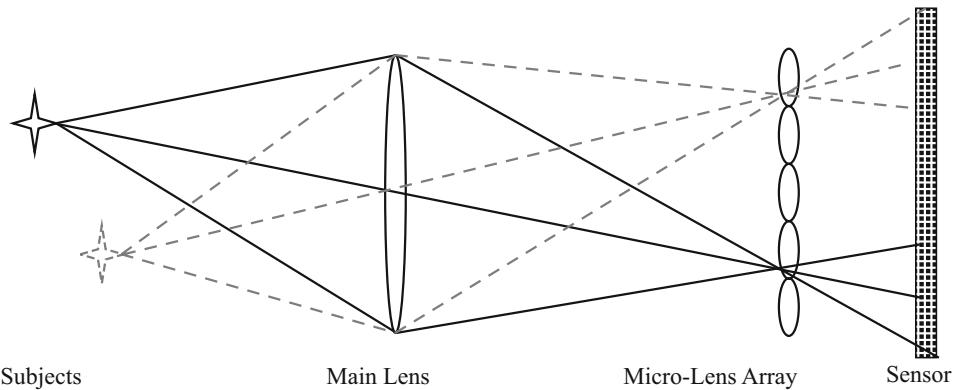


Figure 1.16 A plenoptic camera illustration. Multiple independent subjects in the scene can be processed from the same sensor image. Depth of field and focus can be computed for each subject independently after the image is taken, yielding perspective and focal plane adjustments within the 3D light field

In addition to volume and surface renderings of the light field, a 2D slice from the 3D field or volume can be processed in the frequency domain by way of the Fourier Projection Slice Theorem [437], as illustrated in Fig. 1.17. This is the basis for medical imaging methods in processing 3D MRI and CAT scan data. Applications of the Fourier Projection Slice method to volumetric and 3D range data are described by Levoy [434, 437] and Krig [129]. The basic algorithm is described as follows:

1. The volume data is forward transformed, using a 3D FFT into magnitude and phase data.
2. To visualize, the resulting 3D FFT results in the frequency volume are rearranged by *octant shifting* each cube to align the frequency 0 data around the center of a 3D Cartesian coordinate system in the center of the volume, similar to the way 2D frequency spectrums are *quadrant shifted* for frequency spectrum display around the center of a 2D Cartesian coordinate system.
3. A planar 2D slice is extracted from the volume parallel to the FOV plane where the slice passes through the origin (center) of the volume. The angle of the slice taken from the frequency domain volume data determines the angle of the desired 2D view and the depth of field.
4. The 2D slice from the frequency domain is run through an inverse 2D FFT to yield a 2D spatial image corresponding to the chosen angle and depth of field.

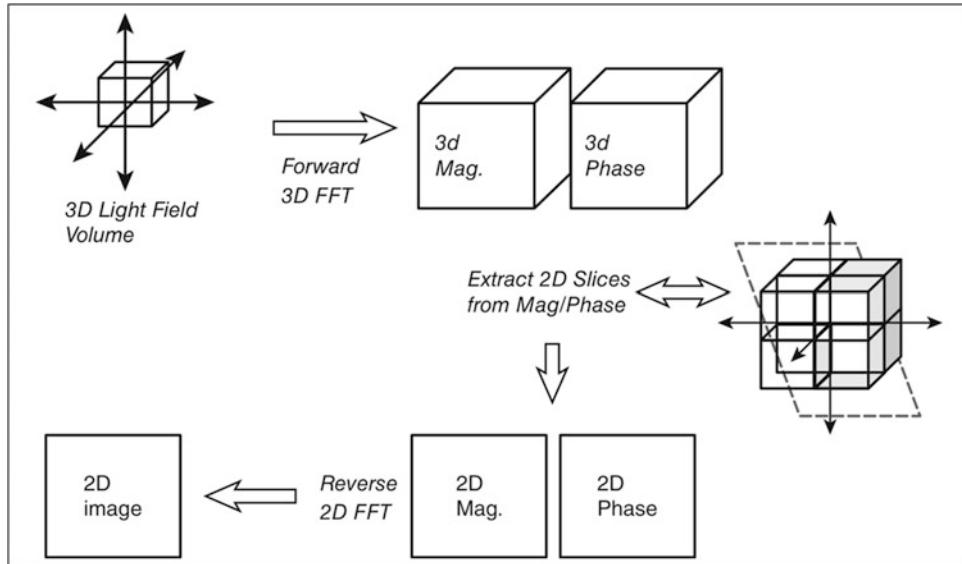


Figure 1.17 Graphic representation of the algorithm for the Fourier Projection Slice Theorem, which is one method of light field processing. The 3D Fourier space is used to filter the data to create 2D views and renderings [129, 434, 437]. (Image used by permission, © Intel Press, from Building Intelligent Systems)

3D Depth Processing

For historical reasons, several terms with their acronyms are used in discussions of depth sensing and related methods, so we cover some overlapping topics in this section. Table 1.1 earlier provided a summary at a high level of the underlying physical means for depth sensing. Regardless of the depth-sensing method, there are many similarities and common problems. Post-processing the depth information is critical, considering the calibration accuracy of the camera system, the geometric model of the depth field, the measured accuracy of the depth data, any noise present in the depth data, and the intended application.

We survey several interrelated depth-sensing topics here, including:

- Sparse depth-sensing methods
- Dense depth-sensing methods
- Optical flow
- Simultaneous localization and mapping (SLAM)
- Structure from motion (SFM)
- 3D surface reconstruction, 3D surface fusion
- Monocular depth sensing
- Stereo and multi-view stereo (MVS)
- Common problems in depth sensing

Human depth perception relies on a set of innate and learned visual cues, which are outside the scope of this work and overlap into several fields, including optics, ophthalmology, and psychology [446]; however, we provide an overview of the above selected topics in the context of depth processing.

Overview of Methods

For this discussion of depth-processing methods, depth sensing falls into two major categories based on the methods shown in Table 1.1:

- **Sparse depth methods**, using computer vision methods to extract local interest points and features. Only selected points are assembled into a sparse depth map or point cloud. The features are tracked from frame to frame as the camera or scene moves, and the sparse point cloud is updated. Usually only a single camera is needed.
- **Dense depth methods**, computing depth at every pixel. This creates a dense depth map, using methods such as stereo, TOF, or MVS. It may involve one or more cameras.

Many sparse depth methods use standard monocular cameras and computer vision feature tracking, such as optical flow and SLAM (which are covered later in this section), and the feature descriptors are tracked from frame to frame to compute disparity and sparse depth. Dense depth methods are usually based more on a specific depth camera technology, such as stereo or structured light. There are exceptions, as covered next.

Problems in Depth Sensing and Processing

The depth-sensing methods each have specific problems; however, there are some common problems we can address here. To begin, one common problem is *geometric modeling* of the depth field, which is complex, including perspective and projections. Most depth-sensing methods treat the entire field as a Cartesian coordinate system, and this introduces slight problems into the depth solutions. A camera sensor is a 2D Euclidean model, and discrete voxels are imaged in 3D Euclidean space; however, mapping between the camera and the real world using simple Cartesian models introduces geometric distortion. Other problems include those of *correspondence*, or failure to match features in separate frames, and *noise* and *occlusion*. We look at such problems in this next section.

The Geometric Field and Distortions

Field geometry is a complex area affecting both depth sensing and 2D imaging. For commercial applications, geometric field problems may not be significant, since locating faces, tracking simple objects, and augmenting reality are not demanding in terms of 3D accuracy. However, military and industrial applications often require high precision and accuracy, so careful geometry treatment is in order. To understand the geometric field problems common to depth-sensing methods, let us break down the major areas:

- Projective geometry problems, dealing with perspective
- Polar and spherical geometry problems, dealing with perspective as the viewing frustum spreads with distance from the viewer
- Radial distortion, due to lens aberrations
- Coordinate space problems, due to the Cartesian coordinates of the sensor and the voxels, and the polar coordinate nature of casting rays from the scene into the sensor

The goal of this discussion is to enumerate the problems in depth sensing, not to solve them, and to provide references where applicable. Since the topic of geometry is vast, we can only provide a few examples here of better methods for modeling the depth field. It is hoped that, by identifying the geometric problems involved in depth sensing, additional attention will be given to this important

topic. The complete geometric model, including corrections, for any depth system is very complex. Usually, the topic of advanced geometry is ignored in popular commercial applications; however, we can be sure that advanced military applications such as particle beam weapons and missile systems do not ignore those complexities, given the precision required.

Several researchers have investigated more robust nonlinear methods of dealing with projective geometry problems [447, 448] specifically by modeling epipolar geometry-related distortion as 3D *cylindrical distortion*, rather than as planar distortion, and by providing reasonable compute methods for correction. In addition, the work of Lovegrove and Davison [466] deals with the geometric field using a *spherical mosaicking* method to align whole images for depth fusion, increasing the accuracy due to the spherical modeling.

The Horopter Region, Panum's Area, and Depth Fusion

As shown in Fig. 1.18, the *Horopter* region, first investigated by Ptolemy and others in the context of astronomy, is a curved surface containing 3D points that are the same distance from the observer and at the same focal plane. *Panum's area* is the region surrounding the Horopter where the human visual system fuses points in the retina into a single object at the same distance and focal plane. It is a small miracle that the human vision system can reconcile the distances between 3D points and synthesize a common depth field! The challenge with the Horopter region and Panum's area lies in the fact that a post-processing step to any depth algorithm must be in place to correctly fuse the points the way the human visual system does. The margin of error depends on the usual variables, including baseline and pixel resolution, and the error is most pronounced toward the boundaries of the depth field and less pronounced in the center. Some of the spherical distortion is due to lens aberrations toward the edges, and can be partially corrected as discussed earlier in this chapter regarding geometric corrections during early sensor processing.

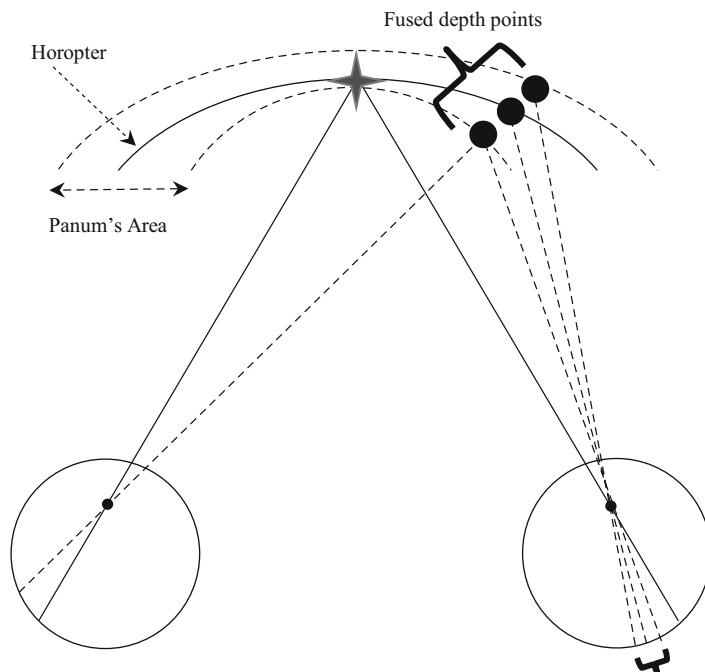


Figure 1.18 Problems with stereo and multi-view stereo methods, showing the Horopter region and Panum's area, and three points in space that appear to be the same point from the left eye's perspective but different from the right eye's perspective. The three points surround the Horopter in Panum's area and are fused by humans to synthesize apparent depth

Cartesian vs. Polar Coordinates: Spherical Projective Geometry

As illustrated in Fig. 1.19, a 2D sensor as used in a TOF or monocular depth-sensing method has specific geometric problems as well; the problems increase toward the edges of the field of view. Note that the depth from a point in space to a pixel in the sensor is actually measured in a spherical coordinate system using polar coordinates, but the geometry of the sensor is purely Cartesian, so that geometry errors are *baked into the cake*.

Because stereo and MVS methods also use single 2D sensors, the same problems as affect single sensor depth-sensing methods also affect multi-camera methods, compounding the difficulties in developing a geometry model that is accurate and computationally reasonable.

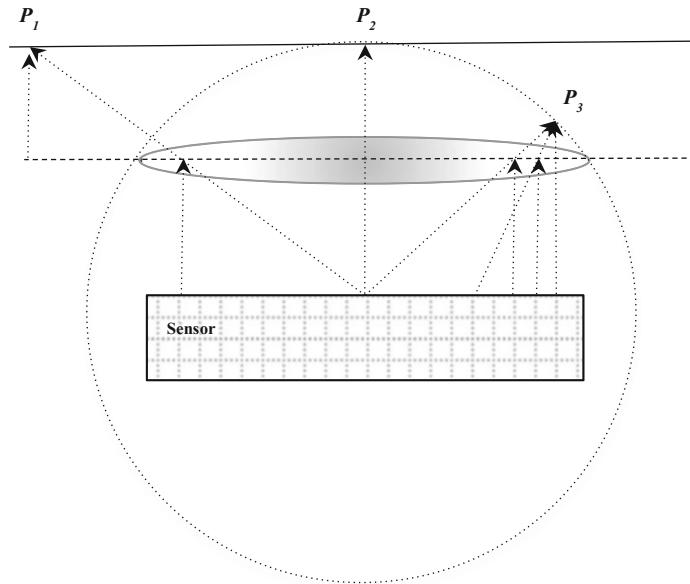


Figure 1.19 A 2D depth sensor and lens with exaggerated imaging geometry problems dealing with distance, where depth is different depending on the angle of incidence on the lens and sensor. Note that P_1 and P_2 are equidistant from the focal plane; however, the distance of each point to the sensor via the optics is not equal, so computed depth will not be accurate depending on the geometric model used

Depth Granularity

As shown in Fig. 1.20, simple Cartesian depth computations cannot resolve the depth field into a linear uniform grain size; in fact, the depth field granularity increases exponentially with the distance from the sensor, while the ability to resolve depth at long ranges is much less accurate.

For example, in a hypothetical stereo vision system with a baseline of 70 mm using 480p video resolution, as shown in Fig. 1.20, depth resolution at 10 m drops off to about 1/2 m; in other words, at 10 m away, objects may not appear to move in Z unless they move at least plus or minus 1/2 m in Z. The depth resolution can be doubled simply by doubling the sensor resolution. As distance increases, humans increasingly use monocular depth *cues* to determine depth, such as for size of objects, rate of an object's motion, color intensity, and surface texture details.

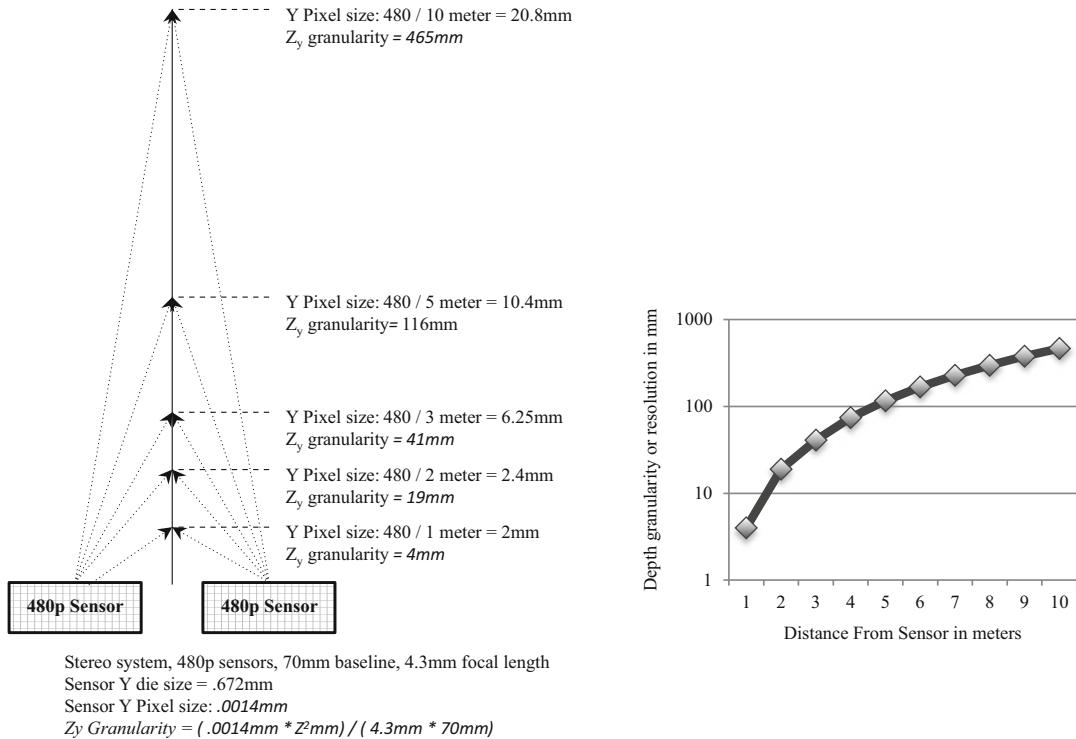


Figure 1.20 Z depth granularity nonlinearity problems for a typical stereo camera system. Note that practical depth sensing using stereo and MVS methods has limitations in the depth field, mainly affected by pixel resolution, baseline, and focal length. At 10 m, depth granularity is almost $\frac{1}{2}$ m, so an object must move at least + or - $\frac{1}{2}$ m in order for a change in measured stereo depth to be computed

Correspondence

Correspondence, or feature matching, is common to most depth-sensing methods, see Mayer [877] for novel deep learning approaches. For a taxonomy of stereo feature matching algorithms, see Scharstein and Szeliski [422]. Here, we discuss correspondence along the lines of feature descriptor methods and triangulation as applied to stereo, multi-view stereo, and structured light.

Subpixel accuracy is a goal in most depth-sensing methods, so several algorithms exist [450]. It is popular to correlate two patches or intensity templates by fitting the surfaces to find the highest match; however, Fourier methods are also used to correlate phase [449, 451], similar to the intensity correlation methods.

For stereo systems, the image pairs are rectified prior to feature matching so that the features are expected to be found along the same line at about the same scale, as shown in Fig. 1.10; descriptors with little or no rotational invariance are suitable [112, 207]. A feature descriptor such as a correlation template is fine, while a powerful method such as the SIFT feature description method [153] is overkill. The feature descriptor region may be a rectangle favoring disparity in the x -axis and expecting little variance in the y -axis, such as a rectangular 3×9 descriptor shape. The disparity is expected in the x -axis, not the y -axis. Several window sizing methods for the descriptor shape are used, including fixed size and adaptive size [422].

Multi-view stereo systems are similar to stereo; however, the rectification stage may not be as accurate, since motion between frames can include scaling, translation, and rotation. Since scale and rotation may have significant correspondence problems between frames, other approaches to feature description have been applied to MVS, with better results. A few notable feature descriptor methods applied to multi-view and wide baseline stereo include the MSER [186] method (also discussed in Chap. 6), which uses a blob-like patch, and the SUSAN [156, 157] method (also discussed in Chap. 6), which defines the feature based on an object region or segmentation with a known centroid or nucleus around which the feature exists.

For structured light systems, the type of light pattern will determine the feature, and correlation of the phase is a popular method [451]. For example, structured light methods that rely on phase-shift patterns using phase correlation [449] template matching claim to be accurate to 1/100th of a pixel. Other methods are also used for structured light correspondence to achieve subpixel accuracy [449].

Holes and Occlusion

When a pattern cannot be matched between frames, a *hole* exists in the depth map. Holes can also be caused by occlusion. In either case, the depth map must be repaired, and several methods exist for doing that. A *hole map* should be provided, showing where the problems are. A simple approach, then, is to fill the hole uses use bilinear interpolation within local depth map patches. Another simple approach is to use the last known-good depth value in the depth map from the current scan line.

More robust methods for handling occlusion exist [453, 454] using more computationally expensive but slightly more accurate methods, such as adaptive local windows to optimize the interpolation region. Yet another method of dealing with holes is *surface fusion* into a depth volume [455] (covered next), whereby multiple sequential depth maps are integrated into a depth volume as a cumulative surface, and then a depth map can be extracted from the depth volume.

Surface Reconstruction and Fusion

A general method of creating surfaces from depth map information is *surface reconstruction*. Computer graphics methods can be used for rendering and displaying the surfaces. The basic idea is to combine several depth maps to construct a better surface model, including the RGB 2D image of the surface rendered as a *texture map*. By creating an iterative model of the 3D surface that integrates several depth maps from different viewpoints, the depth accuracy can be increased, occlusion can be reduced or eliminated, and a wider 3D scene viewpoint is created.

The work of Curless and Levoy [455] presents a method of fusing multiple range images or depth maps into a 3D volume structure. The algorithm renders all range images as *iso-surfaces* into the volume by integrating several range images. Using a signed distance function and weighting factors stored in the volume data structure for the existing surfaces, the new surfaces are integrated into the volume for a cumulative best-guess at where the actual surfaces exist. Of course, the resulting surface has several desirable properties, including reduced noise, reduced holes, reduced occlusion, multiple viewpoints, and better accuracy (see Fig. 1.21).

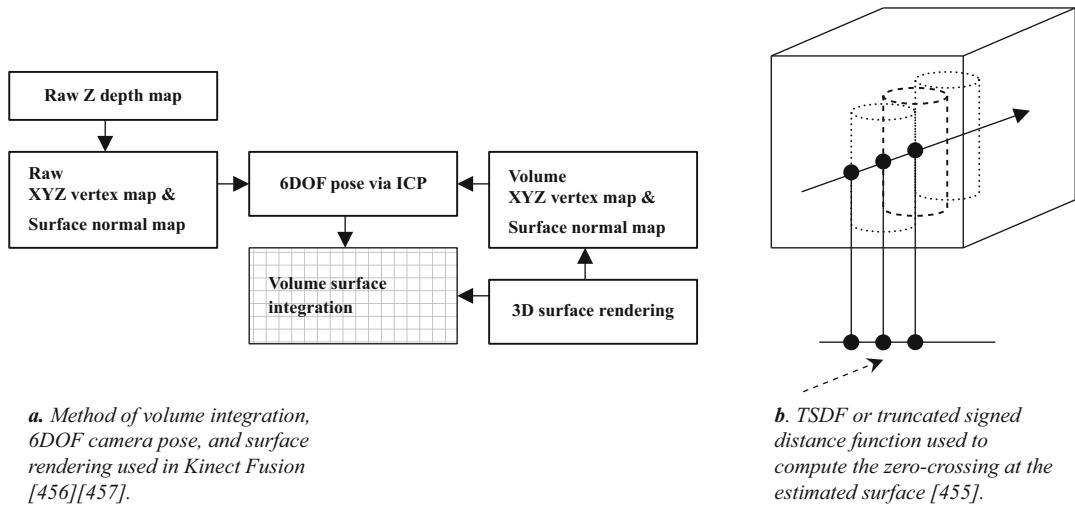


Figure 1.21 (Right) The Curless and Levoy [455] method for surface construction from range images, or depth maps. Shown here are three different weighted surface measurements projected into the volume using ray casting. (Left) Processing flow of Kinect Fusion method

A derivative of the Curless and Levoy method applied to SLAM is the Kinect Fusion approach [456], as shown in Fig. 1.22, using compute-intensive SIMD parallel real-time methods to provide not only surface reconstruction but also camera tracking and the 6DOF or *6-degrees-of-freedom* camera pose via surface alignment from frame to frame. Raytracing and texture mapping are used for surface renderings. There are yet other methods for surface reconstruction from multiple images [462, 533].

Noise

Noise is another problem with depth sensors [391], and various causes include low illumination and, in some cases, motion noise, as well as inferior depth sensing algorithms or systems. Also, the depth maps are often very fuzzy, so image preprocessing may be required, as discussed in Chap. 2, to reduce apparent noise. Many prefer the bilateral filter for depth map processing [294], since it respects local structure and preserves the edge transitions. In addition, other noise filters have been developed to remedy the weaknesses of the bilateral filter, which are well suited to removing depth noise, including the Guided Filter [468], which can perform edge-preserving noise filtering like the bilateral filter, the Edge-Avoiding Wavelet method [470], and the Domain Transform filter [471].

Monocular Depth Processing

Monocular, or single sensor depth sensing, creates a depth map from pairs of image frames using the motion from frame to frame to create the stereo disparity. The assumptions for stereo processing with a calibrated fixed geometry between stereo pairs do not hold for monocular methods, since each time the camera moves the *camera pose* must be recomputed. Camera pose is a *6 degrees-of-freedom* (6DOF) equation, including x , y , and z linear motion along each axis and roll, pitch, and yaw rotational motion about each axis. In monocular depth-sensing methods, the camera pose must be computed for each frame as the basis for comparing two frames and computing disparity.

Note that computation of the 6DOF matrix can be enhanced using *inertial sensors*, such as the accelerometer and MEMS gyroscope [465], as the coarse alignment step, followed by visual feature-based surface alignment methods discussed later in regard to optical flow. Since commodity inertial sensors are standard with mobile phones and tablets, inertial pose estimation will become more effective and commonplace as the sensors mature. While the accuracy of commodity accelerometers is not very good, monocular depth-sensing systems can save compute time by taking advantage of the inertial sensors for pose estimation.

Multi-view Stereo

The geometry model for most monocular multi-view stereo (MVS) depth algorithms is based on projective geometry and epipolar geometry; a good overview of both are found in the classic text by Hartley and Zisserman [419]. A taxonomy and accuracy comparison of six MVS algorithms is provided by Seitz et al. [460]. We look at a few representative approaches in this section.

Sparse Methods: PTAM

Sparse MVS methods create a sparse 3D point cloud, not a complete depth map. The basic goals for sparse depth are simple: track the features from frame to frame, compute feature disparity to create depth, and perform 6DOF alignment to localize the new frames and get the camera pose. Depending on the application, sparse depth may be ideal to use as part of a feature descriptor to add invariance to perspective viewpoint or to provide sufficient information for navigating that is based on a few key landmarks in the scene. Several sparse depth-sensing methods have been developed in the robotics community under the terms *SLAM*, *SFM*, and *optical flow* (discussed below).

However, we first illustrate sparse depth sensing in more detail by discussing a specific approach: *Parallel Tracking and Mapping* (PTAM) [438, 439], which can both track the 6DOF camera pose and generate a sparse depth map suitable for light-duty augmented reality applications, allowing avatars to be placed at known locations and orientations in the scene from frame to frame. The basic algorithm consists of two parts, which run in parallel threads: a tracking thread for updating the pose, and a mapping thread for updating the sparse 3D point cloud. We provide a quick overview of each next.

The *mapping thread* deals with a history buffer of the last N keyframes and an N-level image pyramid for each frame in a history buffer, from which the sparse 3D point cloud is continually refined using the latest incoming depth features via a bundle adjustment process (which simply means fitting new 3D coordinates against existing 3D coordinates by a chosen minimization method, such as the Levenberg–Marquardt [419]). The bundle adjustment process can perform either a local adjustment over a limited set of recent frames or global adjustment over all the frames during times of low scene motion when time permits.

The *tracking thread* scans the incoming image frames for expected features, based on projecting where known-good features last appeared, to guide the feature search, using the 6DOF camera pose as a basis for the projection. A FAST9 [130] corner detector is used to locate the corners, followed by a Shi–Tomasi [149] non-maximal suppression step to remove weak corner candidates (discussed in Chap. 6 in more detail). The feature matching stage follows a coarse-to-fine progression over the image pyramid to compute the 6DOF pose.

Target features are computed in new frames using an 8×8 patch surrounding each selected corner. *Reference features* are computed also as 8×8 patches from the original patch taken from the first-known image where they were found. To align the reference and target patches prior to feature matching, the surface normal of each reference patch is used for pre-warping the patch against the last-known 6DOF camera pose, and the aligned feature matching is performed using zero-mean SSD distance.

One weakness of monocular depth sensing shows up when there is a *failure to localize*; that is, if there is too much motion, or illumination changes too much, the system may fail to localize and the tracking stops. Another weakness is that the algorithm must be initialized entirely for a specific localized scene or workspace, such as a desktop. For initialization, PTAM follows a five-point stereo calibration method that takes a few seconds to perform with user cooperation. Yet another weakness is that the size of the 3D volume containing the point cloud is intended for a small, localized scene or workspace. However, on the positive side, the accuracy of the 3D point cloud is very good, close to the pixel size; the pose is accurate enough for AR or gaming applications; and it is possible to create a 360° perspective point cloud by walking around the scene. PTAM has been implemented on a mobile phone [438] using modest compute and memory resources, with trade-offs for accuracy and frame rate.

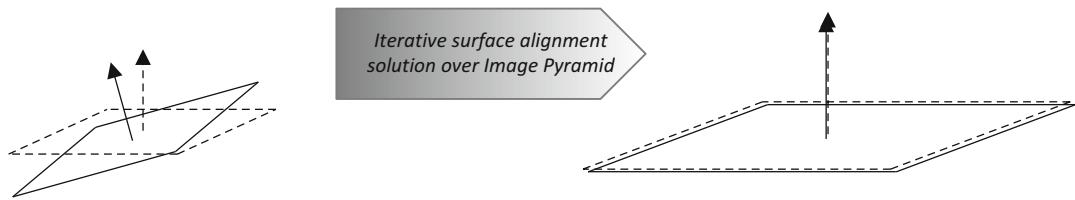


Figure 1.22 Graphic representation of the dense whole-image alignment solution of adjacent frames to obtain the 6DOF camera pose using ESM [467]

Dense Methods: DTAM

Dense monocular depth sensing is quite compute-intensive compared to sparse methods, so the research and development are much more limited. The goals are about the same as for sparse monocular depth—namely, compute the 6DOF camera pose for image alignment, but create a dense every-pixel depth map instead of a sparse point cloud. For illustration, we highlight key concepts from a method for Dense Tracking and Mapping (DTAM), developed by Newcombe et al. [464].

While the DTAM goal is to compute dense depth at each pixel rather than sparse depth, DTAM shares some of the same requirements with PTAM [439], since both are monocular methods. Both DTAM and PTAM are required to compute the 6DOF pose for each new frame in order to align the new frames to compute disparity. DTAM also requires a user-assisted monocular calibration method for the scene, and it uses the PTAM calibration method. And DTAM is also intended for small, localized scenes or workspaces. DTAM shares several background concepts taken from the Spherical Mosaicking method of Lovegrove and Davison [466], including the concept of *whole image alignment*, based on the Efficient Second Order Minimization (ESM) method [467], which is reported to find a stable surface alignment using fewer iterations than LK methods [440] as part of the process to generate the 6DOF pose.

Apparently, both DTAM and Spherical Mosaicking use a spherical coordinate geometry model to mosaic the new frames into the dense 3D surface proceeding from coarse to fine alignment over the image pyramid to iterate toward the solution of the 6DOF camera pose change from frame to frame. The idea of whole-image surface alignment is shown in Fig. 1.22. The new and existing depth surfaces are integrated using a localized guided-filter method [468] into the cost volume. That is, the guided filter uses a guidance image to merge the incoming depth information into the cost volume.

DTAM also takes great advantage of SIMD instructions and highly thread-parallel SIMT GPGPU programming to gain the required performance necessary for real-time operation on commodity GPU hardware.

Optical Flow, SLAM, and SFM

Optical flow measures the motion of features and patterns from frame to frame in the form of a displacement *vector*. Optical flow is similar to sparse monocular depth-sensing methods, and it can be applied to wide baseline stereo matching problems [445]. Since the field of optical flow research and its applications is vast [441–443], we provide only an introduction here with an eye toward describing the methods used and features obtained.

Optical flow can be considered a sparse feature-tracking problem, where a feature can be considered a *particle* [444], so optical flow and particle flow analysis are similar. Particle flow analysis is applied to diverse particle field flow-analysis problems, including weather prediction, simulating combustion and explosives, hydro-flow dynamics, and robot navigation. Methods exist for both 2D and 3D optical flow. The various optical flow algorithms are concerned with tracking-feature descriptors or matrices, rather than with individual scalars or pixels, within consecutive fields of discrete scalar values. For computer vision, the input to the optical flow algorithms is a set of sequential 2D images and pixels, or 3D volumes and voxels, and the output is a set of vectors showing direction of movement of the tracked features.

Many derivations and alternatives to the early Lucas Kanade (LK) method [440–443] are used for optical flow (see [156] for example); however, this remains the most popular reference point, as it uses local features in the form of correlation templates (as discussed in Chap. 6). Good coverage of the state-of-the-art methods based on LK is found in *Lucas Kanade 20 years on*, by Baker and Matthews [462]. The Efficient Second Order Minimization (ESM) method [467] is related to the LK method. ESM is reported to be a stable solution using fewer iterations than LK. LK does not track individual pixels; rather, it relies on the pixel neighborhood, such as a 3×3 matrix or template region, and tries to guess which direction the features have moved, iteratively searching the local region and averaging the search results using a least-squares solution to find the best guess.

While there are many variations on the LK method [441–443], key assumptions of most LK-derived optical flow methods include small displacements of features from frame to frame, rigid features, and sufficient texture information in the form of localized gradients in order to identify features. Various methods are used to find the local gradients, such as Sobel and Laplacian (discussed in Chap. 2). Fields with large feature displacements from frame to frame and little texture information are not well suited to the LK method. That is because the LK algorithm ignores regions with little gradient information by examining the eigenvalues of each local matrix to optimize the iterative solution. However, more recent and robust research methods are moving beyond the limitations of LK [441, 442], and include Deepflow [336], which is designed for deformable features and large displacement optical flow [372], using multilayer feature scale hierarchies [386] similar to convolutional networks [331].

Applications of surface reconstruction to localization and mapping are used in *simultaneous localization and mapping* (SLAM) and *in structure from motion* (SFM) methods—for example, in robotics navigation. One goal of SLAM is to localize, or find the current position and the 6DOF camera pose. Another goal is to create a local region map, which includes depth. To dig deeper into SLAM and SFM methods, see the historical survey by Bailey and Durrant-Whyte [458, 459].

3D Representations: Voxels, Depth Maps, Meshes, and Point Clouds

Depth information is represented and stored in a variety of convertible formats, depending on the intended use. We summarize here some common formats; see also Fig. 1.23.

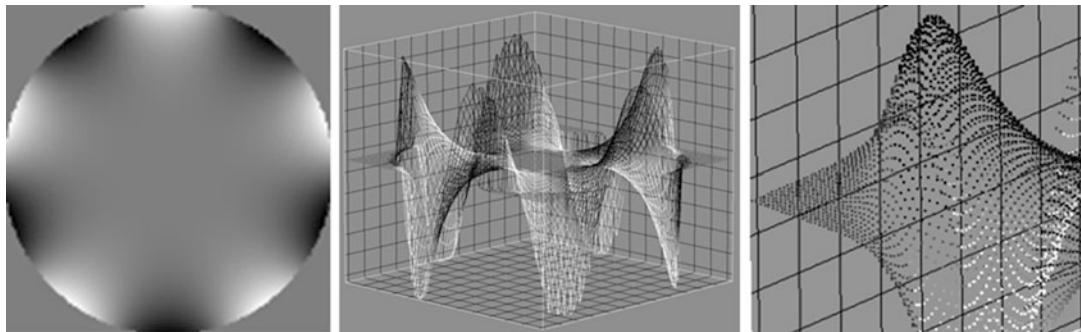


Figure 1.23 Various 3D depth formats. Renderings of a Zernike polynomial. (*Left to right*): A depth map, a polygon mesh rendering using 3D quads, a point cloud rendering equivalent of voxels

The ability to convert between depth formats is desirable for different algorithms and easy to do. Common 3D depth formats include:

- **2D Pixel Array, 3D Depth Map:** A 2D pixel array is the default format for 2D images in memory, and it is the natural storage format for many processing operations, such as convolution and neighborhood filtering. For depth map images, the pixel value is the Z, or depth value. Each point in the array may contain $\{color, depth\}$.
- **3D Voxel Volume:** A 3D volumetric data structure composed of a 3D array of voxels is ideal for several algorithms, including depth map integration for 3D surface reconstruction and raytracing of surfaces for graphical renderings. A voxel is a volume element, like a pixel is a picture element. Each voxel may contain $\{color, normal\}$; the depth coordinates are implicit from the volume structure.
- **3D Polygon Mesh:** Storing 3D points in a standard 3D polygon mesh provides a set of connected points or vertices, each having a surface normal, 3D coordinates, color, and texture. Mesh formats are ideal for rendering surfaces in a GPU pipeline, such as OpenGL or DirectX. Each point in the mesh may contain $\{x, y, z, color, normal\}$, and is associated with neighboring points in a standard pattern such as a quad or triangle describing the surface.
- **3D Point Cloud:** This is a sparse structure that is directly convertible to a standard 3D polygon mesh. The point cloud format is ideal for sparse monocular depth-sensing methods. Each point in the cloud may contain $\{x, y, z, color, normal\}$.

Summary

In this chapter, we survey image sensing methods and sensor image processing methods as the first step in the vision pipeline. We cover the image sensor technologies available, with an eye toward image preprocessing that may be useful for getting the most from the image data, since image sensing methods often dictate the image preprocessing required (More discussion on image preprocessing is provided in Chap. 2.). Sensor configurations used for both 2D and 3D imaging were discussed, as well as a wide range of camera configurations used for computational imaging to create new images after the data is captured, such as HDR images and image refocusing. Depth imaging approaches are covered here as well, and include stereo and time of flight, since mobile

devices are increasingly offering 3D depth camera technology for consumer applications. Depth maps can be used in computer vision to solve many problems, such as 3D feature description and 3D image segmentation of foreground and background objects. The topic of 3D depth processing and 3D features is followed throughout this book; Chap. 6 covers 3D feature descriptors, and Chap. 7 and Appendix B cover 3D ground truth data.

Chapter 1: Learning Assignments

1. Name at least two types of semiconductor materials used to create imaging sensors, and discuss the trade-offs between each sensor material from a manufacturing perspective, and from an end user perspective.
2. Discuss the visible RGB, IR, and UV wavelength response curve of silicon imaging sensors, and optionally draw a diagram showing the spectral responses.
3. Name at least one material that can be used as a near-IR filter for a silicon image sensor.
4. Discuss dynamic range in camera systems, bits per pixel, and when dynamic range becomes critical.
5. Discuss color cell mosaic patterns on image sensors, and some of the implications of the patterns for assembling the cells into color pixels. For example, silicon cell size and arrangement.
6. Describe how color de-mosaicking algorithms work.
7. Describe a range of camera and image sensor calibrations, and how they are established.
8. Name a few sensor calibration adjustments that must be made to the image sensor color cell data after sensor readout, prior to assembling the color cells into RGB pixels.
9. Discuss a few types of corrections that must be made to the assembled pixels after they are assembled from the image sensor.
10. Describe how to compose a high dynamic range (HDR) image from several image frames.
11. Describe how to compute the data rate to read out pixels from an RGB camera, assuming each RGB component contains 16 bits, the frame rate is 60 frames per second, and the frame size is 7680×4320 pixels (UHDTV).
12. Describe at a high level at least three methods for computing depth from camera images, including stereo, multi-view stereo, structured or coded light, and time of flight sensors.
13. Discuss the trade-offs between stereo depth sensing and monocular depth sensing.
14. Discuss the basic steps involved in stereo algorithms, such as image rectification and alignment, and other steps.
15. Describe structured light patterns, and how they work.
16. Describe how the Horopter region and Panum's area affect depth sensing.
17. Discuss problems created by occlusion in stereo processing, such as holes in the stereo field, and how the problems can be solved.
18. Describe how 2D surface fusion of several images can be performed using a 3D voxel buffer.
19. Discuss how monocular depth sensing is similar to stereo depth sensing.
20. Describe the calibration parameters for a stereo camera system, including baseline.
21. Describe how to compute the area a pixel covers in an image at a given distance from the camera.
HINT: camera sensor resolution is one variable.
22. Discuss voxels, depth maps, and point clouds.

"I entered, and found Captain Nemo deep in algebraical calculations of x and other quantities."

—Jules Verne, 20,000 Leagues Under The Sea

This chapter describes the methods used to prepare images for further analysis, including interest point and feature extraction. The focus is on image preprocessing for computer vision, so we do not cover the entire range of image processing topics applied to areas such as computational photography and photo enhancements, so we refer the interested reader to various other standard resources in Digital Image Processing and Signal Processing as we go along [4, 9, 317, 318], and we also point out interesting research papers that will enhance understanding of the topics.

Note Readers with a strong background in image processing may benefit from a light reading of this chapter.

Perspectives on Image Processing

Image processing is a vast field that cannot be covered in a single chapter. So why do we discuss image preprocessing in a book about computer vision? The reason is image preprocessing is typically ignored in discussions of feature description. Some general image processing topics are covered here in light of feature description, intended to illustrate rather than to proscribe, as applications and image data will guide the image preprocessing stage.

Some will argue that image preprocessing is not a good idea, since it distorts or changes the true nature of the raw data. However, intelligent use of image preprocessing can provide benefits and solve problems that ultimately lead to better local and global feature detection. We survey common methods for image enhancements and corrections that will affect feature analysis downstream in the vision pipeline in both favorable and unfavorable ways, depending on how the methods are employed.

Image preprocessing may have dramatic positive effects on the quality of feature extraction and the results of image analysis. Image preprocessing is analogous to the mathematical normalization of a data set, which is a common step in many feature descriptor methods. Or to make a musical analogy, think of image preprocessing as a sound system with a range of controls, such as raw sound with no volume controls; volume control with a simple tone knob; volume control plus treble, bass, and mid;

or volume control plus a full graphics equalizer, effects processing, and great speakers in an acoustically superior room. In that way, this chapter promotes image preprocessing by describing a combination of corrections and enhancements that are an essential part of a computer vision pipeline.

Problems to Solve During Image Preprocessing

Raw image data direct from a camera may have a variety of problems, as discussed in Chap. 1, and therefore it is not likely to produce the best computer vision results. This is why careful consideration of image preprocessing is fundamental. For example, a local binary descriptor using gray scale data will require different preprocessing than will a color SIFT algorithm; additionally, some exploratory work is required to fine-tune the image preprocessing stage for best results. We explore image preprocessing by following the vision pipelines of four fundamental families of feature description methods, with some examples, as follows:

1. **Local Binary Descriptors** (LBP, ORB, FREAK, others)
2. **Spectra Descriptors** (SIFT, SURF, others)
3. **Basis Space Descriptors** (FFT, wavelets, others)
4. **Polygon Shape Descriptors** (blob object area, perimeter, centroid)

These families of feature description metrics are developed into a taxonomy in Chap. 5. Before that, though, Chap. 4 discusses feature descriptor building concepts, while Chap. 3 covers global feature description and then Chap. 6 surveys local feature description. The image preprocessing methods and applications introduced here are samples, but a more developed set of examples, following various vision pipelines, is developed in Chap. 8, including application-specific discussions of the preprocessing stage.

Vision Pipelines and Image Preprocessing

Table 2.1 lists common image preprocessing operations, with examples from each of the four descriptor families, illustrating both differences and commonality among these image preprocessing steps, which can be applied prior to feature description.

Table 2.1 Possible Image Pre-Processing Enhancements and Corrections as Applied to Different Vision Pipelines

Image Pre-Processing	Local Binary (LBP,ORB)	Spectra (SIFT,SURF)	Basis Space (FFT, Codebooks)	Polygon Shape (Blob Metrics)
Illumination corrections	x	x	x	x
Blur and focus corrections	x	x	x	x
Filtering and noise removal	x	x	x	x
Thresholding				x
Edge enhancements		x		x
Morphology				x
Segmentation				x
Region processing and filters		x	x	x
Point processing		x		x
Math and statistical processing		x		x
Color space conversions	x	x	x	x

Local binary features deal with the pixel intensity comparisons of *point-pairs*. This makes the comparisons relatively insensitive to local illumination, brightness, and contrast, so there may not be much need for image preprocessing to achieve good results. Current local binary pattern methods as described in the literature do not typically call for much image preprocessing; they rely on a simple local comparison threshold that can be adjusted to account for illumination or contrast.

Spectra descriptors, such as SIFT (which acts on local region gradients) and SURF (which uses HAAR-like features with integrated pixel values over local regions), offer diverse preprocessing opportunities. Methods that use image pyramids often perform some image preprocessing on the image pyramid to create a scale space representation of the data using Gaussian filtering to smooth the higher levels of the pyramid. Basic illumination corrections and filtering may be useful to enhance the image prior to computing gradients—for example, to enhance the contrast within a band of intensities that likely contain gradient-edge information for the features. But in general, the literature does not report good or bad results for any specific methods used to preprocess the image data prior to feature extraction, and therein resides the opportunity.

Basis space features are usually global or regional, spanning a regular shaped such as a Fourier transform computed over the entire image or block. However, basis space features may be part of the local features, such as the Fourier spectrum of the LBP histogram, which can be computed over histogram bin values of a local descriptor to provide rotational invariance. Another example is the Fourier descriptor used to compute polygon factors for radial line segment lengths showing the roundness of a feature to provide rotational invariance. See Chap. 3, especially Fig. 3.20.

The most complex descriptor family is the polygon shape based descriptors, which potentially require several image preprocessing steps to isolate the polygon structure and shapes in the image for measurement. Polygon shape description pipelines may involve everything from image enhancements to structural morphology and segmentation techniques. Setting up the preprocessing for polygon feature shape extraction typically involves more work than any other method, since thresholds and segmentation require fine-tuning to achieve good results. Also note that polygon shape descriptors are not local patterns but, rather, larger regional structures with features spanning many tens and even hundreds of pixels, so the processing can be more intensive as well.

In some cases, image preprocessing is required to correct problems that would otherwise adversely affect feature description; we look at this next.

Corrections

During image preprocessing, there may be artifacts in the images that should be corrected prior to feature measurement and analysis. Here are various candidates for correction.

- **Sensor corrections.** Discussed in Chap. 1, these include dead pixel correction, geometric lens distortion, and vignetting.
- **Lighting corrections.** Lighting can introduce deep shadows that obscure local texture and structure; also, uneven lighting across the scene might skew results. Candidate correction methods include rank filtering, histogram equalization, and LUT remap.
- **Noise.** This comes in many forms, and may need special image preprocessing. There are many methods to choose from, some of which are surveyed in this chapter.
- **Geometric corrections.** If the entire scene is rotated or taken from the wrong perspective, it may be valuable to correct the geometry prior to feature description. Some features are more robust to geometric variation than others, as discussed in Chaps. 4–6.
- **Color corrections.** It can be helpful to redistribute color saturation or correct for illumination artifacts in the intensity channel. Typically color hue is one of the more difficult attributes to correct, and it may not be possible to correct using simple gamma curves and the sRGB color space. We cover more accurate colorimetry methods later in this chapter.

Enhancements

Enhancements are used to optimize for specific feature measurement methods, rather than fix problems. Familiar image processing enhancements include sharpening and color balancing. Here are some general examples of image enhancement with their potential benefits to feature description.

- **Scale-space pyramids.** When a pyramid is constructed using an octave scale (or a non-octave scale interval) and pixel decimation to subsample images to create the pyramid, subsampling artifacts and jagged pixel transitions are introduced. Part of the scale-space pyramid building process involves applying a Gaussian blur filter to the subsampled images, which removes the jagged artifacts. Also, anti-aliased scaling is available in the GPU hardware.
- **Illumination.** In general, illumination can always be enhanced. Global illumination can be enhanced using simple LUT remapping and pixel point operations and histogram equalizations, and pixel remapping. Local illumination can be enhanced using gradient filters, local histogram equalization, and rank filters.
- **Blur and focus enhancements.** Many well-known filtering methods for sharpening and blurring may be employed at the preprocessing stage. For example, to compensate for pixel aliasing artifacts introduced by rotation that may manifest as blurred pixels which obscure fine detail, sharpen filters can be used to enhance the edge features prior to gradient computations. Or, conversely, the rotation artifacts may be too strong and can be removed by blurring.

In any case, the preprocessing enhancements or corrections are dependent on the descriptor using the images, and the application.

Preparing Images for Feature Extraction

Each family of feature description methods has different goals for the preprocessing stage of the pipeline. Let us look at a few examples from each family here, and examine possible image preprocessing methods for each.

Local Binary Family Preprocessing

The local binary descriptor family is primarily concerned with point-pair intensity value comparisons, and several point-pair patterns are illustrated in Chap. 4 for common methods such as FREAK, BRISK, BRIEF, and ORB. As illustrated in Fig. 2.1, the *comparative difference* ($<$, $>$, $=$) between points is all that matters, so hardly any image preprocessing seems needed. Based on this discussion, here are two approaches for image preprocessing:

1. **Preserve pixels as is.** Do nothing except use a pixel value-difference compare threshold, such as done in the Census transform and other methods, since the threshold takes care of filtering noise and other artifacts.

if $(|point1 - point2| > threshold)$

2. **Use filtering.** In addition to using the compare threshold, apply a suitable filter to remove local noise, such as a smoothing or rank filter. Or, take the opposite approach and use a sharpen filter to amplify small differences, perhaps followed by a smoothing filter. Either method may prove to work, depending on the data and application.

Fig. 2.1 uses center-neighbor point-pair comparisons in a 3×3 local region to illustrate the difference between local threshold and a preprocessing operation for the local binary pattern LBP, as follows:

- Left image: Original unprocessed local 3×3 region data; compare threshold = 5, dark pixels > 5 from center pixel.
- Left center image: Compare threshold = 10; note pattern shape is different simply by changing the threshold.
- Right center image: After a Laplacian sharpening filter is applied to 3×3 region, note that the center pixel value is changed from 52 to 49, so with the compare threshold set to 5 the pattern is now different from original on the left.
- Right image: Threshold on Laplacian filtered data set to 10; note different resulting binary pattern.

35	53	59	35	53	59	35	53	59	35	53	59
38	52	47	38	52	47	38	49	47	38	49	47
48	60	51	48	60	51	48	60	51	48	60	51

Figure 2.1 How the LBP can be affected by preprocessing, showing the compare threshold value effects. (Left) Compare threshold = 5. (Center left) Compare threshold = 10. (Center right) Original data after Laplacian filter applied. (Right) Compare threshold = 5 on Laplacian filtered data

Spectra Family Preprocessing

Due to the wide range of methods in the spectra category, it is difficult to generalize the potential preprocessing methods that may be useful. For example, SIFT is concerned with gradient information computed at each pixel. SURF is concerned with combinations of HAAR wavelets or local rectangular regions of integrated pixel values, which reduces the significance of individual pixel values.

For the integral image-based methods using HAAR-like features such as SURF and Viola Jones, here are a few hypothetical preprocessing options.

1. **Do nothing.** HAAR features are computed from integral images simply by summing local region pixel values; no fine structure in the local pixels is preserved in the sum, so one option is to do nothing for image preprocessing.
2. **Noise removal.** This does not seem to be needed in the HAAR preprocessing stage, since the integral image summing in local regions has a tendency to filter out noise.
3. **Illumination problems.** This may require preprocessing; for example, contrast enhancement may be a good idea if the illumination of the training data is different from the current frame. One preprocessing approach in this situation is to compute a global contrast metric for the images in the training set, and then compute the same global contrast metric in each frame and adjust the image contrast if the contrast diverges beyond a threshold to get closer to the desired global contrast metric. Methods for contrast enhancement include LUT remapping, global histogram equalization, and local adaptive histogram equalization.
4. **Blur.** If blur is a problem in the current frame, it may manifest similar to a local contrast problem, so local contrast enhancement may be needed, such as a sharpen filter. Computing a global statistical metric such as an SDM as part of the ground truth data to measure local or global contrast may be useful; if the current image diverges too much in contrast, a suitable contrast enhancement may be applied as a preprocessing step.

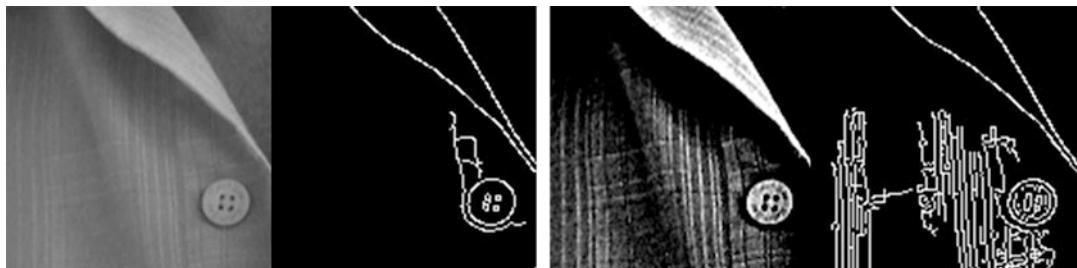


Figure 2.2 The effects of local contrast on gradients and edge detection: (*Left*) Original image and Sobel edges. (*Right*) Contrasted adjusted image to amplify local region details and resulting Sobel edges

Note in Fig. 2.2 that increasing the local-region contrast results in larger gradients and more apparent edges. A feature descriptor that relies on local gradient information is affected by the local contrast.

For the SIFT-type descriptors that use local area gradients, preprocessing may be helpful to enhance the local gradients prior to computation, so as to affect certain features:

1. **Blur.** This will inhibit gradient magnitude computation and may make it difficult to determine gradient direction, so perhaps a local rank filter, high-pass filter, or sharpen filter should be employed.
2. **Noise.** This will exacerbate local gradient computations and make them unreliable, so perhaps applying one of several existing noise-removal algorithms can help.
3. **Contrast.** If local contrast is not high enough, gradient computations are difficult and unreliable. Perhaps a local histogram equalization, LUT remap, rank filter, or even a sharpen filter can be applied to improve results.

Basis Space Family Preprocessing

It is not possible to generalize image preprocessing for basis space methods, since they are quite diverse, according to the taxonomy we are following in this work. As discussed in Chaps. 4–6, basis space methods include Fourier, wavelets, visual vocabularies, KLT, and others. However, here we provide a few general observations on preprocessing.

1. **Fourier Methods, Wavelets, Slant transform, Walsh Hadamard, KLT.** These methods transform the data into another domain for analysis, and it is hard to suggest any preprocessing without knowing the intended application. For example, computing the Fourier spectrum produces magnitude and phase, and phase is shown to be useful in feature description to provide invariance to blur, as reported in the LPQ linear phase quantization method described in Chap. 6, so a blurry image may not be a problem in this case.
2. **Sparse coding and visual vocabularies.** These methods which employ local feature descriptors, which could be SURF, SIFT, LBP, or any other desired feature, are derived from pixels in the spatial domain. Therefore, the method for feature description will determine the best approach for preprocessing. For example, methods that use correlation and raw pixel patches as sparse codes may not require any preprocessing. Or perhaps some minimal preprocessing can be used, such as illumination normalization to balance contrast, local histogram equalization or a LUT contrast remap.

In Fig. 2.3, the contrast adjustment does not have much affect on Fourier methods, since there is no dominant structure in the image. Fourier spectrums typically reveal that the dominant structure and power is limited to lower frequencies that are in the center of the quadrant-shifted 2D plot. For images with dominant structures, such as lines and other shapes, the Fourier power spectrum will show the structure and perhaps preprocessing may be more valuable. Also, the Fourier

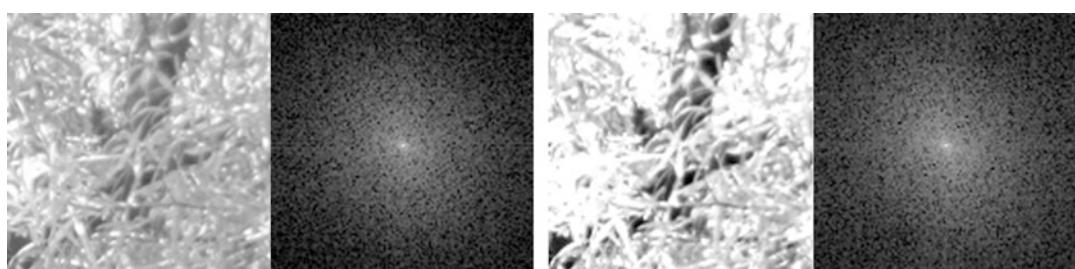


Figure 2.3 In this example, no benefit is gained from preprocessing as shown in the Fourier spectrum; (*Left*) Before. (*Right*) After contrast adjusting the input image

power spectrum display is scaled to a logarithmic value and does not show all the details linearly, so a linear spectrum rendering might show the lower frequencies scaled and magnified better for ease of viewing.

Polygon Shape Family Preprocessing

Polygon shapes are potentially the most demanding features when considering image preprocessing steps, since as shown in Table 2.1, the range of potential preprocessing methods is quite large and the choice of methods to employ is very data-dependent. Possibly because of the challenges and intended use-cases for polygon shape measurements, they are used only in various niche applications, such as cell biology.

One of the most common methods employed for image preparation prior to polygon shape measurements is to physically correct the lighting and select the subject background. For example, in automated microscopy applications, slides containing cells are prepared with fluorescent dye to highlight features in the cells, then the illumination angle and position are carefully adjusted under magnification to provide a uniform background under each cell feature to be measured; the resulting images are then much easier to segment.



Figure 2.4 Use of thresholding to solve problems during image preprocessing to prepare images for polygon shape measurement: (Left) Original image. (Center) Thresholded red channel image. (Right) Perimeter tracing above a threshold



Figure 2.5 Another sequence of morphological preprocessing steps preceding polygon shape measurement: (Left) Original image. (Center) Range thresholded and dilated red color channel. (Right) Morphological perimeter shapes taken above a threshold

As illustrated in Figs. 2.4 and 2.5, if the preprocessing is wrong, the resulting shape feature descriptors are not very useful. Next we list some of the more salient options for preprocessing prior to shape based feature extraction, then we will survey a range of other methods later in this chapter.

1. **Illumination corrections.** Typically critical for defining the shape and outline of binary features. For example, if perimeter tracking or boundary segmentation is based on edges or thresholds, uneven illumination will cause problems, since the boundary definition becomes indistinct. If the illumination cannot be corrected, then other segmentation methods not based on thresholds are available, such as texture-based segmentation.
2. **Blur and focus corrections.** Perhaps not as critical as illumination for polygon shape detection, since the segmentation of object boundary and shape is less sensitive to blur.
3. **Filtering and noise removal.** Shape detection is somewhat tolerant of noise, depending on the type of noise. Shot noise or spot noise may not present a problem, and is easily removed using various noise-cleaning methods.
4. **Thresholding.** This is critical for polygon shape detection methods. Many thresholding methods are employed, ranging from the simple binary thresholding to local adaptive thresholding methods discussed later in this chapter. Thresholding is a problematic operation and requires algorithm parameter fine-tuning in addition to careful control of the light source position and direction to deal with shadows.
5. **Edge enhancements.** May be useful for perimeter contour definition.
6. **Morphology.** One of the most common methods employed to prepare polygon shapes for measurement, covered later in this chapter in some detail. Morphology is used to alter the shapes, presumably for the better, mostly by combinations or pipelines of erosion and dilation operations, as shown in Fig. 2.5. Morphological examples include object area boundary cleanup, spur removal, and general line and perimeter cleanup and smoothing.
7. **Segmentation.** These methods use structure or texture in the image, rather than threshold, as a basis for dividing an image into connected regions or polygons. A few common segmentation methods are surveyed later in this chapter.
8. **Area/Region processing.** Convolution filter masks such as sharpen or blur, as well as statistical filters such as rank filters or media filters, are potentially useful prior to segmentation.
9. **Point processing.** Arithmetic scaling of image data point by point, such as multiplying each pixel by a given value followed by a clipping operation, as well as LUT processing, often is useful prior to segmentation.
10. **Color space conversions.** Critical for dealing accurately with color features, covered later in this chapter.

As shown In Fig. 2.4, a range thresholding method uses the red color channel, since the table background has a lot of red color and can be thresholded easily in red to remove the table top. The image is thresholded by clipping values outside an intensity band; note that the bottom right USB stick is gone after thresholding, since it is red and below the threshold. Also note that the bottom center white USB stick is also mostly gone, since it is white (max RGB values) and above the threshold. The right image shows an attempt to trace a perimeter above a threshold; it is still not very good, as more preprocessing steps are needed.

The Taxonomy of Image Processing Methods

Before we survey image preprocessing methods, it is useful to have a simple taxonomy to frame the discussion. The taxonomy suggested is a set of operations, including point, line, area, algorithmic,

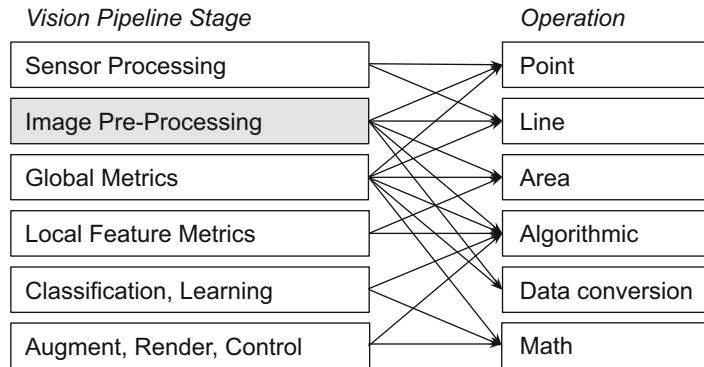


Figure 2.6 Simplified, typical image processing taxonomy, as applied across the vision pipeline

and data conversions, as illustrated in Fig. 2.6. The basic categories of image preprocessing operations introduced in Table 2.1 fit into this simple taxonomy. Note that each stage of the vision pipeline, depending on intended use, may have predominant tasks and corresponding preprocessing operations.

We provide a brief introduction to the taxonomy here, followed by a more detailed discussion in Chap. 5. Note that the taxonomy follows memory layout and memory access patterns for the image data. Memory layout particularly affects performance and power.

Point

Point operations deal with 1 pixel at a time, with no consideration of neighboring pixels. For example, point processing operations can be divided into math, Boolean, and pixel value compare substitution sections, as shown in Table 2.2 in the section later on “Point Filtering.” Other point processing examples include color conversions and numeric data conversions.

Line

Line operations deal with discrete lines of pixels or data, with no regard to prior or subsequent lines. Examples include the FFT, which is a separable transform, where pixel lines and columns can be independently processed in parallel as 1D FFT line operations. If an algorithm requires lines of data, then optimizations for image preprocessing memory layout, pipelined read/write, and parallel processing can be made. Optimizations are covered in Chap. 8.

Area

Area operations typically require local blocks of pixels—for example, spatial filtering via kernel masks, convolution, morphology, and many other operations. Area operations generate specific types of memory traffic, and can be parallelized using fine-grained methods such as common shaders in graphics processors and coarse-grained thread methods.

Algorithmic

Some image preprocessing methods are purely serial or algorithmic code. It is difficult or even impossible to parallelize these blocks of code. In some cases, algorithmic blocks can be split into a few separate threads for coarse-grained parallelism or else pipelined, as discussed in Chap. 8.

Data Conversions

While the tasks are mundane and obvious, significant time can be spent doing simple data conversions. For example, integer sensor data may be converted to floating point for geometric computations or color space conversions. Data conversions are a significant part of image preprocessing in many cases. Example conversions include:

- Integer bit-depth conversions (8/16/32/64)
- Floating point conversions (single precision to double precision)
- Fixed point to integer or float
- Any combination of float to integer and vice versa
- Color conversions to and from various color spaces
- Conversion for basis space compute, such as integer to and from float for FFT

Design attention to data conversions and performance are in order and can provide a good return on investment, as discussed in Chap. 8.

Colorimetry

In this section, we provide a brief overview of color science to guide feature description, with attention to color accuracy, color spaces, and color conversions. If a feature descriptor is using color, then the color representation and processing should be carefully designed, accurate, and suited to the application. For example, in some applications it is possible to recognize an object using color alone, perhaps recognizing an automobile using its paint color, assuming that the vendor has chosen a unique paint color each year for each model. By combining a color descriptor (see [880–882], especially van de Weijer and Schmidt [883]) with another simple feature, such as shape, an effective multivariate descriptor can be devised.

Color Science is a well-understood field defined by international standards and amply described in the literature [241–243]. We list only a few resources here.

- The Rochester Institute of Technology's Munsell Color Science Laboratory is among the leading research institutions in the area of color science and imaging. It provides a wide range of resources and has strong ties to industry imaging giants such as Kodak, Xerox, and others.
- The International Commission on Illumination (CIE) provides standard illuminant data for a range of light sources as it pertains to color science, as well as standards for the well-known color spaces CIE XYZ, CIE Lab, and CIE Luv.

- The ICC International Color Consortium provides the ICC standard color profiles for imaging devices, as well as many other industry standards, including the sRGB color space for color displays.
- Proprietary color management systems, developed by industry leaders, include the Adobe CMM and Adobe RGB, Apple ColorSync, and HP ColorSmart; perhaps the most advanced is Microsoft's Windows Color System, which is based on Canon's earlier Kyuanos system using on CIECAM02.

Overview of Color Management Systems

A full-blown color management system may not be needed for a computer vision application, but the methods of color management are critical to understand when you are dealing with color. As illustrated in Fig. 2.7, a color management system converts colors between the device color spaces, such as RGB or sRGB, to and from a *colorimetric color space*, such as CIE Luv, Lab, Jch, or Jab, so as to perform *color gamut mapping*. Since each device can reproduce color only within a specific gamut or color range, gamut mapping is required to convert the colors to the closest possible match, using the mathematical models of each color device.

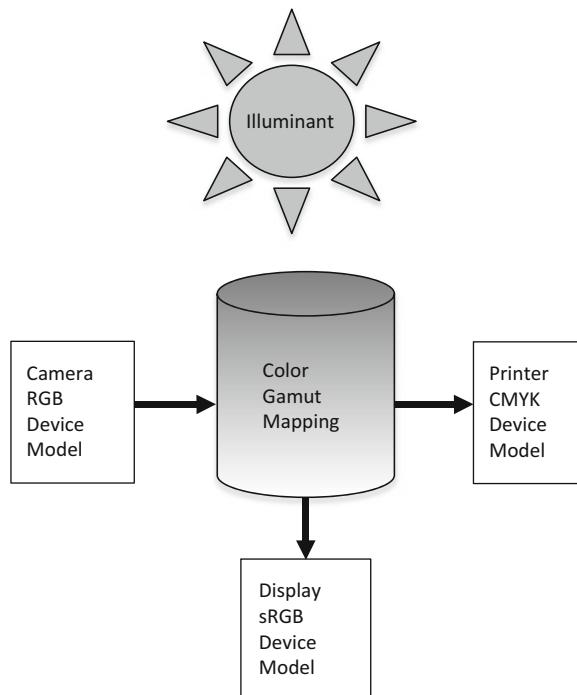


Figure 2.7 Color management system with an RGB camera device model, sRGB display device model, CMYK printer device model, gamut mapping module, and an illuminant model

Illuminants, White Point, Black Point, and Neutral Axis

An *illuminant* is a light source such as natural light or a fluorescent light, defined as the *white point* color by its spectral components and spectral power or color temperature. The white point color value in real systems is never perfectly white and is a measured quantity. The white point value and the oppositional *black point* value together define the endpoints of the *neutral axis* (gray scale intensity) of the color space, which is not a perfectly straight color vector.

Color management relies on accurate information and measurements of the light source, or the illuminant. Color cannot be represented without accurate information about the light source under which the color is measured, since color appears different under fluorescent light versus natural light, and so on. The CIE standards define several values for standard illuminants, such as D65, shown in Fig. 2.8.

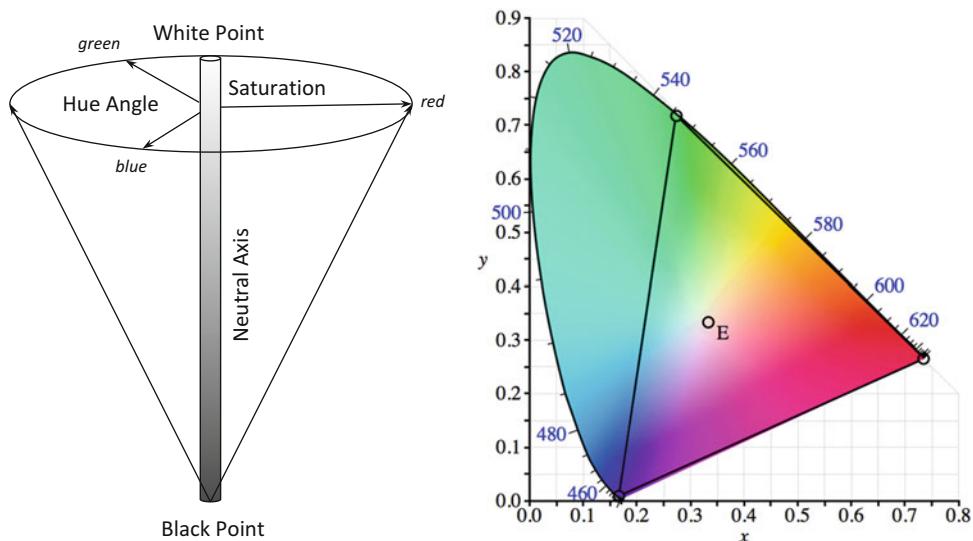


Figure 2.8 (Left) Representation of a color space in three dimensions, neutral axis for the amount of white, hue angle for the primary color, and saturation for amount of color present. (Right) CIE XYZ chromaticity diagram showing values of the standard illuminant D65 OE as the white point, and the color primaries for R, G and B

Device Color Models

Real devices like printers, displays, and cameras conventionally reproduce colors as compared against standard color patches that have been measured using calibrated light sources and spectrographic equipment—for example, the widely used Munsell color patches that define color in terms hue, value, and chroma (HVC) against standard illuminants. In order to effectively manage colors for a given device, a mathematical model or device color model must be created for each device, defining the anomalies in the device color gamut and its color gamut range.

For the color management system to be accurate, each real device must be spectrally characterized and modeled in a laboratory to create a mathematical device model, mapping the color gamut of each device against standard illumination models. The device model is used in the gamut transforms between color spaces.

Devices typically represent color using the primary and secondary colors RGB and CYMK. RGB is a primary, additive color space; starting with black, the RGB color primaries red, green, and blue are added to create colors. CYMK is a secondary color space, since the color components cyan, yellow, and magenta, are secondary combinations of the RGB primary colors; cyan = green plus blue, magenta = red plus blue, and yellow = red plus green. CYMK is also a subtractive color space, since the colors are subtracted from a white background to create specific colors.

Color Spaces and Color Perception

Colorimetric spaces represent color in abstract terms such as lightness, hue or color, and color saturation. Each color space is designed for a different reason, and each color space is useful for different types of analysis and processing. Example simple color spaces include HSV (hue, saturation, value) and HVC (hue, value, chroma). In the case of the CIE color spaces, the RGB color components are replaced by the standardized value CIE XYZ components as a basis for defining the CIE Luv and CIE Lab color spaces.

At the very high end of color science, we have the more recent CIECAM02 color models and color spaces such as Jch and Jab. CIECAM02 goes beyond just the colorimetry of the light source and the color patch itself to offer advanced color appearance modeling considerations that include the surroundings under which colors are measured [241, 246].

While CIECAM02 may be overkill for most applications, it is worth some study. Color perception varies widely based on the surrounding against which the colors are viewed, the spectrum and angles of combined direct and ambient lighting, and the human visual system itself, since people do not all perceive color in the same way.

Gamut Mapping and Rendering Intent

Gamut mapping is the art and science of converting color between two color spaces and getting the best fit. Since the color gamuts of each device are different, gamut mapping is a challenge, and there are many different algorithms in use, with no clear winner. Depending on the intent of the rendering, different methods are useful—for example, gamut mapping from camera color space to a printer color space is different from mapping to an LCD display for viewing.

The CAM02 system provides a detailed model for guidance. For example, a color imaging device may capture the color blue very weakly, while a display may be able to display blue very well. Should the color gamut fitting method use color clipping or stretching? How should the difference between color gamuts be computed? Which color space? For an excellent survey of over 90 gamut mapping methods, see the work of Morovic [244].

In Fig. 2.9 (left image), the sRGB color space is shown as fitting inside the Adobe RGB color space, illustrating that sRGB does not cover a gamut as wide as Adobe RGB. Each color gamut reproduces color differently, and each color space may be linear or warped internally. The right image in Fig. 2.9 illustrates one gamut mapping method to determine the nearest color common to both color gamuts, using Euclidean distance and clipping; however, there are many other gamut mapping distance methods as well. Depending on the surrounding light and environment, color perception changes further complicating gamut mapping.

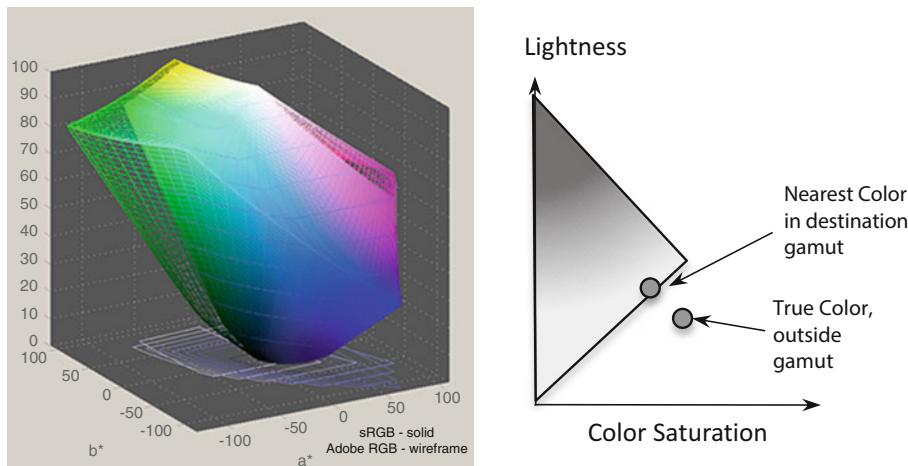


Figure 2.9 The central problem of gamut mapping: (Left) Color sRGB and Adobe RGB color gamuts created using Gamutvision software. (Right) Gamut mapping details

In gamut mapping there is a source gamut and a destination gamut. For example, the source could be a camera and the destination could be an LCD display. Depending on the rendering intent of the gamut conversion, different algorithms have been developed to convert color from source to destination gamuts. Using the *perceptual intent*, color saturation is mapped and kept within the boundaries of the destination gamut in an effort to preserve relative color strength; and out-of-gamut colors from the source are compressed into the destination gamut, which allows for a more reversible gamut map translation. Using the *colorimetric intent*, colors may be mapped straight across from source to destination gamut, and colors outside the destination gamut are simply clipped.

A common method of color correction is to rely on a simple gamma curve applied to the intensity channel to help the human eye better visualize the data, since the gamma curve brightens up the dark regions and compresses the light regions of the image, similar to the way the human visual system deals with light and dark regions. However, gamut correction bears no relationship to the true sensor data, so a calibrated, colorimetrically sound approach is recommended instead.

Practical Considerations for Color Enhancements

For image preprocessing, the color intensity is usually the only color information that should be enhanced, since the color intensity alone carries a lot of information and is commonly used. In addition, color processing cannot be easily done in RGB space while preserving relative color. For example, enhancing the RGB channels independently with a sharpen filter will lead to Moiré fringe artifacts when the RGB channels are recombined into a single rendering. So to sharpen the image, first *forward-convert* RGB to a color space such as HSV or YIQ, then sharpen the V or Y component, and then *inverse-convert* back to RGB. For example, to correct illumination in color, standard image processing methods such as LUT remap or histogram equalization will work, provided they are performed in the intensity space.

As a practical matter, for quick color conversions to gray scale from RGB, here are a few methods.
(1) The G color channel is a good proxy for gray scale information, since as shown in the sensor discussion in Chap. 1, the RB wavelengths in the spectrum overlap heavily into the G wavelengths.
(2) Simple conversion from RGB into gray scale intensity I can be done by taking $I = (R+G+B)/3$.

(3) The YIQ color space, used in the NTSC television broadcast standards, provides a simple forward/backward method of color conversion between RGB and a gray scale component Y, as follows:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.9663 & 0.6210 \\ 1 & -0.2721 & -0.6474 \\ 1 & -1.1070 & 1.7046 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.595716 & -0.274453 & -0.321263 \\ 0.211456 & -0.522591 & 0.311135 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Color Accuracy and Precision

If color accuracy is important, 8 bits per RGB color channel may not be enough. It is necessary to study the image sensor vendor's data sheets to understand how good the sensor really is. At the time of this writing, common image sensors are producing 10–14 bits of color information per RGB channel. Each color channel may have a different spectral response, as discussed in Chap. 1.

Typically, green is a good and fairly accurate color channel on most devices; red is usually good as well and may also have near infrared sensitivity if the IR filter is removed from the sensor; and blue is always a challenge since the blue wavelength can be hardest to capture in smaller silicon wells, which are close to the size of the blue wavelength, so the sensor vendor needs to pay special attention to blue sensing details.

Spatial Filtering

Filtering on discrete pixel arrays is considered *spatial filtering*, or time domain filtering, in contrast to filtering in the frequency domain using Fourier methods. Spatial filters are alternatives to frequency domain methods, and versatile processing methods are possible in the spatial domain.

Convolutional Filtering and Detection

Convolution is a fundamental signal processing operation easily computed as a discrete spatial processing operation, which is practical for 1D, 2D, and 3D processing. The basic idea is to combine, or convolve, two signals together, changing the source signal to be more like the filter signal. The source signal is the array of pixels in the image; the filter signal is a weighted kernel mask, such as a gradient peak shape and oriented edge shape or an otherwise weighted shape. For several examples of filter kernel mask shapes, see the section later in the chapter that discusses Sobel, Scharr, Prewitt, Roberts, Kirsch, Robinson, and Frei-Chen filter masks.

Convolution is typically used for filtering operations such as low-pass, band pass, and high-pass filters, but many filter shapes are possible to detect features, such as edge detection kernels tuned sensitive to edge orientation, or even point, corner, and contour detectors. Convolution is used as a detector in the method of convolution networks [77], as discussed in Chap. 4.

35	43	49		-1	-1	-1		35	43	49
38	52	47	*	-1	8	-1	=	38	67	47
42	44	51		-1	-1	-1		42	44	51

$$-(35 + 43 + 49 + 47 + 51 + 44 + 42 + 38) + (52 \cdot 8) = 67$$

Figure 2.10 Convolution, in this case a sharpen filter: (*Left to right*) Image data, sharpen filter, and resulting image data

The sharpen kernel mask in Fig. 2.10 (center image) is intended to amplify the center pixel in relation to the neighboring pixels. Each pixel is multiplied by its kernel position, and the result (right image) shows the center pixel as the sum of the convolution, which has been increased or amplified in relation to the neighboring pixels.

A convolution operation is typically followed up with a set of postprocessing point operations to clean up the data. Following are some useful postprocessing steps; many more are suggested in the “Point Filters” section that follows later in the chapter.

```

switch (post_processor)
{
case RESULT_ASIS:
    break;
case RESULT_PLUS_VALUE:
    sum += value;
    break;
case RESULT_MINUS_VALUE:
    sum -= value;
    break;
case RESULT_PLUS_ORIGINAL_TIMES_VALUE:
    sum = sum + (result * value);
    break;
case RESULT_MINUS_ORIGINAL_TIMES_VALUE:
    sum = sum - (result * value);
    break;
case ORIGINAL_PLUS_RESULT_TIMES_VALUE:
    sum = result + (sum * value);
    break;
case ORIGINAL_MINUS_RESULT_TIMES_VALUE:
    sum = result - (sum * value);
    break;
case ORIGINAL_LOW_CLIP:
    sum = (result < value ? value : result);
    break;
case ORIGINAL_HIGH_CLIP:
    sum = (result > value ? value : result);
    break;
}
switch (post_processing_sign)
{
```

```

case ABSOLUTE_VALUE:
    if (sum < 0) sum = -sum;
    if (sum > limit) sum = limit;
    break;
case POSITIVE_ONLY:
    if (sum < 0) sum = 0;
    if (sum > limit) sum = limit;
    break;
case NEGATIVE_ONLY:
    if (sum > 0) sum = 0;
    if (-sum > limit) sum = -limit;
    break;
case SIGNED:
    if (sum > limit) sum = limit;
    if (-sum > limit) sum = -limit;
    break;
}

```

Convolution is used to implement a variety of common filters including:

- **Gradient or sharpen filters**, which amplify and detect maxima and minima pixels. Examples include Laplacian.
- **Edge or line detectors**, where lines are connected gradients that reveal line segments or contours. Edge or line detectors can be steerable to a specific orientation, like vertical, diagonal, horizontal, or omnidirectional; steerable filters as basis sets are discussed in Chap. 3.
- **Smoothing and blur filters**, which take neighborhood pixels.

Kernel Filtering and Shape Selection

Besides convolutional methods, kernels can be devised to capture regions of pixels generically for statistical filtering operations, where the pixels in the region are sorted into a list from low to high value. For example, assuming a 3×3 kernel region, we can devise the following statistical filters:

```

sort(&kernel, &image, &coordinates, &sorted_list);
switch (filter_type)
{
case RANK_FILTER:
    // Pick highest pixel in the list, rank = 8 for a  $3 \times 3$  kernel 0..8
    // Could also pick the lowest, middle, or other rank
    image[center_pixel] = sorted_list[rank];
    break;
case MEDIAN_FILTER:
    // Median value is kernel size / 2,  $(3 \times 3 = 9)/2 = 4$  in this case
    image[center_pixel] = sorted_list[median];
    break;
case MAJORITY_FILTER:
    // Find the pixel value that occurs most often, count sorted pixel values
    count(&sorted_list, &counted_list);
    image[center_pixel] = counted_list[0];
    break;
}

```

The rank filter is a simple and powerful method that sorts each pixel in the region and substitutes a pixel of desired rank for the center pixel, such as substitution of the highest pixel in the region for the center pixel, or the median value or the majority value.

Shape Selection or Forming Kernels

Any regional operation can benefit from shape selection kernels to select pixels from the region and exclude others. Shape selection, or forming, can be applied as a preprocessing step to any image preprocessing algorithm or to any feature extraction method. Shape selection kernels can be binary truth kernels to select which pixels from the source image are used as a group, or to mark pixels that should receive individual processing. Shape selection kernels, as shown in Fig. 2.11, can be applied to local feature descriptors and detectors also; similar but sometimes more complex local region pixel selection methods are often used with local binary descriptor methods, as discussed in Chap. 4.

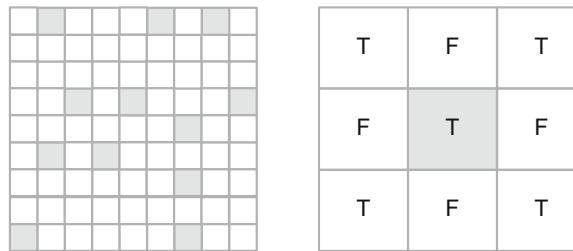


Figure 2.11 Truth and shape kernels: (*Left*) A shape kernel gray kernel position indicating a pixel to process or use—for example, a pixel to convolve prior to a local binary pattern point-pair comparison detector. (*Right*) A truth shape kernel specifying pixels to use for region average, favoring diagonals—T means use this pixel, F means do not use

Point Filtering

Individual pixel processing is typically overlooked when experimenting with image preprocessing. Point processing is amenable to many optimization methods, as will be discussed in Chap. 8. Convolution, as discussed above, is typically followed by point postprocessing steps. Table 2.2 illustrates several common pixel point processing methods in the areas of math operations, Boolean operations, and compare and substitution operations, which seem obvious but can be quite valuable for exploring image enhancement methods to enhance feature extraction.

Table 2.2 Possible Point Operations

// Math ops	// Compare & Substitution ops
<pre> NAMES math_ops[] = { "src + value -> dst", "src - value -> dst", "src * value -> dst", "src / value -> dst", "(src + dst) * value -> dst", "(src - dst) * value -> dst", "(src * dst) * value -> dst", "(src / dst) * value -> dst", "sqrt(src) + value -> dst", "src * src + value -> dst", "exp(src) + value -> dst", "log(src) + value -> dst", "log10(src) + value -> dst", "pow(src ^ value) -> dst", "sin(src) + value -> dst", "cos(src) + value -> dst", "tan(src) + value -> dst", "(value / max(all_src)) * src -> dst", "src - mean(all_src) -> dst", "absval(src) + value -> dst", }; // Boolean ops NAMES bool_ops[] = { "src AND value -> dst", "src OR value -> dst", "src XOR value -> dst", "src AND dst -> dst", "src OR dst -> dst", "src XOR dst -> dst", "NOT(src) -> dst", "LO_CLIP(src, value) -> dst", "LO_CLIP(src, dst) -> dst", "HI_CLIP(src, value) -> dst", "HI_CLIP(src, dst) -> dst", }; </pre>	<pre> NAMES change_ops[] = { "if (src = thresh) value -> dst", "if (src = dst) value -> dst", "if (src != thresh) value -> dst", "if (src != dst) src -> dst", "if (src != dst) value -> dst", "if (src != dst) src -> dst", "if (src >= thresh) value -> dst", "if (src >= thresh) src -> dst", "if (src >= dst) value -> dst", "if (src >= dst) src -> dst", "if (src <= thresh) value -> dst", "if (src <= thresh) src -> dst", "if (src <= dst) value -> dst", "if (src <= dst) src -> dst", "if (lo <= src <= hi) value -> dst", "if (lo <= src <= hi) src -> dst", }; </pre>

Noise and Artifact Filtering

Noise is usually an artifact of the image sensor, but not always. There are several additional artifacts that may be present in an image as well. The goal of noise removal is to remove the noise without distorting the underlying image, and the goal of removing artifacts is similar. Depending on the type of noise or artifact, different methods may be employed for preprocessing. The first step is to classify the noise or artifact, and then to devise the right image preprocessing strategy.

- **Speckle, random noise.** This type of noise is apparently random, and can be removed using a rank filter or median filter.
- **Transient frequency spike.** This can be determined using a Fourier spectrum and can be removed using a notch filter over the spike; the frequency spike will likely be in an outlier region of the spectrum, and may manifest as a bright spot in the image.

- **Jitter and judder line noise.** This is an artifact particular to video streams, usually due to telecine artifacts, motion of the camera or the image scene, and is complex to correct. It is primarily line oriented rather than just single-pixel oriented.
- **Motion blur.** This can be caused by uniform or nonuniform motion and is a complex problem; several methods exist for removal; see Ref. [297].

Standard approaches to noise removal are discussed by Gonzalez [4]. The most basic approach is to remove outliers, and various approaches are taken, including thresholding and local region based statistical filters such as the rank filter and median filter. Weighted image averaging is also sometime used for removing noise from video streams; assuming the camera and subjects are not moving, it can work well. Although deblurring or Gaussian smoothing convolution kernels are sometimes used to remove noise, such methods may cause smearing and may not be the best approach.

A survey of noise-removal methods and a performance comparison model are provided by Buades et al. [493]. This source includes a description of the author's NL-means method, which uses nonlocal pixel value statistics in addition to Euclidean distance metrics between similar weighted pixel values over larger image regions to identify and remove noise.

Integral Images and Box Filters

Integral images are used to quickly find the average value of a rectangular group of pixels. An integral image is also known as a *summed area table*, where each pixel in the integral image is the integral sum of all pixels to the left and above the current pixel. The integral image can be calculated quickly in a single pass over the image. Each value in the summed area table is calculated using the current pixel value from the image $i(n,m)$ combined with previous entries $s(n,m)$ made into the summed area table, as follows:

$$s(x,y) = i(x,y) + s(x-1,y) + s(x,y-1) - s(x-1,y-1)$$

As shown in Fig. 2.12, to find a HAAR rectangle feature value from the integral image, only four points in the integral image table A, B, C, D are used, rather than tens or hundreds of points from the image. The integral image sum of a rectangle region can then be divided by the size of the rectangle region to yield the average value, which is also known as a *box filter*.

05	02	05	02
03	06	03	06
05	02	05	02
03	06	03	06

05	07	12	14
08	16	24	32
13	23	36	46
16	32	48	64

05	07	12	14
08	16	24	32
13	23	36	46
16	32	48	64

Figure 2.12 (Left) Pixels in an image. (Center) Integral image. (Right) Region where a box filter value is computed from four points in the integral image: $\text{sum} = s(A) + s(D) - s(B) - s(C)$

Integral images and box filters are used in many computer vision methods, such as HAAR filters and feature descriptors. Integral images are also used as a fast alternative to a Gaussian filter of a small region, as a way to lower compute costs. In fact, descriptors with a lot of overlapping region processing, such as BRISK [123], make effective use of integral images for descriptor building and use integral images as a proxy for a fast Gaussian blur or convolution.

Edge Detectors

The goal of an edge detector is to enhance the connected gradients in an image, which may take the form of an edge, contour, line, or some connected set of edges. Many edge detectors are simply implemented as kernel operations, or convolutions, and we survey the common methods here.

Kernel Sets: Sobel, Scharr, Prewitt, Roberts, Kirsch, Robinson, and Frei–Chen

The Sobel operator detects gradient magnitude and direction for edge detection. The basic method is shown here.

1. Perform two directional Sobel filters (x and y axis) using basic derivative kernel approximations such as 3×3 kernels, using values as follows:

$$S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

2. Calculate the total gradient as $G_v = |S_x| + |S_y|$
3. Calculate the gradient direction as $\theta = \text{ATAN}(S_y / S_x)$
4. Calculate gradient magnitude $G_m = \sqrt{S_y^2 + S_x^2}$

Variations exist in the area size and shape of the kernels used for Sobel edge detection. In addition to the Sobel kernels shown above, other similar kernel sets are used in practice, so long as the kernel values cancel and add up to zero, such as those kernels proposed by Scharr, Prewitt, Roberts, Robinson, and Frei–Chen, as well as Laplacian approximation kernels. The Frei–Chen kernels are designed to be used together at a set, so the edge is the weighted sum of all the kernels. See Ref. [4] for more information on edge detection masks. Some kernels have compass orientations, such as those developed by Kirsch, Robinson, and others. See Fig. 2.13.

$$\begin{aligned}
& \begin{pmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{pmatrix} \begin{pmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{pmatrix} \text{Scharx} \\
& \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \text{Roberts} \\
& \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} \text{Prewitt} \\
& \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} .5 & 1 & .5 \\ 1 & -6 & 1 \\ .5 & 1 & .5 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} -2 & 1 & -2 \\ 1 & 4 & 1 \\ -2 & 1 & -2 \end{pmatrix} \text{Laplacians} \\
& \begin{pmatrix} 5 & 5 & 5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{pmatrix} \begin{pmatrix} 5 & 5 & -3 \\ 5 & 0 & -3 \\ -3 & -3 & -3 \end{pmatrix} \begin{pmatrix} 5 & -3 & 5 \\ 5 & 0 & -3 \\ 5 & -3 & -3 \end{pmatrix} \begin{pmatrix} -3 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & 5 & -3 \end{pmatrix} \text{Kirsch Compass} \\
& \begin{pmatrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ 5 & 5 & 5 \end{pmatrix} \begin{pmatrix} -3 & -3 & -3 \\ -3 & 0 & 5 \\ -3 & 5 & 5 \end{pmatrix} \begin{pmatrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{pmatrix} \begin{pmatrix} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{pmatrix} \text{Kirsch Compass} \\
& \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{pmatrix} \text{Robinson Compass} \\
& \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -2 \end{pmatrix} \begin{pmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{pmatrix} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix} \text{Robinson Compass} \\
& \frac{1}{2\sqrt{2}} \begin{pmatrix} 1 & \sqrt{2} & 1 \\ 0 & 0 & 0 \\ -1 & -\sqrt{2} & -1 \end{pmatrix}, \quad \frac{1}{2\sqrt{2}} \begin{pmatrix} -1 & 0 & 1 \\ -\sqrt{2} & 0 & \sqrt{2} \\ -1 & 0 & 1 \end{pmatrix}, \quad \frac{1}{2\sqrt{2}} \begin{pmatrix} 0 & -1 & \sqrt{2} \\ 1 & 0 & -1 \\ -\sqrt{2} & 1 & 0 \end{pmatrix} \text{Fre - Chen} \\
& \frac{1}{2\sqrt{2}} \begin{pmatrix} \sqrt{2} & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & -\sqrt{2} \end{pmatrix}, \quad \frac{1}{2} \begin{pmatrix} 0 & -1 & 0 \\ -1 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix}, \quad \frac{1}{2} \begin{pmatrix} -1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{pmatrix} \text{Frei - Chen} \\
& \frac{1}{6} \begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix}, \quad \frac{1}{6} \begin{pmatrix} -2 & 1 & -2 \\ 1 & 4 & 1 \\ -2 & 1 & -2 \end{pmatrix}, \quad \frac{1}{3} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \text{Frei - Chen}
\end{aligned}$$

Figure 2.13 Several edge detection kernel masks

Canny Detector

The Canny method [146] is similar to the Sobel-style gradient magnitude and direction method, but it adds postprocessing to clean up the edges.

1. Perform a Gaussian blur over the image using a selected convolution kernel (7×7 , 5×5 , etc.), depending on the level of low-pass filtering desired.
2. Perform two directional Sobel filters (x and y axis) and find the edge strength as $|G| = |G_x| + |G_y|$ and edge direction as $\theta = \text{ATAN}(G_y/G_x)$ and round the direction to one of the four directions 0, 90, 180, or 270.
3. Perform non-maximal value suppression in the direction of the gradient to set to zero (0) pixels not on an edge (minima values).
4. Perform hysteresis thresholding within a band (high, low) of values along the gradient direction to eliminate edge aliasing and outlier artifacts and to create better connected edges.

Transform Filtering, Fourier, and Others

This section deals with basis spaces and image transforms in the context of image filtering, the most common and widely used being the Fourier transform. A more comprehensive treatment of basis spaces and transforms in the context of feature description is provided in Chap. 3. A good reference for transform filtering in the context of image processing is provided by Pratt [9].

Why use transforms to switch domains? To make image preprocessing easier or more effective, or to perform feature description and matching more efficiently. In some cases, there is no better way to enhance an image or describe a feature than by transforming it to another domain—for example, for removing noise and other structural artifacts as outlier frequency components of a Fourier spectrum, or to compactly describe and encode image features using HAAR basis features.

Fourier Transform Family

The Fourier transform is very well known and covered in the standard reference by Bracewell [219], and it forms the basis for a family of related transforms. Several methods for performing fast Fourier transform (FFT) are common in image and signal processing libraries. Fourier analysis has touched nearly every area of world affairs, through science, finance, medicine, and industry, and has been hailed as “the most important numerical algorithm of our lifetime” [282]. Here, we discuss the fundamentals of Fourier analysis, and a few branches of the Fourier transform family with image preprocessing applications.

The Fourier transform can be computed using optics, at the speed of light [498]. However, we are interested in methods applicable to digital computers.

Fundamentals

The basic idea of Fourier analysis [4, 9, 219] is concerned with decomposing periodic functions into a series of sine and cosine waves (Fig. 2.14). The Fourier transform is bidirectional, between a periodic wave and a corresponding series of harmonic basis functions in the frequency domain, where each basis function is a sine or cosine function, spaced at whole harmonic multiples from the base frequency. The result of the forward FFT is a complex number composed of magnitude and phase data for each sine and cosine component in the series, also referred to as *real data* and *imaginary data*.

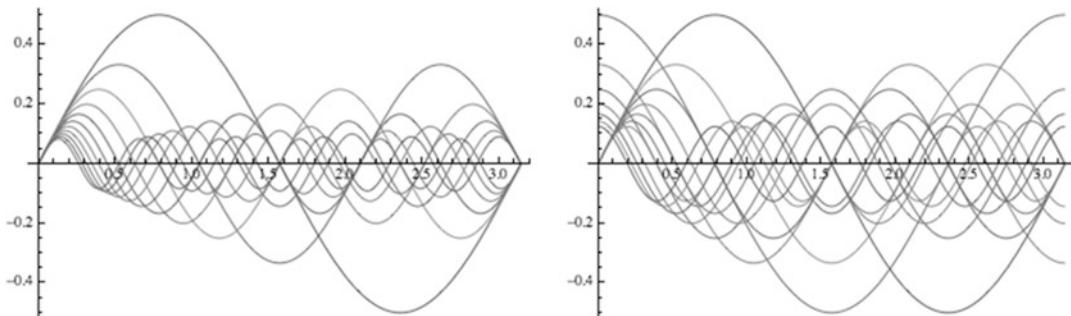


Figure 2.14 (Left) Harmonic series of sine waves. (Right) Fourier harmonic series of sine and cosine waves

Arbitrary periodic functions can be synthesized by summing the desired set of Fourier basis functions, and periodic functions can be decomposed using the Fourier transform into the basic functions as a Fourier series. The Fourier transform is invertible between the time domain of discrete pixels and the frequency domain, where both magnitude and phase of each basis function are available for filtering and analysis, magnitude being the most commonly used component.

How is the FFT implemented for 2D images or 3D volumes? The Fourier transform is a *separable transform* and so can be implemented as a set of parallel 1D FFT line transforms. So, for 2D images and 3D volumes, each dimension, such as the x , y , z dimension, can be computed in place, in parallel as independent x lines, then the next dimension or y columns can be computed in place as parallel lines, then the z dimension can be computed as parallel lines in place, and the final results are scaled according to the transform. Any good 1D FFT algorithm can be set up to process 2D images or 3D volumes using parallelization.

$$2\left(\theta\left(\frac{x}{L}\right) - \theta\left(\frac{x}{L} - 1\right)\right) - 1$$

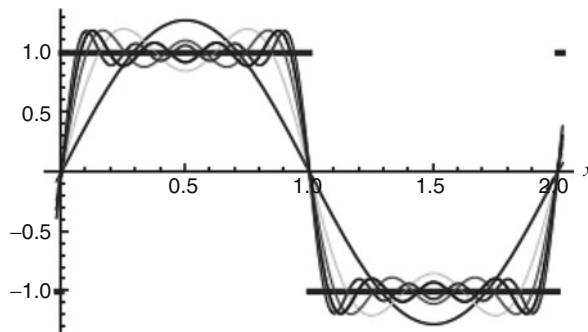


Figure 2.15 Fourier series and Fourier transform concepts showing a square wave approximated from a series of Fourier harmonics

For accuracy of the inverse transform to go from frequency space back to pixels, the FFT computations will require two double precision 64-bit floating point buffers to hold the magnitude and phase data, since transcendental functions such as sine and cosine require high floating point precision for accuracy; using 64-bit double precision floating point numbers for the image data allows a forward transform of an image to be computed, followed by an inverse transform, with no loss of precision compared to the original image—of course, very large images will need more than double precision.

Since 64-bit floating point is typically slower and of higher power, owing to the increased compute requirements and silicon real estate in the ALU, as well as the heavier memory bandwidth load, methods for FFT optimization have been developed using integer transforms, and in some cases fixed point, and these are good choices for many applications.

Note in Fig. 2.16 that the low-pass filter (center right) is applied to preserve primarily low-frequency information toward the center of the plot and it reduces high-frequency components toward the edges, resulting in the filtered image at the far right.

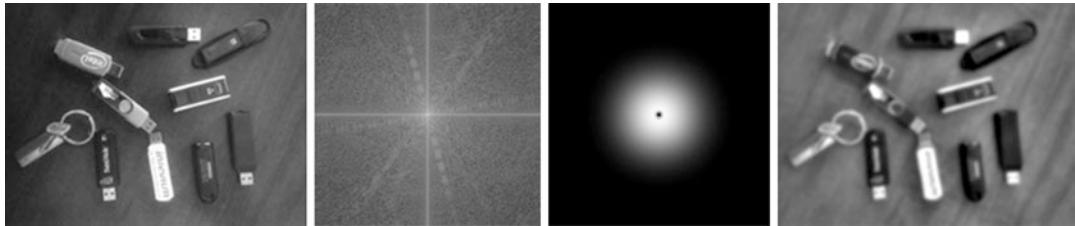


Figure 2.16 Basic Fourier filtering: (*Left*) Original. (*Center left*) Fourier spectrum. (*Center right*) Low-pass filter shape used to multiply against Fourier magnitude. (*Right*) Inverse transformed image with low-pass filter

A key Fourier application is filtering, where the original image is forward-transformed into magnitude and phase; the magnitude component is shown as a Fourier power spectrum of the magnitude data, which reveals structure in the image as straight lines and blocks, or outlier structures or spots that are typically noise. The magnitude can be filtered by various filter shapes, such as high-pass, low-pass, band pass, and spot filters to remove spot noise, to affect any part of the spectrum.

In Fig. 2.16, a circular symmetric low-pass filter shape is shown with a smooth distribution of filter coefficients from 1 to 0, with high multiplicands in the center at the low frequencies, ramping down to zero toward the high frequencies at the edge. The filter shape is multiplied in the frequency domain against the magnitude data to filter out the higher frequency components, which are toward the outside of the spectrum plot, followed by an inverse FFT to provide the filtered image. The low-frequency components are toward the center; typically these are most interesting and so most of the image power is contained in the low-frequency components. Any other filter shape can be used, such as a spot filter, to remove noise or any of the structure at a specific location of the spectrum.

Fourier Family of Transforms

The Fourier transform is the basis for a family of transforms [4], some of which are:

1. **DFT, FFT.** The discrete version of the Fourier transform, often implemented as a fast version, or FFT, commonly used for image processing. There are many methods of implementing the FFT [219].
2. **Sine transform.** Fourier formulation composed of only sine terms.
3. **Cosine transform.** Fourier formulation composed of only cosine terms.
4. **DCT, DST, MDCT.** The discrete Fourier transform is implemented in several formulations: discrete sine transform (DST), discrete cosine transform (DCT), and the modified discrete cosine transform (MDCT). These related methods operate on a macroblock, such as 16×16 or 8×8 pixel region, and can therefore be highly optimized for compute use with integers rather than floating point. Typically the DCT is implemented in hardware for video encode and decode applications for motion estimation of the macro blocks from frame to frame. The MDCT operates on overlapping macroblock regions for compute efficiency.
5. **Fast Hartley transform, DHT.** This was developed as an alternative formulation of the Fourier transform for telephone transmission analysis about 1925, forgotten for many years, then rediscovered and promoted again by Bracewell [219] as an alternative to the Fourier transform. The Hartley transform is a symmetrical formulation of the Fourier transform, decomposing a signal into two sets of sinusoidal functions taken together as a *cosine-and-sine* or *cas()* function, where $\text{cas}(vx) \equiv \cos(vx) + \sin(vx)$. This includes positive and negative frequency components and operates entirely on real numbers for input and output. The Hartley formulation avoids complex numbers as used in the Fourier complex exponential $\exp(j \omega x)$. The Hartley transform has been

developed into optimized versions called the DHT, shown to be about equal in speed to an optimized FFT.

Other Transforms

Several other transforms may be used for image filtering, including wavelets, steerable filter banks, and others that will be described in Chap. 3, in the context of feature description. Note that transforms often have many common uses and applications that overlap, such as image description, image coding, image compression, and feature description.

Morphology and Segmentation

For simplicity, we define the goal of morphology as shape and boundary definition, and the goal of segmentation is to define regions with internal similarity, such as textural or statistical similarity. *Morphology* is used to identify features as polygon shaped regions that can be described with shape metrics, as will be discussed in Chaps. 3 and 6, distinct from local interest point and feature descriptors using other methods. An image is *segmented into regions* to allow independent processing and analysis of each region according to some policy or processing goal. Regions cover an area smaller than the global image but usually larger than local interest point features, so an application might make use of global, regional, and small local interest point metrics together as an *object signature*.

An excellent review of several segmentation methods can be found in work by Haralick and Shapiro [313]. In practice, segmentation and morphology are not easy: results are often less useful than expected, trial and error is required, too many methods are available to provide any strict guidance, and each image is different. So here we only survey the various methods to introduce the topic and illustrate the complexity. An overview of region segmentation methods is shown in Table 2.3.

Table 2.3 Segmentation Methods

Method	Description
Morphological segmentation	The region is defined based on thresholding and morphology operators.
Texture-based segmentation	The texture of a region is used to group like textures into connected regions.
Transform-based segmentation	Basis space features are used to segment the image.
Edge boundary segmentation	Gradients or edges alone are used to define the boundaries of the region with edge linking in some cases to form boundaries.
Color segmentation	Color information is used to define regions.
Super-Pixel Segmentation	Kernels and distance transforms are used to group pixels and change their values to a common value.
Gray scale / luminance segmentation	Grayscale thresholds or bands are used to define the regions.
Depth segmentation	Depth maps and distance from viewer is used to segment the image into foreground, background, or other gradations of inter-scene features.

Binary Morphology

Binary morphology operates on binary images, which are created from other scalar intensity channel images. Morphology [9] is used to *morph* a feature shape into a new shape for analysis by removing shape noise or outliers, and by strengthening predominant feature characteristics. For example, isolated pixels may be removed using morphology, thin features can be fattened, and the predominant shape is still preserved. Note that morphology all by itself is quite a large field of study, with applications to general object recognition, cell biology, medicine, particle analysis, and automated microscopy. We introduce the fundamental concepts of morphology here for binary images, and then follow this section with applications to gray scale and color data.

Binary morphology starts with binarizing images, so typically thresholding is first done to create images with binary-valued pixels composed of 8-bit black and white values, 0-value = black and 255-value = white. Thresholding methods are surveyed later in this chapter, and thresholding is critical prior to morphology.

Binary morphology is a neighborhood operation, and can use a forming kernel with truth values, as shown in Fig. 2.17. The forming kernel guides the morphology process by defining which surrounding pixels contribute to the morphology. Fig. 2.17 shows two forming kernels: kernel a, where all pixels touching the current pixel are considered, and kernel b, where only orthogonally adjacent pixels are considered.

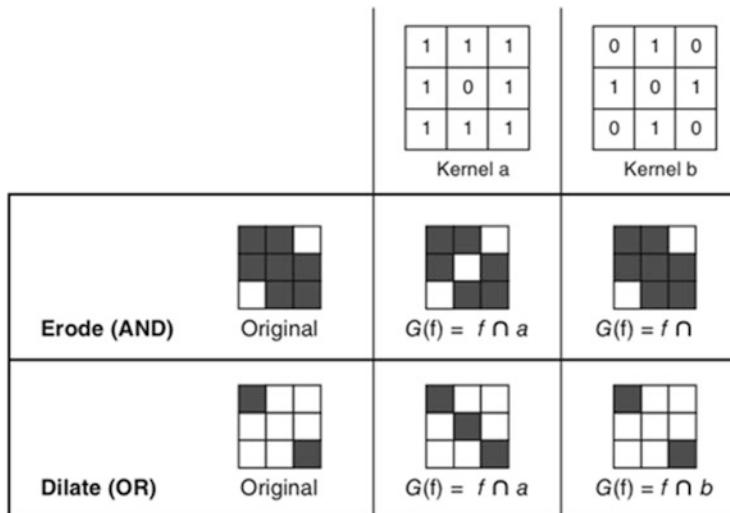


Figure 2.17 3×3 forming kernels and binary erosion and dilation using the kernels; other kernel sizes and data values may be useful in a given application. (Image used by permission, © Intel Press, from Building Intelligent Systems)

The basic operations of morphology include Boolean AND, OR, NOT. The notation used for the fundamental morphological operations is \cup for *dilation* and \cap for *erosion*. In binary morphology, dilation is a Boolean OR operator, while erosion is a Boolean AND operator. In the example provided in Fig. 2.17, only kernel elements with a “1” are used in the morphology calculation, allowing for neighborhood contribution variations. For erosion, the pixels under all true forming kernel elements are AND’d together; the result is 1 if all are true and the pixel feature remains, otherwise the pixel feature is eroded or set to 0.

All pixels under the forming true kernel must be true for erosion of the center pixel. Erosion attempts to reduce sparse features until only strong features are left. Dilatation attempts to inflate sparse features to make them fatter, only 1 pixel under the forming kernel elements must be true for dilation of the center pixel, corresponding to Boolean OR.

Based on simple erosion and dilation, a range of morphological operations are derived as shown here, where \oplus = dilation and \ominus = erosion.

Erode	$G(f) = f \ominus b$
Dilate	$G(f) = f \oplus b$
Opening	$G(f) = (f \oplus b) \ominus b$
Closing	$G(f) = (f \ominus b) \oplus b$
Morphological gradient	$G(f) = f \ominus b$ or $G(f) = f \oplus b - f \ominus b$
Morphological internal gradient	$G_i(f) = f - f \ominus b$
Morphological external gradient	$G_e(f) = f \oplus b - f$

Gray Scale and Color Morphology

Gray scale morphology is useful to synthesize and combine pixels into homogeneous intensity bands or regions with similar intensity values. Gray scale morphology can be used on individual color components to provide color morphology affecting hue, saturation, and color intensity in various color spaces.

For gray scale morphology or color morphology, the basic operations are MIN, MAX, and MINMAX, where pixels above the MIN are changed to the same value and pixels below the MAX are changed to the same value, while pixels within the MINMAX range are changed to the same value. MIN and MAX are a form of thresholding, while MINMAX allows bands of pixel values to be coalesced into equal values forming a homogenous region.

Morphology Optimizations and Refinements

Besides simple morphology [9], there are other methods of morphological segmentation using adaptive methods [246–248]. Also, the MorphoLibJ package (also a plugin for imageJ Fiji) contains one of the most comprehensive and high quality suites of morphological methods including segmentation, filtering, and labeling. The simple morphology methods rely on using a fixed kernel across the entire image at each pixel and assume the threshold is already applied to the image; while the adaptive methods combine the morphology operations with variable kernels and variable thresholds based on the local pixel intensity statistics. This allows the morphology to adapt to the local region intensity and, in some cases, produce better results. Auto-thresholding and adaptive thresholding methods are discussed later in this chapter and are illustrated in Figs. 2.24 and 2.26.

Euclidean Distance Maps

The distance map, or Euclidean distance map (EDM), converts each pixel in a binary image into the distance from each pixel to the nearest background pixel, so the EDM requires a binary image for input. The EDM is useful for segmentation, as shown in Fig. 2.18, where the EDM image is thresholded based on the EDM values—in this case, similar to the ERODE operator.



Figure 2.18 Preprocessing sequence: (*Left*) Image after thresholding and erosion. (*Center*) EDM showing gray levels corresponding to distance of pixel to black background. (*Right*) Simple binary thresholded EDM image

Super-pixel Segmentation

A super-pixel segmentation method [249–253] attempts to collapse similar pixels in a local region into a larger super-pixel region of equal pixel value, so similar values are subsumed into the larger super-pixel. Super-pixel methods are commonly used for digital photography applications to create a scaled or watercolor special effect. Super-pixel methods treat each pixel as a node in a graph, and edges between regions are determined based on the similarity of neighboring pixels and graph distance. See Fig. 2.19.



Figure 2.19 Comparison of various super-pixel segmentation methods (Image © Dr. Radhakrishna Achanta, used by permission)

Feature descriptors may be devised based on super-pixels, including super-pixel value histograms, shape factors of each polygon shaped super-pixel, and spatial relationships of neighboring super-pixel values. Apparently little work has been done on super-pixel based descriptors; however, the potential for several degrees of robustness and invariance seems good. We survey a range of super-pixel segmentation methods next.

Graph-Based Super-pixel Methods

Graph-based methods structure pixels into trees based on the distance of the pixel from a centroid feature or edge feature for a region of like-valued pixels. The compute complexity varies depending on the method.

- **SLIC Method** [250] Simple Linear Iterative Clustering (SLIC) creates super-pixels based on a 5D space, including the CIE Lab color primaries and the XY pixel coordinates. The SLIC algorithm takes as input the desired number of super-pixels to generate and adapt well to both gray scale and RGB color images. The clustering distance function is related to the size of the desired number of super-pixels and uses a Euclidean distance function for grouping pixels into super-pixels.
- **Normalized Cuts** [254, 255] Uses a recursive region partitioning method based on local texture and region contours to create super-pixel regions.
- **GS-FH Method** [256] The graph-based Felzenszwalb and Huttenlocher method attempts to segment image regions using edges based on perceptual or psychological cues. This method uses the minimum length between pixels in the graph tree structure to create the super-pixel regions. The computational complexity is $O(n \log n)$, which is relatively fast.
- **SL Method** [257] The Super-pixel Lattice (SL) method finds region boundaries within tiled image regions or strips of pixels using the graph cut method.

Gradient-Ascent-Based Super-pixel Methods

Gradient ascent methods iteratively refine the super-pixel clusters to optimize the segmentation until convergence criteria are reached. These methods use a tree graph structure to associate pixels together according to some criteria, which in this case may be the RGB values or Cartesian coordinates of the pixels, and then a distance function or other function is applied to create regions. Since these are iterative methods, the performance can be slow.

- **Mean-Shift** [258] Works by registering off of the region centroid based on a kernel-based mean smoothing approach to create regions of like pixels.
- **Quick-Shift** [259] Similar to the mean-shift method but does not use a mean blur kernel and instead uses a distance function calculated from the graph structure based on RGB values and XY pixel coordinates.
- **Watershed** [260] Starts from local region pixel value minima points to find pixel value-based contour lines defining watersheds, or basin contours inside which similar pixel values can be substituted to create a homogeneous pixel value region.
- **Turbopixels** [261] Uses small circular seed points placed in a uniform grid across the image around which super-pixels are collected into assigned regions, and then the super-pixel boundaries are gradually expanded into the unassigned region, using a geometric flow method to expand the boundaries using controlled boundary value expansion criteria, so as to gather more pixels together into regions with fairly smooth and uniform geometric shape and size.

Depth Segmentation

Depth information, such as a depth map as shown in Fig. 2.20, is ideal for segmenting objects based on distance. Depth maps can be computed from a wide variety of depth sensors and methods, including a single camera, as discussed in Chap. 1. Depth cameras, such as the Microsoft Kinect camera, are becoming more common. A depth map is a 2D image or array, where each pixel value is the distance or Z value.

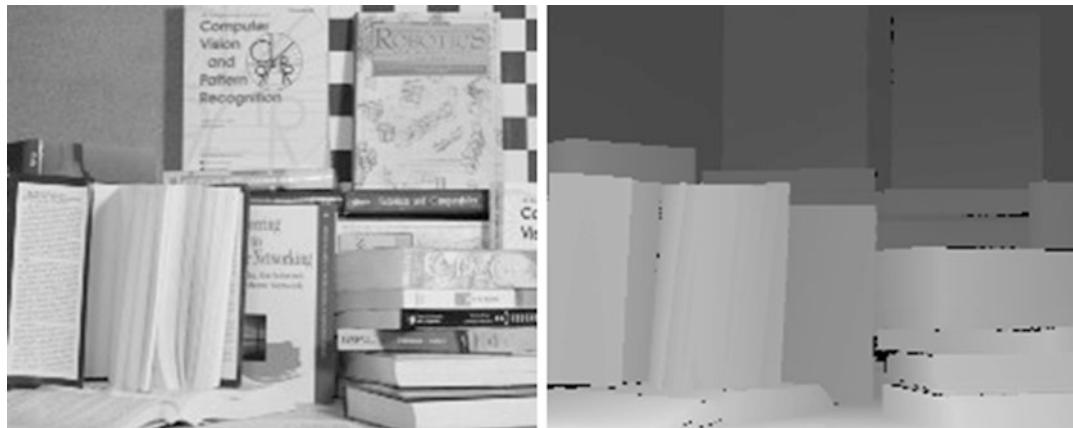


Figure 2.20 Depth images from Middlebury Data set: (*Left*) Original image. (*Right*) Corresponding depth image. Data courtesy of Daniel Scharstein and used by permission

Many uncertainties in computer vision arise out of the problems in locating three-dimensional objects in a two-dimensional image array, so adding a depth map to the vision pipeline is a great asset. Using depth maps, images can be easily segmented into the foreground and background, as well as be able to segment specific features or objects—for example, segmenting by simple depth thresholding.

Depth maps are often very fuzzy and noisy, depending on the depth sensing method, so image preprocessing may be required. However, there is no perfect filtering method for depth map cleanup. Many practitioners prefer the bilateral filter [294] and variants, since it preserves local structure and does a better job of handling the edge transitions.

Color Segmentation

Sometime color alone can be used to segment and threshold. Using the right color component can easily filter out features from an image. For example, in Fig. 2.5, we started from a red channel image from an RGB set, and the goal was to segment out the USB sticks from the table background. Since the table is brown and contains a lot of red, the red channel provides useful contrast with the USB sticks allowing segmentation via red. It may be necessary to color-correct the image to get the best results, such as gamut corrections or boosting the hue or saturation of each color to accentuate difference.

Thresholding

The goal of thresholding is to segment the image at certain intensity levels to reveal features such as foreground, background, and specific objects. A variety of methods exist for thresholding, ranging from global to locally adaptive. In practice, thresholding is very difficult and often not satisfactory by itself, and must be tuned for the dataset and combined with other preprocessing methods in the vision pipeline.

One of the key problems in thresholding is nonuniform illumination, so applications that require thresholding, like cell biology and microscopy, pay special attention to cell preparation, specimen

spacing, and light placement. Since many images do not respond well to global thresholding involving simple methods, local methods are often required, which use the local pixel structure and statistical relationships to create effective thresholds. Both global and local adaptive methods for thresholding are discussed here. A threshold can take several forms:

- **Floor** Lowest pixel intensity allowed
- **Ceiling** Highest pixel intensity allowed
- **Ramp** Shape of the pixel ramp between floor and ceiling, such as linear or log
- **Point** May be a binary threshold point with no floor, ceiling, or ramp

Global Thresholding

Thresholding entire images at a globally determined thresholding level is sometimes a good place to start to explore the image data, but typically local features will suffer and be unintelligible as a result. Thresholding can be improved using statistical methods to determine the best threshold levels. Lookup tables (LUT) can be constructed, guided by statistical moments to create the floor, ceiling, and ramps and the functions to perform rapid LUT processing on images, or false-color the images for visualization.

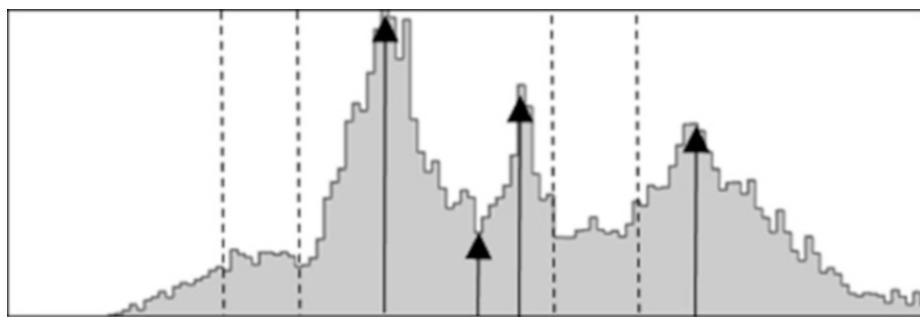


Figure 2.21 Histogram annotated with *arrows* showing peaks and valleys, and *dotted lines* showing regions of similar intensities defined using hysteresis thresholds

Histogram Peaks and Valleys, and Hysteresis Thresholds

Again we turn to the old stand-by, the image histogram. Peaks and valleys in the histogram may indicate thresholds useful for segmentation and thresholding [311]. A hysteresis region marks pixels with similar values, and is easy to spot in the histogram, as shown in Fig. 2.21. Also, many image processing programs have interactive sliders to allow the threshold point and even regions to be set with the pointer device.¹ Take some time and get to know the image data via the histogram and become familiar with using interactive thresholding methods.

¹ See the open-source package ImageJ2, and menu item Image → Adjust-Brightness/Contrast for interactive thresholding.

If there are no clear valleys between the histogram peaks, then establishing two thresholds, one on each side of the valley, is a way to define a region of hysteresis. Pixel values within the hysteresis region are considered inside the object. Further, the pixels can be classified together as a region using the hysteresis range and morphology to ensure region connectivity.

LUT Transforms, Contrast Remapping

Simple lookup tables (LUTs) are very effective for contrast remapping and global thresholding, and interactive tools can be used to create the LUTs. Once the interactive experimentation has been used to find the best floor, ceiling, and ramp function, the LUTs can be generated into table data structures and used to set the thresholds in fast code. False-coloring the image using pseudo-color LUTs is common and quite valuable for understanding the thresholds in the data. Various LUT shapes and ramps can be devised. See Fig. 2.22 for an example using a linear ramp function.

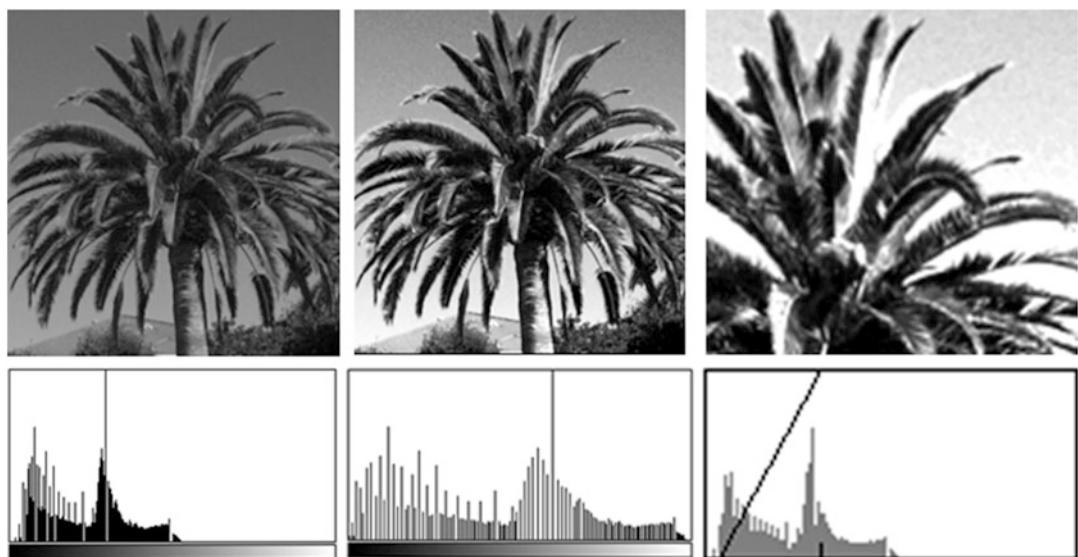


Figure 2.22 Contrast corrections: (*Left*) Original image shows palm frond detail compressed into a narrow intensity range obscuring details. (*Center*) Global histogram equalization restores some detail. (*Right*) LUT remap function spreads the intensity values to a narrower range to reveal details of the palm fronds. The section of the histogram under the diagonal line is stretched to cover the full intensity range in the right image; other intensity regions are clipped. The contrast corrected image will yield more gradient information when processed with a gradient operator such as Sobel

Histogram Equalization and Specification

Histogram equalization spreads pixel values between a floor and ceiling using a contrast remapping function, with the goal of creating a histogram with approximately equal bin counts approaching a straight-line distribution. See Fig. 2.23. While this method works well for gray scale images, color images should be equalized in the intensity channel of a chosen color space, such as HSV V. Equalizing each RGB component separately and re-rendering will produce color moiré artifacts. Histogram equalization uses a fixed region and a fixed remapping for all pixels in the region; however, adaptive local histogram equalization methods are available [306].

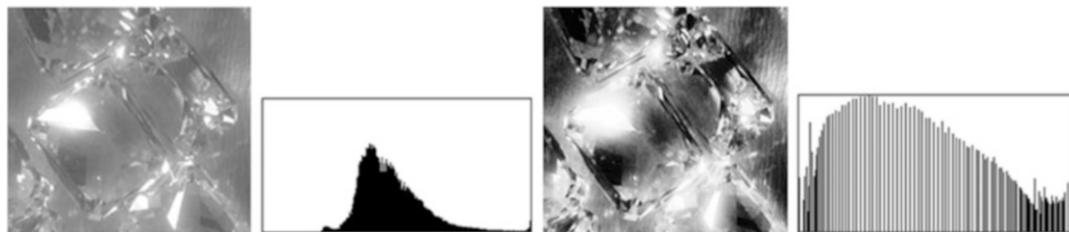


Figure 2.23 (Left) Original image and histogram. (Right) Histogram equalized image and histogram

It is possible to create a desired histogram shape or value distribution, referred to as *histogram specification*, and then remap all pixel values from the source image to conform to the specified histogram shape. The shape may be created directly, or else the histogram shape from a second image may be used to remap the source image to match the second image. With some image processing packages, the histogram specification may be interactive, and points on a curve may be placed and adjusted to create the desired histogram shape.

Global Auto Thresholding

Various methods have been devised to automatically find global thresholds based on statistical properties of the image histogram [312, 495–497] and in most cases the results are not very good unless some image preprocessing precedes the auto thresholding. Table 2.4 provides a brief survey of auto thresholding methods, while Fig. 2.24 displays renderings of each method.

Table 2.4 Selected Few Global Auto-Thresholding Methods Derived from Basic Histogram Features [295]

Method	Description
Default	A variation of the IsoData method, also known as iterative intermeans.
Huang	Huang's method of using fuzzy thresholding.
Intermodes	Iterative histogram smoothing.
IsoData	Iterative pixel averaging of values above and below a threshold to derive a new threshold above the composite average.
Li	Iterative cross-entropy thresholding.
MaxEntropy	Kapur-Sahoo-Wong (Maximum Entropy) algorithm.
Mean	Uses mean gray level as the threshold.
MinError	Iterative method from Kittler and Illingworth to converge on a minimum error threshold.
Minimum	Iterative histogram smoothing, assuming a bimodal histogram.
Moments	Tsai's thresholding algorithm intending to threshold and preserve the original image moments.
Otsu	Otsu clustering algorithms to set local thresholds by minimizing variance.
Percentile	Adapts the threshold based on pre-set allocations for foreground and background pixels.
RenyiEntropy	Another entropy-based method.
Shanbhag	Uses fuzzy set metrics to set the threshold.
Triangle	Uses image histogram peak, assumes peak is not centered, sets threshold in largest region on either side of peak.

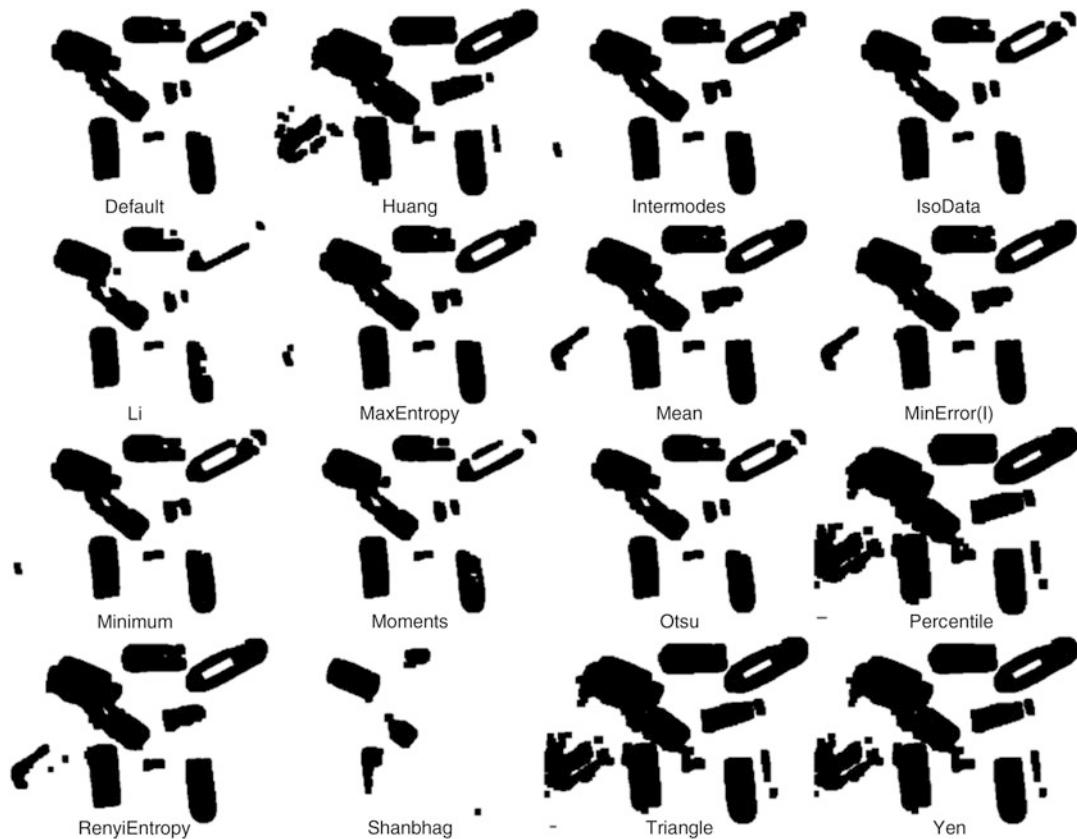


Figure 2.24 Renderings of selected auto-thresholding methods (Images generated using ImageJ auto threshold plug-ins [295])

Local Thresholding

Local thresholding methods take input from the local pixel region and threshold each pixel separately. Here are some common and useful methods.

Local Histogram Equalization

Local histogram equalization divides the image into small blocks, such as 32×32 pixels, and computes a histogram for each block, then re-renders each block using histogram equalization. However, the contrast results may contain block artifacts corresponding to the chosen histogram block size. There are several variations for local histogram equalization, including Contrast Limited Adaptive Local Histogram Equalization (CLAHE) [296].

Integral Image Contrast Filters

A histogram-related method uses integral images to compute local region statistics without the need to compute a histogram, then pixels are remapped accordingly, which is faster and achieves a similar effect as shown in Fig. 2.25.

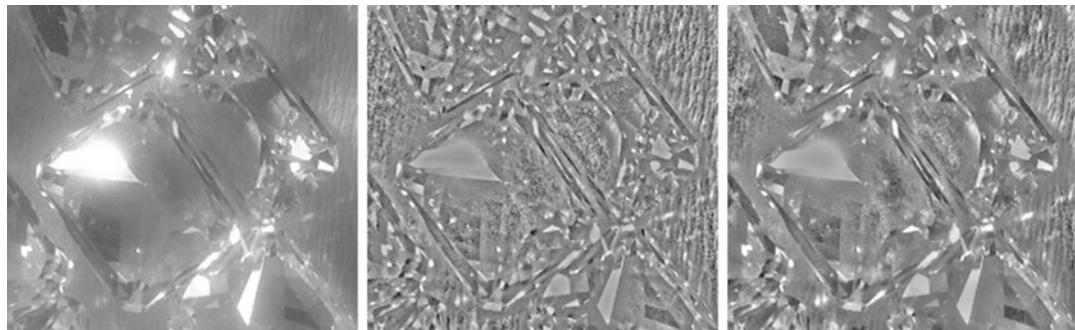


Figure 2.25 Integral image filter from ImageJ to remap contrast in local regions, similar to histogram equalization: (Left) Original. (Center) 20×20 regions. (Right) 40×40 regions

Local Auto Threshold Methods

Local thresholding adapts the threshold based on the immediate area surrounding each target pixel in the image, so local thresholding is more like a standard area operation or filter [495–497]. Local auto thresholding methods are available in standard software packages.² Fig. 2.26 provides some example adaptive local thresholding methods, summarized in Table 2.5.

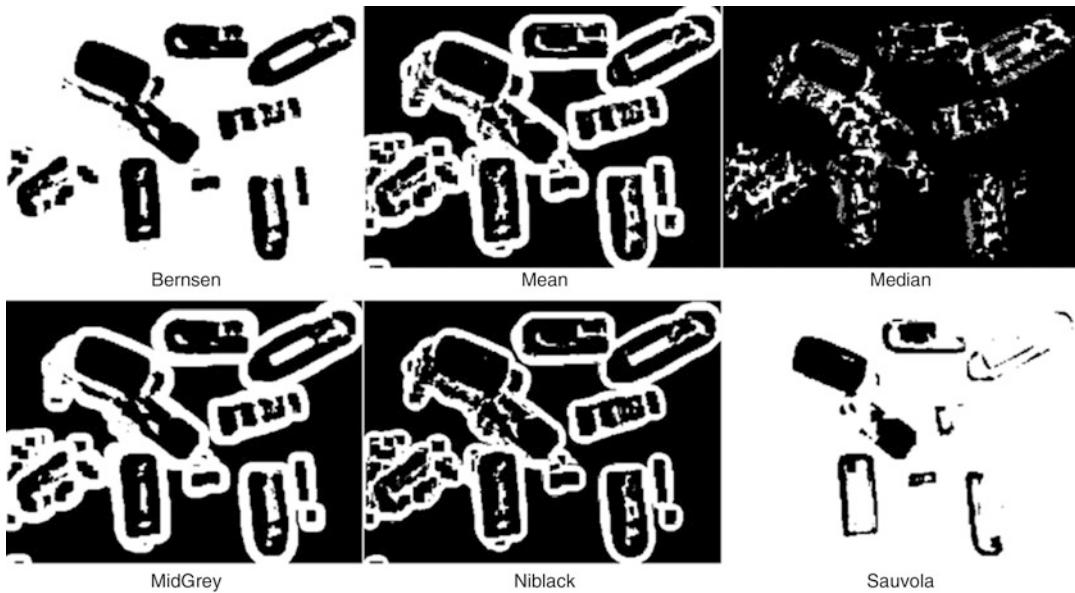


Figure 2.26 Renderings of a selected few local auto and local thresholding methods using ImageJ plug-ins [295]

² See the open-source package ImageJ2, menu item Image → Adjust → Auto Local Threshold | Auto Threshold.

Table 2.5 Selected Few Local Auto-thresholding Methods [295]

Method	Description
Bernsen	Bernsen's algorithm using circular windows instead of rectangles and local midgray values
Mean	Uses the local gray level mean as the threshold
Median	Uses the local gray level mean as the threshold
MidGrey	Uses the local area gray level mean - C (where C is a constant)
Niblack	Niblack's algorithm is: $p = (p > \text{mean} + k * \text{standard_deviation} - c) ? \text{object} : \text{background}$
Sauvola	Sauvola's variation of Niblack: $p = (p > \text{mean} * (1 + k * (\text{standard_deviation} / r - 1))) ? \text{object} : \text{background}$

Summary

In this chapter, we survey image processing as a preprocessing step that can improve image analysis and feature extraction. We develop a taxonomy of image processing methods to frame the discussion, and apply the taxonomy to examples in the four fundamental vision pipelines, as will be developed in the taxonomy of Chap. 5, including (1) local binary descriptors such as LBP, ORB, FREAK; (2) spectra descriptors such as SIFT, SURF; (3) basis space descriptors such as FFT, wavelets; and (4) polygon shape descriptors such as blob object area, perimeter, and centroid. Common problems and opportunities for image preprocessing are discussed. Starting with illumination, noise, and artifact removal, we cover a range of topics including segmentation variations such as depth segmentation and super-pixel methods, binary, gray scale and color morphology, spatial filtering for convolutions and statistical area filters, and basis space filtering.

Chapter 2: Learning Assignments

1. Discuss why image processing is used to improve computer vision pipelines to make the images more amenable to specific feature descriptors.
2. Discuss problems that image preprocessing can solve for gradient based features descriptors such as SIFT.
3. Discuss how image processing goals are influenced in part by the image sensor.
4. Describe some goals for image processing in a general sense, such as goals for corrections and goals for enhancements.
5. Discuss why image preprocessing is important for optimizing a system for making effective use of a given feature descriptor such as SIFT, and why the image preprocessing should be designed specifically for the given the feature descriptor.
6. Describe a hypothetical computer vision application, sketch out an architecture including the feature descriptors used, and describe the goals for image preprocessing prior to feature extraction, and discuss the image processing algorithms chosen to reach the goals.
7. Describe an image processing pipeline to improve the quality of images taken from a very high speed camera (4000 fps) in a low light environment (poor indoor lighting), including the objectives for selecting each algorithm, and alternatives to each algorithm.
8. Describe an image processing pipeline for correcting color images in an outdoor environment in very bright direct sunlight, including the objectives for selecting each algorithm, and alternatives to each algorithm.
9. Discuss how noise affects feature descriptor algorithms.
10. Discuss algorithms to reduce noise, and algorithms to amplify noise.
11. Discuss how noise is related to contrast.
12. Discuss general illumination problems in images, how to detect illumination problems using statistical and image analysis methods, and general approaches to correct the illumination.
13. Discuss how contrast remapping works, and how it can be used to improve image contrast.
14. Describe an image processing pipeline to prepare images for a segmentation algorithm that is based on following connected gradients or intensity ridges.
15. A basic taxonomy for image processing operations can be described based on the region: (1) point operations, (2) line operations, and (3) area operations. Describe each of the three types of region operations in a general sense, describe the limitations of each of the three approaches, and name at least one example algorithm for each of the three approaches.
16. Describe the following color spaces: RGB (additive), CYMK (subtractive), and HSV.
17. Discuss how color image processing works in color intensity space, and why processing in intensity space is usually most effective, compared to processing other color space components such as saturation or hue.
18. Discuss why color processing in RGB space leads to color *moire* effects.
19. Describe the goals of a color management system, including why color management is needed, and provide a few examples.
20. Describe the basic components of a color management system, including the illumination model, the input color space model, and the output color space model.
21. Describe how color gamut mapping works in general, and the problems encountered.
22. Describe how rendering intent is related to gamut mapping.
23. Describe illumination model parameters including white point, black point, and neutral axis.
24. Discuss color saturation, including causes and mitigation strategies.
25. Discuss color resolution, 8-bit color vs. 16-bit color, and when color resolution is critical.

26. Describe a few examples when image processing over local spatial regions is advantageous.
27. Describe how the dot product and convolution are related, and how they are implemented by sketching out an algorithm.
28. Provide the kernel matrix values of a few 3×3 convolution kernels, including a sharpen filter kernel and a blur filter kernel.
29. Discuss why the values of a convolutional filtering kernel should sum to zero.
30. Discuss useful post-processing numerical conditioners applied to convolution results, such as absolute value.
31. Describe how to detect noise in the image (for example histograms and other methods), and spatial filtering approaches for noise removal.
32. Compare the Sobel edge detector algorithm and the Canny edge detector algorithm.
33. Provide the kernel matrix for a few types of edge detectors used for convolutional filtering.
34. Compare Fourier Transform filtering in the frequency domain with convolutional kernel filtering in the discrete spatial domain, and describe the comparative strengths and weakness of each method for image processing.
35. Describe the integral image algorithm and how the integral image is used to implement box filters.
36. Discuss the general goals for image segmentation, and describe at least one segmentation algorithm using pseudo-code.
37. Describe the *binary* morphology operations ERODE and DILATE, discuss the intended use, and provide example 3×3 *binary* kernels for ERODE and DILATE.
38. Describe the *gray-scale* morphology operations MIN and MAX, discuss the intended use, and provide example 3×3 *gray-scale* kernels for MIN and MAX.
39. Discuss in general how a super-pixel algorithm works.
40. Discuss contrast remapping, and how it can be implemented using lookup tables.
41. Compare histogram equalization of global and local regions.
42. Describe the histogram specification algorithm.

Measure twice, cut once.

—Carpenter's saying

This chapter covers the metrics of general feature description, often used for whole images and image regions, including textural, statistical, model based, and basis space methods. Texture, a key metric, is a well-known topic within image processing, and it is commonly divided into structural and statistical methods. Structural methods look for features such as edges and shapes, while statistical methods are concerned with pixel value relationships and statistical moments. Methods for modeling image texture also exist, primarily useful for image synthesis rather than for description. Basis spaces, such as the Fourier space, are also used for feature description.

It is difficult to develop clean partitions between the related topics in image processing and computer vision that pertain to global vs. regional vs. local feature metrics; there is considerable overlap in the applications of most metrics. However, for this chapter, we divide these topics along reasonable boundaries, though those borders may appear to be arbitrary. Similarly, there is some overlap between discussions here on global and regional features and topics that are covered in Chap. 2 on image processing and that are discussed in Chap. 6 on local features. In short, many methods are used for local, regional, and global feature description, as well as image processing, such as the Fourier transform and the LBP.

But we begin with a brief survey of some key ideas in the field of texture analysis and general vision metrics.

Historical Survey of Features

To compare and contrast global, regional, and local feature metrics, it is useful to survey and trace the development of the key ideas, approaches, and methods used to describe features for machine vision. This survey includes image processing (textures and statistics) and machine vision (local, regional, and global features). Historically, the choice of feature metrics was limited to those that were computable at the time, given the limitations in compute performance, memory, and sensor technology. As time passed and technology developed, the metrics have become more complex to compute, consuming larger memory footprints. The images are becoming *multimodal*, combining intensity, color, multiple spectrums, depth sensor information, multiple-exposure settings, high dynamic range imagery, faster frame rates, and more precision and accuracy in x , y , and Z depth. Increases in memory bandwidth and compute performance, therefore, have given rise to new ways to describe feature metrics and perform analysis.

Many approaches to texture analysis have been tried; these fall into the following categories:

- **Structural**, describing texture via a set of micro-texture patterns known as texels. Examples include the numerical description of natural textures such as fabric, grass, and water. Edges, lines, and corners are also structural patterns, and the characteristics of edges within a region, such as edge direction, edge count, and edge gradient magnitude, are useful as texture metrics. Histograms of edge features can be made to define texture, similar to the methods used in local feature descriptors such as SIFT (described in Chap. 6).
- **Statistical**, based on gray level statistical moments describing point pixel area properties, and includes methods such as the co-occurrence matrix or SDM. For example, regions of an image with color intensity within a close range could be considered as having the same texture. Regions with the same histogram could be considered as having the same texture.
- **Model based**, including fractal models, stochastic models, and various semi-random fields. Typically, the models can be used to generate synthetic textures, but may not be effective in recognizing texture, and we do not cover texture generation.
- **Transform or basis based**, including methods such as Fourier, Wavelets, Gabor filters, Zernike, and other basis spaces, which are treated here as a subclass of the statistical methods (statistical moments); however, basis spaces are used in transforms for image processing and filtering as well.

Key Ideas: Global, Regional, and Local Metrics

Let us take a brief look at a few major trends and milestones in feature metrics research. While this brief outline is not intended to be a precise, inclusive look at all key events and research, it describes some general trends in mainstream industry thinking and academic activity.

1960s, 1970s, 1980s—Whole-Object Approaches

During this period, metrics describe mostly whole objects, larger regions, or images; pattern matching was performed on large targets via FFT spectral methods and correlation; recognition methods included object, shape, and texture metrics; and simple geometric primitives were used for object composition. Low-resolution images such as NTSC, PAL, and SECAM were common—primarily gray scale with some color when adequate memory was available. Some satellite images were available to the military with higher resolution, such as LANDSAT images from NASA and SPOT images from France.

Some early work on pattern recognition began to use local interest points and features: notably, Moravic [502] developed a local interest point detector in 1981, and in 1988 Harris and Stephens [148] developed local interest point detectors. Commercial systems began to appear, particularly the View PRB in the early 1980s, which used digital correlation and scale space super-pixels for coarse to fine matching, and real-time image processing and pattern recognition systems were introduced by Imaging Technology. Rack-mounted imaging and machine vision systems began to be replaced by workstations and high-end PCs with add-on imaging hardware, array processors, and software libraries and applications by companies such as Krig Research.

Early 1990s—Partial-Object Approaches

Compute power and memory were increasing, enabling more attention to local feature methods, such as developments from Shi and Tomasi [149] improving the Harris detector methods, Kitchen and Rosenfeld [200] developing gray level corner detection methods, and methods by Wang and Brady [205]. Image moments over polygon shapes were computed using Zernike polynomials in 1990 by

Khotanzad and Hong [268]. Scale space theory was applied to computer vision by Lindberg [502], and many other researchers followed this line of thinking into the future, such as Lowe [153] in 2004.

Metrics described smaller pieces of objects or object components and parts of images; there was increasing use of local features and interest points. Large sets of sub-patterns or basis vectors were used and corresponding metrics were developed. There was increased use of color information; more methods appeared to improve invariance for scale, rotational, or affine variations; and recognition methods were developed based on finding parts of an object with appropriate metrics. Higher image resolution, increased pixel depths, and color information were increasingly used in the public sector (especially in medical applications), along with new affordable image sensors, such as the KODAK MEGA-PLUS, which provided a 1024×1024 image.

Mid-1990s—Local Feature Approaches

More focus was put on metrics that identify small local features surrounding interest points in images. Feature descriptors added more details from a window or patch surrounding each feature, and recognition was based on searching for sets of features and matching descriptors with more complex classifiers. Descriptor spectra included gradients, edges, and colors.

Late 1990s—Classified Invariant Local Feature Approaches

New feature descriptors were developed and refined to be invariant to changes in scale, lightness, rotation, and affine transformations. Work by Schmidt and Mohr [340] advanced and generalized the local feature description methods. Features acted as an alphabet for spelling out complex feature descriptors or vectors whereby the vectors were used for matching. The feature matching and classification stages were refined to increase speed and effectiveness using neural nets and other machine learning methods [134].

Early 2000s—Scene and Object Modeling Approaches

Scenes and objects were modeled as sets of feature components or patterns with well-formed descriptors; spatial relationships between features were measured and used for matching; and new complex classification and matching methods used boosting and related methods to combine strong and weak features for more effective recognition. The SIFT [153] algorithm from Lowe was published; SURF was also published by Bay et al. [152], taking a different approach using HAAR features rather than just gradients. The Viola–Jones method [486] was published, using HAAR features and a boosted learning approach to classification, accelerating matching. The OpenCV library for computer vision was developed by Bradski at INTEL™, and released as open source.

Mid-2000s—Finer-Grain Feature and Metric Composition Approaches

The number of researchers in this field began to mushroom; various combinations of features and metrics (bags of features) were developed by Czurka et al. [226] to describe scenes and objects using key points as described by Sivic [503]; new local feature descriptors were created and old ones refined; and there was increased interest in real-time feature extraction and matching methods for commercial applications. Better local metrics and feature descriptors were analyzed, measured, and used together for increased pattern match accuracy. Also, feature learning and sparse feature codebooks were developed to decrease pattern space, speed up search time, and increase accuracy.

Post-2010—Multimodal Feature Metrics Fusion

There has been increasing use of depth sensor information and depth maps to segment images and describe features and create VOXEL metrics for example see Rusu et al. [380], for example 2D texture metrics are expressed in 3-space. 3D depth sensing methods proliferate, increasing use of

high-resolution images and high dynamic range (HDR) images to enhance feature accuracy, and greater bit depth and accuracy of color images allows for valuable color-based metrics and computational imaging. Increased processing power and cheap, plentiful memory handle larger images on low-cost compute platforms. Faster and better feature descriptors using binary patterns have been developed and matched rapidly using Hamming distance, such as FREAK by Alahi et al. [122] and ORB by Rublee et al. [112]. Multimodal and multivariate descriptors [770, 771] are composed of image features with other sensor information, such as accelerometers and positional sensors.

Future computing research may even come full circle, when sufficient compute and memory capacity exist to perform the older methods, like correlation across multiple scales and geometric perspectives in real-time using parallel and fixed-function hardware methods. This would obviate some of the current focus on small invariant sets of local features and allow several methods to be used together, synergistically. Therefore, the history of development in this field is worth knowing, since it might repeat itself in a different technological embodiment.

Since there is no single solution for obtaining the right set of feature metrics, all the methods developed over time have applications today and are still in use.

Textural Analysis

One of the most basic metrics is *texture*, which is the description of the surface of an image channel, such as color intensity, like an elevation map or terrain map. Texture can be expressed globally or within local regions. Texture can be expressed *locally* by statistical relationships among neighboring pixels in a region, and it can be expressed *globally* by summary relationships of pixel values within an image or region. For a sampling of the literature covering a wide range of texture methods, see Refs. [13, 16–20, 52, 53, 302, 304, 305].

According to Gonzalez [4], there are three fundamental classes of texture in image analysis: statistical, structural, and spectral. *Statistical* measures include histograms, scatter plots, and SDMs. *Structural* techniques are more concerned with locating patterns or structural primitives in an image, such as parallel lines, regular patterns, and so on. These techniques are described in [1, 5, 8, 11]. *Spectral* texture is derived from analysis of the frequency domain representation of the data. That is, a fast Fourier transform is used to create a frequency domain image of the data, which can then be analyzed using Fourier techniques.

Histograms reveal overall pixel value distributions but say nothing about spatial relationships. Scatter plots are essentially two-dimensional histograms, and do not reveal any spatial relationships. A good survey is found in Ref. [307].

Texture has been used to achieve several goals:

- Texture-based segmentation (covered in Chap. 2).
- Texture analysis of image regions (covered in this chapter).
- Texture synthesis, creating images using synthetic textures (not covered in this book).

In computer vision, texture metrics are devised to describe the perceptual attributes of texture by using discrete methods. For instance, texture has been described *perceptually* with several properties, including:

- Contrast
- Color
- Coarseness

- Directionality
- Line-likeness
- Roughness
- Constancy
- Grouping
- Segmentation

If textures can be recognized, then image regions can be segmented based on texture and the corresponding regions can be measured using shape metrics such as area, perimeter, and centroid (as discussed in Chap. 6). Chapter 2 included a survey of segmentation methods, some of which are based on texture. Segmented texture regions can be recognized and compared for computer vision applications. Micro-textures of a local region, such as the LBP discussed in detail in Chap. 6, can be useful as a feature descriptor, and macro-textures can be used to describe a homogenous texture of a region such as a lake or field of grass, and therefore have natural applications to image segmentation. In summary, texture can be used to describe global image content, image region content, and local descriptor region content. The distinction between a feature descriptor and a texture metric may be small.

Sensor limitations combined with compute and memory capabilities of the past have limited the development of texture metrics to mainly 2D gray scale metrics. However, with the advances toward pervasive computational photography in every camera providing higher resolution images, higher frame rates, deeper pixels, depth imaging, more memory, and faster compute, we can expect that corresponding new advances in texture metrics will be made.

Here is a brief historical survey of texture metrics.

1950s Through 1970s—Global Uniform Texture Metrics

Auto-correlation or cross-correlation was developed by Kaizer [26] in 1955 as a method of looking for randomness and repeating pattern features in aerial photography, where auto-correlation is a statistical method of correlating a signal or image with a time-shifted version of itself, yielding a computationally simple method to analyze ground cover and structures.

Bajcsy [25] developed Fourier spectrum methods in 1973 using various types of filters in the frequency domain to isolate various types of repeating features as texture.

Gray level spatial dependency matrices, GLCMs, SDMs or co-occurrence matrices [6] were developed and used by Haralick in 1973, along with a set of summary statistical metrics from the SDMs to assist in numerical classification of texture. Some, but not all, of the summary metrics have proved useful; however, analysis of SDMs and development of new SDM metrics have continued, involving methods such as 2D visualization and filtering of the SDM data within spatial regions [23], as well as adding new SDM statistical metrics, some of which are discussed in this chapter.

1980s—Structural and Model-Based Approaches for Texture Classification

While early work focused on micro-textures describing statistical measures between small kernels of adjacent pixels, macro-textures developed to address the structure of textures within a larger region. K. Laws developed *texture energy-detection* methods in 1979 and 1980 [27–29], as well as *texture classifiers*, which may be considered the forerunners of some of the modern classifier concepts. The Laws method could be implemented as a texture classifier in a parallel pipeline with stages for taking gradients via of a set of convolution masks over Gaussian filtered images to isolate texture micro features, followed by a Gaussian smoothing stage to deal with noise, followed by the energy calculation from the combined gradients, followed by a classifier which matched texture descriptors.

Eigenfilters were developed by Ade [30] in 1983 as an alternative to the Laws gradient or energy methods and SDMs; eigenfilters are implemented using a covariance matrix representation of local 3×3 pixel region intensities, which allows texture analysis and aggregation into structure based on the variance within eigenvectors in the covariance matrix.

Structural approaches were developed by Davis [31] in 1979 to focus on gross structure of texture rather than primitives or micro-texture features. *Hough transforms* were invented in 1972 by Duda and Hart [220] as a method of finding lines and curves, and it was used by Eichmann and Kasparis [32] in 1988 to provide invariant texture description.

Fractal methods and *Markov random field* methods were developed into texture descriptors, and while these methods may be good for texture synthesis, they do not map well to texture classification, since both Fractal and Markov random field methods use random fields, thus there are limitations when applied to real-world textures that are not random.

1990s—Optimizations and Refinements to Texture Metrics

In 1993, Lam and Ip [33, 39] used pyramid segmentation methods to achieve spatial invariance, where an image is segmented into homogenous regions using Voronoi polygon tessellation and irregular pyramid segmentation techniques around Q points taken from a binary thresholded image; five shape descriptors are calculated for each polygon: area, perimeter, roundness, orientation, and major/minor axis ratio, combined into texture descriptors.

Local binary patterns (LBP) were developed in 1994 by Ojala et al. [165] as a novel method of encoding both pattern and contrast to define texture [15, 16, 35, 36]; since then, hundreds of researchers have added to the LBP literature in the areas of theoretical foundations, generalization into 2D and 3D, domain-specific interest point descriptors used in face detection, and spatiotemporal applications to motion analysis [34]. LBP research remains quite active at this time. LBPs are covered in detail in Chap. 6. There are many applications for the powerful LBP method as texture metric, a feature descriptor, and an image processing operator, the latter which was discussed in Chap. 2.

2000 to Today—More Robust Invariant Texture Metrics and 3D Texture

Feature metrics research is investigating texture metrics that are invariant to scale, rotation, lighting, perspective, and so on to approach the capabilities of human texture discrimination. In fact, texture is used interchangeably as a feature descriptor in some circles. The work by Pun and Lee [37] is an example of development of rotational invariant texture metrics, as well as scale invariance. Invariance attributes are discussed in the general taxonomy in Chap. 5.

The next wave of metrics being developed increasingly will take advantage of 3D depth information. One example is the surface shape metrics developed by Spence [38, 304] in 2003, which provide a bump-map type metric for affine invariant texture recognition and texture description with scale and perspective invariance. Chapter 6 also discusses some related 3D feature descriptors.

Statistical Methods

The topic of statistical methods is vast, and we can only refer the reader to selected literature as we go along. One useful and comprehensive resource is the online NIST National Institute of Science and Technology Engineering Statistics Handbook,¹ including examples and links to additional resources and tools.

¹ See the NIST online resource for engineering statistics: <http://www.itl.nist.gov/div898/handbook/>

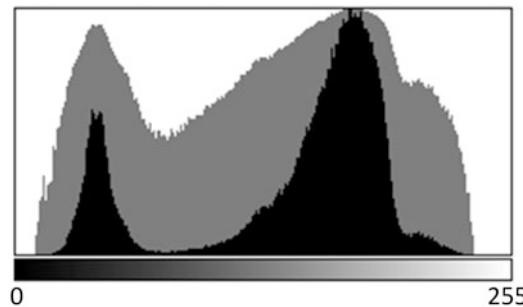


Figure 3.1 Histogram with linear scale values (*black*) and log scale values (*gray*), illustrating how the same data is interpreted differently based on the chart scale

Statistical methods may be drawn upon at any time to generate novel feature metrics. Any feature, such as pixel values or local region gradients, can be expressed statistically by any number of methods. Simple methods, such as the histogram shown in Fig. 3.1, are invaluable. Basic statistics such as minimum, maximum, and average values can be seen easily in the histogram shown in Chap. 2 in Fig. 2.21. We survey several applications of statistical methods to computer vision here.

Texture Region Metrics

Now we look in detail at the specific metrics for feature description based on texture. Texture is one of the most-studied classes of metrics. It can be thought of in terms of the surface—for example, a burlap bag compared to silk fabric. There are many possible textural relationships and signatures that can be devised in a range of domains, with new ones being developed all the time. In this section we survey some of the most common methods for calculating texture metrics:

- Edge metrics
- Cross-correlation
- Fourier spectrum signatures
- Co-occurrence matrix, Haralick features, extended SDM features
- Laws texture metrics
- Tessellation
- Local binary patterns (LBP)
- Dynamic textures

Within an image, each image region has a *texture signature*, where texture is defined as a common structure and pattern within that region. Texture signatures may be a function of position and intensity relationships, as in the spatial domain, or be based on comparisons in some other function basis and feature domain, such as frequency space using Fourier methods.

Texture metrics can be used to both segment and describe regions. Regions are differentiated based on texture homogeneousness, and as a result, texture works well as a method for region segmentation. Texture is also a good metric for feature description, and as a result it is useful for feature detection, matching, and tracking.

[Appendix B](#) contains several ground truth datasets with example images for computing texture metrics, including the CUReT reflectance and texture database from Columbia University. Several

key papers describe the metrics used against the CUReT dataset [21, 40–42] including the appearance of a surface as a bidirectional reflectance distribution function (BRDF) and a bidirectional texture function (BTF).

These metrics are intended to measure texture as a function of direction and illumination, to capture coarse details and fine details of each surface. If the surface texture contains significant sub-pixel detail not apparent in single pixels or groups of pixels, the BRDF reflectance metrics can capture the *coarse reflectance* details. If the surface contains pixel-by-pixel difference details, the BTF captures the *fine texture* details.

Edge Metrics

Edges, lines, contours, or ridges are basic textural features [308, 309]. A variety of simple metrics can be devised just by analyzing the edge structure of regions in an image. There are many edge metrics in the literature, and a few are illustrated here.

Computing edges can be considered on a continuum of methods from interest point to edges, where the interest point may be a single pixel at a gradient maxima or minima, with several connected gradient maxima pixels composed into corners, ridges line segments, or a contours. In summary, a *gradient point* is a degenerate edge, and an edge is a collection of connected gradient points.

The edge metrics can be computed locally or globally on image regions as follows:

- Compute the gradient $\mathbf{g}(\mathbf{d})$ at each pixel, selecting an appropriate gradient operator $\mathbf{g}()$ and select the appropriate kernel size or distance \mathbf{d} to target either micro or macro edge features.
- The distance \mathbf{d} or kernel size can be varied to achieve different metrics; many researchers have used 3×3 kernels.
- Compute edge orientation by binning gradient directions for each edge into a histogram; for example, use 45° angle increment bins for a total of 8 bins at $0^\circ, 45^\circ, 90^\circ, 135^\circ, 180^\circ, 225^\circ, 270^\circ$.

Several other methods can be used to compute edge statistics. The representative methods are shown here; see also Shapiro and Stockton [499] for a standard reference.

Edge Density

Edge density can be expressed as the average value of the gradient magnitudes g_m in a region.

$$E_d = \frac{g_m(d)}{\text{pixels in region}}$$

Edge Contrast

Edge contrast can be expressed as the ratio of the average value of gradient magnitudes to the maximum possible pixel value in the region.

$$E_c = \frac{E_d}{\text{max pixel value}}$$

Edge Entropy

Edge randomness can be expressed as a measure of the Shannon entropy of the gradient magnitudes.

$$E_e = \sum_{i=0}^n g_m(x_i) \log_b g_m(x_i)$$

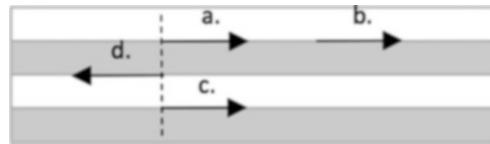


Figure 3.2 Gradient direction of edges a, b, c, d used to illustrate relationships for edge metrics

Edge Directivity

Edge directivity can be expressed as a measure of the Shannon entropy of the gradient directions.

$$E_e = \sum_{i=0}^n g_d(x_i) \log_b g_d(x_i)$$

Edge Linearity

Edge linearity measures the co-occurrence of collinear edge pairs using gradient direction, as shown by edges a–b in Fig. 3.2.

$$E_l = \text{cooccurrence of colinear edge pairs}$$

Edge Periodicity

Edge periodicity measures the co-occurrence of identically oriented edge pairs using gradient direction, as shown by edges a–c in Fig. 3.2.

$$E_p = \text{cooccurrence of identically oriented edge pairs}$$

Edge Size

Edge size measures the co-occurrence of opposite oriented edge pairs using gradient direction, as shown by edges a–d in Fig. 3.2.

$$E_s = \text{cooccurrence of opposite oriented edge pairs}$$

Edge Primitive Length Total

Edge primitive length measures the total length of all gradient magnitudes along the same direction.

$$E_t = \text{total length of gradient magnitudes with same direction}$$

Cross-Correlation and Auto-correlation

Cross-correlation [26] is a metric showing similarity between two signals with a time displacement between them. *Auto-correlation* is the cross-correlation of a signal with a time-displaced version of itself. In the literature on signal processing, cross-correlation is also referred to as a *sliding inner product* or *sliding dot product*. Typically, this method is used to search a large signal for a smaller pattern.

$$f * g = \bar{f}(-t) * g(t)$$

Using the Wiener–Khinchin theorem as a special case of the general cross-correlation theorem, cross-correlation can be written as simply the Fourier transform of the absolute square of the function f_v , as follows:

$$c(t) = \mathcal{F}_v[|f_v|^2](t)$$

In computer vision, the feature used for correlation may be a 1D line of pixels or gradient magnitudes, a 2D pixel region, or a 3D voxel volume region. By comparing the features from the current image frame and the previous image frame using cross-correlation derivatives, we obtain a useful texture change correlation metric.

By comparing displaced versions of an image with itself, we obtain a set of either local or global auto-correlation texture metrics. Auto-correlation can be used to detect repeating patterns or textures in an image, and also to describe the texture in terms of fine or coarse, where coarse textures show the auto-correlation function dropping off more slowly than fine textures. See also the discussion of correlation in Chap. 6 and Fig. 6.20.

Fourier Spectrum, Wavelets, and Basis Signatures

Basis transforms, such as the FFT, decompose a signal into a set of basis vectors from which the signal can be synthesized or reconstructed. Viewing the set of basis vectors as a spectrum is a valuable method for understanding image texture and for creating a signature. Several basis spaces are discussed in this chapter, including Fourier, HAAR, wavelets, and Zernike.

Although computationally expensive and memory intensive, the Fast Fourier Transform (FFT) is often used to produce a frequency spectrum signature. The FFT spectrum is useful for a wide range of problems. The computations typically are limited to rectangular regions of fixed sizes, depending on the radix of the transform (see Bracewell [219]).

As shown in Fig. 3.3, Fourier spectrum plots reveal definite image features useful for texture and statistical analysis of images. For example, Fig. 3.10 shows an FFT spectrum of LBP pattern metrics. Note that the Fourier spectrum has many valuable attributes, such as rotational invariance, as shown

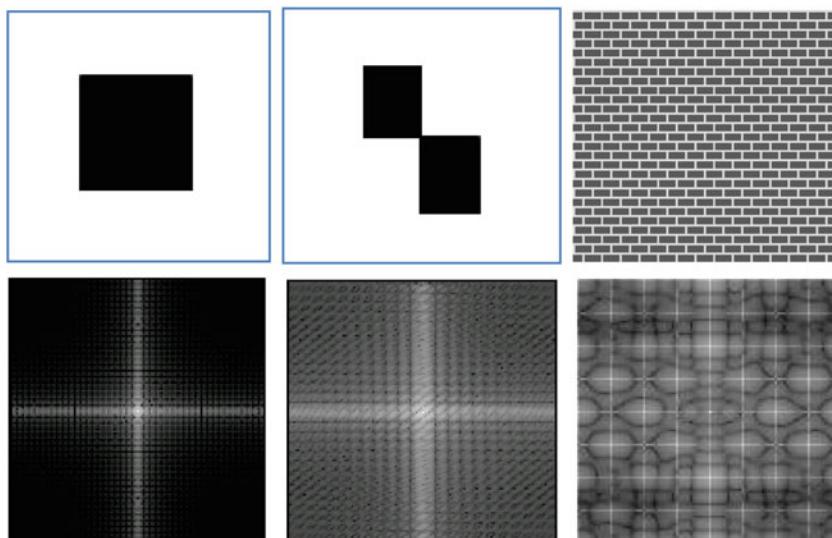


Figure 3.3 (Top row) Example images with texture. (Bottom row) Texture and shape information revealed in the corresponding FFT power spectrums

in Fig. 3.3, where a texture image is rotated 90° and the corresponding FFT spectrums exhibit the same attributes, only rotated 90°.

Wavelets [219] are similar to Fourier methods, and have become increasingly popular for texture analysis [303], discussed later in the section on basis spaces.

Note that the FFT spectrum as a texture metric or descriptor is rotational invariant, as shown in the bottom left image of Fig. 3.3. FFT spectra can be taken over rectangular 2D regions. Also, 1D arrays such as annuli or Cartesian coordinates of the shape taken around the perimeter of an object shape can be used as input to an FFT and as an FFT descriptor shape metric.

Co-occurrence Matrix, Haralick Features

Haralick [6] proposed a set of 2D texture metrics calculated from directional differences between adjacent pixels, referred to as *co-occurrence* matrices, *spatial dependency matrices* (SDM), or gray level co-occurrence matrices (GLCM). A complete set of four (4) matrices are calculated by evaluating the difference between adjacent pixels in the *x*, *y*, *diagonal x* and *diagonal y* directions, as shown in Fig. 3.4, and further illustrated with a 4×4 image and corresponding co-occurrence tables shown in Fig. 3.5.

One benefit of the SDM as a texture metric is that it is easy to calculate in a single pass over the image. The SDM is also fairly invariant to rotation, which is often a difficult robustness attribute to attain. Within a segmented region or around an interest point, the SDM plot can be a valuable texture metric all by itself, therefore useful for texture analysis, feature description, noise detection, and pattern matching.

For example, if a camera has digital-circuit readout noise, it will show up in the SDM for the *x* direction only if the lines are scanned out of the sensor one at a time in the *x* direction, so using the SDM information will enable intelligent sensor processing to remove the readout noise. However, it should be noted that SDM metrics are not always useful alone, and should be qualified with additional feature information. The SDM is primarily concerned with spatial relationships, with regard to spatial orientation and frequency of occurrence. So, it is primarily a statistical measure.

The SDM is calculated in four orientations, as shown in Fig. 3.4. Since the SDM is only concerned with adjacent pairs of pixels, these four calculations cover all possible spatial orientations. SDMs could be extended beyond 2×2 regions by using forming kernels extending into 5×5 , 7×7 , 9×9 , and other dimensions.

A *spatial dependency matrix* is basically a count of how many times a given pixel value occurs next to another pixel value. Fig. 3.5 illustrates the concept. For example, assume we have an 8-bit image (0..255). If an SDM shows that pixel value *x* frequently occurs adjacent to pixels within the range $x + 1$ to $x - 1$, then we would say that there is a “smooth” texture at that intensity. However, if

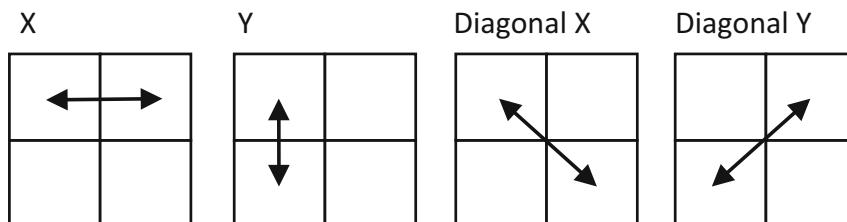


Figure 3.4 Four different vectors used for the Haralick texture features, where the difference of each pixel in the image is plotted to reveal the texture of the image

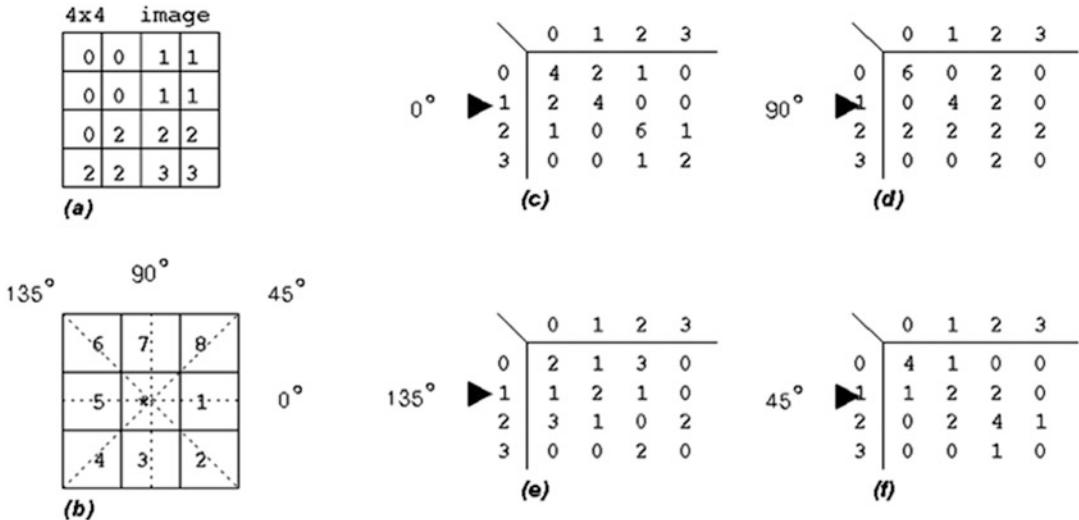


Figure 3.5 (a) 4×4 pixel image, with gray values in the range 0–3. (b) Nearest neighbor angles corresponding to SDM tables. (c–f) With neighborhood counts for each angle

pixel value x frequently occurs adjacent to pixels within the range $x + 70$ to $x - 70$, we would say that there is quite a bit of contrast at that intensity, if not noise.

A critical point in using SDMs is to be sensitive to the varied results achieved when sampling over small vs. large image areas. By sampling the SDM over a smaller area (say 64×64 pixels), details will be revealed in the SDMs that would otherwise be obscured. The larger the size of the sample image area, the more the SDM will be populated. And the more samples taken, the more likely that detail will be obscured in the SDM image plots. Actually, smaller areas (i.e., 64×64 pixels) are a good place to start when using SDMs, since smaller areas are faster to compute and will reveal a lot about local texture.

The Haralick metrics are shown in Fig. 3.6.

The statistical characteristics of the SDM have been extended by several researchers to add more useful metrics [23], and SDMs have been applied to 3D volumetric data by a number of researchers with good results [22].

Extended SDM Metrics (Krig SDM Metrics)

Extensions to the Haralick metrics have been developed by the author [23], primarily motivated by a visual study of SDM plots as shown in Fig. 3.7. Applications for the extended SDM metrics include texture analysis, data visualization, and image recognition. The visual plots of the SDMs alone are valuable indicators of pixel intensity relationships, and are worth using along with histograms to get to know the data.

The extended SDM metrics include centroid, total coverage, low-frequency coverage, total power, relative power, locus length, locus mean density, bin mean density, containment, linearity, and linearity strength. The extended SDM metrics capture key information that is best observed by looking at the SDM plots. In many cases the extended SDM metric are be computed four times, once for each SDM direction of 0° , 45° , 90° , and 135° , as shown in Fig. 3.5.

The SDMs are interesting and useful all by themselves when viewed as an image. Many of the texture metrics suggested are obvious after viewing and understanding the SDMs; others are neither

Angular Second Moment	$\sum_i \sum_j p(i,j)^2$
Contrast	$\sum_{n=0}^{N_x-1} n^2 \left\{ \sum_{i=1}^{N_x} \sum_{j=1}^{N_x} p(i,j) \right\}, i-j = n$
Correlation	$\frac{\sum_i \sum_j (ij)^2 p(i,j) - \mu_x \mu_y}{\sigma_x \sigma_y}$ Where μ_x, μ_y, σ_x , and σ_y are the means and std. deviations of p_x and p_y , the partial probability density functions
Sum of Squares: Variance	$\sum_i \sum_j (i-\mu)^2 p(i,j)$
Inverse Difference Moment	$\sum_i \sum_j \frac{1}{1+(i-j)^2} p(i,j)$
Sum Average	$\sum_{i=2}^{2N_x} i p_{x+y}(i)$ Where x and y are the coordinates (row and column) of an entry in the co-occurrence matrix, and $p_{x+y}(i)$ is the probability of co-occurrence matrix coordinates summing to $x+y$
Sum Variance	$\sum_{i=2}^{2N_x} (i-f_x)^2 p_{x+y}(i)$
Sum Entropy	$-\sum_{i=2}^{2N_x} i p_{x+y}(i) \log(p_{x+y}(i)) = f_x$
Entropy	$-\sum_i \sum_j p(i,j) \log(p(i,j))$
Difference Variance	$\sum_{i=0}^{N_x-1} i^2 p_{x-y}(i)$
Difference Entropy	$-\sum_{i=0}^{N_x-1} p_{x-y}(i) \log(p_{x-y}(i))$
Info. Measure of Correlation 1	$\frac{HX - XY}{\max\{HX, HY\}}$
Info. Measure of Correlation 2	$(1 - \exp[-2(HXY2 - HXY)])^{\frac{1}{2}}$ Where $HXY = -\sum_i \sum_j p(i,j) \log(p(i,j))$, HX , HY are the entropies of p_x and p_y , $HXY2 =$ $-\sum_i \sum_j p(i,j) \log(p_x(i)p_y(j))$, $HXY2 =$ $-\sum_i \sum_j p_x(i)p_y(j) \log(p_x(i)p_y(j))$
Max. Correlation coeff.	Square root of the second largest eigenvalue of Q Where $Q(i,j) = \sum_k \frac{p(i,k)p(j,k)}{p_x(i)p_y(k)}$

Figure 3.6 Haralick texture metrics. (Image used by permission, © Intel Press, from Building Intelligent Systems)

obvious nor apparently useful until developing a basic familiarity with the visual interpretation of SDM image plots. Next, we survey the following:

- **Example SDMs showing four directional SDM maps:** A complete set of SDMs would contain four different plots, one for each orientation. Interpreting the SDM plots visually reveals useful information. For example, an image with a smooth texture will yield a narrow diagonal band of co-occurrence values; an image with wide texture variation will yield a larger spread of values; a noisy image will yield a co-occurrence matrix with outlier values at the extrema. In some cases, noise may only be distributed along one axis of the image—perhaps, across rows or the x axis, which could indicate sensor readout noise as each line is read out of the sensor, suggesting a row- or line-oriented image preparation stage in the vision pipeline to compensate for the camera.

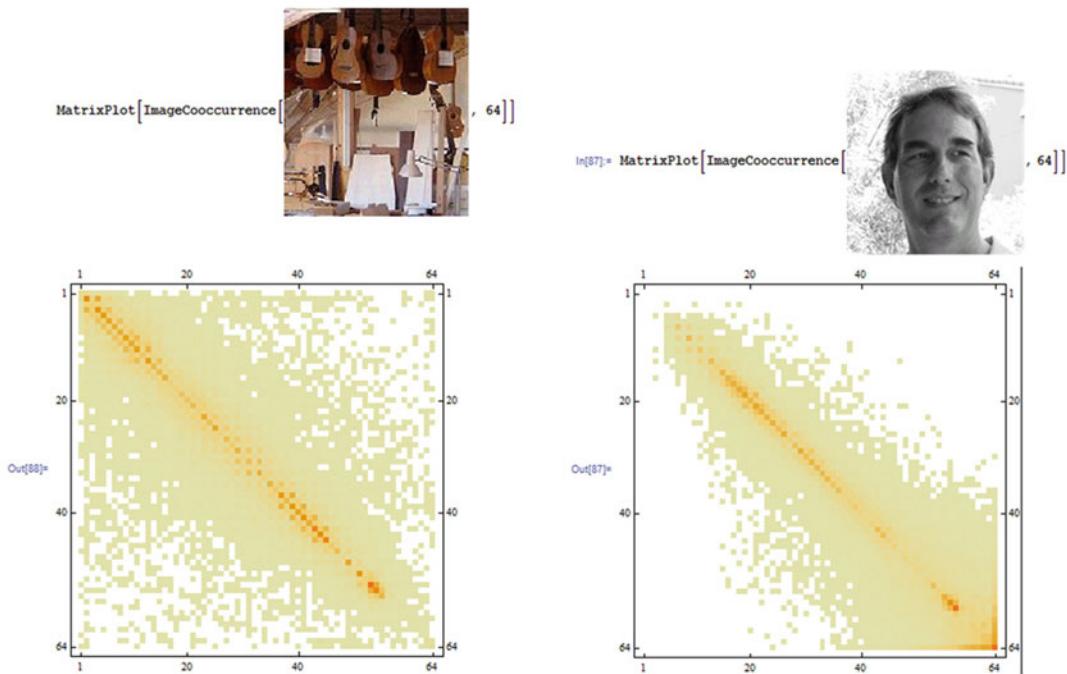


Figure 3.7 Pair of image co-occurrence matrix plots (x-axis plots) computed over 64 bins in the *bottom row* corresponding to the images in the *top row*

- **Extended SDM texture metrics:** The addition of 12 other useful statistical measures to those proposed by Haralick.
- **Some code snippets:** These illustrate the extended SDM computations, full source code is shown in [Appendix D](#).

In Fig. 3.7, several of the extended SDM metrics can be easily seen, including containment and locus mean density. Note that the right image does not have a lot of outlier intensity points or noise (good containment); most of the energy is centered along the diagonal (tight locus), showing a rather smooth set of image pixel transitions and texture, while the left image shows a wider range of intensity values. For some images, wider range may be noise spread across the spectrum (poor containment), revealing a wider band of energy and contrast between adjacent pixels.

Metric 1: Centroid

To compute the centroid, for each SDM bin $p(i,j)$, the count of the bin is multiplied by the bin coordinate for x,y and also the total bin count is summed. The centroid calculation is weighted to compute the centroid based on the actual bin counts, rather than an unweighted “binary” approach of determining the center of the binning region based on only bin data presence. The result is the weighted center of mass over the SDM bins.

$$\text{centroid} = \sum_{i=0}^n \sum_{j=0}^m \begin{pmatrix} x = jp(i,j) \\ y = ip(i,j) \\ z = p(i,j) \end{pmatrix}$$

$$\text{centroid}_y = \frac{y}{z}$$

$$\text{centroid}_x = \frac{x}{z}$$

Metric 2: Total Coverage

This is a measure of the spread, or range of distribution, of the binning. A small coverage percentage would be indicative of an image with few gray levels, which corresponds in some cases to image smoothness. For example, a random image would have a very large coverage number, since all or most of the SDM bins would be hit. The coverage feature metrics (2, 3, 4), taken together with the linearity features suggested below (11, 12), can give an indication of image smoothness.

$$\text{coverage}_c = \sum_{i=0}^n \sum_{j=0}^m \begin{pmatrix} 1 & \text{if } 0 < p(i,j), \\ 0 & \text{otherwise} \end{pmatrix}$$

$$\text{coverage}_t = \frac{\text{coverage}_c}{(n * m)}$$

Metric 3: Low-Frequency Coverage

For many images, any bins in the SDM with bin counts less than a threshold value, such as 3, may be considered as noise. The low-frequency coverage metric, or noise metric, provides an idea how much of the binning is in this range. This may be especially true as the sample area of the image area increases. For whole images, a threshold of 3 has proved to be useful for determining if a bin contains noise for a data range of 0-255, and using the SDM over smaller local kernel regions may use all the values with no thresholding needed.

$$\text{coverage}_c = \sum_{i=0}^n \sum_{j=0}^m \text{if } 0 < p(i,j) < 3 \begin{pmatrix} 1, \\ \text{else } 0 \end{pmatrix}$$

$$\text{coverage}_l = \frac{\text{coverage}_c}{(n * m)}$$

Metric 4: Corrected Coverage

Corrected coverage is the total coverage with noise removed.

$$\text{coverage}_n = \text{coverage}_t - \text{coverage}_l$$

Metric 5: Total Power

The power metric provides a measure of the swing in value between adjacent pixels in an image, and is computed in four directions. A smooth image will have a low power number because the differences between pixels are smaller. Total power and relative power are inter-related, and relative power is computed using the total populated bins (z) and total difference power (t).

$$\text{power}_c = \sum_{i=0}^n \sum_{j=0}^m \text{if } p(i,j) \neq 0 \left(\begin{array}{l} z+ = 1, \\ t+ = |i - j| \end{array} \right)$$

$$\text{power}_t = t$$

Metric 6: Relative Power

The relative power is calculated based on the scaled total power using nonempty SDM bins t , while the total power uses all bins.

$$\text{power}_r = \frac{t}{z}$$

Metric 7: Locus Mean Density

For many images, there is a “locus” area of high-intensity binning surrounding the bin axis (locus axis is where adjacent pixels are of the same value $x = y$) corresponding to a diagonal line drawn from the upper left corner of the SDM plot. The degree of clustering around the locus area indicates the amount of smoothness in the image. Binning from a noisy image will be scattered with little relation to the locus area, while a cleaner image will show a pattern centered about the locus.

$$\text{locus}_c = \sum_{i=0}^n \sum_{j=0}^m \text{if } 0 < |i - j| < 7 \left(\begin{array}{l} z+ = 1, \\ d+ = p(i,j) \end{array} \right)$$

$$\text{locus}_d = \frac{d}{z}$$

The locus mean density is an average of the bin values within the locus area. The locus is the area around the center diagonal line, within a band of 7 pixels on either side of the identity line ($x = y$) that passes down the center of each SDM. However, the number 7 is not particularly special, but based upon experience, it just gives a good indication of the desired feature over whole images. This feature is good for indicating smoothness.

Metric 8: Locus Length

The locus length measures the range of the locus concentration about the diagonal. The algorithm for locus length is a simple count of bins populated in the locus area; a threshold band of 7 pixels about the locus has been found useful.

```

y = length = 0;
while (y < 256) {
    x = count = 0;
    while (x < 256) {
        n = |y-x|;
        if (p[i,j] == 0) && (n < 7) count++;
        x++;
    }
    if (!count) length++;
    y++;
}

```

Metric 9: Bin Mean Density

This is simply the average bin count from nonempty bins.

$$\text{density}_c = \sum_{i=0}^n \sum_{j=0}^m \text{if } p(i,j) \neq 0 (v = p(i,j), z+ = 1)$$

$$\text{density}_b = \frac{v}{z}$$

Metric 10: Containment

Containment is a measure of how well the binning in the SDM is contained within the boundaries or edges of the SDM, and there are four edges or boundaries, for example assuming a data range [0...255], there are containment boundaries along rows 0 and 255, and along columns 0 and 255. Typically, the bin count m is 256 bins, or possibly less such as 64. To measure containment, basically the perimeters of the SDM bins are checked to see if any binning has occurred, where the perimeter region bins of the SDM represent extrema values next to some other value. The left image in Fig. 3.7 has lower containment than the right image, especially for the low values.

$$\text{containment}_1 = \sum_{i=0}^m \text{if } p(i,0) \neq 0 (c_1+ = 1)$$

$$\text{containment}_2 = \sum_{i=0}^m \text{if } p(i,m) \neq 0 (c_2+ = 1)$$

$$\text{containment}_3 = \sum_{i=0}^m \text{if } p(0,i) \neq 0 (c_3+ = 1)$$

$$\text{containment}_4 = \sum_{i=0}^m \text{if } p(m,i) \neq 0 (c_4+ = 1)$$

$$\text{containment}_t = 1.0 - \frac{(c_1+ + c_2+ + c_3+ + c_4+)}{4m}$$

If extrema are hit frequently, this probably indicates some sort of overflow condition such as numerical overflow, sensor saturation, or noise. The binning is treated unweighted. A high containment number indicates that all the binning took place within the boundaries of the SDM. A lower number indicates some bleeding. This feature appears visually very well in the SDM plots.

Metric 11: Linearity

The linearity characteristic may only be visible in a single orientation of the SDM, or by comparing SDMs. For example, the image in Fig. 3.8 reveals some linearity variations across the set of SDMs. This is consistent with the image sensor used (older tube camera).

$$\text{linearity}_c = \sum_{j=0}^m \text{if } p(jm,j) > 1 \left(\begin{array}{l} z+ = 1, \\ l+ = p(256j,j) \end{array} \right)$$

$$\text{linearity}_{\text{normalized}} = \frac{z}{m}$$

$$\text{linearity}_{\text{strength}} = \frac{l}{z} * m^{-1}$$

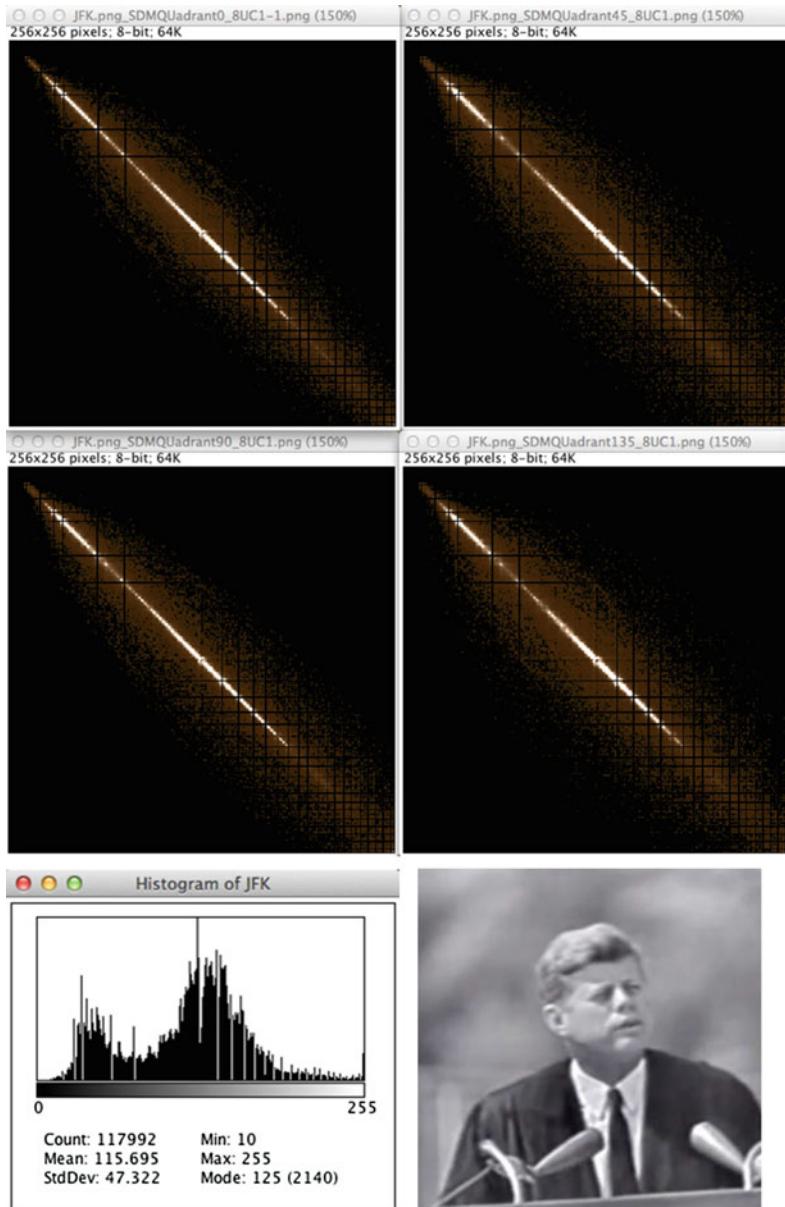


Figure 3.8 SDMs from old tube camera showing linearity variations in the sensor, includes full set of 0°, 45°, 90°, and 135° SDMs. (Public domain image from National Archives)

Metric 12: Linearity Strength

The algorithm for linearity strength is shown in Metric 11. If there is any linearity present in a given angle of SDM, both linearity strength and linearity will be comparatively higher at this angle than the other SDM angles (Table 3.1).

Table 3.1 Extended SDM metrics from Fig. 3.8

METRIC	0 Deg	45 Deg	90 Deg.	135 Deg.	Ave.
xcentroid	115	115	115	115	115
ycentroid	115	115	115	115	115
low_frequency_coverage	0.075	0.092	0.103	0.108	0.095
total_coverage	0.831	0.818	0.781	0.780	0.803
corrected_coverage	0.755	0.726	0.678	0.672	0.708
total_power	2.000	3.000	5.000	5.000	3.750
relative_power	17.000	19.000	23.000	23.000	20.500
locus_length	71	72	71	70	71
locus_mean_density	79	80	74	76	77
bin_mean_density	21	19	16	16	18
containment	0.961	0.932	0.926	0.912	0.933
linearity	0.867	0.848	0.848	0.848	0.853
linearity_strength	1.526	1.557	0.973	1.046	1.276

Laws Texture Metrics

The Laws metrics [24, 27–29] provide a structural approach to texture analysis, using a set of masking kernels to measure texture energy or variation within fixed sized local regions, similar to the 2×2 region SDM approach but using larger pixel areas to achieve different metrics.

The basic Laws algorithm involves classifying each pixel in the image into texture based on local energy, using a few basic steps:

1. The mean average intensity from each kernel neighborhood is subtracted from each pixel to compensate for illumination variations.
2. The image is convolved at each pixel using a set of kernels, each of which sums to zero, followed by summing the results to obtain the absolute average value over each kernel window.
3. The difference between the convolved image and the original image is measured, revealing the Laws energy metrics.

Laws defines a set of nine separable kernels to produce a set of texture region energy metrics, and some of the kernels work better than others in practice. The kernels are composed via matrix multiplication from a set of four vector masks L5, E5, S5, and R5, described below. The kernels were originally defined as 5×5 masks, but 3×3 approximations have been used also, as shown below.

5×5 form

L5	Level Detector	[1 4 6 4 1]
E5	Edge Detector	[-1 -2 0 2 1]
S5	Spot Detector	[-1 0 2 0 1]
R5	Ripple Detector	[1 -4 6 -4 1]

3×3 approximations of 5×5 form

L3	Level Detector	[1 2 1]
E3	Edge Detector	[-1 0 1]
S3	Spot Detector	[-1 2 -1]
R3	Ripple Detector	[*NOTE: cannot be reproduced in 3x3 form]

$$\begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} * [1, 2, 1] = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

E3L3 $\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$	E3S3 $\begin{pmatrix} 1 & 0 & -1 \\ -2 & 0 & 2 \\ 1 & 0 & -1 \end{pmatrix}$	L3S3 $\begin{pmatrix} -1 & -2 & -1 \\ 2 & 4 & 2 \\ -1 & -2 & -1 \end{pmatrix}$
E5L5 $\begin{pmatrix} -1 & -2 & 0 & 2 & 1 \\ -4 & -8 & 0 & 8 & 4 \\ -6 & -12 & 0 & 12 & 6 \\ -4 & -8 & 0 & 8 & 4 \\ -1 & -2 & 0 & 2 & 1 \end{pmatrix}$	E5S5 $\begin{pmatrix} 1 & 2 & 0 & -2 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ -2 & -4 & 0 & 4 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & -2 & -1 \end{pmatrix}$	L5S5 $\begin{pmatrix} -1 & -4 & -6 & -4 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 8 & 12 & 8 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & -4 & -6 & -4 & -1 \end{pmatrix}$

Figure 3.9 L3E3 kernel composition example

To create 2D masks, vectors L_n , E_n , S_n , and R_n (as shown above) are convolved together as separable pairs into kernels; a few examples are shown in Fig. 3.9.

Note that Laws texture metrics have been extended into 3D for volumetric texture analysis [43, 44].

LBP Local Binary Patterns

In contrast to the various structural and statistical methods of texture analysis, the LBP operator [18, 50] computes the local texture around each region as an LBP binary code, or *micro-texture*, allowing simple micro-texture comparisons to segment regions based on like micro-texture. (See the very detailed discussion on LBP in Chap. 6 for details and references to the literature, and especially Fig. 6.6.) The LBP operator [165] is quite versatile, easy to compute, consumes a low amount of memory, and can be used for texture analysis, interest points, and feature description. As a result, the LBP operator is discussed in several places in this book.

As shown in Fig. 3.10, the uniform set of LBP operators, composed of a subset of the possible LBPs that are by themselves rotation invariant, can be binned into a histogram, and the corresponding bin values are run through an FFT as a 1D array to create an FFT spectrum, which yields a robust metric with strong rotational invariance.

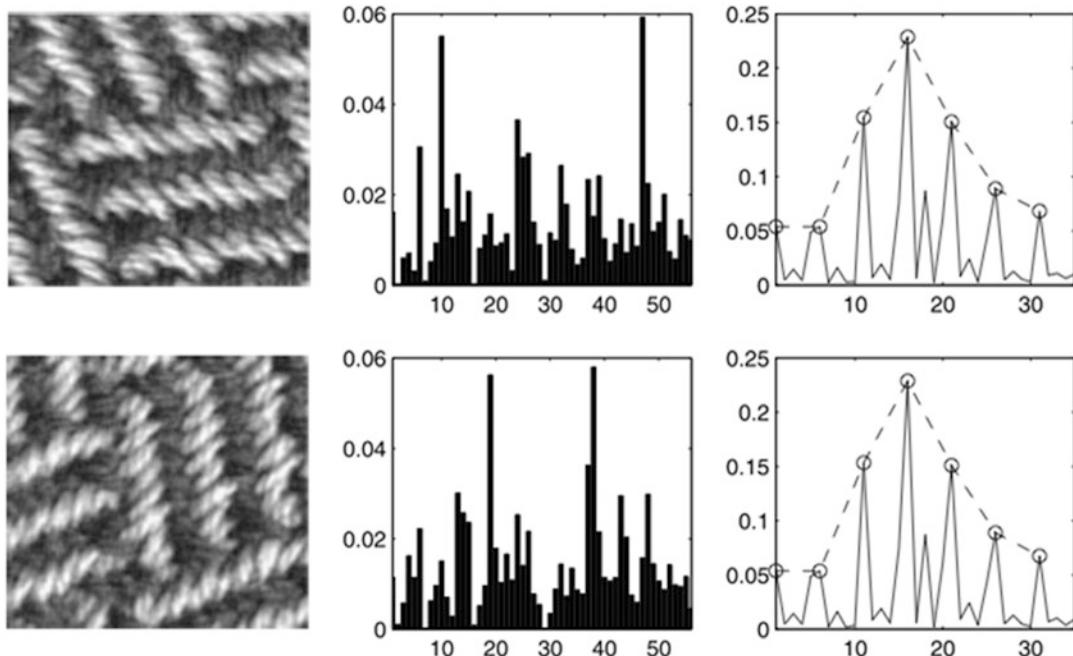


Figure 3.10 (Left) texture images. (Center) LBP histograms. (Right) FFT spectrum plots of the histograms which reveal the rotational invariance property of the LBP histograms. Note that while the histogram binning looks different for the rotated images, the FFT spectrums look almost identical. (Image © Springer-Verlag London Limited from Computer Vision Using Local Binary Patterns)

Dynamic Textures

Dynamic textures are a concept used to describe and track textured regions as they change and morph dynamically from frame to frame [13–15, 45]. For example, dynamic textures may be textures in motion, like sea waves, smoke, foliage blowing in the wind, fire, facial expressions, gestures, and poses. The changes are typically tracked in spatiotemporal sets of image frames, where the consecutive frames are stacked into volumes for analysis as a group. The three dimensions are the XY frame sizes, and the Z dimension is derived from the stack of consecutive frames $n - 2, n - 1, n$.

A close cousin to dynamic texture research is the field of *activity recognition* (discussed in Chap. 6), where features are parts of moving objects that compose an activity—for example, features on arms and legs that are tracked frame to frame to determine the type of motion or activity, such as walking or running. One similarity between activity recognition and dynamic textures is that the features or textures change from frame to frame over time, so for both activity recognition and dynamic texture analysis, tracking features and textures often requires a spatiotemporal approach involving a data structure with a history buffer of past and current frames, which provides a volumetric representation to the data.

For example, VLBP and LBP-TOP (discussed in Chap. 6) provide methods for dynamic texture analysis by using the LBP constructed to operate over three dimensions in a volumetric structure, where the volume contains image frames $n - 2, n - 1$, and n stacked into the volume.

Statistical Region Metrics

Describing texture in terms of statistical metrics of the pixels is a common and intuitive method. Often a simple histogram of a region will be sufficient to describe the texture well enough for many applications. There are also many variations of the histogram, which lend themselves to a wide range of texture analysis. So this is a good point at which to examine histogram methods. Since statistical mathematics is a vast field, we can only introduce the topic here, dividing the discussion into image moment features and point metric features.

Image Moment Features

Image moments [4, 500] are scalar quantities, analogous to the familiar statistical measures such as mean, variance, skew, and kurtosis. Moments are well suited to describe polygon shape features and general feature metric information such as gradient distributions. Image moments can be based on either scalar point values or basis functions such as Fourier or Zernike methods discussed later in the section on basis space.

Moments can describe the projection of a function onto a *basis space*—for example, the Fourier transform projects a function onto a basis of harmonic functions. Note that there is a conceptual relationship between 1D and 2D moments in the context of shape description. For example, the 1D mean corresponds to the 2D centroid, and the 1D minimum and maximum correspond to the 2D major and minor axis. The 1D minimum and maximum also correspond to the 2D bounding box around the 2D polygon shape (also see Fig. 6.29).

In this work, we classify image moments under the term *polygon shape descriptors* in the taxonomy (see Chap. 5). Details on several image moments used for 2D shape description are covered in Chap. 6, under “Object Shape Metrics for Blobs and Objects.”

Common properties of moments in the context of 1D distributions and 2D images include:

- Zeroth order moment is the mean or 2D centroid.
- Central moments describe variation around the mean or 2D centroid.
- First order central moments contain information about 2D area, centroid, and size.
- Second order central moments are related to variance and measure 2D elliptical shape.
- Third order central moments provide symmetry information about the 2D shape, or skewness.
- Fourth order central moments measure 2D distribution as tall, short, thin, short, or fat.
- Higher-level moments may be devised and composed of moment ratios, such as co-variance.

Moments can be used to create feature descriptors that are invariant to several robustness criteria, such as scale, rotation, and affine variations. The taxonomy of robustness and invariance criteria is provided in Chap. 5. For 2D shape description, in 1961 Hu developed a theoretical set of seven 2D planar moments for character recognition work, derived using invariant algebra, that are invariant under scale, translation, and rotation [7]. Several researchers have extended Hu’s work. An excellent resource for this topic is *Moments and Moment Invariants in Pattern Recognition*, by Jan Flusser et al. [500].

Point Metric Features

Point metrics can be used for the following: (1) feature description, (2) analysis and visualization, (3) thresholding and segmentation, and (4) image processing via programmable LUT functions (discussed in Chap. 2). Point metrics are often overlooked. Using point metrics to understand the structure of the image data is one of the first necessary steps toward devising the image preprocessing pipeline to prepare images for feature analysis. Again, the place to start is by analysis of the histogram, as shown in Figs. 3.1 and 3.11. The basic point metrics can be determined visually, such as minima, maxima, peaks, and valleys. False coloring of the histogram regions for data visualization is simple using color lookup tables to color the histogram regions in the images.

Here is a summary of statistical point metrics:

- **Quantiles, median, rescale:** By sorting the pixel values into an ordered list, as during the histogram process, the various quartiles can be found, including the median value. Also, the pixels can be rescaled from the list and used for pixel remap functions (as described in Chap. 2).

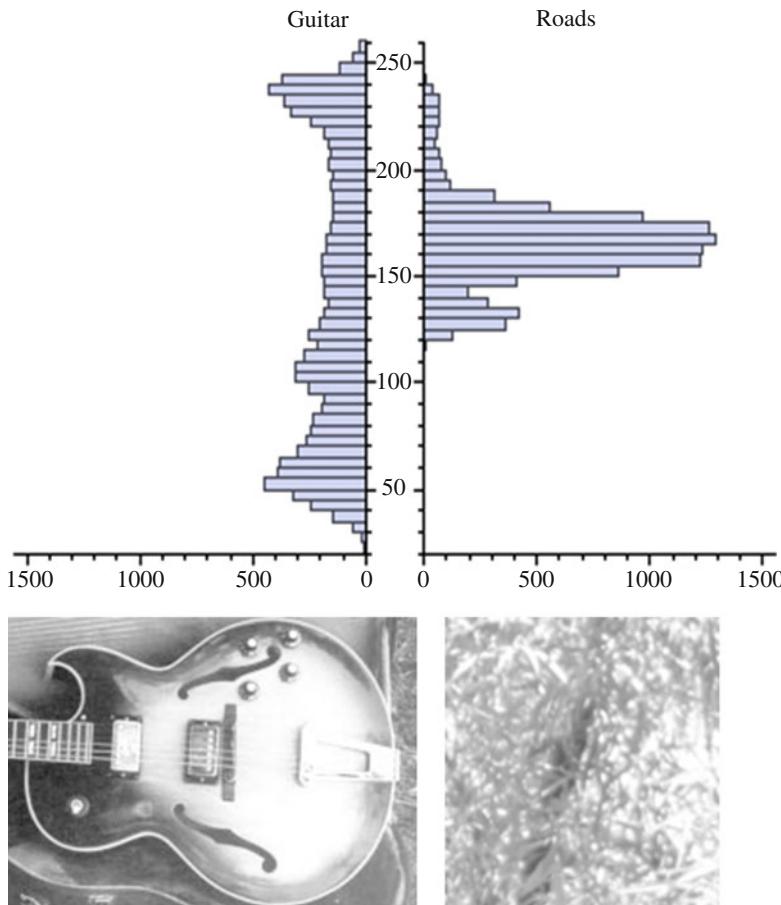


Figure 3.11 Two image histograms side by side, for analysis

- **Mix, max, mode:** The minimum and maximum values, together with histogram analysis, can be used to guide image preprocessing to devise a threshold method to remove outliers from the data. The mode is the most common pixel value in the sorted list of pixels.
- **Mean, harmonic mean, and geometric mean:** Various formulations of the mean are useful to learn the predominant illumination levels, dark or light, to guide image preprocessing to enhance the image for further analysis.
- **Standard deviation, skewness, and kurtosis:** These moments can be visualized by looking at the SDM plots.
- **Correlation:** Topic was covered earlier in this chapter under cross-correlation and auto-correlation.
- **Variance, covariance:** The variance metric provides information on pixel distribution, and covariance can be used to compare variance between two images. Variance can be visualized to a degree in the SDM, also as shown in this chapter.
- **Ratios and multivariate metrics:** Point metrics by themselves may be useful, but multivariate combinations or ratios using simple point metrics can be very useful as well. Depending on the application, the ratios themselves form key attributes of feature descriptors (as described in Chap. 6). For example, mean: min, mean: max, median: mean, area: perimeter.

Global Histograms

Global histograms treat the entire image. In many cases, image matching via global histograms is simple and effective, using a distance function such as SSD. As shown in Fig. 3.12, histograms reveal quantitative information on pixel intensity, but not structural information. All the pixels in the region

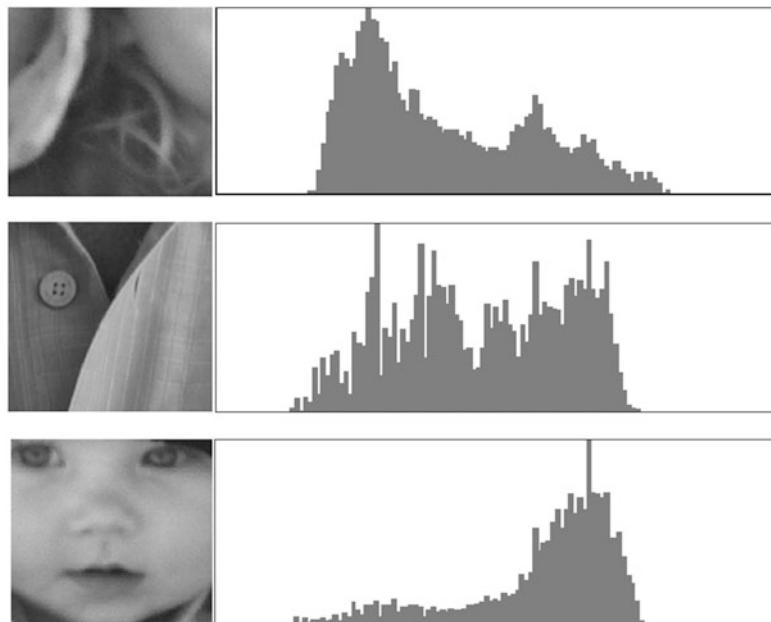


Figure 3.12 2D histogram shapes for different images

contribute to the histogram, with no respect to the distance from any specific point or feature. As discussed in Chap. 2, the histogram itself is the basis of histogram modification methods, allowing the shape of the histogram to be stretched, compressed, or clipped as needed, and then used as an inverse lookup table to rearrange the image pixel intensity levels.

Local Region Histograms

Histograms can also be computed over *local regions* of pixels, such as rectangles or polygons, as well as over sets of feature attributes, such as gradient direction and magnitude or other spectra. To create a polygon region histogram feature descriptor, first a region may be segmented using morphology to create a mask shape around a region of interest, and then only the masked pixels are used for the histogram.

Local histograms of pixel intensity values can be used as attributes of a feature descriptor, and also used as the basis for remapping pixel values from one histogram shape to another, as discussed in Chap. 2, by reshaping the histogram and reprocessing the image accordingly. Chapter 6 discusses a range of feature descriptors such as SIFT, SURF, and LBP which make use of feature histograms to bin attributes such as gradient magnitude and direction.

Scatter Diagrams, 3D Histograms

The *scatter diagram* can be used to visualize the relationship or similarity between two image datasets for image analysis, pattern recognition, and feature description. Pixel intensity from two images or image regions can be compared in the scatter plot to visualize how well the values correspond. Scatter diagrams can be used for feature and pattern matching under limited translation invariance, but they are less useful for affine, scale, or rotation invariance. Fig. 3.13 shows an example using a scatter diagram to look for a pattern in an image, the target pattern is compared at different offsets, the smaller the offset, the better the correspondence. In general, tighter sets of peak features indicate a strong structural or pattern correspondence; more spreading of the data indicates weaker correspondence. The farther away the pattern offset moves, the lower the correspondence.

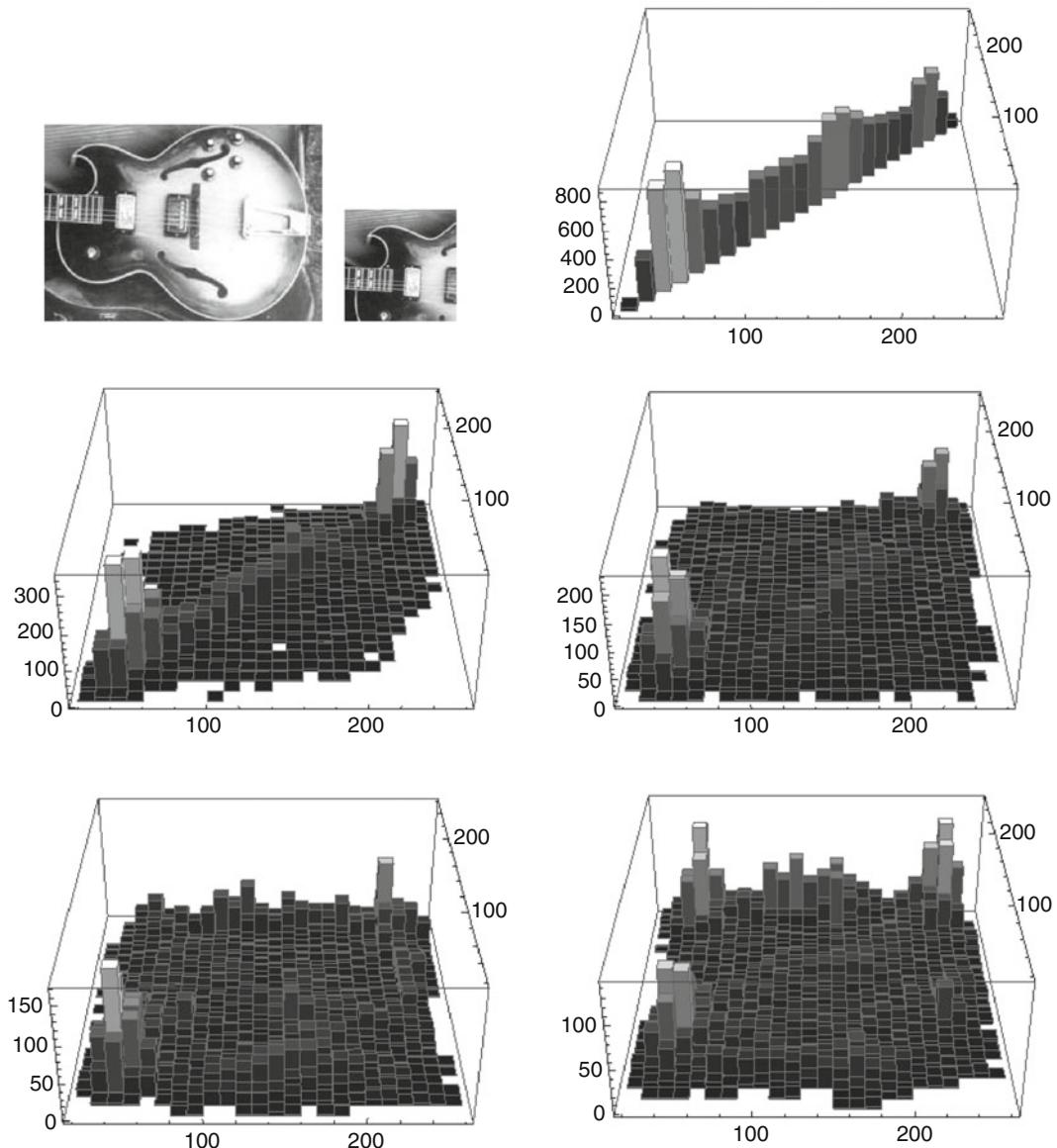


Figure 3.13 Scatter diagrams, rendered as 3D histograms, of an image and a target pattern at various displacements. Top row: (left) image, (center) target pattern from image, (right) SDM of pattern with itself. Center row: (left) target and image offset 1,1 (right) target and image offset 8,8. Bottom row: (left) target and image offset 16,16 (right) target and image offset 32,32

Note that by analyzing the peak features compared to the low-frequency features, correspondence can be visualized. Fig. 3.14 shows scatter diagrams from two separate images. The lack of peaks along the axis and the presence of spreading in the data show low structural or pattern correspondence.

The scatter plot can be made, pixel by pixel, from two images, where pixel pairs form the Cartesian coordinate for scatter plotting using the pixel intensity of image 1 is used as the x coordinate, and the pixel intensities of image 2 as the y coordinate, then the count of pixel pair correspondence is binned in the scatter plot. The bin count for each coordinate can be false colored for visualization. Fig. 3.15 provides some code for illustration purposes.

For feature detection, as shown in Fig. 3.12, the scatter plot may reveal enough correspondence at coarse translation steps to reduce the need for image pyramids in some feature detection and pattern matching applications. For example, the step size of the pattern search and compare could be optimized by striding or skipping pixels, searching the image at 8 or 16 pixel intervals, rather than at every pixel, reducing feature detection time. In addition, the scatter plot data could first be thresholded to a binary image, masked to show just the peak values, converted into a bit vector, and measured for correspondence using HAMMING distance for increased performance.

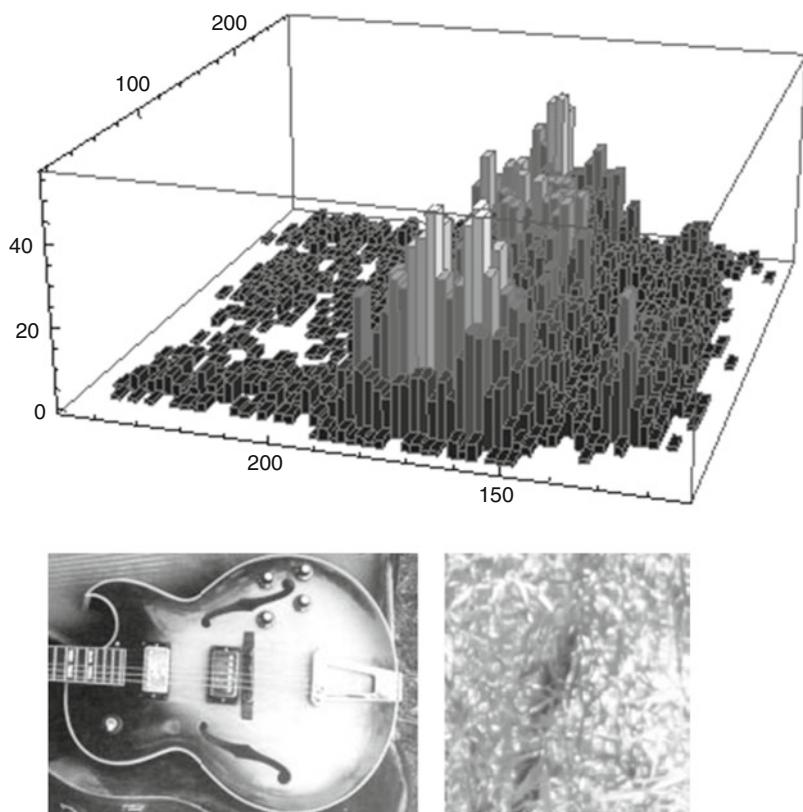


Figure 3.14 Scatter diagram from two different images showing low correspondence along diagonal

```

r1.x = sarea.x;
r1.y = sarea.y;
r1.z = sarea.z;
r1.dx = dx;
r1.dy = 1;
r1.dz = 1;

r2.x = darea.x;
r2.y = darea.y;
r2.z = darea.z;
r2.dx = dx;
r2.dy = 1;
r2.dz = 1;

/* INITIALIZE DATA */
for (x=0; x < 0x10000; mbin[x] = (int)0, x++);

gf = c->grain;
if (gf <= 0) gf = 1;
if (gf > dx) gf = dx;

z=0;
while (z < dz) {
    r1.y = sarea.y;
    r2.y = darea.y;
    y=0;
    while (y < dy) {
        pix_read(c->soid, &r1, data1);
        pix_read(c->doid, &r2, data2);
        for (x=0; x < dx; mbin[ ((data2[x] << 8)&0xff00) + (data1[x] & 0xff) ]++, x += gf);
        y += gf;
        r1.y += gf;
        r2.y += gf;
    }
    z += gf;
    r1.z += gf;
    r2.z += gf;
}

```

Figure 3.15 Code to illustrate binning 8-bit data for a scatter diagram comparing two images pixel by pixel and binning the results for plotting

Multi-resolution, Multi-scale Histograms

Multi-resolution histograms have been used for texture analysis [46], and also for feature recognition [47]. The PHOG descriptor described in Chap. 6 makes use of multi-scale histograms of feature spectra—in this case, gradient information. Note that the multi-resolution histogram provides scale invariance for feature description. For texture analysis [46], multi-resolution histograms are constructed using an image pyramid, and then a histogram is created for each pyramid level and concatenated together [10], which is referred to as a *multi-resolution histogram*. This histogram has the desirable properties of algorithm simplicity, fast computation, low memory requirements, noise tolerance, and high reliability across spatial and rotational variations. See Fig. 3.16. A variation on the pyramid is used in the method of Zhao and Pietikainen [15], employing a multidimensional pyramid image set from a volume.

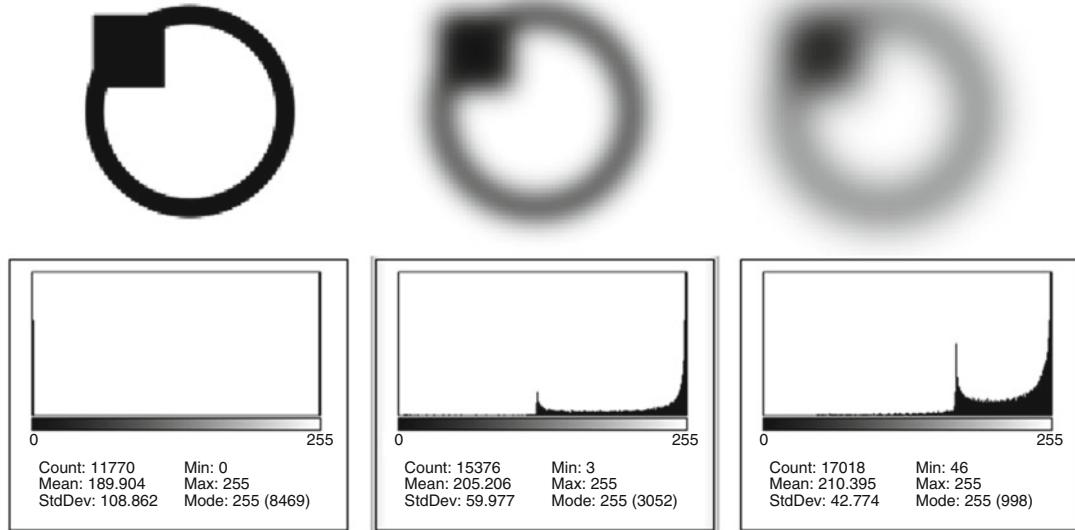


Figure 3.16 Multi-resolution histogram image sequence. Note that the multiple histograms are taken at various Gaussian blur levels in an attempt to create more invariant feature descriptors

Steps involved in creating and using multi-resolution histograms are as follows:

1. Apply Gaussian filter to image.
2. Create an image pyramid.
3. Create histograms at each level.
4. Normalize the histograms using L1 norm.
5. Create cumulative histograms.
6. Create difference histograms or DOG images (differences between pyramid levels).
7. Renormalize histograms using the difference histograms.
8. Create a feature vector from the set of difference histograms.
9. Use L1 norm as distance function for comparisons between histograms.

Radial Histograms

For some applications, computing the histogram using radial samples originating at the shape centroid can be valuable [128, 129]. To do this, a line is cast from the centroid to the perimeter of the shape, and pixel values are recorded along each line and then binned into histograms. See Fig. 3.17.

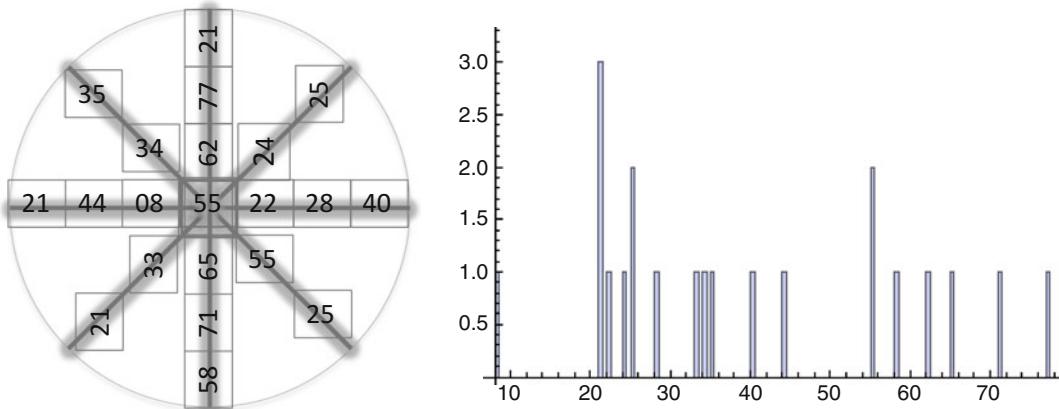


Figure 3.17 Radial histogram illustrations [128, 129]

Contour or Edge Histograms

The perimeter or shape of an object can be the basis of a shape histogram, which includes the pixel values of each point on the perimeter of the object binned into the histogram. Besides recording the actual pixel values along the perimeter, the chain code histogram (CCH) that is discussed in Chap. 6 shows the direction of the perimeter at connected edge point coordinates. Taken together, the CCH and contour histograms provide useful shape information.

Basis Space Metrics

Features can be described in a *basis space*, which involves transforming pixels into an alternative basis and describing features in the chosen basis, such as the frequency domain. What is a basis space and what is a transform? Consider the decimal system, which is base 10, and the binary system which is base 2. We can change numbers between the two number systems by using a transform. A Fourier transform uses sine and cosine as basis functions in frequency space, so that the Fourier transform can move pixels between the time-domain pixel space and the frequency space. Basis space moments describe the projection of a function onto a basis space [500]—for example, the Fourier transform projects a function onto a basis of harmonic functions.

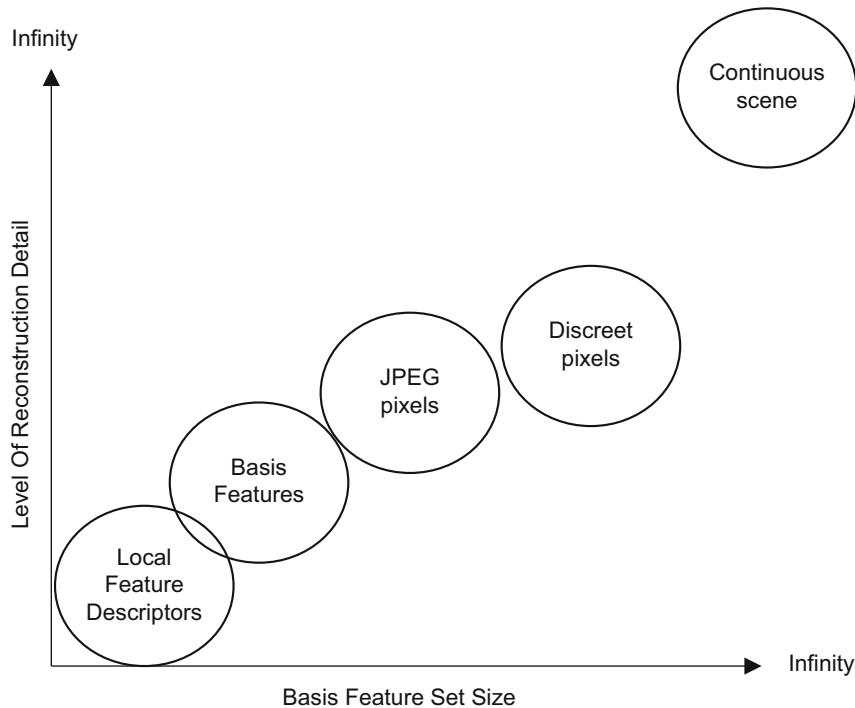


Figure 3.18 An oversimplified spectrum of basis space options, showing feature set size and complexity of description and reconstruction

Basis spaces and transforms are useful for a wide range of applications, including image coding and reconstruction, image processing, feature description, and feature matching. As shown in Fig. 3.18, image representation and image coding are closely related to feature description. Images can be described using *coding methods* or *feature descriptors*, and images also can be reconstructed from the encodings or from the feature descriptors. Many methods exist to reconstruct images from alternative basis space encodings, ranging from lossless RLE methods to lossy JPEG methods; in Chap. 4, we provide illustrations of images that have been reconstructed from only local feature descriptors (see Figs. 4.12, 4.13 and 4.14).

As illustrated in Fig. 3.18, a spectrum of basis spaces can be imagined, ranging from a continuous real function or live scene with infinite complexity, to a complete raster image, a JPEG compressed image, a frequency domain, or other basis representations, down to local feature descriptor sets. Note that the more detail that is provided and used from the basis space representation, the better the real scene can be recognized or reconstructed. So the trade-off is to find the best representation or description, in the optimal basis space, to reach the invariance and accuracy goals using the least amount of compute and memory.

Transforms and basis spaces are a vast field within mathematics and signal processing, covered quite well in other works, so here we only introduce common transforms useful for image coding and feature description. We describe their key advantages and applications, and refer the reader to the literature as we go. See Fig. 3.19.

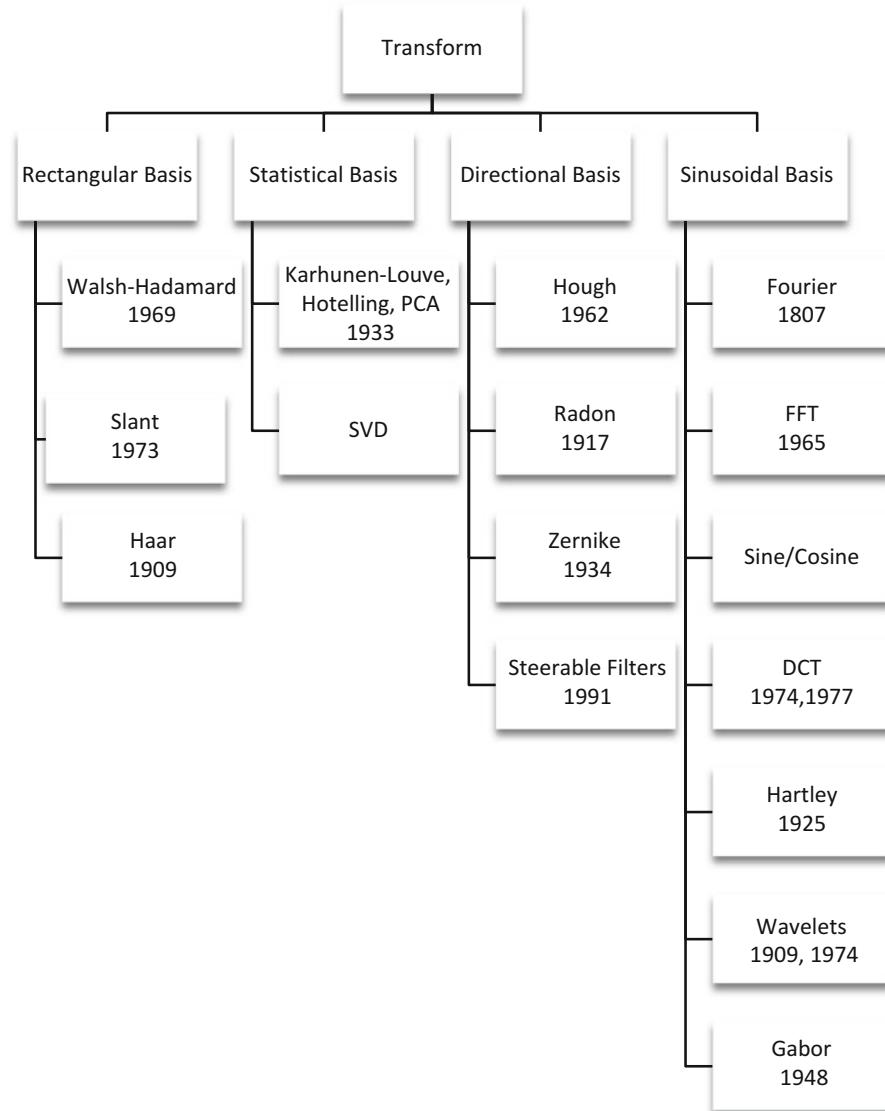


Figure 3.19 Various basis transforms used in image processing and computer vision

Since we are dealing with discrete pixels in computer vision, we are primarily interested in discrete transforms, especially those which can be accelerated with optimized software or fixed-function hardware. However, we also cover a few integral transform methods that may be slower to compute and less used. Here is an overview:

- **Global or local feature description.** It is possible to use transforms and basis space representations of images as a global feature descriptor, allowing scenes and larger objects to be recognized and compared. The 2D FFT spectrum is only one example, and it is simple to compare FFT spectrum features using SAD or SSD distance measures.
- **Image coding and compression.** Many of the transforms have proved valuable for image coding and image compression. The basic method involves transforming the image, or block regions of

the image, into another basis space. For example, transforming blocks of an image into the Fourier domain allows the image regions to be represented as sine and cosine waves. Then, based on the amount of energy in the region, a reduced amount of frequency space components can be stored or coded to represent the image. The energy is mostly contained in the lower-frequency components, which can be observed in the Fourier power spectrum such as shown in Fig. 2.16; the high-frequency components can be discarded and the significant lower-frequency components can be encoded, thus some image compression is achieved with a small loss of detail. Many novel image coding methods exist, such as that using a basis of scaled Laplacian features over an image pyramid [310].

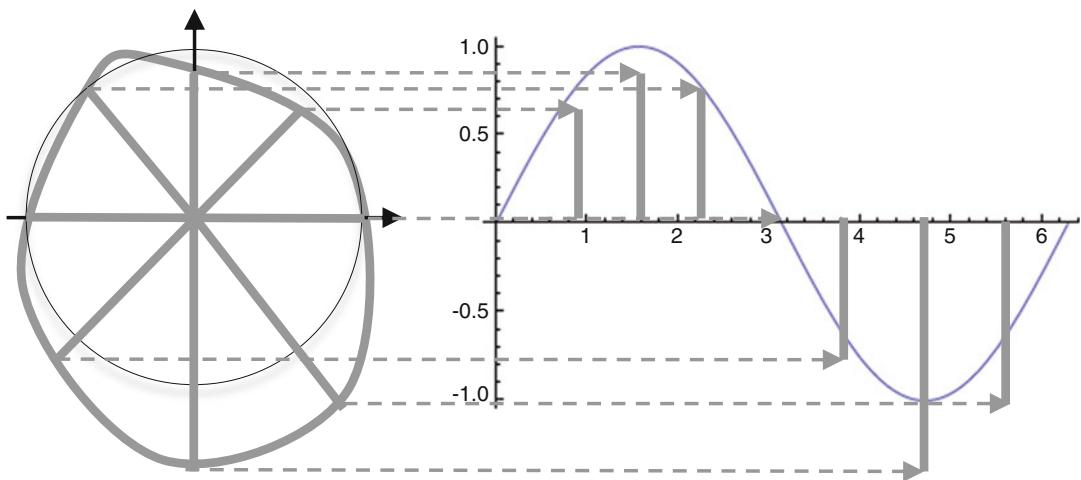


Figure 3.20 Fourier descriptor of the odd shaped polygon surrounding the circle on the *left*

Fourier Description

The Fourier family of transforms was covered in detail in Chap. 2, in the context of image preprocessing and filtering. However, the Fourier frequency components can also be used for feature description. Using the forward Fourier transform, an image is transformed into frequency components, which can be selectively used to describe the transformed pixel region, commonly done for image coding and compression, and for feature description.

The Fourier descriptor provides several invariance attributes, such as rotation and scale. Any array of values can be fed to an FFT to generate a descriptor—for example, a histogram. A common application is illustrated in Fig. 3.20, describing the circularity of a shape and finding the major and minor axis as the extrema frequency deviation from the sine wave. A related application is finding the endpoints of a flat line segment on the perimeter by fitting FFT magnitude's of the harmonic series as polar coordinates against a straight line in Cartesian space.

In Fig. 3.20, a complex wave is plotted as a dark gray circle unrolled around a sine wave function or a perfect circle. Note that the Fourier transform of the lengths of each point around the complex function yields an approximation of a periodic wave, and the Fourier descriptor of the shape of the complex wave is visible. Another example illustrating Fourier descriptors is shown in Fig. 6.29.

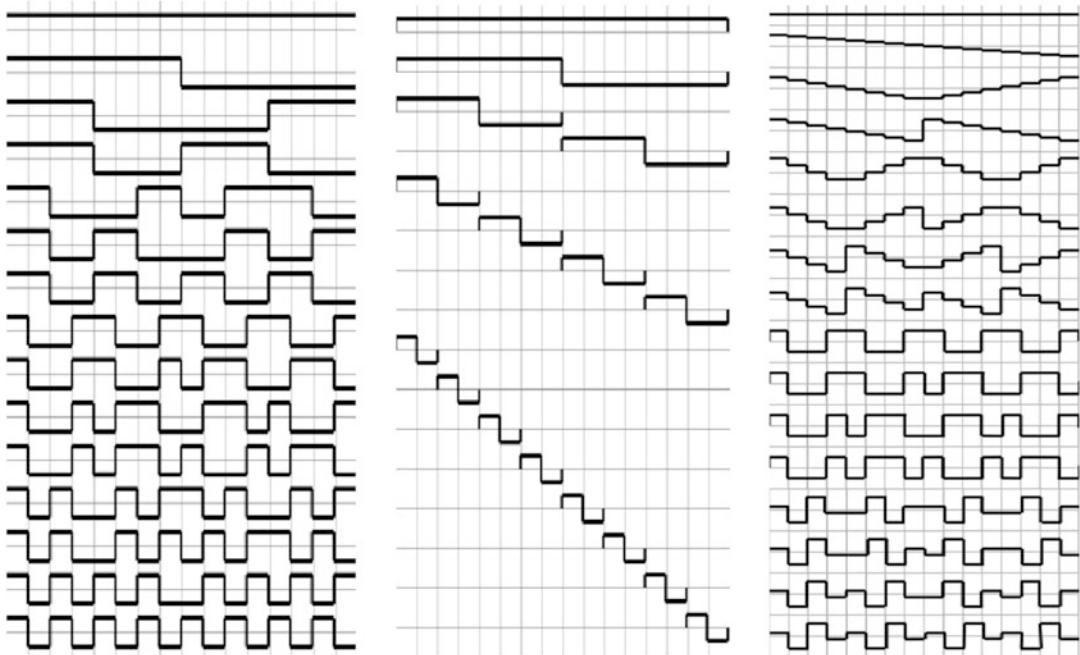


Figure 3.21 (Left) Walsh-Hadamard basis set. (Center) HAAR basis set. (Right) Slant basis set

Walsh–Hadamard Transform

The Hadamard transform [4, 9] uses a series of square waves with the value of +1 or -1, which is ideal for digital signal processing. It is amenable to optimizations, since only signed addition is needed to sum the basis vectors, making this transform much faster than sinusoidal basis transforms. The basis vectors for the harmonic Hadamard series and corresponding transform can be generated by sampling Walsh functions, which make up an orthonormal basis set; thus, the combined method is commonly referred to as the Walsh–Hadamard transform; see Fig. 3.21.

HAAR Transform

The HAAR transform [4, 9] is similar to the Fourier transform, except that the basis vectors are HAAR features resembling square waves, and similar to wavelets. HAAR features, owing to their orthogonal rectangular shapes, are suitable for detecting vertical and horizontal image features that have near-constant gray level. Any structural discontinuities in the data, such as edges and local texture, cannot be resolved very well by the HAAR features; see Figs. 3.21 and 6.21.

Slant Transform

The Slant transform [276], as illustrated in Fig. 3.21, was originally developed for television signal encoding, and was later applied to general image coding [4, 275]. The Slant transform is analogous to

the Fourier transform, except that the basis functions are a series of slant, sawtooth, or triangle waves. The slant basis vector is suitable for applications where image brightness changes linearly over the length of the function. The slant transform is amenable to discrete optimizations in digital systems. Although the primary applications have been image coding and image compression, the slant transform is amenable to feature description. It is closely related to the Karhunen–Loeve transform and the Slant–Hadamard transform [494].

Zernike Polynomials

Fritz Zernike, 1953 Nobel Prize winner, devised Zernike polynomials during his quest to develop the phase contrast microscope, while studying the optical properties and spectra of diffraction gratings. The Zernike polynomials [264–266] have been widely used for optical analysis and modeling of the human visual system, and for assistance in medical procedures such as laser surgery. They provide an accurate model of optical wave aberrations expressed as a set of basis polynomials, illustrated in Fig. 3.22.

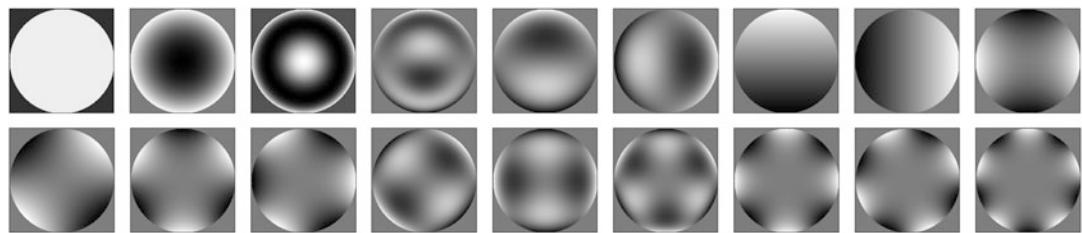


Figure 3.22 The first 18 Zernike modes. Note various aberrations from a perfect filter; *top left image* is the perfect filter. (Images © Dr. Thomas Salmon at Northeastern State University and used by permission)

Zernike polynomials are analogous to steerable filters [370], which also contain oriented basis sets of filter shapes used to identify oriented features and take moments to create descriptors. The Zernike model uses radial coordinates and circular regions, rather than rectangular patches as used in many other feature description methods.

Zernike methods are widely used in optometry to model human eye aberrations. Zernike moments are also used for image watermarking [270] and image coding and reconstruction [271, 273]. The Zernike features provide scale and rotational invariance, in part due to the radial coordinate symmetry and increasing level of detail possible within the higher-order polynomials. Zernike moments are used in computer vision applications by comparing the Zernike basis features against circular patches in target images [268, 269].

Fast methods to compute the Zernike polynomials and moments exist [267, 272, 274], which exploit the symmetry of the basis functions around the x and y axes to reduce computations, and also to exploit recursion.

Steerable Filters

Steerable filters are loosely considered as basis functions here, and can be used for both filtering or feature description. Conceptually similar to Zernike polynomials, steerable filters [370, 382] are composed by synthesizing steered or oriented linearly combinations of chosen basis functions, such as quadrature pairs of Gaussian filters and oriented versions of each function, in a simple transform.

Many types of filter functions can be used as the basis for steerable filters [371, 373]. The filter transform is created by combining together the basis functions in a filter bank, as shown in Fig. 3.23. Gain is selected for each function, and all filters in the bank are summed, then adaptively applied to the image. Pyramid sets of basis functions can be created to operate over scale. Applications include convolving oriented steerable filters with target image regions to determine filter response strength, orientation and phase. Other applications include filtering images based on orientation of features, contour detection, and feature description.

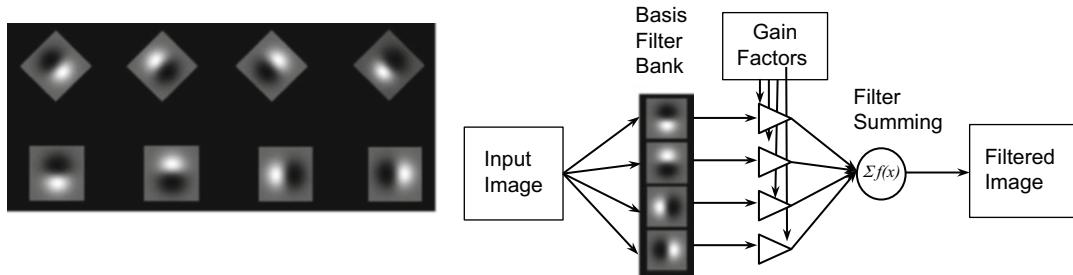


Figure 3.23 (Left) Steerable filters basis set showing eight orientations of the first-order Gaussian filter. (Right) How steerable filters can be combined for directional filtering. Filter images generated using ImageJ Fiji SteerableJ plugin from Design of Steerable Filters for Feature Detection Using Canny-Like Criteria, M. Jacob, M. Unser, PAMI 2004

For feature description, there are several methods that could work—for example, convolving each steerable basis function with an image patch. The highest one or two filter responses or moments from all the steerable filters can then be chosen as the set-ordinal feature descriptor, or all the filter responses can be used as a feature descriptor. As an optimization, an interest point can first be determined in the patch, and the orientation of the interest point can be used to select the one or two steerable filters closest to the orientation of the interest point; then the closest steerable filters are used as the basis to compute the descriptor.

Karhunen–Loeve Transform and Hotelling Transform

The Karhunen–Loeve transform (KLT)[4, 9] was devised to describe a continuous random process as a series expansion, as opposed to the Fourier method of describing periodic signals. Hotelling later devised a discrete equivalent of the KLT using principal components. “KLT” is the most common name referring to both methods.

The basis functions are dependent on the eigenvectors of the underlying image, and computing eigenvectors is a compute-intensive process with no established fast transform known. The KLT is not separable to optimize over image blocks, so the KLT is typically used for PCA on small datasets such as feature vectors used in pattern classification, clustering, and matching.

Wavelet Transform and Gabor Filters

Wavelets, as the name suggests, are short waves or wave-lets [326]. Think of a wavelet as a short-duration pulse such as a seismic tremor, starting and ending at zero, rather than a continuous or resonating wave. Wavelets are convolved with a given signal, such as an image, to find similarity and statistical moments. Wavelets can therefore be implemented like convolution kernels in the spatial domain. See Fig. 3.24.

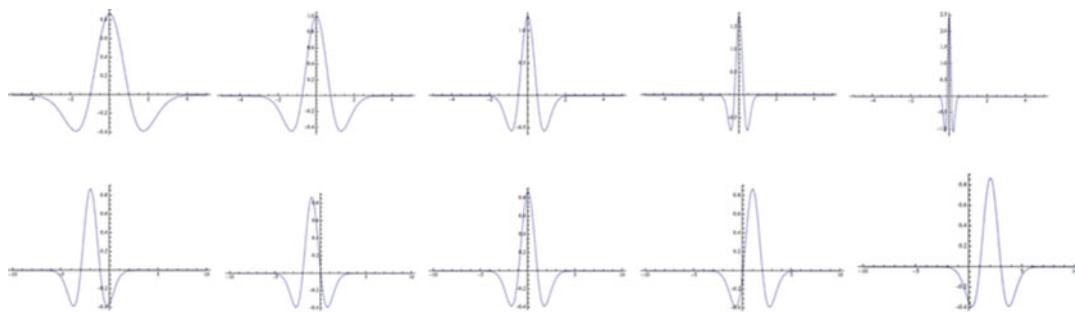


Figure 3.24 Wavelet concepts using a “Mexican top hat” wavelet basis. (*Top*) A few scaled Mexican top hats derived from the mother wavelet. (*Bottom*) A few translated wavelets

Wavelet analysis is a vast field [283, 284] with many applications and useful resources available, including libraries of wavelet families and analysis software packages [281]. Fast wavelet transforms (FWTs) exist in common signal and image processing libraries. Several variants of the wavelet transform include:

- Discrete wavelet transform (DWT)
- Stationary wavelet transform (SWT)
- Continuous wavelet transform (CWT)
- Lifting wavelet transform (LWT)
- Stationary wavelet packet transform (SWPT)
- Discrete wavelet packet transform (DWPT)
- Fractional Fourier transform (FRFT)
- Fractional wavelet transform (FRWT)

Wavelets are designed to meet various goals, and are crafted for specific applications; there is no single wavelet function or basis. For example, a set of wavelets can be designed to represent the musical scale, where each note (such as middle C) is defined as having a duration of an eighth note wavelet pulse, and then each wavelet in the set is convolved across a signal to locate the corresponding notes in the musical scale.

When designing wavelets, the mother wavelet is the basis of the wavelet family, and then daughter wavelets are derived using translation, scaling, or compression of the mother wavelet. Ideally, a set of wavelets are overlapping and complementary so as to decompose data with no gaps and be mathematically reversible.

Wavelets are used in transforms as a set of nonlinear basis functions, where each basis function can be designed as needed to optimally match a desired feature in the input function. So, unlike transforms which use a uniform set of basis functions—as the Fourier transform uses sine and cosine functions—wavelets use a dynamic set of basis functions that are complex and nonuniform in nature. See Fig. 3.25.

Wavelets have been used as the basis for scale and rotation invariant feature description [280], image segmentation [277, 278], shape description [279], and obviously image and signal filtering of all the expected varieties, denoising, image compression, and image coding. A set of application-specific wavelets could be devised for feature description.



Figure 3.25 Various 2D wavelet shapes: (*left to right*) Top hat, Shannon, Daubechies, Smylet, Coiflett

Gabor Functions

Wavelets can be considered an extension of the earlier concept of Gabor functions [285, 325], which can be derived for imaging applications as a set of 2D oriented bandpass filters. Gabor's work was centered on the physical transmission of sound and problems with Fourier methods involving time-varying signals like sirens that could not be perfectly represented as periodic frequency information. Gabor proposed a more compact representation than Fourier analysis could provide, using a concept called *atoms* that recorded coefficients of the sound that could be transmitted more compactly. See Fig. 3.26.

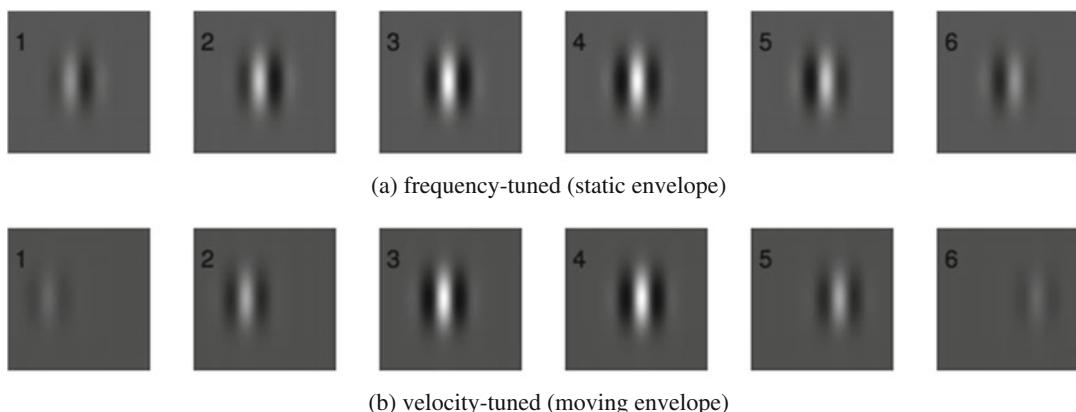


Figure 3.26 This figure showing Gabor filters (*top*) frequency tuned, and (*bottom*) velocity tuned. Images © - Springer-Verlag, taken from CVPR 2010, “Facial expression recognition using Gabor motion energy filters, Tingfan Wu, Bartlett, M.S. Movellan, Javier R.”

Hough Transform and Radon Transform

The Hough transform [220–222] and the Radon transform [291] are related, and the results are equivalent, in the opinion of many [287, 292]; see Fig. 3.27. The Radon transform is an integral transform, while the Hough transform is a discrete method, therefore much faster. The Hough method is widely used in image processing, and can be accelerated using a GPU [290] with data parallel methods. The Radon algorithm is slightly more accurate and perhaps more mathematically sound, and is often associated with X-ray tomography applied to reconstruction from X-ray projections. We focus primarily on the Hough transform, since it is widely available in image processing libraries.

Key applications for the Hough and Radon transforms are shape detection and shape description of lines, circles, and parametric curves. The main advantages include:

- Robust to noise and partial occlusion
- Fill gaps in apparent lines, edges, and curves
- Can be parameterized to handle various edge and curve shapes

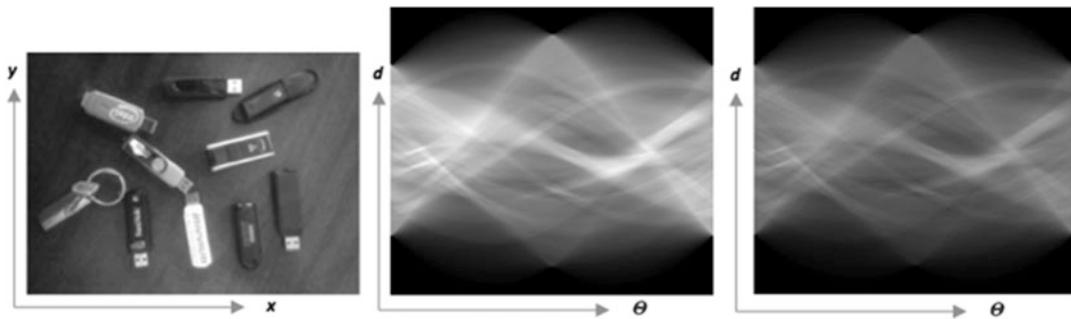


Figure 3.27 Line detection: (Left) Original image. (Center) Radon Transform. (Right) Hough Transform. The brightness of the transform images reveals the relative strength of the accumulators, and the sinusoidal line intersections indicate the angular orientation of features

The disadvantages include:

- Look for one type or parameterization of a feature at a time, such as a line
- Colinear segments are not distinguished and lumped together
- May incorrectly fill in gaps and link edges that are not connected
- Length and position of lines are not determined, but this can be done in image space

The Hough transform is primarily a global or regional descriptor and operates over larger areas. It was originally devised to detect lines, and has been subsequently generalized to detect parametric shapes [293], such as curves and circles. However, adding more parameterization to the feature requires more memory and compute. Hough features can be used to mark region boundaries described by regular parametric curves and lines. The Hough transform is attractive for some applications, since it can tolerate gaps in the lines or curves and is not strongly affected by noise or some occlusion, but morphology and edge detection via other methods is often sufficient, so the Hough transform has limited applications.

The input to the Hough transform is a gradient magnitude image, which has been thresholded, leaving the dominant gradient information. The gradient magnitude is used to build a map revealing all the parameterized features in the image—for example, lines at a given orientation or circles with a given diameter. For example, to detect lines, we map each gradient point in the pixel space into the Hough parameter space, parameterized as a single point (d, θ) corresponding to all lines with orientation angle θ at distance d from the origin. Curve and circle parameterization uses different variables [293]. The parameter space is quantized into cells or accumulator bins, and each accumulator is updated by summing the number of gradient lines passing through the same Hough points. The accumulator method is modified for detecting parametric curves and circles. Thresholding the accumulator space and reprojecting only the highest accumulator values as overlays back onto the image is useful to highlight features.

Summary

This chapter provides a selected history of global and regional metrics, with the treatment of local feature metrics deferred until Chaps. 4 and 6. Some historical context is provided on the development of structural and statistical texture metrics, as well as basis spaces useful for feature description, and several common regional and global metrics. A wide range of topics in texture analysis and statistical analysis are surveyed with applications to computer vision.

Since it is difficult to cleanly partition all the related topics in image processing and computer vision, there is some overlap of topics in here and in Chaps. 2, 4, 5, and 6.

Chapter 3: Learning Assignments

1. Discuss when to use a global image processing operation vs. a local or regional image processing operation.
2. Discuss in general how global image statistics can guide image preprocessing for computer vision applications, and specifically name one global image metric and discuss how it can be applied.
3. Compare global image feature metrics and local feature descriptors in general, and discuss a specific example global feature metric and compare it to a specific local feature descriptor.
4. Describe global image texture in general terms.
5. Discuss how a 2d histogram of an image can be used to understand image texture.
6. Discuss how the 2d Fourier Series of an image is used to understand image texture.
7. Discuss how the Haralick texture metrics based on the co-occurrence matrix are used to understand image texture.
8. Discuss how Spatial Dependency Matrix (SDM) plots are used to understand image texture.
9. Discuss statistical moments of an image histogram, including at least the mean value and variance, and how these features are useful as global image descriptors.
10. Describe a multi-resolution histogram built from an image pyramid, and how to interpret the results of the histogram.
11. Describe how a Fourier description of the shape of a circle is created from the Fourier Series, and how it is useful as a shape descriptor.
12. Describe basis features for the HAAR transform, Slant Transform, and Walsh–Hadamard Transform.
13. Compare Wavelet features to Fourier Series features.
14. Describe the Hough Transform and the Radon Transform algorithms, and how they are used as a global image metric for shape detection.

“Science, my boy, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth.”

—Jules Verne, Journey to the Center of the Earth

In this chapter we examine several concepts related to local feature descriptor design—namely local patterns, shapes, spectra, distance functions, classification, matching, and object recognition. The main focus is *local feature metrics*, as shown in Fig. 4.1. This discussion follows the general vision taxonomy that is presented in Chap. 5, and includes key fundamentals for understanding interest point detectors and feature descriptors, as surveyed in Chap. 6, including selected concepts common to both detector and descriptor methods. Note that the opportunity always exists to modify as well as mix and match detectors and descriptors to achieve the best results.

Local Features

We focus on the design of *local feature descriptors* and how they are used in training, classification, and machine learning. The discussion follows the feature taxonomy as is presented in Chap. 5 and as is illustrated in Fig. 5.1. The main elements are: (1) *shape* (for example, rectangle or circle); (2) *pattern* (either dense sampling or sparse sampling); and (3) *spectra* (binary values, scalars, sparse codes, or other values). A dense patterned feature will use each pixel in the local region, such as each pixel in a rectangle, while a sparse feature will use only selected pixels from the region.

In addition to the many approaches to shape and pattern, there are numerous approaches taken for the spectra, ranging from gradient-based patch methods to sparse local binary pattern methods. The main topics covered here include:

- **Detectors**, used to locate interesting features in the image.
- **Descriptors**, used to describe the regions surrounding interesting features.
- **Descriptor attributes**, such as feature robustness and invariance.
- **Classification**, used to create databases of features and optimal feature matching.
- **Recognition**, used to match detected features in target images against trained features.
- **Feature learning**, or machine learning methods.

Based on the concepts presented this chapter, the vision taxonomy offered in Chap. 5 provides a way to summarize and analyze the feature descriptors and their attributes, thereby enabling limited comparison between the different approaches.

Vision Pipeline Stages

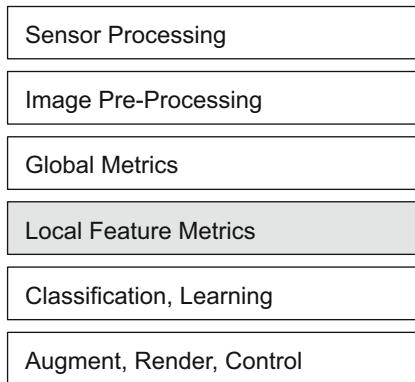


Figure 4.1 Various stages in the vision pipeline; this chapter focuses on local feature metrics and classification and learning

Detectors, Interest Points, Keypoints, Anchor Points, Landmarks

A *detector* finds interesting features in the image. The terminology in the literature for discussing an “interesting feature” includes several interchangeable terms, such as *keypoint*, *landmark*, *interest point*, or *anchor point*, all of which refer to features such as corners, edges, or patterns that can be found repeatedly with high likelihood. In Chap. 6, we survey many detector methods, along with various design approaches. In some cases, the keypoint detector is used to determine the orientation vector of the surrounding feature descriptor—for example, by computing the overall gradient orientation of the corner. The uncertain or low-quality keypoints are commonly filtered out prior to feature description. Note that many keypoint methods operate on smaller pixel regions, such as 3×3 for the LBP and 7×7 for FAST.

The keypoint location itself may not be enough for feature matching; however, some methods discussed here rely on *keypoints only*, without a feature descriptor. Feature description provides more information around each keypoint, and may be computed over larger regions and multiple scales, such as SIFT and ORB.

Descriptors, Feature Description, Feature Extraction

A feature *descriptor* can be computed at each key point to provide more information about the pixel region surrounding the keypoint. However, in methods that compute features across a fixed-size pixel grid such as the Viola–Jones method [138], no interest point is necessary, since the grid defines the descriptor region. Feature description typically uses some combination of color or gray scale intensity channels, as well as local information such as gradients and colors. Feature description takes place over a shape, such as a square or circle. In some cases, pixel point-pair sample patterns are used to compute or compare selected pixel values to yield a *descriptor vector*—for example, as shown later, in Fig. 4.8.

Typically, an interest point provides some amount of invariance and robustness—for example, in scale and rotation. In many cases, the orientation of the descriptor is determined from the interest point, and the descriptor provides other invariance attributes. Combining the interest point with the

descriptor provides a larger set of invariance attributes. And if several descriptors are associated together from the same object, object recognition is possible.

For example, a descriptor may contain multivariate, multidimensional, and multigeometric quantities calculated over several intensity channels, multiple geometric scales, and multiple perspectives (see Varma [771] and Vedaldi [770] for more on multivariate descriptors). A *multivariate* descriptor may contain RGBD data (red, green, blue, and Z depth data); a *multidimensional* descriptor may contain feature descriptions at various levels of zoom across an image pyramid; and a *multigeometry* descriptor may contain a set of feature descriptions computed across affine transforms of the local image patch or region.

There is no right or wrong method for designing features; many approaches are taken. For example, a set of metrics including region shape, region texture, and region color of an object may be helpful in an application to locate fruit, while another application may not need color or shape and can rely instead on sets of interest points, feature descriptors, and their spatial relationships. In fact, combining several weaker descriptor methods into a multivariate descriptor is often the best approach.

Computing feature descriptors from an image is commonly referred to as *feature extraction*.

Sparse Local Pattern Methods

While some methods describe features densely within regular sampling grids across an image, such as the PHOG [183] method discussed in Chap. 6, other methods such as FREAK [122] use *sparse local patterns* to sample pixels anchored at interest points to create the descriptor. Depending on the method, the shapes may be trained, learned, or chosen by design, and many topologies of shapes and patterns are in current use.

To frame the discussion on sparse local pattern and descriptor methods, notice that there is a contrast with global and regional descriptor methods, which typically do *not* rely on sparse local patterns. Instead, global and regional methods typically use dense sampling of larger shapes such as rectangles or other polygons. For example, polygon shape descriptors, as discussed in Chap. 6, may delineate or segment the feature region using dense methods such as mathematical morphology and region segmentation. Global and regional descriptor metrics, such as texture metrics, histograms, or SDMs discussed in Chap. 3, are typically computed across cohesive, dense regions rather than sparse regions.

Local Feature Attributes

This section discusses how features are chosen to provide the desired attributes of feature goodness, such as invariance and robustness.

Choosing Feature Descriptors and Interest Points

Both the interest point detector and the feature description method must be chosen to work well together, and to work well for the type of images being processed. Robustness attributes such as contrast, scale, and rotation must be considered for both the detector and the descriptor pair. As shown in Appendix A, each interest point detector is best designed to find specific types of features, and therefore no single method is good for all types of images.

For example, FAST and Harris methods typically find many small *mono-scale* interest points, while other methods, such as that used in SIFT find fewer, larger and finely tuned *multi-scale* interest points. Some tuning of the interest point detector parameters is expected, so as to make them work at all, or else some preprocessing of the images maybe needed to help the detector find the interest points in the first place. (Chapter 6 provides a survey of interest point methods and background mathematical concepts.)

Feature Descriptors and Feature Matching

Feature description is foundational to *feature matching*, leading to image understanding, scene analysis, and object tracking. The central problems in feature matching include how to determine if a feature is differentiated from other similar features, and if the feature is part of a larger object.

The method of determining a feature match is critical, for many reasons; these reasons include compute cost, memory size, repeatability, accuracy, and robustness. While a perfect match is ideal, in practice a relative match is determined by a *distance function*, where the incoming set of feature descriptors is compared with known feature descriptors. But we'll discuss several distance functions later in this chapter.

Table 4.1 Some Attributes for Good Feature Descriptors and Interest Points. (See also Figure 5-2 for the general robustness criteria.)

Good Feature Metric Attributes	Details
Scale invariance	Should be able to find the feature at different scales
Perspective invariance	Should be able to find the feature from different perspectives in the field of view
Rotational invariance	The feature should be recognized in various rotations within the image plane
Translation invariance	The feature should be recognized in various positions in the FOV
Reflection invariance	The feature should be recognized as a mirror image of itself
Affine invariance	The feature should be recognized under affine transforms
Noise invariance	The feature should be detectable in the presence of noise
Illumination invariance	The feature should be recognizable in various lighting conditions including changes in brightness and contrast
Compute efficiency	The feature descriptor should be efficient to compute and match
Distinctiveness	The feature should be distinct and detectable, with a low probability of mis-match, amenable to matching from a database of features
Compact to describe	The feature should not require large amounts of memory to hold details
Occlusion robustness	The feature or set of features can be described and detected when parts of the feature or feature set are occluded
Focus or blur robustness	The feature or set of features can be detected at varying degrees of focus (i.e, image pyramids can provide some of this capability)
Clutter and outlier robustness	The feature or set of features can be detected in the presence of outlier features and clutter

Criteria for Goodness

Measuring the goodness of features can be done *one attribute at a time*. A general list of goodness attributes for feature landmarks is provided in Table 4.1. Note that this list is primarily about invariance and robustness: these are the key concepts, since performance can be tuned using various optimization methods, as discussed in Chap. 8. Of course, in a given application some attributes of goodness are more important than others; this is discussed in Chap. 7, in connection with ground truth data.

How do we know a feature is *good* for an application? We may divide the discussion between the interest point methods and the descriptor method, and the combined robustness and invariance attributes provided by both as shown in Table 4.1. The interest point at which the feature is anchored is critical, since if the anchor is not good and cannot be easily and repeatedly found, the resulting descriptor is calculated at a suboptimal location.

Note that in many cases, image preprocessing is key to creating a good feature as shown in Fig. 4.2. If the image data has problems that can be corrected or improved, the feature description should be done after the image preprocessing. Note that many feature description methods rely on local image enhancements during descriptor creation, such as Gaussian blur of regions around keypoints for noise removal, so image preprocessing should complement the descriptor method. Each preprocessing method has drawbacks and advantages; see Table 2.1 and Chap. 2 for information on image preprocessing.

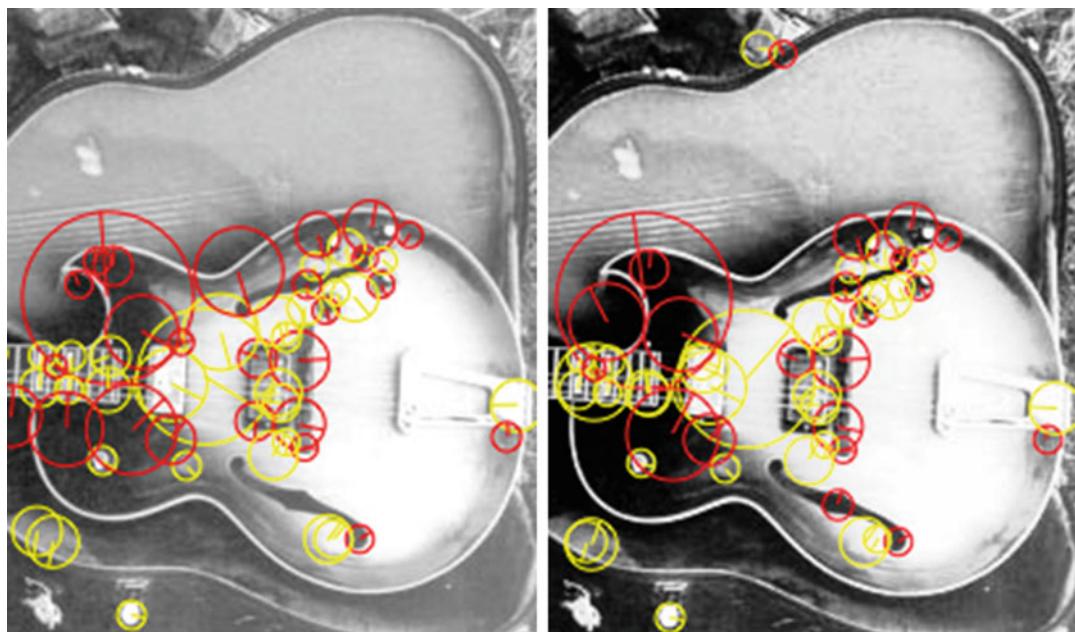


Figure 4.2 (Left) SURF feature descriptors calculated over original image. (Right) Image has been preprocessed using histogram equalization prior to feature extraction and therefore a different but overlapping set of features is found

Repeatability, Easy vs. Hard to Find

Ideally the feature will be easy to find in an image, meaning that the feature description contains sufficient information to be robust under various conditions as shown in Table 4.1, such as contrast and brightness variations, scale, and rotation. *Repeatability* applies particularly to interest point detection, so the choice of interest point detector method is critical. (Appendix A contains example images showing interesting nonrepeatability anomalies for several common interest point detectors.)

Some descriptors, such as SIFT [153, 170], are known to be robust under many imaging conditions. This is not too surprising, since SIFT is designed to be discriminating over multiple dimensions, such as scale and rotation, using carefully composed sets of local region gradients with a weighting factor applied to increase the importance of gradients closer to the center of the feature. But the robustness and repeatability come at a compute cost. SIFT [153, 170] is one of the most computationally expensive methods; however, Chap. 6 surveys various SIFT optimizations and variations.

Distinctive vs. Indistinctive

A descriptor is distinctive if:

- The feature can be differentiated from other, similar feature regions of the image.
- Different feature vectors are unique in the feature set.
- The feature can be matched effectively using a suitable distance function.

A feature is indistinct if similar features cannot be distinguished; this may be caused by a lack of suitable image preprocessing, insufficient information in the descriptor, or an unsuitable distance function chosen for the matching stage. Of course, adding information into a simpler descriptor to make the descriptor a hybrid multivariate or multi-scale descriptor may be all that is needed to improve distinctiveness. For example, color information can be added to distinguish between skin tones.

Relative and Absolute Position

Positional information, such as coordinates, can be critical for feature goodness. For example, to associate features together using constraints on the corner position of human eyes, interest point coordinates are needed. These enable more accurate identification and location of the eyes by using, as part of an intelligent matching process, the distance and angles between the eye corner locations.

With the increasing use of depth sensors, simply providing the Z or depth location of the feature in the descriptor itself may be enough to easily distinguish a feature from the background. Position in the depth field is a powerful bit of information, and since computer vision is often concerned with finding 3D information in a 2D image field, the Z depth information can be an invaluable attribute for feature goodness.

Matching Cost and Correspondence

Feature matching is a measurement of the correspondence between two or more features using a *distance function* (discussed next in this section). Note here that feature matching has a cost in terms

of memory and compute time. For example, if a feature descriptor is composed of an array of 8-bit bytes, such as an 18×18 pixel correlation template, then the feature matching cost is the compute time and memory required to compare two 18×18 (324) pixel regions against each other, where the matching method or distance function used may be SAD, SSD, or similar difference metric. Another example involves local binary descriptors such as the LBP (linear binary patterns), which are stored as bit vectors, where the matching cost is the time to perform the Hamming distance function, which operates by comparing two binary vectors via Boolean XOR followed by a bit count to provide the match metric.

In general, distance functions are well-known mathematical functions that are applied to computer vision; however, some are better suited than others in terms of computability and application to a specific vision task. For example, SSD, SAD, cosine distance, and Hamming distance metrics have been implemented in silicon as computer machine language instructions in some architectures, owing to their wide applicability. So choosing a distance function that is accelerated in silicon can be an advantage.

The feature database is another component of the matching cost, so the organization of the database and feature search contribute to the cost. We briefly touch on this topic later in this chapter.

Distance Functions

This section provides a general discussion of distance functions used for clustering, classification, and feature matching. Often the appropriate distance function for an application is unknown, therefore several distance functions should be tried to find the best one, or a new one should be devised. For example, a distance function can be augmented to selectively compare distance only for non-zero datums (intersection), or where one datum is zero and the other is not (outliers), or only for datums which exceed a threshold. Be creative. Note that distance functions can be taken over several dimensions—for example, 2D image arrays for feature descriptor matching, 3D voxel volumes for point cloud matching, and multidimensional spaces for multivariate descriptors. Since this is a brief overview, a deeper treatment is available by Pele [530], Varma [771], Vedaldi [770], Cha [884], Duda [824], and Deza [885].

Note that kernel machines [353, 354], discussed later in this chapter, and in more detail in Chap. 10 in the section Kernel Functions, Kernel Machines, SVM, provide an automated framework to classify a feature space and substitute chosen distance function kernels.

Early Work on Distance Functions

In 1968, Rosenfeld and Pfaltz [113] developed novel methods for determining the distance between image features, which they referred to as “a given subset of the picture,” where the feature shapes used in their work included diamonds, squares, and triangles. The distance metrics they studied include some methods that are no longer in common use today:

- Hexagonal distance from a single point (Cartesian array)
- Hexagonal distance from a single point (staggered array)
- Octagonal distance from a single point
- City block distance from blank areas
- Square distances from blank areas
- Hexagonal distance from blank areas

- Octagonal distance from blank areas
- Nearest integer to Euclidean distance from a single point

This early work by Rosenfeld and Pfaltz is fascinating, since the output device used to render the images was ASCII characters printed on a CRT terminal or line printer, as shown in Fig. 4.3.

```

1.9.7.5.3.1.9.7.5555555555555555555555555555555.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.....5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3333333333333333333333333333.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.....3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.111111111111111111.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.1111111111111111111111.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.9999999999999999999.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.....9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.77777777777777777.9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.....7.9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.555555555555.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.5.....5.7.9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.5.33333333.5.7.9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.5.3.....3.5.7.9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.5.3.1.111111111111111111111111.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.5.3.1.111111111111111111111111111111.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.5.3.1.9.7.5.33333333.5.7.9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.5.3.....5.7.9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.....5.7.9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.....7.9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.....9.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.999999999999999999.1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.....1.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.11111111111111111111111111.3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.....3.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.5.3333333333333333333333.5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.5.....5.7.9.1.3.5.7.9.1
1.9.7.5.3.1.9.7.5.3.1.9.7.5.333333333333333333333333.7.9.1.3.5.7.9.1

```

"Square" distances (d_2) from a single point.

Figure 4.3 An early Rosenfeld and Pfaltz rendering that illustrates a distance function (square distance in this case) using a line printer as the output device. (Image © reprinted from Rosenfeld and Pfaltz, Pattern Recognition (Oxford: Pergamon Press, 1968), 1:33–61. Used with permission from Elsevier)

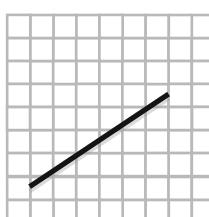
Euclidean or Cartesian Distance Metrics

The Euclidean distance metrics include basic Euclidean geometry identities in Cartesian coordinate spaces; in general, these are simple and obvious to use.

Euclidean Distance

This is the simple distance between two points.

$$\text{Euclidean Distance}[\{a, b\}, \{x, y\}] = \sqrt{(a - x)^2 + (b - y)^2}$$



Squared Euclidean Distance

This is faster to compute, and omits the square root.

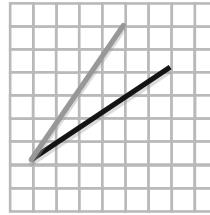
$$\text{Squared Euclidean Distance}[\{a, b\}, \{x, y\}] = (a - x)^2 + (b - y)^2$$

Cosine Distance or Similarity

This is angular distance, or the normalized dot product between two vectors to yield similarity of vector angle; also useful for 3D surface normal and viewpoint matching.

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$

$$\text{Cosine Distance}[\{a, b\}, \{x, y\}] = 1 - \frac{ax + by}{\sqrt{a^2 + b^2} \sqrt{x^2 + y^2}}$$



Sum of Absolute Differences (SAD) or L1 Norm

The difference between vector elements is summed and taken as the total distance between the vectors. Note that SAD is equivalent to Manhattan distance.

$$\text{SAD}(d_1, d_2) = \sum_{i=0}^{n1} \sum_{j=0}^{n2} (d_1[i, j] - d_2[i, j])$$

Sum of Squared Differences (SSD) or L2 Norm

The difference between vector elements is summed and squared and taken as the total distance between the vectors; commonly used in video decoding for motion estimation.

$$\text{SSD}(d_1, d_2) = \sum_{i=0}^{n1} \sum_{j=0}^{n2} (d_1[i, j] - d_2[i, j])^2$$

Correlation Distance

This is the correlation difference coefficient between two vectors, similar to cosine distance.

$$C[u, v] = \frac{1 - (u - \text{Mean}[u]) \cdot (v - \text{Mean}[v])}{\|u - \text{Mean}[u]\| \|v - \text{Mean}[v]\|}$$

$$C[\{a, b\}, \{x, y\}] = \frac{(a + \frac{1}{2}(-a-b))(x + \frac{1}{2}(-x-y)) + (\frac{1}{2}(-a-b)+b)(\frac{1}{2}(-x-y)+y)}{\sqrt{\text{Abs}[a + \frac{1}{2}(-a-b)]^2 + \text{Abs}[\frac{1}{2}(-a-b)+b]^2} \sqrt{\text{Abs}[x + \frac{1}{2}(-x-y)]^2 \text{Abs}[\frac{1}{2}(-x-y)+y]^2}}$$

Hellinger Distance

An effective alternative to Euclidean distance, Hellinger distance sometimes yields better accuracy for histogram-type distance metrics, as reported in the ROOTSIFT [166] optimization of SIFT. Hellinger distance, which can be formulated in a few different forms, is defined for L1 normalized histogram vectors as:

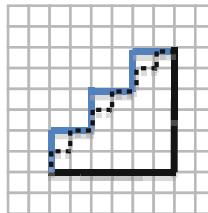
$$H(x, y) = \sum_{i=1}^n (\sqrt{x_i} - \sqrt{y_i})^2$$

Grid Distance Metrics

These metrics calculate distance analogous to paths on grids. Therefore the distance is measured as grid steps.

Manhattan Distance

Also known as city block difference or rectilinear distance, this measures distance via the route along a grid; there may be more than one path along a grid with equal distance.

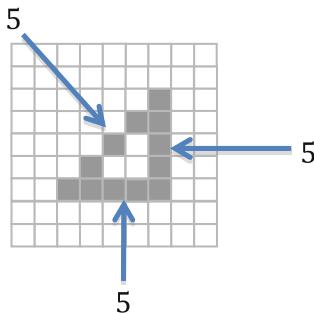


$$\text{Manhattan Distance}[\{a, b\}, \{x, y\}] = \text{Abs}(a - x) + \text{Abs}(b - y)$$

Chebyshev Distance

Also known as chessboard difference, this measures the greatest difference along a grid between two vectors. Note in the illustration below that each side of the triangle would have a Chebyshev distance, or length of 5, but in Euclidean space, one of the lines, the hypotenuse, is longer than the others.

$$\text{Chebyshev Distance}[\{a, b\}, \{x, y\}] = \text{Max}[\text{Abs}(a - x), \text{Abs}(b - y)]$$



Statistical Difference Metrics

These metrics are based on statistical features of the vectors, and therefore the distance metrics need not map into a Euclidean space.

Earth Movers Distance (EMD) or Wasserstein Metric

Earth movers distance measures the cost to transform a multidimensional vector, such as a histogram, into another vector. The analogy is an earth mover (bulldozer) moving dirt between two groups of piles to make the piles of dirt in each group the same size. The EMD assumes there is a ground distance between the features in the vector—for example, the distance between bins in a histogram. The EMD is computed to be the minimal cost for the transform, which integrates the distance moved $d \times$ the amount moved f , subject to a few constraints.

$$\text{COST}(P, Q, F) = \sum_{i=1}^m \sum_{j=1}^n d_{ij} f_{ij}$$

Once the cost is computed, the result is normalized.

$$\text{EMD}(P, Q) = \frac{\sum_{i=1}^m \sum_{j=1}^n d_{ij} f_{ij}}{\sum_{i=1}^m \sum_{j=1}^n f_{ij}}$$

The EMD has a high compute cost and can be useful for image analysis, but EMD is not an efficient metric for feature matching.

Mahalanobis Distance

Also known as quadratic distance, this computes distance using mean and covariance; it is scale invariant.

$$d_{ij} = \left((\bar{x}_i - \bar{x}_j)^T S^{-1} (\bar{x}_i - \bar{x}_j) \right)^{\frac{1}{2}}$$

$$\text{SSD}(d_1, d_2) = \sum_{i=-n1}^{n1} \sum_{j=-n2}^{n2} f \left((x + i, y + j) - g(x + i - d_1, y_j - d_2) \right)^2$$

where \bar{x}_i = mean of feature vector 1, and \bar{x}_j = mean of feature vector 2.

Bray Curtis Distance

This is equivalent to a ratio of the sums of absolute differences and sums, such as a ratio of norms of Manhattan distances. Bray Curtis dissimilarity is sometimes used for clustering data.

$$\text{BrayCurtisDistance}[\{a, b, c\}, \{x, y, z\}] = \frac{\text{Abs}(a - x) + \text{Abs}(b - y) + \text{Abs}(c - z)}{\text{Abs}(a + x) + \text{Abs}(b + y) + \text{Abs}(c + z)}$$

Canberra Distance

This measures the distance between two vectors of equal length:

$$\text{CanberraDistance}[\{a, b\}, \{x, y\}] = \frac{\text{Abs}(a - x)}{\text{Abs}(a) + \text{Abs}(x)} + \frac{\text{Abs}(b - y)}{\text{Abs}(b) + \text{Abs}(y)}$$

Binary or Boolean Distance Metrics

These metrics rely on set comparisons and Boolean algebra concepts, which makes this family of metrics attractive for optimization on digital computers.

L0 Norm

The L0 norm is a count of non-zero elements in a vector and is used in the Hamming Distance metric and other binary or Boolean metrics.

$$\|x_0\| = \#\{i | x_i \neq 0\}$$

Hamming Distance

This measures the binary difference or agreement between vectors of equal length—for example, string or binary vectors. Hamming distance for binary bit vectors can be efficiently implemented in digital computers with either complete machine language instructions or as an XOR operation followed by a bit count operation. Hamming distance is a favorite for matching local binary descriptors, such as LBP, FREAK, CENSUS, BRISK, BRIEF, and ORB.

String distance: 5 = 0001100111 = compare “HelloThere” and “HelpsThing”

Binary distance: 3 = 10100010 = **(01001110)** XOR **(11001100)**

Bit count of (u XOR v)

Jaccard Similarity and Dissimilarity

The ratio of pairwise similarity of a binary set (0,1 or true, false) over the number of set elements. Set 1 below contains two bits with the same pairwise value as Set 2, so the similarity is 2/5 and the dissimilarity is 3/5. Jaccard similarity can be combined with Hamming distance.

Set 1: {1,0,1,1,0}

Set 2: {1,1,0,1,1}

Jaccard Similarity: 2 / 5 = .4

Jaccard Dissimilarity: 3 / 5 = .6

Descriptor Representation

This section discusses how information is represented in the descriptors, including coordinates spaces useful for feature description and matching, with some discussion of multimodal data and feature pyramids.

Coordinate Spaces, Complex Spaces

There are many coordinate systems used in computer vision, so being able to transform data between coordinate systems is valuable. Coordinate spaces are analogous to basis spaces. Often, choosing the right coordinate system provides advantages for feature representation, computation, or matching. Complex spaces may include multivariate collections of scalar and vector variables, such as gradients, color, binary patterns, and statistical moments of pixel regions. See Fig. 4.4.

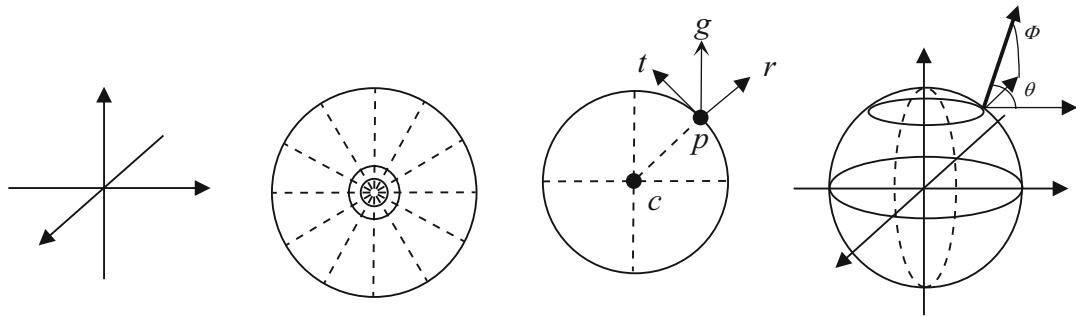


Figure 4.4 Coordinate spaces, Cartesian, polar, radial, and spherical

Cartesian Coordinates

Images are typically captured in the time domain in a Cartesian space, and for many applications translating to other coordinate spaces is needed. The human visual system views the world as a complex 3D spherical coordinate space, and humans can, through a small miracle, map the 3D space into approximate or relative Cartesian coordinates. Computer imaging systems capture data and convert it to Cartesian coordinates, but depth perception and geometric accuracy are lost in the conversion. (Chapter 1 provided a discussion of depth-sensing methods and 3D imaging systems, including geometric considerations.)

Polar and Log Polar Coordinates

Many descriptors mentioned later in Chap. 6 use a circular descriptor region to match the human visual system. Therefore, polar coordinates are logical candidates to bin the feature vectors. For example, the GLOH [136] method uses polar coordinates for histogram gradient binning, rather than Cartesian coordinates as used in the original SIFT [153] method. GLOH can be used as a retrofit for SIFT and has proved to increase accuracy [136]. Since the circular sampling patterns tend to provide better rotational invariance, polar coordinates and circular sampling are a good match for descriptor design.

Radial Coordinates

The RIFF descriptor (described later in Chap. 6) uses a local radial coordinate system to describe rotationally invariant gradient-based feature descriptors. The radial coordinate system is based on a radial gradient transform (RGT) that normalizes vectors for invariant binning.

As shown in Figs. 4.4 and 6.27, the RGT creates a local coordinate system within a patch region c , and establishes two orthogonal basis vectors (r, t) relative to any point p in the patch, r for the radial vector, and t for the tangential vector. The measured gradients g at all points p are projected onto the radial coordinate system (r, t) , so that the gradients are represented in a locally invariant fashion relative to the interest point c at the center of the patch. When the patch is rotated about c , the gradients rotate also, and the invariant representation holds.

Spherical Coordinates

Spherical coordinates, also referred to as 3D polar coordinates, can be applied to the field of 3D imaging and depth sensing to increase the accuracy for description and analysis. For example, depth cameras today typically only provide (x,y) , and Z depth information for each sample. However, this is woefully inadequate to describe the complex geometry of space, including warping, radial distortion and nonlinear distance between samples. Chapter 1 discussed the complexities of 3D space, depth measurements, and coordinate systems.

Gauge Coordinates

The G-SURF methods [180] use a differential geometry concept [182] of a local region Gauge coordinate system to compute the features. Gauge coordinates are local to the image feature, and they carry advantages for geometrical accuracy. Gauge derivatives are rotation and translation invariant.

Multivariate Spaces, Multimodal Data

Multivariate spaces combine several quantities, such as Tensor spaces which combine scalar and vector values, and are commonly used in computer vision. While raw image data may be scalar values only, many feature descriptors compute local gradients at each pixel, so the combination of pixel scalar value and gradient vector forms a tensor or multivariate space. For example, color spaces (see Chap. 2) may represent color as a set of scalar and vector quantities, such as the hue, saturation, and value (HSV) color space illustrated in Fig. 2.8, where the vectors include \mathbf{HS} with \mathbf{H} hue as the vector angle and \mathbf{S} saturation as the vector magnitude. \mathbf{V} is another vector with two purposes, first as the axis origin for the \mathbf{HS} vector and second as the color intensity or gray scale vector \mathbf{V} . It is often useful to convert raw RGB data into such color spaces for ease of analysis—for example, to be able to uniformly change the color intensity of all colors together so as to affect brightness or contrast.

In general, by increasing the dimensions of the feature space, more discrimination and robustness can be added. For example, the LBP pattern as described later in Chap. 6 can be extended into multiple variables by adding features such as a rotational invariant representation (RILBP); or by replicating the LBP across RGB color channels as demonstrated in the color LBP descriptor; or by extending the LBP pattern into spatiotemporal 3-space, like the LBP-TOP to add geometric distortion invariance.

Multimodal sensor data is becoming widespread with the proliferation of mobile devices that have built-in GPS, compass, temperature, altimeter, inertial and other sensors. An example of a multimodal, multivariate descriptor is the SIFT-GAFD [237] method, as illustrated in Fig. 4.5, which adds accelerometer information in the form of a gravity vector to the SIFT descriptor. The gravity vector is referred to as *global orientation*, and the SIFT local pixel region gradient is referred to as the *local orientation*.

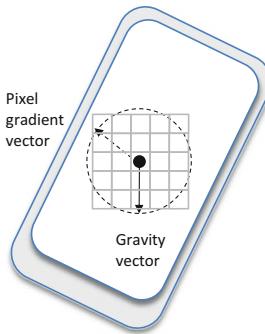


Figure 4.5 Multimodal descriptor using accelerometer data in the form of a gravity vector, in a feature descriptor as used in the SIFT-GAFD method [237]. The gravity vector of global orientation can be used for feature orientation with respect to the environment

Feature Pyramids

Many feature descriptors are computed in a mono-scale fashion using pixel values at a given scale only, and then for feature detection and matching the feature is searched for in a scale space image pyramid. However, by computing the descriptor at multiple scales and storing multiple scaled descriptors together in a *feature pyramid*, the feature can be detected on mono-scale images with scale variance without using a scale space pyramid.

For interest point and feature descriptor methods, *scale invariance* can be addressed either by: (1) scaling the images prior to searching, as in the scale space pyramid methods discussed later in this chapter; or (2) scaling and pyramiding multiple scales of the feature in the descriptor. Shape-based methods are by nature more scale invariant than interest point and feature descriptor methods, since shape-based methods depend on larger polygon structures and shape metrics.

Descriptor Density

Depending on the image data, there will be a different number of good interest points and features, since some images have more pronounced texture. And depending on the detector method used, images with high texture structure, or wider pixel intensity range differences, will likely generate more interest points than images with low contrast and smooth texture.

A good rule of thumb is that between 0.1 and 1 % of the pixels in an image can yield raw, unfiltered interest points. The more sensitive detectors such as FAST and the Harris detector family are at the upper end of this range (see Appendix A). Of course, detector parameters are tuned to reduce unwanted detection for each application.

Interest Point and Descriptor Culling

In fact, even though the interest point looks good, the corresponding descriptor computed at the interest point may not be worth using and will be discarded in some cases. Both interest points and descriptors are culled. So tuning the detector and descriptor together are critical trial-and-error processes. Using our base assumption of 0.1–1 % of the pixels yielding valid raw interest points, we can estimate the possible detected interest points based on video resolution, as shown in Table 4.2.

Table 4.2 Possible Range of Detected Interest Points per Image

	480p NTSC	1080p HD	2160p 4kUHD	4320p 8kUHD
Resolution	640 x 480	1920 x 1080	3840 x 2160	7680 x 4320
Pixels	307200	2073600	8294400	33177600
Interest points	300 – 3k	2k – 21k	8k – 83k	33k – 331k

Depending on the approach, the detector may be run only at mono-scale or across a set of scaled images in an image pyramid scale space. For scale space search methods, the interest point detector is run at each pixel in each image in the pyramid. What methods can be used to cull interest points to reduce the interest point density to a manageable number?

One method to select the best interest points is to use an *adaptive detector tuning method* (as discussed in Chap. 6 under “Interest Point Tuning”). Other approaches include only choosing interest points at a given threshold distance apart—for example, an interest point that cannot be adjacent to another interest point within a five-pixel window, with the best candidate point selected within the threshold.

Another method is to vary the search strategy as discussed in this chapter—for example, search for features at a lower resolution of the image pyramid, identify the best features, and record their positions, and perhaps search at higher levels of the pyramid to confirm the feature location, then compute the descriptors. This last method has the drawback of missing fine-grain features by default, since features may only be present at full image resolution.

Yet another method is to look for interest points every other pixel or within grid-sized regions. All of the above methods are used in practice, and other methods exist besides.

Dense vs. Sparse Feature Description

A *dense* descriptor makes use of all the pixels in the region or patch. By “dense” we mean that the kernel sampling pattern includes all the pixels, since a sparse kernel may select specific pixels to use or ignore. SIFT and SURF are classic examples of dense descriptors, since all pixels in rectangular regions contribute to the descriptor computation.

Many feature description methods, especially local binary descriptor methods, are making use of *sparse patterns*, where selected pixels are used from a region rather than all the pixels. The FREAK descriptor demonstrates one of the most ingenious methods of sparse sampling by modeling the human visual system, using a circular search region, and leveraging the finer resolution sampling closer to the center of the region, as well as tuning a hierarchy of local sampling patterns of increasing resolution for optimal results. Not only can sparse features potentially use less memory and reduce computations, but the sparse descriptor can be spread over a wider area to compensate for feature anomalies that occur in smaller regions.

Descriptor Shape Topologies

For this discussion, we view descriptor shape *topology* with an eye toward surveying the various shapes of the pixel regions used for descriptor computations. Part of the topology is the shape or boundary, and part of the topology is the choice of dense vs. sparse sampling patterns, discussed later

in this chapter. Sampling and patterning methods range from the simple rectangular regions up to the more complex sparse local binary descriptor patterns. As discussed in Chap. 6, both 2D and 3D descriptors are being designed to use a wide range of topologies. Let us look at a few topological design considerations, such as patch shape, sub-patches, strips, and deformable patches.

Which shape is better? The answer is subjective and we do not attempt to provide absolute answers, just offer a survey.

Correlation Templates

An obvious shape is the simple rectangular regions commonly used by correlation template matching methods. The descriptor is thus the *mugshot*, or actual image in the template region. To select sub-spaces within the rectangle, a mask can be used—for example, it could be a circular mask inside the bounding rectangle to mask off peripheral pixels from consideration.

Patches and Shape

The literature commonly refers to the feature shape as a *patch*, and usually a rectangular shape is assumed. Patch shapes are commonly rectangular owing to the ease of coding 2D array memory access. Circular patches are widely used in the local binary descriptor methods.

However, many descriptors also compute features *over multiple patches* or regions, not just a single patch. Here are some common variations on patch topology.

Single Patches, Sub-patches

Many descriptors limit the patch count to a single 2D patch. This is true of most common descriptors that are surveyed in Chap. 6. However, some of the local binary descriptors use a set of integral image *sub-patches* at specific points within the larger patch—for example, BRIEF uses a 5×5 integral image sub-patch at each sample point in the local binary pattern, within the larger 31×31 pixel patch region, so the value of each sub-patch becomes the value used for the point-pair comparison. The goal is to filter the values at each point to remove noise.

Deformable Patches

Rather than use a rigid shape, such as a fixed-size rectangle or a circle, feature descriptors can be designed with deformation in mind, such as scale deformations [337, 338], and affine or homographic deformation [212], to enable more robust matching. Examples include the DeepFlow [326, 376] deep matching method, and RFM2.3, as discussed in Chap. 6. Also, the D-NETS [127] method, using the fully connected or sparse connected topology, can be considered to be deformable in terms of invariance of the placement of the strip patterns; see Fig. 4.7 and the discussion of D-nets in Chap. 6. Many feature learning methods discussed later in this chapter also use deformed features for training.

Fixed descriptor shapes, such as rigid rectangles and circles, can detect motion under a rigid motion hypothesis, where the entire descriptor is expected to move with some amount of variance, such as in scale or affine transformation. However, for activity recognition and motion, a more deformable descriptor model is needed, and DeepFlow [336, 376] bridges the gap between descriptor matching methods and optical flow matching methods, using deformable patches and deep matching along the lines of deep learning networks.

Multi-patch Sets

The SIFT descriptor uses multi-patch sets of three patches from adjacent DoG images taken from the scale space pyramid structure, as shown in Fig. 6.15. Several other methods, such as the LBP-TOP and VLBP shown in Fig. 6.12, use sets of patches spread across a volume structure. LBP-TOP uses patches from adjacent planes, and the VLBP uses intersecting patches in 3-space. Dynamic texture methods use sets of three adjacent patches from spatiotemporal image frame sets, as frame $n - 2$, frame $n - 1$, and frame -0 (current frame).

TPLBP, FPLBP

The three-patch LBP TPLBP and four-patch LBP FPLBP [236] utilize novel multi-patch sampling patterns to add sparse local structure into a composite LBP descriptor. As shown in Fig. 4.6, the three-patch LBP uses a radial set of LBP patterns composed using alternating sets of three patches, and the four-patch LBP uses a more distributed pairing of patches over a wider range.

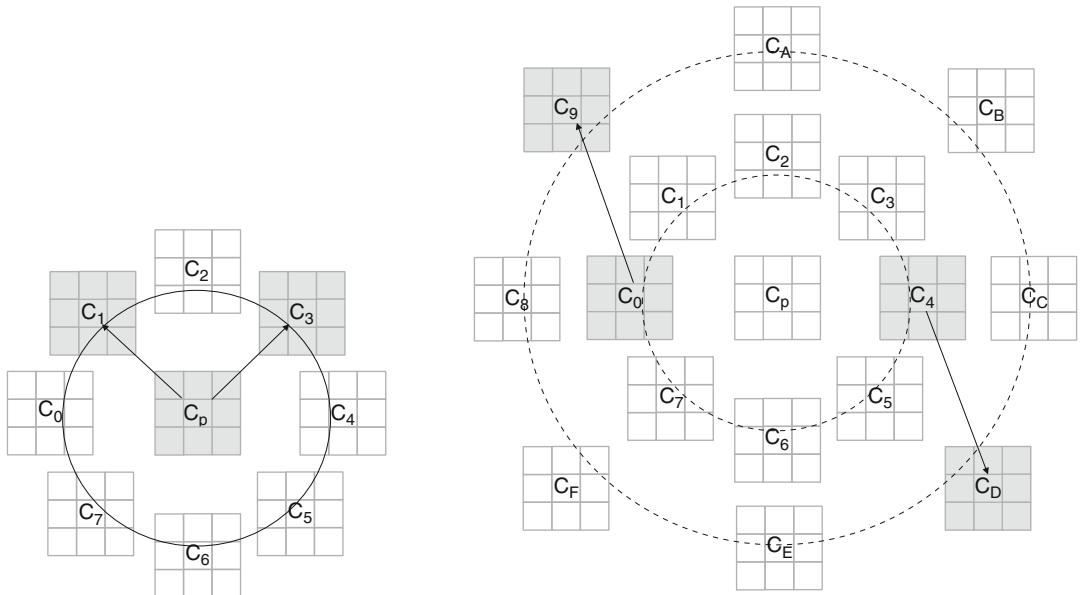


Figure 4.6 Novel multi-patch sets developed by Wolf et al. [236]. (*Left*) The TPLBP compares the values from three-patch sets around the ring to compute the LBP code, eight sets total, so there is one set for each LBP bit. (*Right*) The four-patch LBP uses four patches to computed bits using two symmetrically distributed patches from each ring, to produce each bit in the LBP code. The radius of each ring is a variable, the patch pairing is a variable, and the number of patches per ring is a variable; here, there are eight patches per ring

Strip and Radial Fan Shapes

Radial fans or spokes originating at the feature interest point location or shape centroid can be used as the descriptor sampling topology—for example, with Fourier shape descriptors (as discussed in Chap. 6; see especially Fig. 6.29).

D-NETS Strip Patterns

The D-NETS method developed by Hundelshausen and Sukthankar [127] uses a connected graph-shaped descriptor pattern with variations in the sampling pattern possible. The authors suggest that the method is effective using three different patterns, as shown in Fig. 4.7:

1. Fully connected graph at interest points
2. Sparse or iterative connected graph at interest points
3. Densely sampled graph over a chosen grid



Figure 4.7 Reduced resolution examples of the D-NETS [127] sampling patterns. (*Left*) Full dense connectivity at interest points. (*Center*) Sparse connectivity at interest points. (*Right*) Dense connectivity over a regular sampling grid. The D-NETS authors note that a dense sampling grid with 10 pixel spacing is preferred over sampling at interest points

The descriptor itself is composed of a set of *d-tokens*, which are strips of raw pixel values rather than a value from a patch region: the strip is the region, and various orientations of lines are the pattern. The sampling along the strip is between 80 and 20 % of the strip length rather than the entire length, omitting the endpoints, which is claimed to reduce the contribution of noisy interest points. The sampled points are combined into a set s of uniform chunks of pixels and normalized and stored into a discrete d-token descriptor.

Object Polygon Shapes

The object and polygon shape methods scan globally and regionally to find the shapes in the entire image frame or region. The goal is to find an object or region that is cohesive. A discussion of the fundamental methods for segmentation polygon shapes for feature descriptors is provided here, including:

- Morphological object boundary methods
- Texture or regional structural methods
- Superpixel or pixel similarity methods
- Depth map segmentation

Chapter 6 provides details on a range of object shape factors and metrics used to statistically describe the features of polygon shape. Note that this topic is often discussed in the literature as “image moments”; a good source of information is Flusser et al. [500].

Morphological Boundary Shapes

One method for defining polygon shapes is to use morphology. Morphological segmentation is a common method for region delineation, either as a binary object or as a gray scale object. Morphological shapes are sometimes referred to as *blobs*. In both binary and gray scale cases, thresholding is often used as a first step toward defining the object boundary, and morphological reshaping operations such as ERODE and DILATE are used to grow, shrink, and clean up the shape boundary. Morphological segmentation is threshold- and edge-feature driven. (Chapter 3 provided a discussion of the methods used for morphology and thresholding.)

Texture Structure Shapes

Region texture is also used to segment polygon shapes. Texture segmentation is a familiar image-processing method for image analysis and classification, and is an ideal method for segmentation in a nonbinary fashion. Texture reveals structure that simple thresholding ignores. As shown in Fig. 6.6, the LBP operator can detect local texture, and the texture can be used to segment regions such as sky, water, and land. Texture segmentation is based on local image pixel relationships. (Several texture segmentation methods were surveyed in Chap. 3.)

Super-Pixel Similarity Shapes

Segmenting a region using super-pixel methods is based on the idea of collapsing similar pixels together—for example, collapsing pixels together with similar colors into a larger shape. The goal is to segment the entire image region into super-pixels. Super-pixel methods are based on similarity. (Several super-pixel processing methods were discussed in Chap. 3.)

Local Binary Descriptor Point-Pair Patterns

Local binary descriptor shapes and sampling patterns, such as those employed in FREAK, BRISK, ORB, and BRIEF, are good examples to study in order to understand the various trade-offs and design approaches. We examine local binary shape and pattern concepts here. (Chapter 6 provides a more detailed survey of each descriptor.)

Local binary descriptors use a *point-pair sampling method*, where pairs of pixels are assigned to each other for a binary comparison. Note that a drawback of local binary descriptors and point-pair comparisons is that small changes in the image pixel values in the local region may manifest as *binary artifacts*. Seemingly insignificant changes in a set of pixel values may cause problems during matching that are pronounced for: (1) noisy images, and (2) images with constant gray level. However, each local binary descriptor method attempts to mitigate the binary artifact problems. For example, BRISK (see Fig. 4.10) and ORB (see Fig. 4.11) compute a *filtered region* surrounding each interest point to reduce the noise component prior to the binary comparison.

Another method to mitigate the binary artifact problem of constant gray level is used in a modification of the LBP method called the local trinary pattern operator, or LTP [504] (see also reference [165], Section 2.9.3), which uses *trinary values* of $\{-1, 0, 1\}$ to describe regions. A threshold band is established for the LTP to describe near-constant gray values as 0, values above the threshold band as 1, and values below the threshold band as -1 . The LTP can be used to describe both smooth regions of constant gray level and contrasted regions in the standard LBP. In addition, the compare threshold for point-pairs can be tuned to compensate for noise, illumination, and contrast, as employed in nearly all local binary descriptor methods.

Figure 4.8 (left image) illustrates a hypothetical descriptor pattern to include selected pixels as the black values, while the center left image shows a strip-oriented shape and pattern where the descriptor calculates the descriptor over pixels along a set of line segments with no particular symmetry like the DNETS [127] method.

In Fig. 4.8 also, the center right image illustrates a convolution kernel where the filter shape and filter goal are specified, while the right image is a blob shape using radial pixel sampling lines originating at the shape centroid and terminating on the blob perimeter. Note that a 1D Fourier descriptor can be computed from an array containing the length of each radial line segment from the centroid to the perimeter to describe shape, or just an array of raw pixel values can be kept, or else D-nets can be computed.

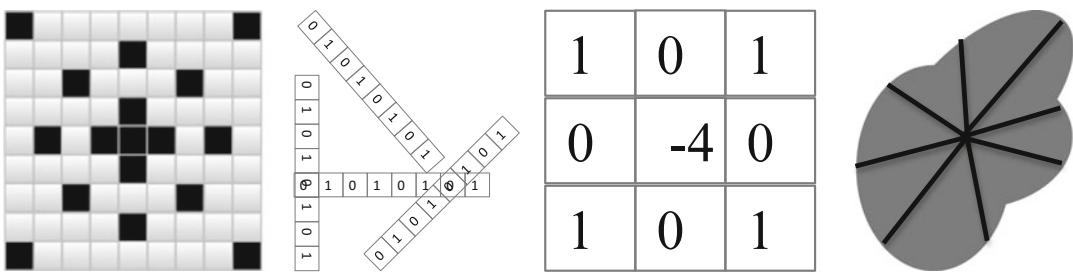


Figure 4.8 Illustrating various descriptor patterns and shapes. (Left) Sparse. (Center left) Nets or strips. (Center right) Kernels. (Right) Radial spokes

A feature descriptor may be designed by using one or more shapes and patterns together. For example, the hypothetical descriptor pattern in Fig. 4.8 (left image) uses one pattern for pixels close to the interest point, another pattern uses pixels farther away from the center to capture circular pattern information, and another pattern covers a few extrema points. An excellent example of tuned sampling patterns is the FREAK descriptor, discussed next.

FREAK Retinal Patterns

The FREAK [122] descriptor shape, also discussed in some detail in Chap. 6, uses local binary patterns based on the human retinal system, as shown in Fig. 4.9, where the density of the receptor cells in the human visual system is greater in the center and decreases with distance from center. FREAK follows a similar pattern when building the local binary descriptors, referred to as a *coarse-to-fine* descriptor pattern, with fine detail in the center of the patch and coarse detail moving outward. The coarse-to-fine method also allows for the descriptor to be matched in coarse-to-fine segments. The coarse part is matched first, and if the match is good enough, the fine feature components are matched as well.

FREAK descriptors can be built with several patterns within this framework. For FREAK, the actual pattern shape and point-pairing are designed during a training phase where the best point-pair patterns are learned using a method similar to ORB [126] to find point-pairs with high variance. The pattern is only constrained by the training data; only 45 point-pairs are used from the 32×31 image patch region.

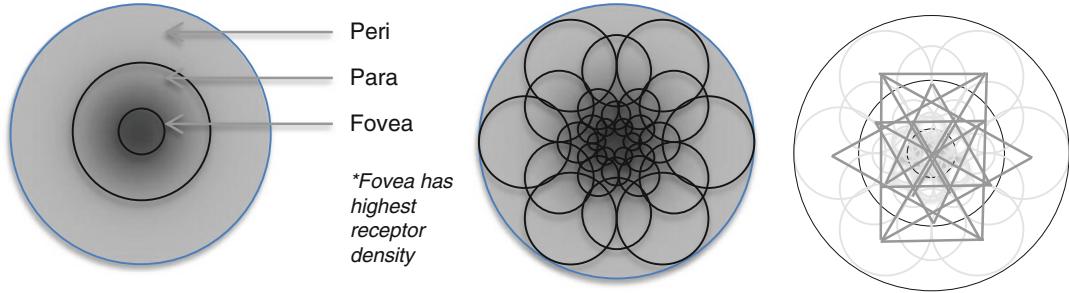


Figure 4.9 (Left) The human visual system concentration of receptors in the center Fovea region with less receptor density moving outward to periphery vision regions of Para and Peri. (Center) FREAK [122] local binary pattern sampling regions, six regions in each of six overlapping distance rings from the center, size of ring denotes compare point averaging area. (Right) Hypothetical example of a FREAK-style point-pair pattern

As illustrated in Fig. 4.9, the pairs of points at the end of each line segment are compared, the set of compare values are composed into a binary descriptor vector using 16 bytes, and a cascade of four separate 16-byte coarse-to-fine patterns are included in the descriptor set. Typically, the coarse pattern alone effectively rejects bad matches, and the finer patterns are used to qualify only the closest matches.

Brisk Patterns

The BRISK descriptor [123] point-pair sampling shape is symmetric and circular, composed of 60 total points arranged in four concentric rings, as shown in Fig. 4.10. Surrounding each of the 60 points is a circular sampling region, the sampling regions increase in size with distance from the center, and also proportional to the distance between sample points. Within the sampling regions, Gaussian smoothing is applied to the pixels and a local gradient is calculated over the smoothed region.

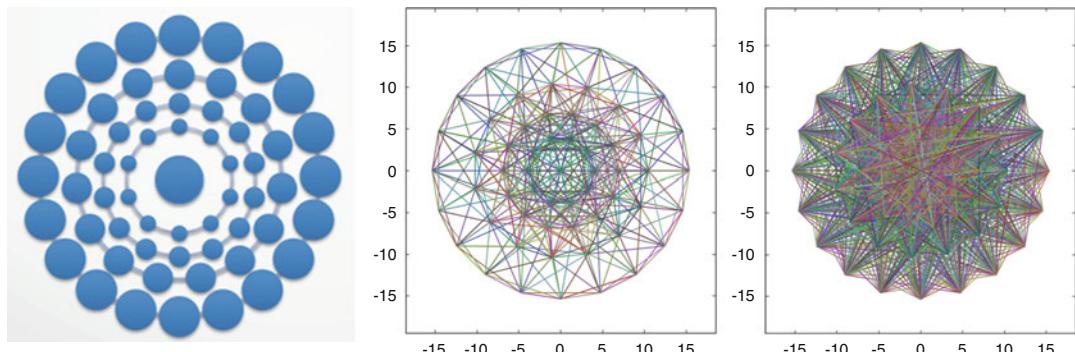


Figure 4.10 (Left) BRISK concentric sampling grid pattern. (Center) Short segment pairs. (Right) Long distance pairs. Note that the size of the region (*left image*) for each selected point increases in diameter with distance from the center, and the binary comparison is computed from the center point of each Gaussian-sampled circular region, rather than from each solitary center point (*Center and right images* used by permission © Josh Gleason [135])

Like other local binary descriptors, BRISK compares pairs of points to form the descriptor. The point-pairs are specified in two groups: (1) *long segments*, which are used together with the region gradients to determine angle and direction of the descriptor, the angle is used to rotate the descriptor area, and then the pair-wise sampling pattern is applied; (2) *short segments*, which can be pair-wise compared and composed into the 512-bit binary descriptor vector.

ORB and BRIEF Patterns

ORB [126] is based in part on the BRIEF descriptor [124, 125], thus the name **Oriented Brief**, since ORB adds orientation to the BRIEF method and provides other improvements as well. For example, ORB also improves the interest point method by qualifying FAST corners using Harris corner methods, and improves corner orientation using Rosin's method [53] in order to steer the BRIEF descriptor to improve rotational invariance (BRIEF is known to be sensitive to rotation).

ORB also provides a very good point-pair training method, which is an improvement over BRIEF. In BRIEF, as shown in Fig. 4.11, the sample points are specified in a random distribution pattern based on a Gaussian distribution about the center point within the 31×31 patch region; the chosen number of sample points is 256. Selected sample point-pairs are compared to each other to form the binary descriptor vector. The value of each point is calculated via an integral image method to smooth a 5×5 region into the point value.

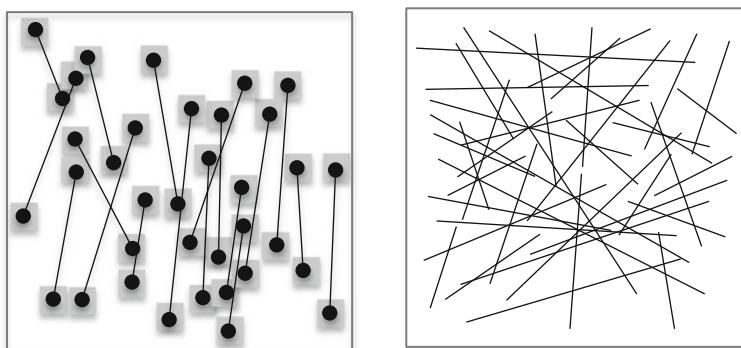


Figure 4.11 (Left) An ORB style pattern at greatly reduced point pair count resolution, using <32 points instead of the full 256 points. (Right) A BRIEF style pattern using randomized point-pairs

To learn the descriptor point-pair sample and compare pattern, ORB uses a training algorithm to find uncorrelated points in the training set with high variance, and selects the best 256 points to define the pairwise sampling patterns used to create the binary feature vector. So the shape and pattern are nonsymmetric, as shown in Fig. 4.11, similar to some DNETS patterns. The ORB point-pair patterns are dependent on the training data.

Note in Fig. 4.11 that a BRIEF style pattern (right image) uses random point-pairs. Several methods for randomizing point-pairs are suggested by the developers [124]. The ORB pattern shown in Fig. 4.11 is based on choosing point-pairs that have high statistical variance within a bounding 31×31 image patch, where the smaller 5×5 gray image patch regions are centered at the chosen interest points. Then each 5×5 region is smoothed using an integral image method to yield a single value for the point.

Descriptor Discrimination

How discriminating is a descriptor? By *discrimination* we mean how well the descriptor can uniquely describe and differentiate between other features. Depending on the application, more or less discrimination is needed, thus it is possible to *over-describe* a feature by providing more information and invariance than is useful, or to *under-describe* the feature by limiting the robustness and invariance attributes. Feature descriptor discrimination for a given set of robustness criteria may be important and interesting, but discrimination is not always the right problem to solve in some cases.

The need for increased discrimination in the descriptor can be balanced in favor of using a cascade of simple descriptors like correlation templates under the following assumptions.

1. **Assuming cheap massively parallel compute**, deformable descriptors such as Taylor and Rosin's RFM2.3 [212] become a more attractive option, allowing simple weakly discriminating correlation templates or pixel patches to be used and deformed in real-time in silicon using the GPU texture sampler for scale, affine and homographic transforms. Matching and correspondence under various pose variations and lighting variations can be easily achieved using parallel GPU SIMD/SIMT compute and convolution kernels. So the GPU can effectively allow a simple correlation patch to be warped and contrast enhanced to be used as a deformable descriptor and compared against target features.
2. **Assuming lots of fast and cheap memory**, such as large memory cache systems, many nondiscriminating descriptors or training patterns can be stored in the database in the memory cache. Various weighting schemes such as those used in neural networks and convolutional networks can be effectively employed to achieve desired correspondence and quality. Also, other boosting schemes can be employed in the classifier, such as the Adaboost method, to developed strong classifiers from weakly discriminating data.

In summary, both highly discriminating feature descriptors and cascades of simple weakly discriminating feature descriptors may be the right choice for a given application, depending on the target system.

Spectra Discrimination

One dimension of feature discrimination is the chosen descriptor spectra or values used to represent the feature. We refer to *spectra* simply as values within a spectrum or over a continuum. A feature descriptor that only uses a single spectra, such as a histogram of intensity values, will have discrimination to intensity distributions, with no discrimination for other attributes such as shape or affine transforms. For example, a feature descriptor may increase the level of discrimination by combining a multivariate set of spectra such as *RGB* color, depth, and local area gradients of color intensity, (see Varma [771] and Vedaldi [770] for more on multivariate descriptors).

It is well known [240] that the human visual system discriminates and responds to gradient information in a scale and rotationally invariant manner across the retina, as demonstrated in SIFT and many other feature description methods. Thus the use of gradients is a common and preferred spectra for computer vision.

Spectra may be taken over a range of variables, where simple scalar ranges of values are only one type of spectra:

1. Gray scale intensity
2. Color channel intensity
3. Basis function domains (frequency domain, HAAR, etc.)
4. 2D or 3D gradients
5. 3D surface normals
6. Shape factors and morphological measures
7. Texture metrics
8. Area integrals
9. Statistical moments of regions
10. Hamming codes from local binary patterns

Each of the above spectra types, along with many others that could be enumerated, can be included in a multivariate feature descriptor to increase discrimination. Of course, discrimination requirements for a chosen application will guide the design of the descriptor. For example, an application that identifies fruit will be more effective using color channel spectra for fruit color, shape factors to identify fruit shapes, and texture metrics for skin texture.

One way to answer the question of discrimination is to look at the information contained in the descriptor. Does the descriptor contain multivariate collections of spectra, and how many invariance attributes are covered, such as orientation or scale?

Region, Shapes, and Pattern Discrimination

Shape and pattern of the feature descriptor are important dimensions affecting discrimination. Each feature shape has advantages and disadvantages depending on the application. Surprisingly, even a single pixel can be used as a feature descriptor shape (see Fig. 1.7). Let us look at other dimensions of discrimination.

Shapes and patterns may be classified as follows:

1. A single pixel (discussion of single pixel description methods to follow)
2. A line of pixels
3. A rectangular region of pixels
4. A polygon shape or region of pixels
5. A pattern or set of unconnected pixels, such as foveal patterns

The shape of the descriptor determines attributes of discrimination. For example, a rectangular descriptor will be limited in the rotational invariance attribute compared to a circular shaped descriptor. Also, a smaller shape for the descriptor limits the range to a smaller area, and also limits scale invariance. A larger size descriptor area carries more pixels which can increase discrimination.

Descriptor shape, pixel sampling pattern, sampling region size, and pixel metrics have been surveyed by several other researchers [120–122]. In this section, we dig deeper and wider into specific methods used for feature descriptor tuning, paying special attention to local binary feature descriptors, which hold promise for low power and high performance.

Geometric Discrimination Factors

The shape largely determines the amount of rotational invariance possible. For example, a rectangular shape typically begins to fall off in rotational discrimination at around 15° , while a circular pattern typically performs much better under rotational variations. Note that any poorly discriminating shape or pattern descriptor can be enhanced and made more discriminating by incorporating more than one shape or pattern into the descriptor vector.

A shape and pattern such as a HAAR wavelet, as used in the Viola–Jones method, integrates all pixels in a rectangular region, yielding the composite value of all the pixels in the region. Thus there is no local fine-detail pattern information contained in the descriptor, leading to very limited local area discrimination and poor rotational invariance or discrimination.

Another example of poor rotational discrimination is the rectangular correlation template method, which compares two rectangular regions pixel by pixel. However, several effective descriptor methods use a rectangular-shaped region.

In general, rectangles are a limitation to rotational invariance. However, SURF uses a method of determining the dominant orientation of the rectangular HAAR wavelet features within a circular neighborhood to achieve better rotational invariance. And SIFT uses a method to improve rotational invariance and accuracy by applying a circular weighting function to the rectangular regions during the binning stage.

It should also be noted that descriptors with low discrimination are being used very effectively in targeted applications, such as correlation methods for motion estimation in video encoding. In this case, the rectangle shape is a great match to the encoding problem and lends itself to highly optimized fixed function hardware implementations, since frame-to-frame motion can be captured very well in rectangular regions, and there is typically little rotation or scale change from frame to frame for at 30 Hz frame rates, just translation.

With this discussion in mind, descriptor discrimination should be *fitted* appropriately to the application, since adding discrimination comes at a cost of compute and memory.

Feature Visualization to Evaluate Discrimination

Another way to understand discrimination is to use the feature descriptor itself to reconstruct images from the descriptor information alone, where we may consider the collection of descriptors to be a compressed or encoded version of the actual image. Image compression, encoding, and feature description are related; see Fig. 3.18. Next, we examine a few examples of image reconstruction from the descriptor information alone.



Figure 4.12 Discrimination via a visualization of the HOG description (left image), original image on right. (Image © Carl Vondrick, used by permission.) See also “HOGgles: Visualizing Object Detection Features, Carl Vondrick, Aditya Khosla, Tomasz Malisiewicz, Antonio Torralba, Massachusetts Institute of Technology, Oral presentation at ICCV 2013”

Discrimination via Image Reconstruction from HOG

Figure 4.12 visualizes a reconstruction using the HOG descriptor [98]. The level of detail is coarse and follows line and edge structure that matches the intended use of HOG. One key aspect of the discrimination provided by HOG is that no image smoothing is used on the image prior to calculating the descriptor. The HOG research shows that smoothing the image results in a *loss of discrimination*. Dalal and Triggs [98] highlight their deliberate intention to avoid image smoothing to preserve image detail.

However, some researchers argue that noise causes problems when calculating values such as local area gradients and edges, and further recommend that noise be eliminated from the image by smoothing prior to descriptor calculations; this is the conventional wisdom in many circles. Note that there are many methods to filter noise without resorting to extreme Gaussian-style smoothing, convolution blur, and integral images, which distort the image field.

Some of the better noise-filtering methods include speckle removal filters, rank filtering, bilateral filters, and many other methods that were discussed in Chap. 2. If the input image is left as is, or at least the best noise filtering methods are used, the feature descriptor will likely retain more discrimination power for fine-grained features.

Discrimination via Image Reconstruction from Local Binary Patterns

As shown in Fig. 4.13, d’Angelo and Alahi [119] provide visualizations of images reconstructed from the FREAK and BRIEF local binary descriptors. The reconstruction is rendered entirely from the descriptor information alone, across the entire image. BRIEF uses a more random pattern to compare points across a region, while FREAK uses a trained and more foveal and symmetrical pattern with increased detail closer to the center of the region. And d’Angelo and Alahi [119] note that the reconstruction results are similar to Laplacian filtered versions of the original image, which helps us understand that the discrimination of these features appears to be structurally related to detailed edge and gradient information.

The d’Angelo and Alahi reconstruction method [119] creates an image from a set of overlapping descriptor patches calculated across the original image. To reconstruct the image, the descriptors are first reconstructed using a novel method to render patches, and then the patches are merged by averaging the overlapping regions to form an image, where the patch merge size may vary as desired. For example, note that Fig. 4.13 uses 32×32 patches for the Barbara images in the left column,

and a 64×64 patch size for the cameraman in the center column. Also note that Barbara is not reconstructed with the same discrimination as the cameraman, whose image contains finer details. Other fascinating feature visualization work is provided by Vodrick [888].

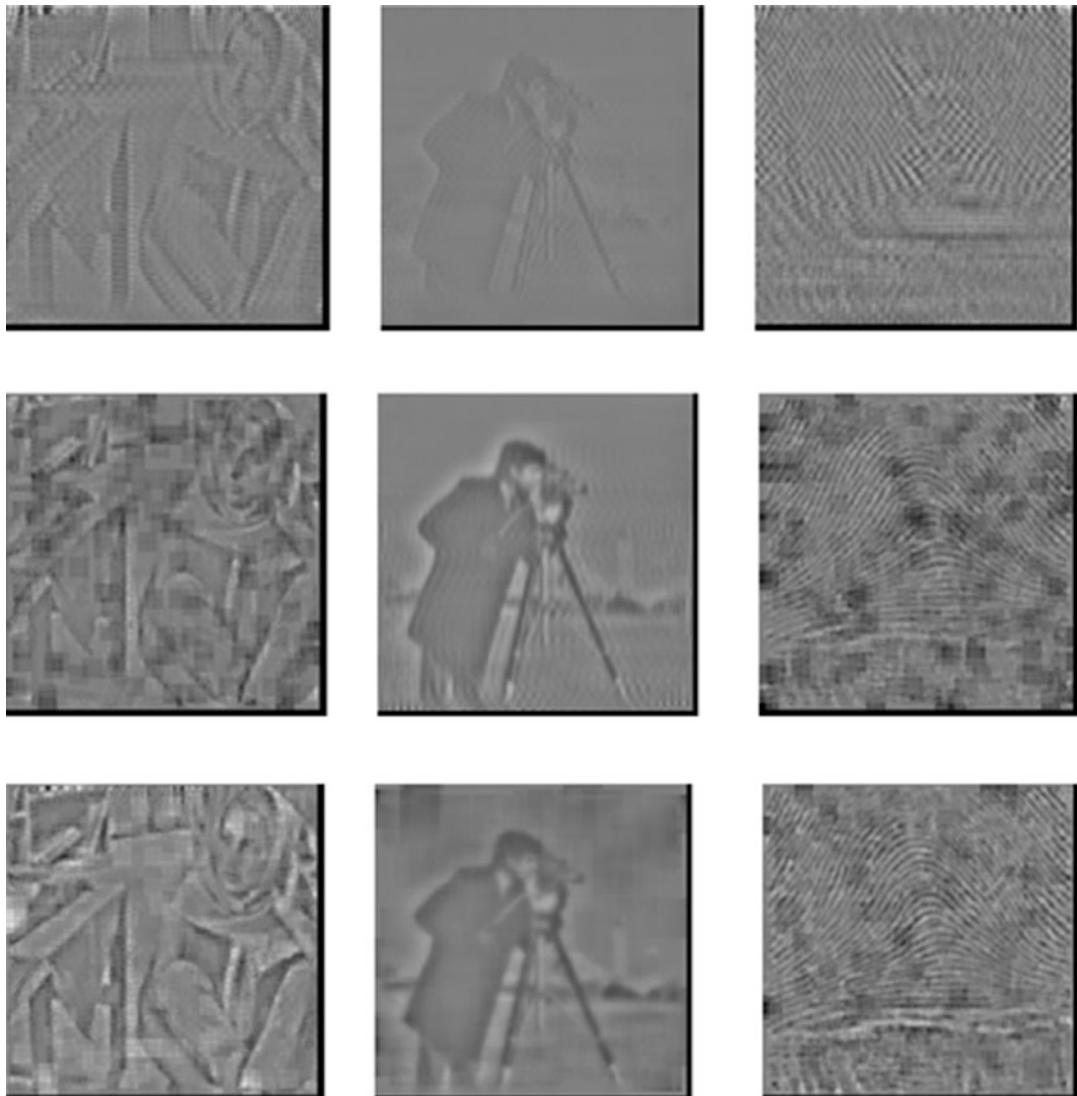


Figure 4.13 Images reconstructed using local binary descriptors using 512 point-pairs. (*Top row*) BRIEF. (*Middle row*) Randomized FREAK (more similar to BRIEF). (*Bottom row*) Binary FREAK using the foveal pattern. Images © Alexandre Alahi, used by permission

Discrimination via Image Reconstruction from SIFT Features

Another method of approximate image reconstruction [97] proves the discrimination capabilities of SIFT descriptors; see Fig. 4.14. The reconstruction method for this research starts by taking an unknown image containing a scene such as a famous building, finding the set of Hessian-affine region detectors in the image, extracting associated SIFT feature descriptors, and then saving a set of elliptical image patch regions around the SIFT descriptors.



Figure 4.14 Image reconstruction of common scenes using combined SIFT descriptors taken from several views of the same object, images © Herve Jegou, used by permission

Next, an image database containing similar and, it is hoped, matching images of the same scene are searched to find the closest matching SIFT descriptors at Hessian-affine interest points. Then a set of elliptical patch regions around each SIFT descriptor is taken. The elliptical patches found in the database are warped into a synthesized image based on a priori interest region geometric parameters of the scenes.

The patches are stitched together via stacking and blending overlapping patches and also via smooth interpolation. Any remaining holes are filled by smooth interpolation. One remarkable result of this method is the demonstration that an image can be reconstructed from a set of patches from different images at different orientations, since the feature descriptors are similar; and in this case, the discrimination of the SIFT descriptor is demonstrated well.

Accuracy, Trackability

Accuracy can be measured in terms of specific feature attributes or robustness criteria; see Tables 4.1 and 7.4. A given descriptor may outperform another descriptor in one area and in not another. In the research literature, the accuracy and performance of each new feature descriptor is usually benchmarked against the standby methods SIFT and SURF. The feature descriptor accuracy is measured using commonly accepted ground truth datasets designed to measure robustness and invariance attributes. (See Appendix B for a survey of standard ground truth datasets, and Chap. 7 for a discussion about ground truth dataset design.)

A few useful accuracy studies are highlighted here, illustrating some of the ways descriptor and interest point accuracy can be measured. For instance, one of the most comprehensive surveys of earlier feature detector and descriptor accuracy and invariance is provided by Mikolajczyk and Schmid [136], covering a range of descriptors including GLOH, SIFT, PCA-SIFT, Shape Context, spin images, Hessian Laplacian GLOH, cross correlation, gradient moments, complex filters, differential invariants, and steerable filters.

In Gauglitz et al. [137], there are invariance metrics for zoom, pan, rotation, perspective distortion, motion blur, static lighting, and dynamic lighting for several feature metrics, including Harris, Shi-Tomasi, DoG, Fast Hessian, FAST, and CenSurE, which are discussed in Chap. 6. There are also metrics for a few classifiers, including randomized trees and FERNS, which are discussed later in this chapter. Figure 4.15 provides some visual comparisons of feature detector and interest point accuracy from Gauglitz [137].

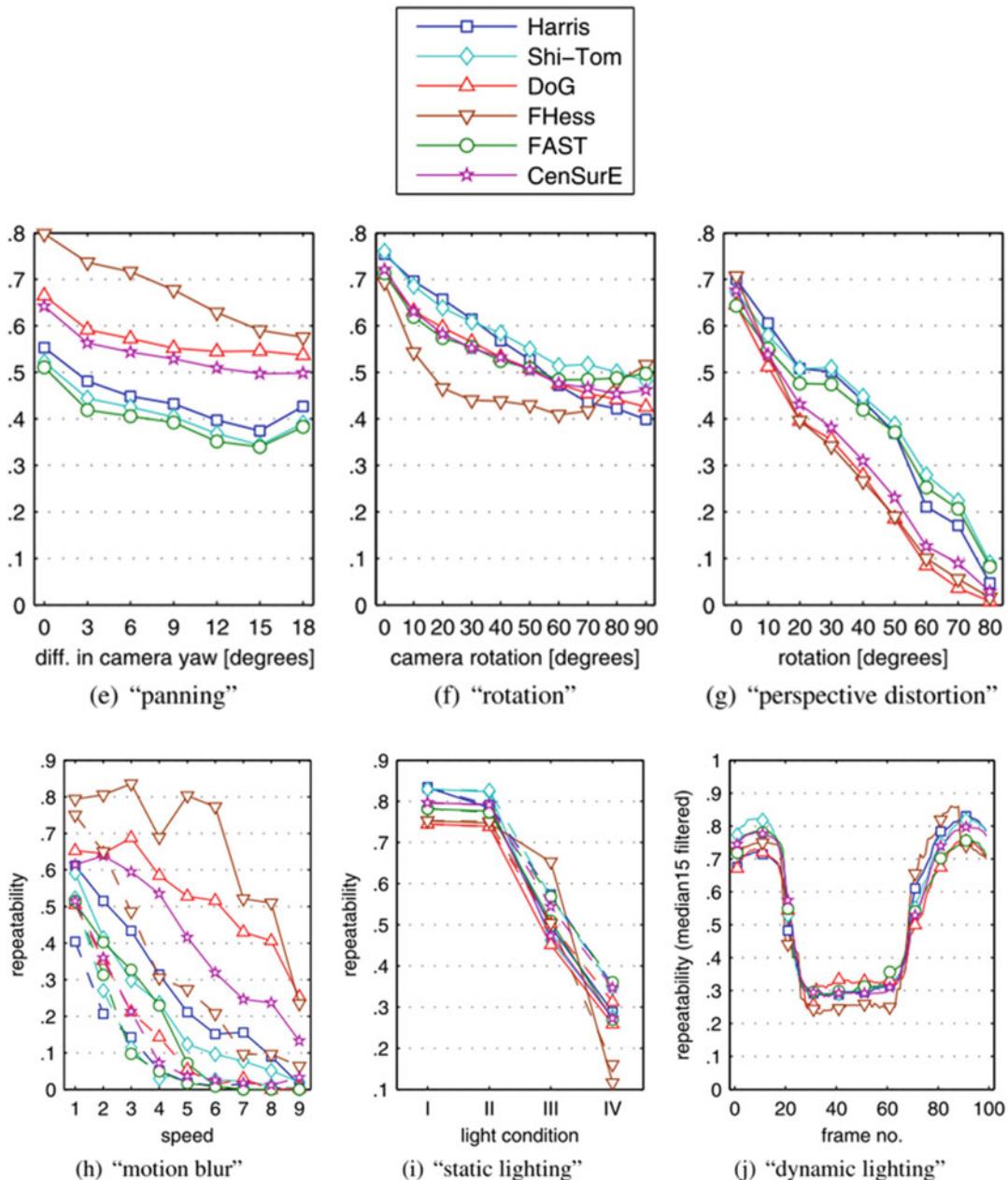


Figure 4.15 Accuracy of feature descriptors over various invariance criteria (From Gauglitz et al. [137], images © Springer Science + Business Media, LLC, used by permission)

Turning to the more recent local binary descriptors, Alahi et al. [122] provide a set of comparisons where FREAK is shown to be superior in accuracy to BRISK, SURF, and SIFT on a particular dataset and set of criteria developed by Mikolajczyk and Schmid [136] for feature accuracy over attributes such as viewpoint, blur, JPEG compression, brightness, rotation, and scale. In Rublee et al. [112], ORB is shown to have better rotational invariance than SIFT, SURF, and BRIEF. In summary, local binary descriptors are proving to be attractive in terms of robustness, accuracy, and compute efficiency.

Accuracy Optimizations, Subregion Overlap, Gaussian Weighting, and Pooling

Various methods are employed to optimize feature descriptor accuracy, and a few methods are discussed here. For example, descriptors often use overlapping sampling pattern subregions, as shown in the FREAK descriptor pattern in Fig. 4.9. By overlapping sampling regions and treating boundaries carefully, accuracy seems to be better in most all cases [153, 170]. Overlapping regions makes sense intuitively, since each point in a region is related to surrounding points. The value of pattern subregion overlapping in feature description seems obvious for local binary pattern type descriptors and spectra descriptor variants such as SURF and others [136, 173]. When the sampling regions used in the descriptor do not overlap, recognition rates are not as accurate [122].

Gaussian weighting is another effective method for increasing accuracy to reduce noise and uncertainty in measurements. For example, the SIFT [153, 170] descriptor applies a Gaussian-based weighting factor to each local area gradient in the descriptor region to favor gradients nearer the center and reduce the weighting of gradients farther away. In addition, the SIFT weighting is applied in a circularly symmetric pattern, which adds some rotational invariance; see Fig. 6.17.

Note that Gaussian weighting is different from Gaussian filtering; a Gaussian filter both reduces noise and eliminates critical fine details in the image, but such filtering has been found to be counterproductive in the HOG method [98]. A Gaussian weighting factor, such as used by SIFT on the gradient bins, can simply be used to qualify data rather than change the data. In general, a weighting factor can be used to scale the results and fine-tune the detector or descriptor. The subregion overlap in the sampling pattern and Gaussian weighting schemes are complementary.

Accuracy can be improved by relying on groups of nearby features together rather than just a single feature. For example, in convolutional networks, several nearby features may be pooled for a joint decision to increase accuracy via chosen robustness or invariance criteria [339]. The pooling concept may also be referred to as *neighborhood consensus* or *semi-local constraints* in the literature, and it can involve joint constraints, such as the angle and distance among a combined set of local features [340–342].

Sub-pixel Accuracy

Some descriptor and recognition methods can provide sub-pixel accuracy in matching the feature location [139–142]. Common methods to compute sub-pixel accuracy include cross-correlation, sum-absolute difference, Gaussian fitting, Fourier methods, and rigid body transforms and ICP. In general, sub-pixel accuracy is not a common feature in popular, commercial applications and is needed only in high-end applications like industrial inspection, aerospace, and military systems.

For example, SIFT provides sub-pixel accuracy for the location of keypoints. Digital correlation methods and template matching are well known and used in industrial applications for object tracking, and can be extended to compute correlations over a range of one-pixel offset areas to yield a set of correlations that can be fit into a curve and interpolated to find the highest match to yield sub-pixel accuracy.

Sub-pixel accuracy is typically limited to translation. Rotation and scale are much more difficult to quantify in terms of sub-pixel accuracy. Typical sub-pixel accuracy results for translation only achieve better than $\frac{1}{4}$ pixel resolution, but resolution accuracy can be finer grained, and in some methods translation accuracy is claimed to be as high as 1/20th of a pixel using FFT registration methods [143].

Also, stereo disparity methods benefit from improved sub-pixel accuracy, especially at long ranges, since the granularity of Z distance measurements increases exponentially with distance.

Thus the calculated depth field contains coarser information as the depth field increases, and the computed depth field is actually nonlinear in Z . Therefore, sub-pixel accuracy in stereo and multi-view stereo disparity calculations is quite desirable and necessary for best accuracy.

Search Strategies and Optimizations

As shown in Fig. 5.1, a feature may be sparse, covering a local area, or it may cover a regional or global area. The search strategy used to isolate each of these feature types is different. For a global feature, there is no search strategy: the entire frame is used as the feature. For a regional descriptor, a region needs to be chosen or segmented (discussed in Chap. 2). For sparse local features, the search strategy becomes important. Search strategies for sparse local regions fall into a few major categories, as follows (also included in the taxonomy in Chap. 5).

Dense Search

In a dense search, each pixel in the image is checked. For example, an interest point is calculated at each pixel, the interest points are then qualified and sorted into a candidate list, and a feature descriptor is calculated for each candidate. Dense search is used by local binary descriptors and common descriptors such as SIFT.

In stereo matching and depth sensing, each pixel is searched in a dense manner for calculating disparity and closest points. For example, stereo algorithms use a dense search for correspondence to compute disparity, line by line and pixel by pixel; monocular depth-sensing methods such as PTAM [319] use a dense search for interest points, followed by a sparse search for known features at predicted locations.

Dense methods may also be applied across an image pyramid, where the lower resolution pyramids are usually searched first and finer-grain pyramids are searched later. Dense methods in general are preferred for accuracy and robustness when feature locations are not known and cannot be predicted.

Grid Search

In grid search methods, the image is divided into a regular grid or tiles, and features are located based on the tiles. A novel grid search method is provided in the OpenCV library, using a grid search adapter (discussed in Chap. 6 and Appendix A). This allows for repeated trial searches within a grid region for the best features, and has the capability of adjusting detector parameters before each trial run. One possible disadvantage of a grid search from the perspective of accuracy is that features do not line up into grids, so features can be missed or truncated along the grid boundary, decreasing accuracy and robustness overall.

Grid search can be used in many ways. For example, a regular grid is used as anchor points with the grid topology of D-NETS, as illustrated in Fig. 4.7. Or, a grid is used to form image tile patches and a descriptor is computed for each tile, such as in the HOG method, as shown in Fig. 4.12. Also the Viola–Jones method [138] computes HAAR features on a grid.

Multi-scale Pyramid Search

The idea behind the multi-scale image pyramid search is either to accelerate searching by starting at a lower resolution or to truly provide multi-scale images to allow for features to be found at appropriate scale. Methods to reduce image scale include pixel decimation, bilinear interpolation, and other multi-sampling methods. Scale space is a popular method for creating image pyramids, and many variations are discussed in the next section; see Fig. 4.16.



Figure 4.16 A five-octave scale pyramid. The image is from Albrecht Durer’s Apocalypse woodcuts, 1498. Note that many methods use non-octave pyramid scales [112]

However, the number of detected features falls off rapidly as the pyramid levels increase, especially for scale space pyramids, which have been Gaussian filtered, since Gaussian filters reduce image texture detail. Also, fewer pixels are present to begin with at higher pyramid levels, so a pyramid scale interval smaller than octaves is sometimes used. See reference [152] for a good discussion of image pyramids.

Scale Space and Image Pyramids

Often, instead of using simple pixel decimation and pixel interpolation to reduce image scale, a *scale space* [505, 506] pyramid representation, originally proposed by Lindberg [529], is built up using Gaussian filtering methods to decrease the scaling artifacts and preserve blob-like features. Scale space is a more formal method of defining a multi-scale set of images, typically using a Gaussian kernel $g()$ convolved with the image $f(x,y)$, as follows:

$$g(x,y : t) = \frac{1}{2\pi t} e^{-(x^2+y^2)/2t}$$

$$L(.,.; t) = g(.,.; t) \times f(.,.),$$

or by an equivalent method:

$$\partial_t L = \frac{1}{2} \nabla^2 L,$$

with the initial state $L(x, y; 0) = f(x, y)$.

A good example of Gaussian filter design for scale space is described in the SURF method [152]. Gaussian filters implemented as kernels with increasing size are applied to the original image at octave-spaced subsampling intervals to create the scale space images—for example, starting with a 9×9 Gaussian filter and increasing to 15×15 , 21×21 , 27×27 , 33×33 , and 39×39 ; see Fig. 4.17.



Figure 4.17 Scale space Gaussian images at scales of 0, 2, 4, 16, 32, 64. Image is from Albrecht Durer’s Apocalypse woodcuts, 1498

One drawback of scale space is the loss of localization and lack of accuracy in higher levels of the image pyramid. In fact, some features are simply missing from higher levels of the image pyramid, owing to a lack of resolution and to the Gaussian filtering. The best example of effective scale space feature matching may be SIFT, which provides for the first pyramid image in the scale to be double the original resolution to mitigate scale space problems, and also provides a good multi-scale descriptor framework. See also Fig. 4.18.

Image pyramids are analogous to texture *mip-maps* used in computer graphics. Variations on the image pyramid are common. Octave and non-octave pyramid spacings are used, with variations on the filtering method also. For example, the SIFT method [153, 170] uses a five-level octave scale $n/2$ image pyramid with Gaussians filtered images in a scale space. Then, the Difference of Gaussians (DoG) method is used to capture the interest point extrema maxima and minima in the adjacent images in the pyramid. SIFT uses a double-scale first pyramid level with linear interpolated pixels at

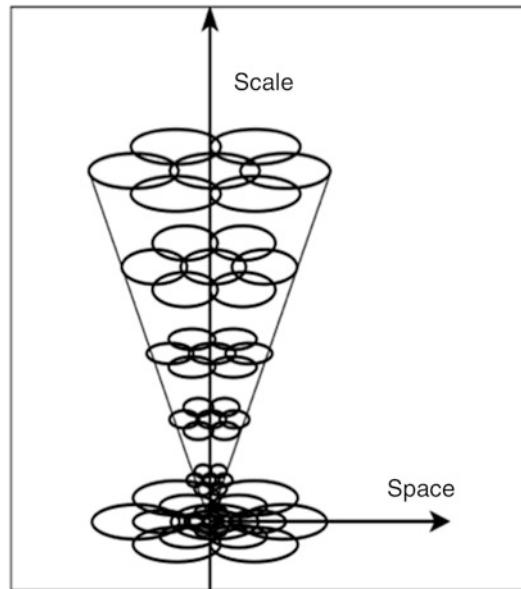


Figure 4.18 Scale and space

$2\times$ original magnification to help preserve fine details. This technique increases the number of stable keypoints by about four times, which is quite significant. In the ORB [112] method, a non-octave scale space is built around a $\sqrt{2}$ scale over a five-level pyramid, which has closer resolution gradations between pyramid levels than an octave scale of two times.

Feature Pyramids

An alternative to scale space pyramids and pyramid searching is to use *feature-space pyramiding*, and build a set of multi-scale feature descriptors stored together in the database. In this approach, the descriptor itself contains the pyramid, and no scale space or image pyramid is needed. Instead, feature searching occurs directly from the mono-scale target image to the multi-scale features. The RFM method [212] discussed in Chap. 6 goes even further and includes *multi-perspective* transformed versions of each patch for each descriptor. In Table 4.3, note that the multi-scale features can be used to match directly on the target images, while the mono-scale features are better to use on an image pyramid.

Table 4.3 Some Tradeoffs in Using a Mono-Scale Feature and a Multi-Scale Feature

Feature Scale	Feature Size	Feature Description Compute Time	Image Pyramid Used for Matching	Mono-Scale Images Used for Matching
Mono-scale feature	Smaller memory footprint	Faster to compute	Yes	No
Multi-scale feature	Larger memory footprint	Slower to compute	No	Yes

Figure 3.16 shows the related concept of a *multi-resolution histogram* [144], created from image regions from a scale space pyramid and with the histograms concatenated in the descriptor that is used to determine texture metrics for feature matching. So in the multi-scale histogram method, no pyramid image set is required at run time; rather, the pyramid search uses histogram features from the descriptor itself to find correspondence with the mono-scale target image.

A wide range of scalar and other metrics can be composed into a multi-scale feature pyramid, such as image intensity patches, color channel intensity patches, gradient magnitude, and gradient orientations. Histograms of textural features have been found useful as affine-invariant metrics as a part of a wider feature descriptor [144].

Sparse Predictive Search and Tracking

In a sparse predictive search pipeline, specific features at known locations, found in previous frames, are searched for in the next frame at the expected positions. For example, in the PTAM [319] algorithm for monocular depth sensing, a sparse 3D point cloud and camera pose are created from sequential video frames from a single camera by locating a set of interest points and feature descriptors. For each new frame, a *prediction* is made of the coordinates where the same interest points and feature detectors might be in the new image, using the prior camera pose matrix. Then, for the new frame, a search or tracking loop is started to locate a *small number* of the predicted interest points using a pyramid coarse to fine search strategy. The predicted interest points and features are searched for within a range around where each is predicted to be, and the camera pose matrix is updated based on the new coordinates where the features are found. Then, a *larger number* of points are predicted using the updated camera pose and a search and tracking loop is entered over a finer scale pyramid image in the set. This process iterates to find points and refine the pose matrix.

Tracking Region-Limited Search

One example of a region-limited search is a video conferencing system that tracks the location of the speaker using stereo microphones to calculate the coarse location via triangulation. Once the coarse speaker position is known, the camera is moved to view the speaker, and only the face region is of interest for further fine positional location adjustments, auto-zoom, auto-focus, and auto-contrast enhancements. In this application, the entire image does not need to be searched or processed for face features. Instead, the center of the FOV is the region where the search is limited to locate the face. For example, if the image is taken from an HD camera with 1920×1080 resolution, only a limited region in the center of the image, perhaps 512×512 pixels, needs to be processed to locate the face features.

Segmentation Limited Search

A segmented region can define the search area, such as a region with specific texture, or pixels of a specific color intensity. In a morphological vision pipeline, regions may be segmented in a variety of ways, such as thresholding and binary erosion + dilation to create binary shapes. Then the binary shapes can be used as masks to segment the corresponding gray scale image regions under the masks for feature searching. Image segmentation methods were covered in Chap. 2.

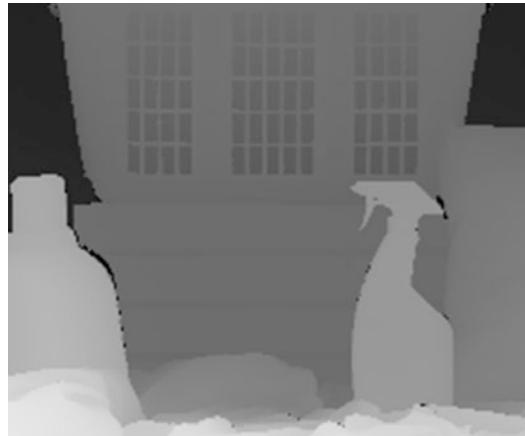


Figure 4.19 Segmentation of image regions based on a depth map. Depth image from Middlebury Data set (Source: D. Scharstein and C. Pal “Learning conditional random fields for stereo” CVPR Conference, 2007. Courtesy of authors)

Depth or Z Limited Search

With the advent of low-cost commercial depth sensors appearing on mobile consumer devices, the Z dimension is available for limiting search ranges. See Fig. 4.19. For example, by segmenting out the background of an image using depth, the foreground features are more easily segmented and identified, and search can be limited by depth segments. Considering how much time is spent in computer vision to extract 3D image information from 2D images, we can expect depth cameras to be used in novel ways to simplify computer vision algorithms.

Computer Vision, Models, Organization

This section contains a high-level overview of selected examples to illustrate how feature metrics are used within computer vision systems. Here, we explore how features are selected, learned, associated together to describe real objects, classified for efficient searching and matching, and used in computer vision pipelines. This section introduces machine learning, but only at a high level using selected examples. A good reference on machine learning is found in [528] by Prince. A good reference for computer vision models, organization, applications, and algorithms is found in Szelinski [316].

Several terms are chosen and defined in this section for the discussion of computer vision models, namely *feature space*, *object models*, and *constraints*. The main topics for this section include:

- Feature spaces and selection of optimal features
- Object recognition via object models containing features and constraints
- Classification and clustering methods to optimize pattern matching
- Training and learning

Note Many of the methods discussed in computer vision research journals and courses are borrowed from other tangent fields and applied, for example, machine learning and statistical analysis. In some cases computer vision is driving the research in such tangent fields. Since these fields are well established and considered beyond the scope of this work, we provide only a brief topical introduction here, with references for completeness [316, 528].

Feature Space

The collection and organization of all features, attributes, and other information necessary to describe objects may be called the *feature space*. Features are typically organized and classified into a feature space during a training or learning phase using ground truth data as a training set. The selected features are organized and structured in a database or a set of data structures, such as trees and lists, to allow for rapid search and feature matching at run time.

The feature space may contain one or more types of *descriptors* using spectra such as histograms, binary pattern vectors, as multivariate composite descriptors (see Varma [771] and Vedaldi [770] for more on multivariate descriptors). In addition, the feature space contains *constraints* used to associate sets of features together to identify objects and classes of objects. A feature space is unique to any given application, and is built according to the types of features used and the requirements of the application; there is no standard method.

The feature space may contain several *parameters* for describing objects; for example:

- **Several types of feature descriptors**, such as SIFT and simple color histograms.
- **Cartesian coordinates** for each descriptor relative to training images.
- **Orientations** of each descriptor.
- **Name of training image** associated with each descriptor.
- **Multimodal information**, such as GPS, temperatures, elevation, acceleration.
- **Feature sets** or lists of associated descriptors.
- **Constraints** between the descriptors in a set, such as the relative distance from each other, relative distance thresholds, angular relationships between descriptors, or relative to a reference point.
- **Object models** to collect and associate parameters for each object.
- **Classes** or associations of objects of the same type, such as automobiles.
- **Labels** for objects or constraints.

Object Models

The task of machine learning is creating models from data. As stated by Wittrock, “The brain is a model builder” (see Wittrock, M.C., Generative learning processes of the brain. Educational Psychologist, 27(4), 531–541). The human brain is an excellent learner, using input from the five senses of touch, smell, sight, taste, smell, as well as inputs from internal nerves and internal thoughts. All of these inputs feed into models we create to make decisions on actions and further thoughts. The models are believed to be true by our minds.

Rather than build models via machine learning, many successful vision systems are designed specifically by experts to solve a problem in hard-coded program logic, and the systems are tuned by experts until the desired goals are achieved. Therefore, machine learning is only one method used to create complete models, see chapters 9 and 10 for more details on machine learning.

Object model describes real objects or classes of objects using parameters from the feature space. For example, an object may contain all parameters required to describe a specific automobile, such as feature descriptor sets, labels, and constraints. A class of objects may associate and label all objects of the same class, such as an automobile of any type. There is no standard or canonical object model to follow, so in this section we describe the overall attributes of computer vision objects and how to model them. A *Generative model* generates model data within a class of objects; a *Discriminative model* differentiates between object classes.

Object models may be composed of sets of individual features; constraints on the related features, such as position or orientation of features within an object model; and perhaps other multimodal information for the objects or descriptors, such as GPS information or time stamps, as shown in Fig. 4.20. The object model can be created using a combination of supervised and unsupervised learning methods [385]; we survey several methods later in this chapter.

One early attempt to formulate object models is known as *parts-based models*, suggested in 1973 by Fischler and Elschlager [512]. These describe and recognize larger objects by first recognizing their parts—for example, a face being composed of parts such as eyes, nose, and mouth. There are several variations on parts-based models; see references [513–515], for example. Machine learning methods are also used to create the object models [528], and are discussed later in this section.

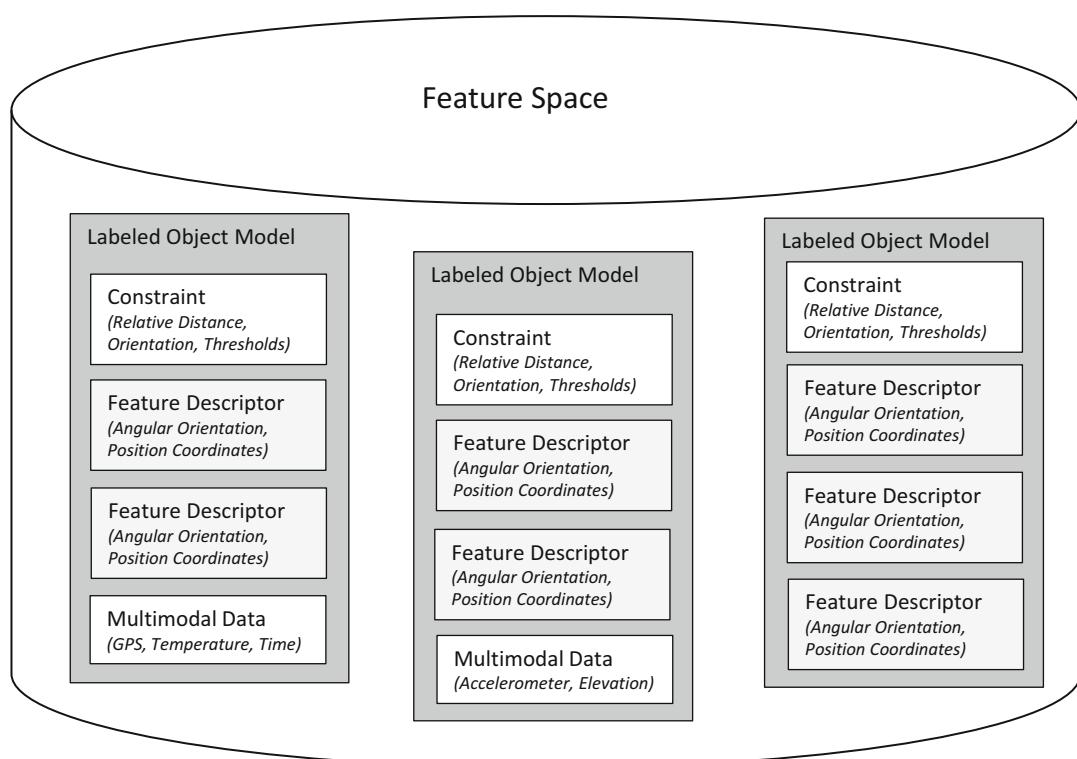


Figure 4.20 Simplified hypothetical feature space showing organization and association of features, constraints, and objects

A simple object model may be composed of only image histograms of whole images, the name or label of each associated image, and possibly a few classification parameters such as the subject matter of the image, GPS location, and date. To identify unknown target images, a histogram of the target image is taken and compared against image histograms from the database. Correspondence is measured using a suitable distance metric such as SAD. In this simple example, brute-force searching or a hash table index may be used to check each histogram in the database against target image histograms, and perhaps other parameters from the object model may be matched along with the histograms, such as the GPS coordinates. No complex machine learning classification, clustering, data reductions, or organization of the database need be done, since the search method is brute-force. However, finding correspondence will become progressively slower as more images are added to the database. And the histogram all by itself is not very discriminative and offers little invariance.

Constraints

Key to object recognition, *constraints* are used to associate and qualify features and related attributes as objects. Features alone are probably insufficient to recognize an object without additional qualification, including *neighborhood consensus* or *semi-local constraints* involving joint constraints, such as the angle and distance among a combined set of local features [340–342]. Constraints associate object model elements together to describe and recognize a larger object [357, 358, 361], such as by minimum feature count thresholds required to ensure that a proper subset of object features are found together, or by using multimodal data constraints such as GPS position, or by voting.

Since there are many approaches for creating constraints, we can only illustrate the concept. For example, Lowe [153] shows recognition examples illustrating how SIFT features can be used to recognize objects containing many tens of distinct features, in some cases using as few as two or three good features. This allows for perspective and occlusion invariance if some of the features describing the object cannot be found, taking into consideration feature orientation and scale as constraints. Another example is wide baseline stereo matching, which requires position and distance constraints on feature pairs in L/R image assuming that the scale and orientation of L/R feature pairs is about equal; in this case, translation would be constrained to be within a range based on depth.

Selection of Detectors and Features

Feature detectors are selected based on a combination of variables, such as the feature detector *design method* and the types of invariance and performance desired. Several approaches or design methods are discussed next.

Manually Designed Feature Detectors

Some feature detectors, such as polygon shape descriptors and sparse local features like SURF, are manually designed and chosen using the intuition, experience, and test results of the practitioner to address the desired invariance attributes for an application. This involves selecting the right spectra to describe the features, determining the shape and pattern of the feature, and choosing the types of regions to search. However, some detectors are statistically and empirically designed, which we cover next.

Statistically Designed Feature Detectors

Statistical methods are used to design and create feature detectors. For example, the binary sampling patterns used in methods such as ORB and FREAK are created from the training dataset based on the statistical characteristics of the possible interest point comparison pairs. Typically, ORB ranks each detected interest point feature pair combination to find terms that are uncorrelated with high variance. This is a statistical sorting or training process to design the feature patterns and tune them for a specific ground truth dataset. See Fig. 4.11 for more details on ORB, and see the discussions of FREAK and ORB earlier in this chapter as well.

SIFT also uses statistical methods to determine, from a training set, the best interest points, dominant orientation of each interest point, and scale of each interest point.

Learned Features

Many systems learn a unique codebook of features, using sparse coding methods to identify a unique set of basis features during a training phase against selected ground truth data. The learned basis features are specific to the application domain or training data, and the chosen detectors and descriptors may simply be pixel regions used as correlation templates. However, any descriptor may be used, such as SIFT. Neural network and convolutional network approaches are popularly used for feature learning, as well as sparse coding methods, which are introduced later in this chapter, and surveyed in detail in Chapters 9 and 10.

Overview of Training

A machine vision system is *trained* to recognize desired features, objects, and activities. Training may be supervised and assisted by an expert, or unsupervised as in the deep learning methods discussed later in this section. Here, we provide an overview of common steps and provide references for more detail. One of the simplest examples of training would be to take image histograms associated with each type of image—for example, a set of histograms that describe a face, animal, or automobile taken from different images.

Training involves collecting a training set of images appropriate for the application domain, and then determining which detectors and descriptors can be tuned to yield the best results. In some cases, the feature descriptor itself may be trainable and designed to match the training data, such as the local binary pattern descriptors ORB, BRIEF, and FREAK, which can use variable pixel sampling patterns optimized and learned from the training data.

In feature learning systems, the entire feature set is learned from the training set. Feature learning methods employ a range of descriptor methods such as simple correlation temples containing pixel regions, or SIFT descriptors. The learned feature set is reduced by keeping only the features that are significantly different from features already in the set. Feature learning methods are covered later in this chapter and in the Chap. 10 discussion on Feature Learning Architectures.

To form larger objects during training, sets of features may be associated together using constraints, such as geometric relationships like angles or distances between features, or the count of features of a given value within a specific region. Objects are determined during training, which involves running detectors and descriptors against chosen ground truth data to find the features, and then determining the constraints to represent objects as a composite set of features. Activities can be recognized by tracking features and their positions within adjacent frames.

In any case, the features obtained through the training phase are *classified* into a searchable feature space using a wide range of statistical and machine learning methods.

Classification of Features and Objects

Classification is a term sometimes used to describe the feature recognition, pattern recognition, or inference stage of the vision pipeline. Classification compares unknown incoming target features against the trained feature space. Several approaches are taken for automatically building classifiers, including support vector machines (SVMs), kernel machines, and neural networks, see Chap. 10.

In general, the size of the training set or ground truth dataset is key to classifier accuracy [328–330]. During system training, first a training set with ground truth data is used to build up the classifier, see Chap. 7 for a discussion on ground truth data. The machine learning community provides a wealth of guidance on training, so we defer to established sources. Key journals to dig deeper into machine learning and testing against ground truth data include NIPS and IEEE PAMI, the latter which goes back to 1979. Machine learning and statistical methods are used to guide the selection, classification, and organization of features during training. If no classification of the feature space is made, the feature match process follows a slow brute-force linear search of new features against known features.

Key classification problems discussed in this section include:

- **Group Distance and Clustering** of similar features using a range of nearest-neighbor methods to assist in organization, fitting, error minimization, searching and matching, and enabling similarity constraints such as geometric proximity, angular relationships, and multimodal cues.
- **Dimensionality Reductions** to avoid over-fitting, cleaning the data to remove outliers and spurious data, and reducing the size of the database.
- **Boosting and Weighting** to increase the accuracy of feature matching.
- **Constraints** describing relationships between descriptors composing an object, such as pose estimators and threshold accept/reject filters.
- **Structuring the Database** for rapid matching vs. brute-force methods.

Group Distance: Clustering, Training, and Statistical Learning

We refer to *group distance* and *clustering* in this discussion, sometimes interchangeably, as methods to describe similarities and differences between groups of data atoms, such as feature descriptors. Applications of group distance and clustering include error minimization, regression, outlier removal, classification, training, and feature matching.

According to Estivill-Castro [343], *clustering* is impossible to define in a mathematical sense, since there are so many diverse methods and approaches to describe a cluster. See Table 4.4 for a summary of related methods. However, we discuss clustering here in the context of computer vision to address data organization, pattern matching, and describing object model constraints (while attempting to not ruffle the feathers of mathematical purists who use different terminology).

To identify similar features in a group, a wide range of clustering algorithms or group distance algorithms are used [345], which may also be referred to as *error minimization* and *regression methods* in some literature. Features are clustered together for computer vision to help solve fundamental problems, including object modeling, finding similar patterns during matching, organizing and classifying similar data, and dimensionality reductions.

One way to describe a cluster is by *similarity*—for example, describing a cluster of related features under some distance metric or regression method. In this sense, clustering overlaps with distance functions: Euclidean distance for position, cosine distance for orientation, and Hamming distance for

binary feature vector comparisons are examples. However, distance functions between two points are differentiated in this discussion from group distance functions, clusters, and group distributions.

Efficiently organizing similar data in feature space for searching and classification is a form of clustering. It can be based on similarity or distance measures of feature vectors or on object constraint similarity, and it is required to speed up feature searching and matching. However, commercial databases and brute-force search may be used as-is for feature descriptors, with no attempt made to optimize. Custom data structures can be built for optimizations via trees, pyramids, lists, and hash tables. (We refer the reader to standard references in computer science covering data organization and searching; see the classic texts *The Art of Computer Programming* by Donald Knuth or *Data structure and Algorithms* by Aho, Ullman, and Hopcroft.)

Another aspect of clustering is the feature space dimension and topology. Since some feature spaces are multivariate and multidimensional, containing scalars and tensors, any strict definition of clustering, error minimization, regression, or distance is difficult; it really depends on the space in which similarity is to be measured.

Group Distance: Clustering Methods Survey, KNN, RANSAC, K-Means, GMM, SVM, Others

A spectrum of alternatives may be chosen for clustering and learning similarities between groups of data atoms, starting at the low end with basic C library searching and sorting functions, and reaching the high end with statistical and machine learning methods such as kernel machines and support vector machines (SVMs) to build complete classifiers, kernel machines are introduced Chapter 10 in the section Kernel Functions, Kernel Machines, SVM. Kernel machines allow various similarity functions to be substituted into a common framework to enable simplified comparison of similarity methods and classification.

Table 4.4 is a summary of selected clustering methods, with a few key references for the interested reader.

Classification Frameworks, REIN, MOPED

Training and classification fall into the following general categories:

- **Supervised.** A human will assist during the training process to make sure the results are correct.
- **Unsupervised.** The classifier can be trained automatically from feature data and parameters [385].

Putting all the pieces together, we see that training the classifiers may be manual or automated, simple or complex, depending on the complexity of the objects and the range of feature metrics used.

An SVM or kernel machine may be the ideal solution, or the problem may be simpler. For example, a machine vision system to identify fruit may contain a classifier for each type of fruit, with features including simple color histograms, shape factors such as area and perimeter and Fourier descriptors, and surface texture metrics, with constraints to associate and quantify all the features for each type of fruit. The training process would involve imaging several pieces of fruit of each type; developing canonical descriptors for color, shape, and surface texture; and devising a top-level classifier perhaps discriminating first on color, next surface texture, and finally shape. A simpler fruit classifier may contain just a set of image histograms of accurate color measurements for each fruit object, and may work well enough if each piece of fruit is imaged with a high-precision color camera against a black conveyor belt background in a factory.

While most published research is based on a wide range of nonstandard classification methods designed for specific applications or to demonstrate research results, some work is being done toward more standardized classification frameworks.

Table 4.4 Clustering, Classification, and Machine Learning Methods

Group Distance Criteria	Methods & References	Description
Distance	K-Nearest Neighbor [356]	Uses a chosen distance function, cluster based on simple distance to k-nearest neighbors in the training set.
Consensus Models	RANSAC [362] PROSAC [355] Levenberg-Marquardt [383]	Use random sample consensus to estimate model parameters from contaminated data sets.
Centroid Models	K-Means [346], Voronoi Tessellation, Delaunay Triangulation Hierarchical K-Means, Nister trees [369]	Use a centroid of distribution as the base of the cluster, which can be very slow for large datasets; can be formulated in a hierarchical tree structure using vocabulary words (Nister method) for much better performance.
Connectivity of Clusters	Hierarchical Clustering [347]	Builds connectivity between other clusters.
Density Models	DBSCAN [344, 377] OPTICS [378]	Locate distributions with maxima and minima density compared to surrounding data.
Distribution Models	Gaussian Mixture Models [348]	Iterative methods of finding maximum likelihood of model parameters.
Neural Methods	Neural Networks [352]	Neural methods defy a single definition, but typically use one or more inputs; adaptive weight factors for each input that can be learned and trained, a neural function to act on the inputs and weights, a bias factor for the neural function; produce one or more outputs.
Bayesian	Naïve Bayesian [365] Randomize Trees [366] FERNS [299]	Learning model recording probabilistic relationships between variables.
Probabilistic, Semantic	[224] Latent Semantic Analysis (pLSA) Latent Dirichlet Allocation (LDA) Hidden Markov Models, HMM [367, 368]	Learning model based on probabilistic relationships between variables.
Kernel Methods, Kernel Machines	Kernel Machines [353] ¹ Various Kernels [354] PCA [349, 350] *SVM is a well-known instance of a kernel machine.	Reduce a distribution to a set of uncorrelated, ranked principal components in a Euclidean space for ease of matching and clustering.
Support Vector Machines	SVM [351]	An SVM may produce structured or multivariate output to classify input.

¹<http://www.kernel-machines.org/>

One noteworthy example of a potentially standard classifier framework developed for robot navigation and object recognition is the REIN method [379], which allows the mixing and matching of detectors, descriptors, and classifiers for determining constraints. REIN provides a plug-in architecture and interfaces to allow for any algorithms, such as OpenCV detectors and descriptors, to be combined in parallel or serial pipelines. Two classification methods are available in REIN as plug-in modules for concurrent use: *Binarized Gradient Grid Pyramids* are introduced as a new method [379], and *View Point Feature Histograms* [380] are also used.

The REIN pipeline provides interfaces for (1) *attention operators* to identify interesting 3D points and reduce the search space; (2) *detectors* for creating feature descriptors; and (3) *pose estimators* to determine geometric constraints for applications like robot motion such as grasping. REIN is available for research as open source; see reference [379].

Another research project, MOPED [381], provides a regular architecture for robotic navigation, including object and pose recognition. MOPED includes optimizations to use all available CPU and GPU compute resources in parallel. Moped provides optimized versions of SIFT and SURF for GPGPU, and makes heavy use of SSE instructions for pose estimation.

Kernel Machines

In machine learning, a *kernel machine* [354] is a framework allowing a set of methods for statistically clustering, ranking, correlating, and classifying patterns or features to be automated. One common example of a kernel machine is the support vector machine (SVM) [333].

The framework for a kernel machine maps descriptor data into a feature space, where each coordinate in the feature space corresponds to a descriptor. Within the feature space, feature matching and feature space reductions can be efficiently carried out using *kernel functions*. Various kernel functions are used within the kernel machine framework, including RBF kernels, Fisher kernels, various polynomial kernels, and graph kernels.

Once the feature descriptors are transformed into the feature space, comparisons, reductions, and clustering may be employed. The key advantage of a kernel machine is that the kernel methods are interchangeable, allowing for many different kernels to be evaluated against the same feature data. There is an active kernel machine community (see kernel-machines.org). See also Chap. 10 for more on SVMs and kernel descriptors.

Boosting, Weighting

Boosting [363] is a machine learning concept that allows a set of classifiers to be used together, organized into combinatorial networks, pipelines, or cascades, and with learned weights applied to each classifier. This results in a higher, synergistic prediction and recognition capability using the combined weighted classifiers. Boosting is analogous to the weighting factors used for neural network inputs; however, boosting methods go further to combine networks of classifiers to create a single, strong classifier.

We illustrate boosting from the Viola–Jones method [138, 178] also discussed in Chap. 6, which uses the ADA-BOOST training method to create a cascaded pattern matching and classification network by generating strong classifiers from many weak learners. This is done through dynamic weighting factors determined in a training phase, and the method of using weighting factors is called *boosting*.

The idea of boosting is to first start out by equally weighting the detected features—in this case, HAAR wavelets—and then matching the detected features against the set of expected features; for example, those features detected for a specific face. Each set of weighted features is a classifier. Classifiers that fail to match correctly are called *weak learners*. For each weak learner during the training phase, new weighting factors are applied to each feature to make the classifier match correctly. Finally, all weak learners are combined linearly into a *cascaded classifier*, which is like a pipeline or funnel of weak classifiers designed to reject bad features early in the pipeline.

The training can take many hours, days or weeks and requires some supervision. While ADA-BOOST solved binary classification problems, the method can be extended into multiclass classification [364].

Table 4.5 Comparison of Various Interest Point, Descriptor, and Classifier Concepts

Technique	FERNS	SIFT	FREAK	Convolutional Network	Polygon Shape Factors
Sparse Keypoints	x	x	x	x	
Feature Descriptor		x	x	x	x
Multi-Scale Representation	x	x		x	
Coarse to Fine Descriptor			x		
Deep Learning Network				x	
Sparse Codebook				x	

Note: The FERNS method does not rely on a local feature descriptor, and instead relies on a classifier using constraints between interest points.

Selected Examples of Classification

We call out a few noteworthy and popular classification approaches here, which are also listed in Table 4.5. See Chap. 10 for more information on feature learning architectures, including classification methods such as visual vocabularies.

Randomized trees is a method using hierarchical patch classifiers [366] based on Bayesian probability methods, taking a set of simple patch features deformed by random homography parameters. Ozuyusal et al. [299] further develop the randomized tree method with optimizations using non-hierarchical organization in the form of *FERNS*, using binary probability tests for patch classifier membership. Matches are evaluated using a naïve Bayesian approach.

FERNS training [299] involves combining training data from multiple viewpoints of each patch to add scale and perspective invariance, using trees with 11 levels and 11 versions of each patch, warped using randomized affine deformation parameters; some Gaussian noise and smoothing are also applied to the deformed patches. Keypoints are then located in each deformed patch, and the keypoints found in the most deformed patches are selected for the training set. The FERNS keypoints use maxima of Laplacian filters at three scales and retain only the strongest 400 keypoints. The Laplacian keypoints do not include orientation or fine-scale estimation. FERNS does not use descriptors, just the strongest Laplacian keypoints computed over the 11 deformed images in each set.

While K-means [346] methods can be very slow, an optimization using hierarchical Nister Trees [369] is a highly scalable alternative for indexing massive numbers of quantized or clustered local descriptors in a hierarchical vocabulary tree. The method is reported to be very discriminative and has been tested on large datasets.

Binary Histogram Intersection Minimization (BHIM) [314] uses pairs of multi-scale local binary patterns (MSLBP) [314] to form pairwise-coupled classifiers based on strong divergence between pairs of MSLBP features. Histogram intersection on pairs of MSLBP features use a distance function such as SAD to find the largest divergence of histogram distance. The BHIM classifier is then composed of a list of “pairs” of MSLBP histograms with large divergence, and MSLBPs are matched into the classifier. BHIM uses features created across multiple scales of training data. It is reported by the authors to be at least as accurate as ADA-BOOST, and the MSLBP features are reported to be more discriminant than LBPs.

Alahi et al. [373] develop a method for classification and matching using a cascaded set of coarse to fine grids of region descriptors called *object descriptors* (ODs). The target application is tracking objects across a set of cameras, such as traffic cameras in a metropolitan area. Each OD is a collection of multi-scale descriptors computed in equal-size regions over multi-scale grids; the grids range over six scales with a 25 % scaling factor difference. Any existing descriptor method can be used in the

OD method, such as SIFT, SURF, or correlation templates. The authors [373] claim improved performance by cascading descriptors in an OD compared with using existing descriptors.

Feature Learning, Sparse Coding, Convolutional Networks

Feature learning methods create a set of basis features (we use the term *basis features* loosely here) derived from the ground truth data during a training phase. The basis features are collected into a set. There are several related approaches taken to create the set, discussed in this section.

The topics introduced in this section are covered in much more detail in Chap. 10 under Feature Learning Architectures.

Terminology: Codebooks, Visual Vocabulary, Bag of Words, Bag of Features

Several related approaches and terminologies are used in the feature learning literature, including variations such as *sparse coding*, *codebooks*, *bag of words*, and *visual vocabularies*. However, for the novice, there is some conceptual overlap in the various approaches and the terminology is subtle, describing minor variations in methods used to learn the features and build the classification networks; see references [106–111]. The sparse codes are analogous to basis features. Many researchers in the areas of activity recognition [61, 67] are using sparse codebooks and extending the field of research.

We describe some of the terminology and concepts, including:

- Dictionaries, codebooks, visual vocabularies, bags of words, bags of features, and feature alphabet, containing sets of features.
- Sparse codes, sparse coding, and minimal sets of features or codes.
- Multilayered sparse coding and deep belief networks, containing multilayered classification networks for hierarchical matching; these are composed of small, medium, and large scale features—perhaps ten or more layers of scale.
- Single-layer sparse coding, with no hierarchy of features, which may be built on top of a multi-scale descriptor such as SIFT.
- Unsupervised feature learning, including various methods of learning the best features for a given application from the ground truth dataset; feature learning has received much attention recently in the Neural Information Processing Systems (NIPS) community, especially as applied to convolutional networks.

Sparse Coding

Some early work in the area of sparse coding for natural images can be found in the work of Olshausen and Field [118], which forms the conceptual basis. To create a *sparse codebook*, first an image feature domain is chosen, such as face recognition or automobile recognition. Then a set of *basis items* (patches, vectors, or functions) are selected and put into a codebook based on a chosen uniqueness function. The sparse coding goal is to contain the smallest set of unique basis items required to achieve the accuracy and performance goals for the system.

When adding a new feature to the codebook during the training stage, candidate features are compared against the features already in the codebook to determine feature uniqueness, using a suitable distance function and empirical threshold. If the feature is sufficiently unique, as measured by the distance function and a threshold, the new feature is added to the codebook.

In work by Bo, Ren, and Fox [116], the training phase for learning features involves using objects such as a cup, which is positioned on a small rotating table. Multiple images are taken of the object from a number of viewpoints and distances to achieve perspective invariance, which then yields a set of patches taken from a variety of poses, from which the unique sparse codewords are created and added to the codebook. See also references [116, 217, 218, 229]. Related work includes a histogram of sparse codes descriptor or HSC [117], as described in Chap. 7, used to retrofit a HOG descriptor. See Chap. 10 for more details on sparse coding architectures.

Visual Vocabularies

Visual vocabularies are analogous to word vocabularies and they share common research [223]. See Chap. 10 for more details on vocabulary architectures. In the area of document analysis, content is analyzed and described based on the histogram of unique word counts in the document. Of course, the histogram can be trimmed and remapped to reduce the quantization and binning. Visual vocabularies follow the same method as word vocabulary methods, representing images globally by the frequency of visual words, as illustrated in Fig. 4.21, where visual word methods use feature descriptors of many types.

To build the visual vocabularies, unique features descriptors are extracted and collected from ground truth images. To be included in the vocabulary, the new feature must have significant statistical differences from the existing features in the vocabulary, so features are added to the vocabulary only if they exceed a difference threshold function.

To quantize the visual vocabulary features for determining their uniqueness, clustering and classification methods are performed on the feature set, and candidate features are selected that are unique so as to reduce the feature space and assist in matching speed. Various statistical methods may be employed to reduce the feature space, such as K-means, KNN, SVM, Bayes, and others.

To collect the visual features, practitioners are using all possible methods of feature description and image search, including sampling the image at regular grids and at interest points, as well as scale space searches. The features used in the vocabularies range from simple rectangular pixel regions, to SIFT features, and everything in between. Applications for the visual vocabularies range from analyzing spatiotemporal images for activity recognition [224, 227] to image classification [108, 110, 225–227].

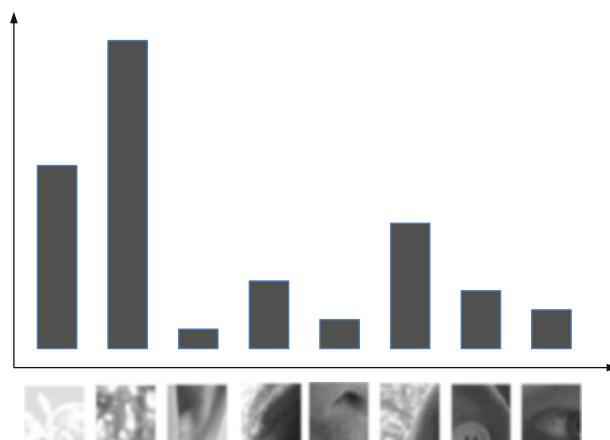


Figure 4.21 Hypothetical, simplified illustration representing a set of visual words, and a histogram showing frequency of use of each visual word in a given image.

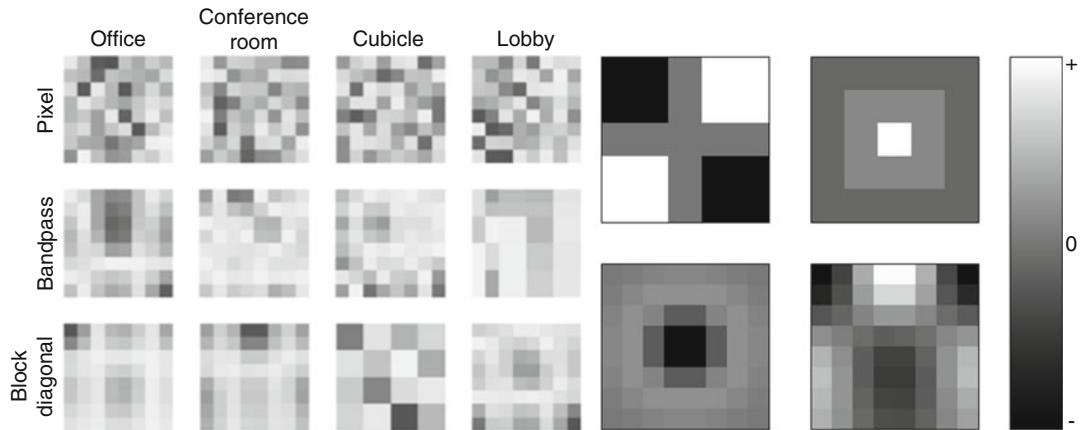


Figure 4.22 (Left) The optimal learned convolution filters for an image of an Office, a conference room, cubicle, and lobby; gray scale values represent filter coefficient magnitudes. (Right) Comparable corner detectors in the top row, difference of Gaussian in the bottom left, and a custom filter which is preferred by the author (Images © Andrew Richardson and Edwin Olson, used by permission)

Learned Detectors via Convolutional Filter Masks

As illustrated in Fig. 4.22, Richardson and Olson [114] developed a method of learning optimal convolutional filters as an interest point detector with applications to stereo visual odometry. This method uses combinations of DCT and HAAR basis features composed together, using random weights to form a set of candidate 8×8 pixel basis functions, each of which is tested against a target feature set resembling 2D barcodes known as AprilTags [509]. Each 8×8 pixel candidate is measured against the AprilTags to find the best convolution masks for each tag to form the basis set. Of course, other target features such as corners could be used for ground truth data instead of AprilTags.

Using the learned convolution masks, the steps in feature detection are as follows: (1) convolve each mask at chosen pixels to get a response; (2) compare convolution response against a threshold; (3) suppress non-extrema response values using a 3×3 spatial filter window. The authors report good accuracy and high performance on the order of a FAST detector, but with the benefit of higher performance for the combined detection and non-maximal suppression stage as feature counts increase.

Convolutional Neural Networks, Neural Networks

Convolutional neural networks which are discussed at length in Chapters 9 and 10, pioneered by Lecun [331] and others, are one method of implementing machine learning algorithms based on neural network theory [352]. Convolutional networks are showing great success in academia and industry [332] for image classification and feature matching.

Convolutional neural networks are one method of modeling a neural network. The main compute elements in the convolutional network are many optimized convolutions in parallel, as well as fast local memory between the compute units. The run-time classification performance can be quite fast, especially for hardware-optimized implementations [510].

As shown in Fig. 4.23 at a high level, one method of modeling each neuron and a network of neurons includes a set of inputs, a set of weighting factors applied to each input, a combinatorial function, and an output. Many neural models exist that map into convolutional networks, we refer the reader to the experts, see Lecun [331]. Neural networks have been devised using several models, but this topic is outside the scope of this work [352]; see the NIPS community research for more.

Neural networks are multilevel, containing several layers and interconnections. As shown in the hypothetical neural network in Fig. 4.23, a bias input is provided to each neural function as a

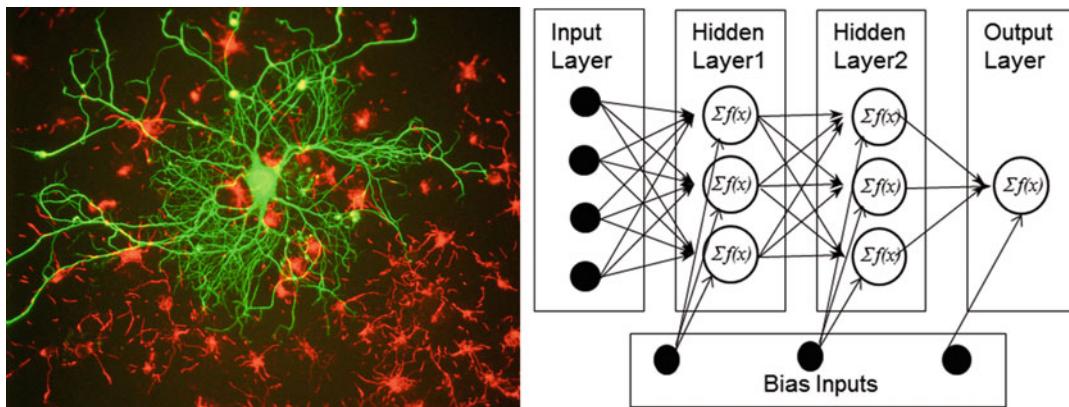


Figure 4.23 (Left) Neurons from a human brain. (Right) One of many possible models of an artificial neural network [352]. Note that each neuron may have several inputs, several outputs, a bias factor, and input/output weight factors (not shown). Human neuron image on left © Gerry Shaw, used by permission

weighting factor. Some neural network configurations use individual weights applied to each individual input, so the weighting factors act as convolution kernel coefficients. In terms of convolutional networks, the neural network paradigm can be mapped into localized patches of raw pixels as feature inputs at the lowest level. For example, the patch size may be 1 pixel or a 5×5 patch of pixels, each input having a convolutional weighting factor.

See Chap. 9 for details on neural networks, including historical background and neuroscience concepts. See Chap. 10 for feature learning architectures employing neural networks.

Summary

In this chapter, we survey background concepts and ideas used to create local feature descriptors and interest point detectors. The key concepts and ideas are also developed into the vision taxonomy suggested in Chap. 5. Distance functions are covered here, as well as useful coordinate systems. We examine the shape and pattern of local descriptors, with an emphasis on local binary descriptors such as ORB, FREAK, and BRISK to illustrate the concepts.

Feature descriptor discrimination is illustrated using image reconstructions from feature descriptor data alone. Search strategies are discussed, such as scale space pyramids and multilevel search, as well as other methods such as grid limited search. Computer vision system models are covered, including concepts such as feature space, object models, feature constraints, statistically designed features, and feature learning. Classification and training are illustrated using several methods, including kernel machines, convolutional networks, and deep learning. Several references to the literature are provided for the interested reader to dig deeper. Practical observations and considerations for designing vision systems are also provided.

In summary, this chapter provides useful background concepts to keep in mind when reading the local feature descriptor survey in Chap. 6, since the concepts discussed here are taken mainly from the current local descriptor methods in use; however, some additional observations and directions for future research are suggested in this chapter as well.

Chapter 4: Learning Assignments

1. Discuss how a local feature descriptor is different than a global image descriptor or global statistical metric, and provide an example comparison between a local feature descriptor and a global feature descriptor.
2. A *feature detector* is equivalently called a local interest point, anchor point, and landmark. Discuss what a feature detector is used for, and describe in general how they work.
3. Discuss how to cull down the set of local interest points (feature detectors), and why culling is critical for effective feature description.
4. Discuss and compare alternatives to using feature detectors to find sparse local interest points, such as using dense feature descriptors computed at each pixel, or grid-aligned feature descriptors.
5. Discuss why it is critical to pair the right combination of feature detector and feature descriptor together.
6. Discuss the difference between feature description and feature extraction.
7. Discuss feature invariance criteria such as scale and rotational invariance, and name at least five (5) other invariance criteria.
8. Discuss why determining the invariance criteria in advance is critical for selecting the interest point and feature descriptor methods for a given application, and describe an example application and describe the relevant invariance criteria.
9. Discuss why interest points, or feature detectors, should be distinct and easy to find with high repeatability.
10. Describe as many distance functions as you can remember, at least Euclidean distance, cosine distance, and SSD difference.
11. Describe how a distance function is used to measure correspondence between feature descriptors to feature matching.
12. Describe how Hamming distance works, and why it is ideal for measuring correspondence between local binary descriptors.
13. Discuss scale space, image pyramids, and feature descriptor pyramids, and provide example applications for each.
14. Discuss feature descriptor shapes, otherwise referred to as *patches*, and the advantages and disadvantages of each shape.
15. Discuss the motivations and goals behind the design of local binary descriptors.
16. Compare local binary descriptors with other feature descriptor methods using other spectra such as pixel values or gradient values.
17. Discuss several examples of spectra used to create feature descriptors, such as gradients, pixel values, and color information.
18. Describe how saccadic dithering is used by the human visual system.
19. Describe the shape of the variable levels of detail detected by the retina, and describe applications variable level of detail to feature descriptor design.
20. Discuss the approach to determining the pixel sampling patterns used in local binary descriptors, including *dense* sample patterns used in the LBP, and *sparse* point-pair pixel patterns used by FREAK and ORB, and discuss the motivations and goals for each approach.
21. Discuss applications for sub-pixel accuracy in feature descriptors, and name specific feature descriptor methods which have been demonstrated to be sub-pixel accurate.
22. Describe feature search approaches including multi-scale image pyramid search, dense pixel search, grid tile search, and sparse local interest point search.

23. Describe image classification, and labeled classes.
24. Describe how clustering of features is used during training to select representative features.
25. Describe how K-MEANS and K-NN (K-nearest neighbor) clustering methods work at a high level, and how they are different.
26. Describe the general operation and goals of the ADA-BOOST method used by the Viola–Jones Method.
27. Describe sparse coding goals as applied to feature classification.
28. Describe visual vocabularies and bag-of-words methods as applied to feature classification.

Taxonomy of Feature Description Attributes

5

“for the Entwives desired order, and plenty, and peace (by which they meant that things should remain where they had set them).”

—J. R. R. Tolkien, The Lord of the Rings

This chapter develops a general *Vision Metrics Taxonomy* for feature description, so as to collect summary descriptor attributes for high-level analysis. The taxonomy includes a set of general *robustness criteria* for feature description and ground truth datasets. The material presented and discussed in this book follows and reflects this taxonomy. By developing a standard vocabulary in the taxonomy, terms and techniques are intended to be consistently communicated and better understood. The taxonomy is used in the survey of feature descriptor methods in Chap. 6 to record “*what*” practitioners are doing.

As shown in Fig. 5.1, the Vision Metrics Taxonomy is based on feature descriptor dimensions using three axes—shape and pattern, spectra, and density—intended to create a simple framework for analysis and discussion. A few new terms and concepts have been introduced where there had been no standard, such as for the term feature descriptor families. These have been broken down into categories of local binary descriptors, spectra descriptors, basis space descriptors, and polygon shape descriptors; these descriptor families are also discussed in detail in Chap. 4. Additionally, the taxonomy borrows some useful terminology from the literature when it exists there, including several terms for the robustness and invariance attributes.

Why create a taxonomy that is guaranteed to be fuzzy, includes several variables, and will not perfectly express the attributes of any feature descriptor? The intent is to provide a framework to describe various design approaches used for feature description. However, the taxonomy is not intended to be used for comparing descriptors in terms of their goodness, performance, or accuracy.

The three axes of the Vision Metrics Taxonomy in Fig. 5.1 are:

1. **Shape and pattern:** How the pixels are taken from the target image.
2. **Density:** The extent of the image required for the descriptor, differentiating among local, regional, and global descriptors.
3. **Spectra:** The scalar and vector quantities used for the metrics, and a summary breakdown of the algorithms and computations.

Feature Descriptor Families

Feature descriptors and metrics have developed along several lines of thinking into separate families. In many cases, the research communities for the various families are working on different problems, and there is little cross-pollination or mutual interest. For example, cell biology and medical

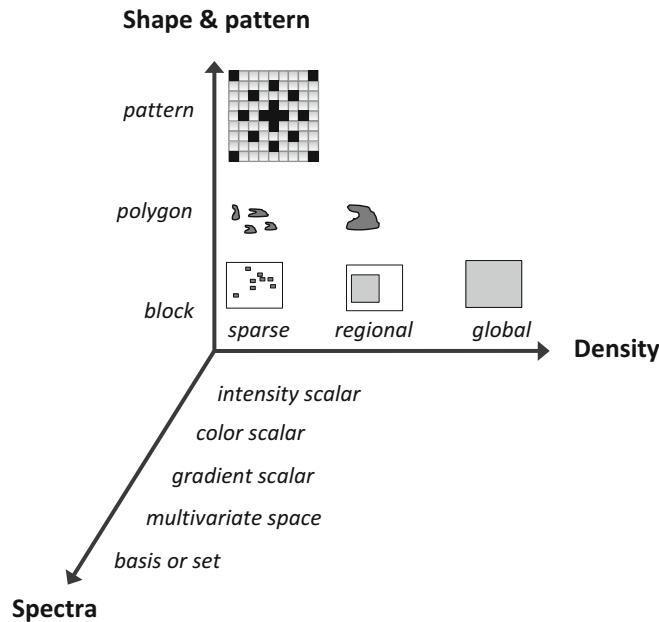


Figure 5.1 Taxonomy for feature descriptor dimensions, including (1) feature density as global, regional, and sparse local; (2) shape and pattern of pixels used to compute the descriptor, which includes rectangles, circles, and sparse sampling patterns; (3) spectra, which includes the spectrum of information contained in the feature itself

applications are typically interested in polygon shape descriptors, also referred to in the literature as image moments. Those involved with trendy augmented reality applications for mobile phones, as discussed in the computer vision literature, may be more interested in local binary descriptors. In some cases, there are common concepts shared by feature detectors and feature descriptors, as will be discussed in detail in Chap. 6; these include the use of gradients and local binary patterns.

Based on the taxonomy shown in Fig. 5.1, we divide features into the following families:

- **Local Binary Descriptors.** These sample point-pairs in a local region and create a binary coded bit vector, 1 bit per compare, amenable to Hamming distance feature matching. Examples include LBP, FREAK, ORB, BRISK, Census.
- **Spectra Descriptors.** These use a wide range of spectra values, such as gradients and region averages. There is no practical limit to the spectra that could be used with these features. One of the most common spectra used in detectors is the local region gradient, such as in SIFT. Gradients are also used in several interest point and edge detectors, such as Harris, Sobel.
- **Basis Space Descriptors.** These methods encode the feature vector into a set of basis functions, such as the familiar Fourier series of sine and cosine magnitude and phase. In addition, existing and novel basis features are being devised in the form of sparse codebooks and visual vocabularies (we use the term basis space loosely).
- **Polygon Shape Descriptors.** These take the shape of objects as measured by statistical metrics, such as area, perimeter, and centroid. Typically, the shapes are extracted using a morphological vision pipeline and regional algorithms, which can be more complex than localized algorithms for feature detectors and feature descriptors (as will be discussed in Chap. 8). *Image moments* [500] is a term often used in the literature to describe shape features.

Prior Work on Computer Vision Taxonomies

Several research papers compare and contrast various aspects of sparse local features, and the field is rich with examples of comparisons of keypoint detectors [85, 298] and feature descriptors [99, 137]. New feature descriptor methods and improvements are usually compared to existing methods, utilizing several robustness and invariance criteria. However, there is a lack of formal taxonomy work to highlight the subtle details affecting design and comparison. For a good survey covering state-of-the-art computer vision methods, see Szelinski [316].

It should be noted that computer vision is a huge field. Several thousand research papers are published every year, and several thousand equally interesting research papers are rejected by conference publishers. Here are a few noteworthy works that survey and organize the field of feature metrics and computer vision.

- **Affine Covariant Interest Point Detectors.** A good taxonomy is provided by Mikolajczyk et al. [145] for affine covariant interest point detectors. Also, Lindberg [142] has studied the area of scale independent interest point methods extensively. We seek a much richer taxonomy, however, to cover design principles for feature descriptors, and we have developed our taxonomy around families of descriptor methods with common design characteristics.
- **Annotated Computer Vision Bibliography.** From USC and maintained by Keith Price, this resource provides a detailed breakdown of computer vision into several branches, as well as links to some key research in the field and computer vision resources.¹
- **CVonline: The Evolving, Distributed, Nonproprietary, Online Compendium of Computer Vision.** This provides a comprehensive and detailed list of topics in computer vision. The website is maintained by Robert Fisher, and indexes the key Wikipedia articles. This may be one of the best online resources currently available.²
- **Local Invariant Feature Detectors: A Survey.** Prepared by Tinne Tuytelaars and Krystian Mikolajczyk [99], this reference provides a good overview of several feature description methods, as well as a discussion of literature on local features, performance and accuracy evaluations of several methods, types of methods (corner detectors, blob detectors, feature detectors), and implementation details.

Robustness and Accuracy

A key goal for computer vision is *robustness*, or the ability of a feature to be recognized under various conditions. Robustness can be broken down into several attributes. For example, detecting a feature should be robust over various criteria that are critical to a given application, such as scale, rotation, or illumination. We might also use the terms *invariant* or *invariance* to describe robustness. The end goal is accurate localization, correspondence, and robustness under invariance criteria.

However, some robustness attributes are dependent on the feature descriptor combined with other variables. For example, many local feature descriptor methods compute position and orientation based on a chosen interest point method, so the descriptor accuracy is interrelated with the interest point method. The distance function and classification method are interrelated as well, to determine final accuracy.

¹ <http://iris.usc.edu/Vision-Notes/bibliography/contents.html>.

² <http://homepages.inf.ed.ac.uk/rbf/CVonline/CVentry.htm>.

Note Since it is not possible to define robustness or accuracy of a feature descriptor in isolation from the interest point method, the classifier, and the distance function, the opportunity exists to mix and match well-known detectors and descriptors, combined with various classifiers, to yield the desired robustness and accuracy.

Robustness and accuracy are a combination of the following factors:

1. **Interest point accuracy**, since many descriptors depend on the keypoint location and orientation.
2. **Descriptor accuracy**, as each descriptor method varies, and can be tuned.
3. **Classifier and distance function accuracy**, as a poor classifier and matching stage can lead to the wrong results.

Part of the challenge for an application, thus, is to define the robustness criteria, attribute by attribute, and then to define the limits and bounds of invariance sought. For example, scale invariance from $1\times$ to $100\times$ magnification may not be needed and hardly possible, but scale invariance from $1\times$ to $4\times$ may be all that is needed and much simpler to reach.

Several attributes of robustness are developed here into a robustness taxonomy. To determine actual robustness, ground truth data is needed as a basis to check the algorithms and measure results. Chapter 7 provides a background in ground truth data selection and design.

General Robustness Taxonomy

Robustness criteria can be expressed in terms of attributes and measured as invariance or robustness to those attributes. (See Chap. 7, Table 7.1, for more information on each of the robustness criteria attributes, with considerations for creating ground truth datasets.) Robustness criteria and attributes are grouped under the following group headings:

- Illumination
- Color
- Incompleteness
- Resolution and distance
- Geometric distortion
- Discrimination and uniqueness

Each robustness criterions group contains several finer-grain attributes, as illustrated in Fig. 5.2.

Illumination	Color	Incompleteness	Resolution, accuracy	Geometric	Discrimination, uniqueness
<ul style="list-style-type: none"> • Uneven illumination • Brightness • Contrast • Vignette 	<ul style="list-style-type: none"> • Color Space Accuracy • Color Channels • Color Bit Depth 	<ul style="list-style-type: none"> • Clutter • Occlusion • Outliers, proximity • Noise • Motion blur • Jitter, Judder 	<ul style="list-style-type: none"> • Location accuracy or position • Shape & thickness distortion • Focal plane or depth • Pixel Depth Resolution 	<ul style="list-style-type: none"> • Scale • Rotation • Geometric warp • Reflection • Radial distortion • Polar distortion 	<ul style="list-style-type: none"> • Quality

Figure 5.2 General robustness criteria and their attributes

Let us take a look at these robustness attributes, along with some practical considerations for design and implementation of feature descriptors and the corresponding ground truth data to address the attributes.

Illumination

Light is the source of all imaging, and it should be the no.1 priority area for analysis and consideration when setting requirements for a given application. Illumination has several facets and is considered separately from color and color spaces. In some cases, the illumination can be corrected by changing the light source, or by adding or relocating light sources. In other cases, image preprocessing is needed to correct the illumination to prepare the image for further analysis and feature extraction.

Attention to illumination cannot be stressed enough; for example, see Fig. 4.2 showing the effects of preprocessing to change the illumination in terms of increasing the contrast for feature extraction. Key illumination attributes are:

- **Uneven illumination:** image contains dark and bright regions, sometimes obscuring a feature that is dependent on a certain range of pixel intensities.
- **Brightness:** there is too much or too little total light, affecting feature detection and matching.
- **Contrast:** intensity bands are too narrow, too wide, or contained in several bands.
- **Vignette:** light is distributed unevenly, such as dark around the edges.

Color Criteria

When color is used, accuracy of color is critical. Color management and color spaces are discussed in Chap. 2, but some major considerations are:

- **Color space accuracy:** which color space should be used—RGB, YIQ, HSV, or a perceptually accurate color space such as CIECAM02 Jch or Jab? Each color space has accuracy and utility considerations, such as the ease of transforming colors to and from color spaces.
- **Color channels:** since cameras typically provide RGB data, extracting the gray scale intensity from the RGB data is often important. There are many methods for converting RGB color to gray scale intensity, and many color spaces to choose from.
- **Color bit depth:** color information, when used, must be accurate enough for the application. For example, 8-bit color may be suitable for most applications, unless color discrimination is necessary, so higher precision color using 10, 12, 14, or 16 bits per channel may be needed.

Also, depending on the camera sensor used, there will be signal characteristics, such as color sensitivity and dynamic range, which differ for each color channel. For demanding color-critical applications, the camera sensor should be well understood and have a known method of calibration. Individual colors may need to be compensated during image pre-processing. (See Chap. 1 for a discussion of camera sensors.)

Incompleteness

Features are not always presented in the image from frame to frame the way they are expected, or in the way they were learned. The features may appear to be incomplete. Key attributes of incompleteness include:

- **Clutter:** the feature is obscured by surrounding image features, and the feature aliases and blends into the surrounding pixels.
- **Occlusion:** the feature is partially hidden; in many cases the application will encounter occluded features or sets of features.

- **Outliers, proximity:** sometimes only features in certain regions are used, and outlying features must be detected and ignored.
- **Noise:** can come from rain, bad image sensors, and many other sources. A constant problem, noise can be compensated for, if it is understood, using a wide range of filter methods during preprocessing.
- **Motion blur:** if it is measured and understood, motion blur can be compensated for using filtering during preprocessing.
- **Jitter, judder:** a motion artifact, jitter or judder can be corrected, but not always; this can be a difficult robustness criterion to meet.

Resolution and Accuracy

Robustness regarding resolution, scale, and distance is often a challenge for computer vision. This is especially true when using feature metrics that rely on discrete pixel sizes over which the pixel area varies with distance. For example, feature metrics that rely on pixel neighborhood structure alone do not scale well or easily, such as correlation templates and most local region kernel methods. Other descriptors, such as those based on shape factors, may provide robustness that pixel region structures cannot achieve. Depending on the application, more than one descriptor method may be required to handle resolution and scale.

To meet the challenge of resolution and distance robustness, various methods are employed in practice, such as scale-space image pyramid collections and feature-space pyramids, which contain multi-scale representations of the feature. Key criteria for resolution and distance robustness include:

- **Location accuracy or position:** how close does the metric need to provide coordinate location under scale, rotation, noise and other criteria? Is pixel accuracy or sub-pixel accuracy needed? Regional accuracy methods of feature description cannot determine positional accuracy as well; for example, methods that use HAAR-like features and integral images can suffer the most, since in computing the HAAR rectangle, all pixels in the rectangle are summed together, throwing away discrimination of individual pixel locations. Pixel-accurate feature accuracy can also be challenging, since as features move and rotate they distort, and the pixel sampling artifacts create uncertainty.
- **Shape and thickness distortion:** distance, resolution, and rotation combine to distort the pixel sample shapes, so a feature may appear to be thicker than it really is or thinner. Distortion is a type of sampling artifact.
- **Focal plane or depth:** depending on distance, the pixel area covered by each pixel changes size. In this case, depth sensors can provide some help when used along with RGB or other sensors.
- **Pixel depth resolution:** for example, processing color channels to preserve the bit accuracy using float or unsigned short int as a minimum can be required.

Geometric Distortion

Perhaps the most common distortion of image features is geometric, since geometric distortions take many forms as the camera moves and as objects move. Geometric attributes for robustness include the following:

- **Scale:** distance from viewpoint, a commonly addressed robustness criteria.
- **Rotation:** important in many applications, such as industrial inspection.
- **Geometric warp:** key area of research in the fields of activity recognition and dynamic texture analysis, as discussed in Chaps. 4 and 6.

- **Reflection:** flipping the image by 180°.
- **Radial distortion:** a key problem in depth sensing and also for 2D camera geometry in general, since depth fields are not uniform or simple; see Chap. 1.
- **Polar distortion:** a key problem in depth sensing geometry; see Chap. 1.

Efficiency Variables, Costs and Benefits

We consider efficiency to be related to compute, memory, and total invariance attributes provided. How efficient is a feature descriptor or feature metric? How much compute is needed to create the metric? How much memory is needed to store the metric? How accurate is the metric? How much robustness and invariance are provided vs. the cost of compute and memory? To answer the above questions is very difficult and depends on how the entire vision pipeline is implemented for an application, as well as the compute resources available. The Vision Metrics Taxonomy provides information to pursue such questions, but as always pursuing the wrong questions may lead to the wrong answers.

Discrimination and Uniqueness

The selection of optimal, discriminating features is achieved using a variety of methods. For example, local feature detector methods filter out only the most discriminating or unique candidates based on criteria such as corner strength; then descriptors are computed at the selected interest points as patches or other shapes; and finally the resulting descriptor is either accepted or rejected based on uniqueness criteria. Uniqueness is also the key criterion for creating sparse codebooks discussed in Chap. 4.

Discrimination can be measured by the ability to recreate an image from only the descriptor information, as discussed in Chap. 4. A descriptor with too little information to adequately recreate an image may be considered weak or nondiscriminating.

General Vision Metrics Taxonomy

To understand feature metrics, we develop a Vision Metrics Taxonomy composed of summary criteria. Each criterion is selected with a practical, engineering perspective in mind to provide information for evaluation and implementation in specific terms, such as algorithm, spectra, memory size, and other attributes. The basic categories of the Vision Metrics Taxonomy are shown in Table 5.1, and also summarized here as a list, and each list item is discussed in separate sections in this chapter:

- Feature Descriptor Family
- Spectra Dimension
- Spectra Value
- Interest Point
- Storage Format
- Data Types
- Descriptor Memory
- Feature Shape
- Feature Pattern
- Feature Density
- Feature Search Method

Table 5.1 Vision Metrics Taxonomy

Feature Descriptor Family	Interest Point	Pattern Pair Sampling
Local Binary Descriptor	Point, edge, or corner	Center – boundary pair
Spectra Descriptor	Contour based, perimeter	Random pair points
Basis Space Descriptor	Other	Foveal centered trained pairs
Polygon Shape Descriptor	No interest point	Trained pairs
Spectra Dimensions	Storage Format	Symmetric pairs
Single variate	Spectra vector	Pattern Region Size
Multivariate	Bit vector	Bounding box (x size, y size)
Spectra Value	Data Types	Distance function
Orientation Vector	Float	Euclidean distance
Sensor, accelerometer data	Integer	Squared Euclidean distance
Multigeometry	Fixed point	Cosine similarity
Multi-scale	Descriptor Memory	Correlation distance
Fourier magnitude	Fixed length or variable length	Manhattan distance
Fourier phase	Byte count range	Chessboard or Chebychev distance
Other basis function	Feature Shape	Earth movers distance
Morphological shape metrics	Rectangle block patch	SAD L1 Norm
Learned binary descriptors	Symmetric polygon region	SSD L2 Norm
Dictionary, codebook, vocabulary	Irregular segmented region	Mahalanobis distance
Region histogram 2D	Volumetric region	Bray Curtis difference
3D histogram	Deformable	Canberra distance
Log polar bins	Feature Search Method	L0 Norm
Cartesian bins	Coarse to fine image pyramid	Hamming distance
Region sum	Scale space pyramid	Jaccard similarity
Region average	Pyramid scale	Run-Time Compute
Region statistical	Dense sliding window	Compute complexity % of SIFT
Binary pattern	Dense grid block search	Feature Density
DoG (1-bit)	Window search	Global
DoG (multi-bit)	Grid block search	Regional
Bit vector of values	Sparse at interest points	Sparse
Gradient magnitude	Sparse at predicted points	Feature Pattern
Gradient direction	Sparse in segmented regions	Rectangular kernel
3D surface normals	Depth segmented regions (Z)	Binary compare pattern
Line segment metric	Super-pixel search	DNET line sample strip set
Gray scale info	Sub-pixel search	Radial line sampling pattern
Color space info	Double-scale 1 st pyramid level	Perimeter or contour edge
		Sample weighting pattern

- Pattern Pair Sampling
- Pattern Region Size
- Distance Function
- Run-Time Compute

Many of the background concepts used in the taxonomy are discussed in Chap. 4, where attributes about the internal structure and goals of common features are analyzed. In addition, this taxonomy is illustrated in the Feature Metric Evaluation (FME) information tables later in this chapter. A small subset of the taxonomy is used in the Chap. 6 survey of feature descriptors to record summary information. The taxonomy in Table 5.1 is a guideline for collecting and summarizing information. No judgment on goodness or performance is recorded or implied.

Feature Descriptor Family

As described at the beginning of this chapter, feature descriptors are classified in this taxonomy as follows:

- Local Binary Descriptors
- Spectra Descriptors
- Basis Space Descriptors
- Polygon Shape Descriptors

Spectra Dimensions

The spectra or values recorded in the feature descriptor vary, and may include one or more types of information or spectra. We divide the categories as follows:

- **Single variate:** stores a single value such as an integral image or region average, or just a simple set of pixel gradients.
- **Multivariate:** multiple spectra are stored; for example, a combination of spectra such as color information, gradient magnitude and direction, and other values.

Spectra Type

The spectral type of feature descriptor is a major axis in this taxonomy, as shown in Fig. 5.1. Here are common spectra, which have been discussed in Chap. 3 and will be discussed in Chap. 6 as well.

- **Gradient magnitude:** a measure of local region texture or difference, used by a wide range of patch-based feature descriptor methods. It is well known [240] that the human visual system responds to gradient information in a scale and rotationally invariant manner across the retina, as demonstrated in SIFT and many other feature description methods, thus the use of gradients is a preferred method for computer vision.
- **Gradient direction:** some descriptor methods compute a gradient direction and others do not. A simple region gradient direction method is used by several feature descriptors and edge detection methods, including Sobel and SIFT, to provide rotational invariance.
- **Orientation vector:** some descriptors are oriented and others are not. Orientation can be computed by methods other than a simple gradient—for example, SURF uses a method of sampling many gradient directions to compute the dominant gradient orientation of the entire patch region as the orientation vector. In the RIFF method, a radial relative orientation is computed. In the SIFT method, any orientations detected within 80 % of the dominant orientation will result in an additional interest point being generated, so the same descriptor may allow multiple interest points differing only in orientation.
- **Sensor data:** data such as accelerometer or GPS information is added to the descriptor. In the GAFD method, a gravity vector computed from an accelerometer is used for orientation.
- **Multigeometry:** multiple geometric transforms of the descriptor data that are stored together in the descriptor, such as several different perspective transforms of the same data as used in the RFM2.3 descriptor; the latter contains the same patch computed over various geometric transforms to increase the scale, rotation, and geometric robustness.
- **Multiscale:** instead of relying on a scale-space pyramid, the descriptor stores a copy of several scaled representations. The multi-resolution histogram method described in Chap. 4 is one such method of approximating feature description over a range of scales, where scale is approximated

using a range of Gaussian blur functions, and their resulting histograms are stored as the multi-scale descriptor.

- **Fourier magnitude:** both the sine and cosine basis functions from the Fourier series can be used in the descriptor—for example, in the polygon shape family of descriptors as illustrated in Fig. 6.29. The magnitude of the sine or cosine alone is a revealing shape factor, without the phase, as illustrated in Fig. 6.6, which shows the histogram of LBPs run through a Fourier series to produce the power spectrum. This illustrates how the LBP histogram power spectrum provides rotational invariance. Other methods related to Fourier series may use alternative arrangements of the computation, such as the discrete cosine transform (DCT), which uses only the cosine component and is amenable to integer computations and hardware acceleration as commonly done for media applications.
- **Fourier phase:** phase information has been shown to be valuable for creating a blur-invariant feature descriptor, as demonstrated in the LPQ method discussed in Chap. 6.
- **Other basis functions:** can be used for feature description. Wavelets are commonly used in place of Fourier methods owing to greater control over the function window and tuning of the basis functions derived from the mother wavelet into the family of related wavelets. See Chap. 2 for a discussion of wavelets compared to other basis functions.
- **Morphological shape metrics:** predominantly used in the polygon shape descriptor family, composed of shape factors, and referred to as *image moments* in some literature. They are computed over the gross features of a polygon image region such as area, perimeter, centroid, and many others. The vision pipeline and image preprocessing used for polygon shape description may include morphological and texture operators, rather than local interest point and descriptor computations.
- **Learned binary descriptors:** created by running ground truth data through a training step, such as developed in ORB and FREAK, to create a set of statistically optimized binary sampling point-pair patterns.
- **Dictionary, codebook, vocabulary from feature learning methods:** build up a visual vocabulary, dictionary, or sparse codebook as a sparse set of unique features using a wide range of descriptor methods, such as simple images correlation patches or SIFT descriptors. When combined as a sparse set, these are representative of the features found in a set of ground truth data for an application domain, such as automobile recognition or face recognition.
- **Region histogram 2D:** used for several types of information, such as binning gradient direction, as in CARD, RFM2.3, and SURF; or for binning linear binary patterns, such as the LBP. The SIFT method of histogramming gradient information uses a fairly large histogram bin region, which provides for some translation invariance, similar to the human visual system treatment of the 3D position of gradients across the retina [240].
- **3D histogram:** used in methods such as used in SIFT, which represents gradient magnitude and orientation together as a 3D histogram.
- **Cartesian bins:** a common method of binning local region information into the descriptor simply based on the Cartesian position of pixels in a patch—for example, histogramming the pixel intensity magnitude of each point in the region.
- **Log polar bins:** instead of binning local region feature information in Cartesian rectangular arrangements, some descriptors such as GLOH use a log polar coordinate system to prepare values for histogram binning, with the goal of adding better rotational invariance to the descriptor.
- **Region sum:** such as an integral image, a method used to quickly sum the local region pixel values, or HAAR feature. The region sum is stored into the feature representing the total value of all the pixels in the region. Note that region summation may be good for coarse-feature description of an area, but the summation process eliminates fine local texture detail.

- **Region average:** average value of the pixels in a region area, also referred to as a box filter, which may be computed from a convolution operation, scaled integral image, or by simply adding up the pixel values in the array.
- **Region statistical:** such as region moments, like standard deviation, variance, or max or min values.
- **Binary pattern:** such as a vector of binary values, or bits—for example, stored as a result of local pixel pair compare computations of local neighborhood pixel values as used in the local binary descriptor family, such as LBP, Census, and ORB.
- **DoG (1-bit quantized):** as used in the FREAK descriptor, a set of DoG or bandpass filter features of different sizes, taken over a local binary region in a retinal sampling pattern similar to the human visual system, compared in pairs, and quantized to a single bit in a histogram vector.
- **DoG (multi-bit):** a type of bandpass filter that is implemented using many variations, where a Gaussian blur filter is applied to the image, then the image is subtracted from (a) a shifted copy of itself, (b) a copy of itself at another Gaussian blur level, or (c) a copy of itself at another image scale as in the SIFT descriptor method.
- **Bit vector of values:** a bit string containing a sequence of values quantized to a single bit, such as a threshold.
- **3D surface normals:** the analog to 2D gradients except in 3D, used in the HON4D method [190] to describe the surface of a 3D object location in the feature descriptor.
- **Line segment metric:** as in the CCH method, used to describe the line segments composing an object perimeter. Or, as used as a shape factor for objects where the length of a set of radial line segments originating at the centroid and extending to the perimeter are recorded in the descriptor, which can be fed into a Fourier transform to yield a power spectrum signature, as shown in Fig. 6.29.
- **Color space info:** some descriptors do not take advantage of color information, which in many cases can provide added discrimination and accuracy. Both the use of simple RGB channels, such as in the RGB-D methods [67, 110], or using color space conversions into more accurate spaces are invaluable. For example, face recognition has problems distinguishing faces from different cultures, and since the skin tone varies across regions, the color value can be measured and added to the descriptor. However, several descriptors make use of color information, such as S-LBP, which operates in a colorimetric, accurate color space such as CIE-Lab, or the F-LBP, which computes a Fourier spectrum of color distance from the center pixel to adjacent pixels, as well as color variants of SIFT and many others.
- **Gray scale info:** the gray scale or color intensity value is the default spectra in almost all descriptors. However, the method used to create the gray scale from color, and the image preprocessing used to prepare intensity for analysis and measurement, are critical for the vision pipeline and were discussed in Chap. 2.

Interest Point

The use of interest points is optional with feature description. Some methods do not use interest points, and sample the image on a fixed grid rather than at every pixel, such as the Viola–Jones method using HAAR-like features. It is also possible to simply create a feature descriptor for every pixel rather than just at interest points, but since the performance impact is considerable, interest points are typically used to find the best location for a feature first.

Several methods for finding interest points are surveyed and discussed in Chap. 6. Categories of interest points for the taxonomy include:

- **Point, edge, or corner:** these methods typically start with locating the local region maxima and minima; methods used include gradients, local curvature, Harris methods, blob detectors, and edge detectors.
- **Contour based, perimeter:** some methods do not start feature description at maxima and minima, and instead look for structure in the image, such as a contour or perimeter, and this is true mainly for the morphological shape based methods.
- **Other:** there are other possibilities for determining interest point location, such as prediction of likely interest point or feature positions, or using grid or tile regions.
- **No interest point:** some methods do not use any interest points at all.

Storage Formats

Storage formats are a practical matter for memory efficiency and engineering real systems and designing data structures. Knowing the storage format can guide efforts during engineering and optimization toward various programming constructs, instruction sets, and memory architecture.

For example, both CPU and GPGPU graphics processors often provide dedicated silicon to support various storage format organizations, such as scatter and gather operations, and sparse and dense data structure support. Understanding the GPGPU capabilities can provide guidelines for designing the storage format, as discussed in Chap. 8. Storage format summary:

- **Spectra vector:** may be a set of histograms, a set of color values, a set of basis vectors.
- **Bit vector:** local binary patterns use bit vector data types, some programming languages include bit vector constructs, and some instruction sets include bit vector handling instructions.
- **Multivariate collection:** a set of values such as statistical moments or shape factors.

Data Types

The data types used for feature description are critical for accuracy, memory use, and compute. However, it is worth noting that data types can be changed as a trade-off for accuracy in some cases. For example, converting floating point to fixed point or integer computations may be more memory efficient, as well as power efficient, since a floating point silicon ALU complex occupies almost four times more die space, thus consuming more power than an integer ALU. The data type summary includes:

- **Float:** many applications require floating point for accuracy. For example, a Fourier transform of images requires at least 64 bits double precision (larger images require more precision); other applications like target tracking may require 32-bit floating point for precision trajectory computations.
- **Integer:** pixel values are commonly represented with 8 bit values, with 16 bits per pixel common as image sensors provide better data. At least 32-bit integers are needed for many data structures and numerical results, such as integral images.
- **Fixed point:** this is an alternative representation to floating point, which saves data space and can be implemented more efficiently in silicon. Most modern GPUs support several fixed-point formats, and some CPUs as well. Fixed-point formats include 8-, 16-, and 24-bit representations. Accuracy may be close enough using fixed point, depending on the application. In addition to fixed-point data types, GPUs and some processors also provide various normalized data types (see manufacturer information).

Descriptor Memory

The total descriptor memory size is part of the efficiency of the descriptor, and compute performance is another component. A descriptor with a large memory footprint, few invariance attributes and heavy compute is inefficient. We are interested in memory size as a practical matter. Key memory-related attributes include:

- **Fixed length or variable length:** some descriptors allows for alternative representations.
- **Byte count:** the length of all data in the descriptor.

Feature Shapes

A range of shapes are used for the pixel sampling pattern; shapes are surveyed in Chap. 4 including the following methods:

- **Rectangle block patch:** simple x, y, dx, dy range.
- **Symmetric polygon region:** may be an octagon, as in the CenSurE method, or a circular region, like FREAK or DAISY.
- **Irregular segmented region:** such as computed using morphological methods following segmented regions or thresholded perimeter.
- **Volumetric region:** some features make use of stacks of images resembling a volume structure. As shown in Fig. 6.12, the VLBP or Volume LBP and the LBP-TOP make use of volumetric data structures. The dynamic texture methods and activity recognition methods often use sets of three adjacent patches from the current frame plus two past frames, organized in a spatiotemporal image frame history, similar to a volume.
- **Deformable:** most features use a rigid shape, such as a fixed-size rectangle or a circle; however, some descriptors are designed with deformation in mind, such as scale deformations [337, 338], and affine or homographic deformation [212], to enable more robust matching.

Feature Pattern

Feature pattern is a major axis in this taxonomy, as shown in Fig. 5.1, since it affects memory architecture and compute efficiency.

Feature shape and pattern are related. Shape refers to the boundary, and pattern refers to the sampling method. Patterns include:

0	1	0
1	-4	1
0	1	0

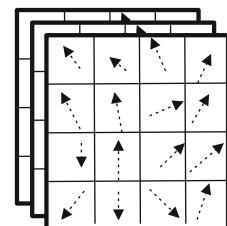
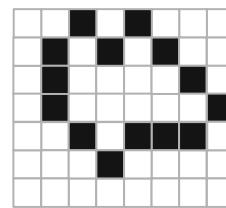


Figure 5.3 Feature shapes. (*Left to right*) Rectangular patch, symmetric polygon region, irregular segmented region, and volumetric region

- **Rectangular kernel:** some methods use a kernel to define which elements in the region are included in the sample; see Fig. 5.3 (left image) showing a kernel that does not use the corner pixels in the region; see also Fig. 4.8.

- **Binary compare pattern:** such as FREAK, ORB, and BRISK, where specific pixels in a region are paired to form a complex sampling pattern.
- **DNET line sample strip set:** where points along a line segment are sampled densely; see Fig. 4.8.
- **Radial line sampling pattern:** where points on radial line segments originating at a center point are sampled densely; for example, used to compute Fourier descriptors for polygon region shape; see Fig. 6.29.
- **Perimeter or contour edge:** where points around the edge of a shape or region are sampled densely.
- **Sample weighting pattern:** as shown in Fig. 6.17, SIFT uses a circular weighting pattern in the histogram bins to decrease the contribution of points farther away from the center of the patch. The D-NETS method uses binary weighting of samples along the line strips, favoring points away from the endpoints and ignoring points close to the end points. Weighting patterns can provide invariance to noise and occlusion.

See Chap. 4 for more illustrations in the section on patches and shapes.

Feature Density

As shown in Fig. 5.1, feature density is a major axis in this taxonomy. The amount of the image used for the descriptor is referred to in this taxonomy as *feature density*. For example, some descriptors are intended to use smaller regions of local pixels, anchored at interest points, and to ignore the larger image. Other methods use larger regions. Density categories include:

- **Global:** covers the entire image, each pixel in the image.
- **Regional:** covers fairly large regions of the image, typically on a grid, or around a segmented structure or region, not anchored at interest points.
- **Sparse:** may be taken at interest points, or in small regions at selected points such as random points in the BRIEF descriptor, trained points such as FREAK and ORB, or a sparse sampling grid as in the RFM2.3 descriptor.

Feature Search Methods

The method used for searching for features in the image is a significant for feature descriptor design. The search method determines a lot about the design of the descriptor, and the compute time required in the vision pipeline. We list several search variations here, and more detailed descriptions and illustrations are provided in Chap. 4. Note that a feature descriptor can make use of multiple search criteria. Feature search related information is summarized as follows:

- **Coarse-to-fine image pyramid:** or multi-scale search, using a pyramid of coarser resolution copies of the original.
- **Scale space pyramid:** the scale space pyramid is a variation of the regular coarse-to-fine image pyramid, where a Gaussian blur function is computed over each pyramid scale image [529] to create a more uniform search space; see Fig. 4.17.
- **Pyramid scale factor:** captures pyramid scale intervals, such as octaves or other scales—for example, ORB uses a $\sim 1.41 \times$ scale.
- **Dense sliding window:** where the search is made over each pixel in the image, often within a sliding rectangular region centered at each pixel.
- **Grid block search:** where the image is divided into a fixed grid or tiles, so the search can be faster but does not discriminate as well as dense methods. For example, see Fig. 6.17 describing the PHOG method, which computes descriptors at different grid resolutions across the entire image.

- **Window search:** limited dense search to particular regions, such as in stereo matching between two L/R frames where the correspondence search range is limited to expected locations.
- **Sparse at interest points:** where a corner detector or other detector is used to determine where valid features may be found.
- **Sparse at predicted points:** such as in tracking and mapping algorithms like PTAM, where the location of interest points is predicted based on motion or trajectory, and then a feature search begins at the predicted points.
- **Sparse in segmented regions:** for example, when morphological shape segmentation methods or thresholding segmentation methods define a region, and a second pass is made through the region looking for features.
- **Depth segmented regions (Z):** when depth camera information is used to threshold the image into foreground and background, and only the foreground regions are searched for features.
- **Super-pixel search:** similar to the image pyramid method, but a multi-scale representation of the image is created by combining pixel values together using super-pixel integration methods, as discussed in Chap. 2.
- **Sub-pixel search:** where sub-pixel accuracy is needed—for example, with region correlation, so several searches are made around a single pixel, with sub-pixel offsets computed for each compare, and in some cases geometric transforms of the pattern are made prior to feature matching.
- **Double-scale first pyramid level:** In the SIFT scale-space pyramid method, the lowest level of the pyramid is computed from a doubled $2 \times$ linear interpolated version of the full-scale image, which has the effect of preserving high-frequency information in the lowest level of the image pyramid, and increasing the number of stable keypoints by about four times, which is quite significant. Otherwise, computing the Gaussian blur across the original image would have the effect of throwing away most of the high-frequency details.

Pattern Pair Sampling

For local binary patterns, pattern pair sampling design is one of the key areas of innovation. Pairs of points are compared using a function such as (center pixel < kernel pixel) using a compare region threshold, and then the result of the comparison forms the binary descriptor vector. Note that many local binary descriptor methods are discussed and illustrated in Chap. 4, to illustrate variations in point-pair sampling configuration and compare functions. The vision taxonomy for point-pair sampling includes:

- **Center—boundary pair:** such as in the LBP family and Census transform.
- **Random pair points:** such as in BRIEF, and semi-random in ORB.
- **Foveal centered trained pairs:** such as in FREAK and Daisy.
- **Trained pairs:** many methods train the point-pairs using ground truth data to meet objective criteria, such as FREAK and ORB.
- **Symmetric pairs:** such as BRISK, which provides short and long line segments spaced symmetrically for point-pair comparisons.

Pattern Region Size

The size of the local pattern region is a critical performance factor, even though memory access is likely from fast-register files and cache. For example, if we are performing a convolution of a 3×3 pattern region, there are nine multiplies per kernel, and possibly one summary multiply to scale the results, for a total of ten multiplies per pixel. For each multiply we have two memory reads, one for the pixel and one for the kernel value; and we have ten memory writes, one for each multiply.

A 640×480 image has 307,200 pixels, and assuming 8 bits per pixel gray scale only, per frame we end up with 3,072,000 multiplies, 60,720,000 memory reads, and 307,200 writes for the result. Larger kernel sizes and larger image sizes of course add more compute.

There are many ways to optimize the performance, which we cover in Chap. 8 on vision pipeline engineering. For this attribute, we are interested in the following:

- **Bounding box (x size, y size)**: for example, the bounding box around a rectangular region, circular region, or polygon shape region.

Distance Function

Computing the pattern matching or correspondence is one of the key performance criteria for a good descriptor. Feature matching is a trade-off between accuracy and performance, with the key variables being the numeric type and size of the feature descriptor vectors, the distance function, and the number of patterns and search optimizations in the feature database. Choosing a feature descriptor amenable to fast matching is a good goal.

In general, the fastest distance functions are the binary family and Hamming distance, which is used in the local binary descriptor family. Some common distance functions are enumerated here; see Chap. 4 for details.

Euclidean or Cartesian Distance Family

- Euclidean distance
- Squared Euclidean distance
- Cosine similarity
- SAD L1 Norm
- SSD L2 Norm
- Correlation distance
- Hellinger distance

Grid Distance Family

- Manhattan distance
- Chessboard or Chebychev distance

Statistical Distance Family

- Earth movers distance
- Mahalanobis distance
- Bray Curtis difference
- Canberra distance

Binary or Boolean Distance Family

- L0 Norm
- Hamming distance
- Jaccard similarity

Feature Metric Evaluation

This section addresses the question of how to summarize feature descriptor information at a high level from the Vision Metrics Taxonomy into a practical Feature Metric Evaluation Framework (FME) Feature metric evaluation (FME) from an engineering and design perspective.

Note The FME is intended as a template to capture high-level information for basic analysis.

Efficiency Variables, Costs and Benefits

Efficiency can be measured for a feature descriptor in simple terms, such as the benefit of the compute cost and memory used vs. what is provided in the way of accuracy, discrimination, robustness, and invariance. How much value does the method provide for the time, space, and power cost? Efficiency metrics include:

- **Costs:** compute, memory, time, power
- **Benefits:** accuracy, robustness, and invariance attributes provided
- **Efficiency:** benefits vs. costs

The effectiveness of the data contained in the descriptor varies—for example, a large memory footprint to contain a descriptor with little invariance is not efficient, and a high compute cost for small amounts of invariance and accuracy also reveals low efficiency. We could say that an efficient feature representation contains the least number of bytes and lowest compute cost providing the greatest amount of discrimination, robustness, and accuracy. Local binary descriptors have demonstrated the best efficiency for many robustness attributes.

Image Reconstruction Efficiency Metric

For a visual comparison of feature descriptor efficiency, we can also reconstruct an image from the feature descriptors, and then visually and statistically analyze the quality of the reconstruction vs. the compute and memory cost. Detailed feature descriptors can provide good visualization and reconstruction of the original image from the descriptor data only. For example, Fig. 4.12 shows how the HOG descriptor captures oriented gradients using 32,780 bytes per 64×128 region, Fig. 4.13 shows image reconstruction illustrating how BRIEF and FREAK capture edge information similar to Laplacian or other edge filters using 64 bytes per descriptor, and Fig. 4.14 shows SIFT image reconstruction using 128 bytes per descriptor.

Although we do not include image reconstruction efficiency in the FME, this topic was covered in Chap. 4, under the discussion of discrimination.

Example Feature Metric Evaluations

Here are a few examples showing how the Vision Metrics Taxonomy and the FME can be used to collect summary descriptor information.

SIFT Example

We use SIFT as an example baseline, since SIFT is widely recognized and carefully designed.

VISION METRIC TAXONOMY FME

Name:	SIFT
Feature Family:	Spectra
Spectra dimensions:	Multivariate
Spectra:	Gradient magnitude and direction, DoG Scale Space Maxima
Storage format:	Orientation and position, gradient orientation histograms

<i>Data type:</i>	<i>Float, integer</i>
<i>Descriptor Memory:</i>	<i>128 bytes for descriptor histogram</i>
<i>Feature shape:</i>	<i>Rectangular region</i>
<i>Search method:</i>	<i>Dense sliding window in 2D and 3D $3 \times 3 \times 3$ image pyramid</i>
<i>Feature density:</i>	<i>Local</i>
<i>Feature pattern:</i>	<i>Rectangular and pyramid-cubic</i>
<i>Pattern pair sampling:</i>	<i>–</i>
<i>Pattern region size:</i>	<i>16×16</i>
<i>Distance function:</i>	<i>Euclidean distance</i>

GENERAL ROBUSTNESS ATTRIBUTES

Total: *5 (scale, illumination, rotation, affine transforms, noise)*

LBP Example

The LBP is a very simple feature detector with many variations, used for texture analysis and feature description. We use the most basic form of 3×3 LBP here as an example.

VISION METRIC TAXONOMY FME

<i>Name:</i>	<i>LBP</i>
<i>Feature Family:</i>	<i>Local Binary</i>
<i>Spectra dimensions:</i>	<i>Single-variate</i>
<i>Spectra:</i>	<i>Pixel pair compares with center pixel</i>
<i>Storage format:</i>	<i>Binary Bit Vector</i>
<i>Data type:</i>	<i>Integer</i>
<i>Descriptor Memory:</i>	<i>1 byte</i>
<i>Feature shape:</i>	<i>Square centered at center pixel</i>
<i>Search method:</i>	<i>Dense sliding window</i>
<i>Feature density:</i>	<i>Local</i>
<i>Feature pattern:</i>	<i>Rectangular kernel</i>
<i>Pattern pair sampling:</i>	<i>Center—boundary pairs</i>
<i>Pattern region size:</i>	<i>3×3 or more</i>
<i>Distance function:</i>	<i>Hamming distance</i>

GENERAL ROBUSTNESS ATTRIBUTES

Total: *3 (brightness, contrast, rotation using RILBP)*

Shape Factors Example

This example uses binary thresholded polygon regions. For this hypothetical example, the preprocessing steps begin with adaptive binary thresholding and morphological shape definition operations, and the measurement steps begin with pixel neighborhood based perimeter following to defined the perimeter edge, followed by centroid computation from perimeter points, followed by determination of 36 radial line segments originating at the centroid reaching to the perimeter. Then each line segment is analyzed to find the shape factors including major/minor axis the Fourier

descriptor. The measurements assume a single binary object is being measured, and real-world images may contain at many objects.

We also assume the memory footprint as follows: angular samples taken around 360° , starting at centroid, at 10° increments for 36 angular samples, 36 floats for FFT spectrum magnitude, 36 integers for line segment length array, four integers for major/minor axis orientation and length, four integers for bounding box (x, y, dx, dy), one integer for perimeter length, two integers for centroid coordinates, TOTAL $36 * 4 + 36 * 2 + 4 * 2 + 4 * 2 + 1 * 2 * 2 * 2 = 238$, assuming 2 byte short integers and 4-byte floats are used.

VISION METRIC TAXONOMY FME

<i>Name:</i>	<i>Shape Factors</i>
<i>Feature Family:</i>	<i>Polygon Shape</i>
<i>Spectra dimensions:</i>	<i>Multivariate</i>
<i>Spectra:</i>	<i>Perimeter following, area, perimeter, centroid, other image moments</i>
<i>Storage format:</i>	<i>complex data structure</i>
<i>Data type:</i>	<i>Float, integer</i>
<i>Descriptor Memory:</i>	<i>Variable, several hundred bytes possible</i>
<i>Feature shape:</i>	<i>Polygon shapes, rectangular bounding box region</i>
<i>Search method:</i>	<i>Dense, recursive</i>
<i>Feature density:</i>	<i>Regional</i>
<i>Feature pattern:</i>	<i>Perimeter contour or edge</i>
<i>Pattern pair sampling:</i>	<i>-</i>
<i>Pattern region size:</i>	<i>Entire image</i>
<i>Distance function:</i>	<i>Multiple methods, multiple comparisons</i>

GENERAL ROBUSTNESS ATTRIBUTES

<i>Total:</i>	<i>8 or more (scale, rotation, occlusion, shape, affine, reflection, noise, illumination)</i>
---------------	---

Summary

In this chapter, a taxonomy is proposed as shown in Fig. 5.1 to describe feature description dimensions as shape, pattern, and spectra. This taxonomy is used to divide the families of feature description methods into polygon shape descriptors, local binary descriptors, and basis space descriptors. The taxonomy is used throughout the book. Also, a general vision metrics taxonomy is proposed for the purpose of summarizing high-level feature descriptor design attributes, such as type of spectra, descriptor pixel region size, distance function, and search method. In addition, a general robustness taxonomy is developed to quantify feature descriptor goodness, one attribute at a time, based on invariance and robustness criteria attributes, including illumination, scale, rotation, and perspective. Since feature descriptor methods are designed to address only some of the invariance and robustness attributes, each attribute should be considered separately when evaluating a feature descriptor for a given application. In addition, the robustness attributes can be applied to the design of ground truth datasets, as discussed in Chap. 7. Finally, the vision metrics taxonomy and the robustness taxonomy are combined to form a feature metric evaluation (FME) table to record feature descriptor attributes in summary form. A simple subset of the FME is used to review the attributes of several feature descriptor methods surveyed in Chap. 6.

Chapter 5: Learning Assignments

1. Describe the difference between the following feature descriptor families:
 - Local Binary Descriptors
 - Spectra descriptors using gradients and other scalar values
 - Basis Space Descriptors
 - Polygon Shape Descriptors
2. Describe at a high level the types of problems that would be exhibited in images under each of the robustness and invariance categories below:
 - Illumination variance
 - Color variance
 - Incompleteness of features
 - Resolution and distance variance
 - Geometric distortion
 - Discrimination and uniqueness
3. Describe a few example feature descriptor region shapes, and discuss the trade-offs involved when designing the shape.
4. Describe illumination problems, and the sources of the problems.
5. Describe geometric distortion, and the sources of the distortion.
6. Describe different spectra, such as gradients and color, that can be used in a feature descriptor, and describe applications for each spectra.
7. Describe pyramid search compared to sliding window search.

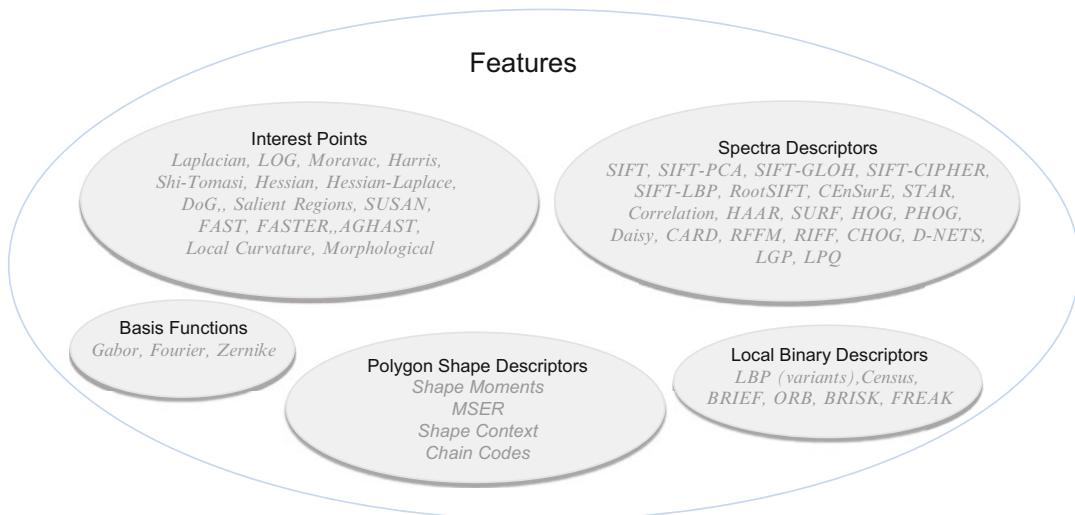
Interest Point Detector and Feature Descriptor Survey

6

“Who makes all these?”

—Jack Sparrow, Pirates of the Caribbean

Many algorithms for computer vision rely on locating interest points, or keypoints in each image, and calculating a feature description from the pixel region surrounding the interest point. This is in contrast to methods such as correlation, where a larger rectangular pattern is stepped over the image at pixel intervals and the correlation is measured at each location. The interest point is the *anchor point*, and often provides the scale, rotational, and illumination invariance attributes for the descriptor; the descriptor adds more detail and more invariance attributes. Groups of interest points and descriptors together describe the actual objects.



However, there are many methods and variations in feature description. Some methods use features that are not anchored at interest points, such as polygon shape descriptors, computed over larger segmented polygon-shaped structures or regions in an image. Other methods use interest points only, without using feature descriptors at all. And some methods use feature descriptors only, computed across a regular grid on the image, with no interest points at all.

Terminology varies across the literature. In some discussions, interest points may be referred to as *keypoints*. The algorithms used to find the interest points maybe referred to as *detectors*, and the

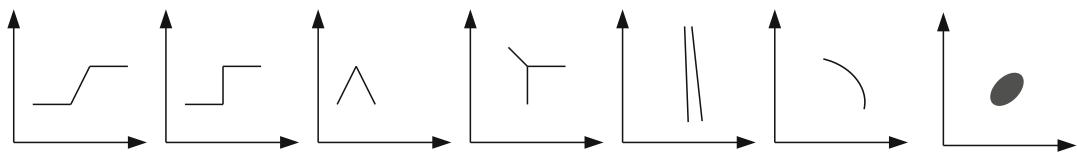


Figure 6.1 Types of keypoints, including corners and interest points. (*Left to right*) Step, roof, corner, line or edge, ridge or contour, maxima region

algorithms used to describe the features may be called *descriptors*. We use the terminology interchangeably in this work. Keypoints may be considered a set composed of (1) interest points, (2) corners, (3) edges or contours, and (4) larger features or regions such as blobs; see Fig. 6.1. This chapter surveys the various methods for designing local interest point detectors and feature descriptors.

Interest Point Tuning

What is a good keypoint for a given application? Which ones are most useful? Which ones should be ignored? Tuning the detectors is not simple. Each detector has different parameters to tune for best results on a given image, and each image presents different challenges regarding lighting, contrast, and image preprocessing. Additionally, each detector is designed to be useful for a different class of interest points, and must be tuned accordingly to filter the results down to a useful set of good candidates for a specific feature descriptor. Each feature detector will work best with certain descriptors, see Appendix A.

So the keypoints are further filtered to be useful for the chosen feature descriptor. In some cases, a keypoint is not suitable for producing a useful feature descriptor, even if the keypoint has a high score and high response. If the feature descriptor computed at the keypoint produces a descriptor score that is too weak, for example, the keypoint and corresponding descriptor should both be rejected. OpenCV provides several novel methods for working with detectors, enabling the user to try different detectors and descriptors in a common framework, and automatically adjust the parameters for tuning and culling as follows:

- **DynamicAdaptedFeatureDetector.** This class will tune supported detectors using an *adjusterAdapter()* to only keep a limited number of features, and iterate the detector parameters several times and redetect features in an attempt to find the best parameters, keeping only the requested number of best features. Several OpenCV detectors have an *adjusterAdapter()* provided, some do not; the API allows for adjusters to be created.
- **AdjusterAdapter.** This class implements the criteria for culling and keeping interest points. Criteria may include KNN nearest neighbor matching, detector response or strength, radius distance to nearest other detected points, number of keypoints within a local region, and other measures that can be included for culling keypoints for which a good descriptor cannot be computed.
- **PyramidAdaptedFeatureDetector.** This class can be used to adapt detectors that do not use a scale-space pyramid, and the adapter will create a Gaussian pyramid and detect features over the pyramid.
- **GridAdaptedFeatureDetector.** This class divides an image into grids and adapts the detector to find the best features within each grid cell.

Interest Point Concepts

An interest point may be composed of various types of corner, edge, and maxima shapes, as shown in Fig. 6.1. In general, a good interest point must be easy to find and ideally fast to compute; it is hoped that the interest point is at a good location to compute a feature descriptor. The interest point is thus the qualifier or *keypoint* around which a feature may be described.

There are various concepts behind the interest point methods currently in use, as this is an active area of research. One of the best analyses of interest point detectors is found in Mikolajczyk et al. [145], with a comparison framework and taxonomy for affine covariant interest point detectors, where *covariant* refers to the elliptical shape of the interest region, which is an affine deformable representation. Scale invariant detectors are represented well in a circular region. Maxima region and blob detectors can take irregular shapes. See the response of several detectors against synthetic interest point and corner alphabets in Appendix A.



Figure 6.2 Candidate edge interest point filters. (Left to right) Laplacian, derivative filter, and gradient filter

Commonly, detectors use *maxima and minima points*, such as gradient peaks and corners; however, edges, ridges, and contours are also used as keypoints, as shown in Fig. 6.2. There is no superior method for interest point detection for all applications. A simple taxonomy provided by Tuytelaars and Van Gool [511] lists edge-based region methods (EBR), maxima or intensity-based region methods (IBR), and segmentation methods to find shape-based regions (SBR) that may be blobs or features with high entropy.

Corners are often preferred over edges or isolated maxima points, since the corner is a structure and can be used to compute an angular orientation for the feature. Interest points are computed over color components as well as gray scale luminance. Many of the interest point methods will first apply some sort of Gaussian filter across the image and then perform a gradient operator. The idea of using the Gaussian filter first is to reduce noise in the image, which is otherwise amplified by gradient operators.

Each detector locates features with different degrees of invariance to attributes such as rotation, scale, perspective, occlusion, and illumination. For evaluations of the quality and performance of interest point detection methods measured against various robustness and invariance criteria on standardized datasets, see Mikolajczyk and Schmid [136] and Gauglitz et al. [137]. One of the key challenges for interest point detection is scale invariance, since interest points change dramatically in some cases over scale. Lindberg [204] has extensively studied the area of scale independent interest point methods.

Affine invariant interest points have been studied in detail by Mikolajczyk and Schmid [99, 133, 136, 145, 298, 303]. In addition, Mikolajczyk and Schmid [501] developed an affine-invariant version of the Harris detector. As shown in [523], it is often useful to combine several interest point detection methods to form a hybrid, for example, using the Harris or Hessian to locate suitable maxima regions,

and then using the Laplacian to select the best scale attributes. Variations are common, Harris-based and Hessian-based detectors may use scale-space methods, while local binary detector methods do not use scale space.

A few fundamental concepts behind many interest point methods come from the field of linear algebra, where the local region of pixels is treated as a matrix. (*Refer to a good linear algebra textbook as background for this section.*) Additional concepts come from other areas of mathematical analysis. Some of the key math useful for locating interest points are illustrated below, however note that in practice various forms of equations and algorithms are used which deviate from those shown here, see the references for more details.

- **A Matrix.** We start with a 2d rectangular pixel region, or matrix, of some dimension x,y :

$$M_{x,y} = \begin{bmatrix} 0,0 & \dots & x,0 \\ \dots & \dots & \dots \\ 0,y & \dots & x,y \end{bmatrix}$$

- **Gradient Magnitude.** This is the first derivative of the pixels in the local interest region, and assumes a direction. This is an unsigned positive number, and is also a Laplacian operator.

$$\left(\frac{\partial M_{x,y}}{\partial x} \right)^2 + \left(\frac{\partial M_{x,y}}{\partial y} \right)^2$$

- **Gradient Direction.** This is the angle or direction of the largest gradient angle from pixels in the local region in the range $+\pi$ to $-\pi$.

$$\tan^{-1} \left(\frac{\partial M_{x,y}}{\partial x} \right)^2 / \left(\frac{\partial M_{x,y}}{\partial y} \right)^2$$

- **Laplacian.** This is the second derivative and can be computed selectively using any of three terms:

$$\begin{aligned} & \frac{\partial^2 f \mathbf{M}_{x,y}}{\partial x^2} \\ & \frac{\partial^2 \mathbf{M}_{x,y}}{\partial y^2} \\ & \frac{\partial^2 \mathbf{M}_{x,y}}{\partial x \partial y} \end{aligned}$$

However, the Laplacian operator does not use the third form above, and computes a signed value of average orientation with respect to x and y partials only, see the Gradient Magnitude operator above.

- **Hessian Matrix or Hessian.** A square matrix containing second order partial derivatives of each pixel within the matrix region, describing surface curvature at each pixel. The Hessian has several interesting properties useful for interest point detection methods discussed in this section, which we can express in \mathbf{L} notation as follows:

$$H(x, \sigma) = \begin{pmatrix} L_{xx}(x, \sigma) & L_{xy}(x, \sigma) \\ L_{xy}(x, \sigma) & L_{yy}(x, \sigma) \end{pmatrix}$$

- **Largest Hessian.** This is based on the second derivative, as is the Laplacian, but the Hessian uses all three terms of the second derivative to compute the direction along which the second derivative is maximum as a signed value.

- **Smallest Hessian.** This is based on the second derivative, is computed as a signed number, and may be a useful metric as a ratio between largest and smallest Hessian.
- **Hessian Orientation, largest and smallest values.** This is the orientation of the largest second derivative in the range $+\pi$ to $-\pi$, which is a signed value, and it corresponds to an orientation without direction. The smallest orientation can be computed by adding or subtracting $\pi/2$ from the largest value.
- **Determinant of Hessian, Trace of Hessian, Laplacian of Gaussian.** All three names are used to describe the trace characteristic of a matrix, which can reveal geometric scale information by the absolute value, and orientation by the sign of the value. See SURF [166] for an application, which we can express in L notation as follows.

$$\text{trace } \mathcal{H}_{\text{norm}} L = t^\gamma \nabla^2 L = t^\gamma (L_{xx} + L_{yy})$$

$$\det \mathcal{H}_{\text{norm}} L = t^{2\gamma} (L_{xx} L_{yy} - L_{xy}^2)$$

- **Eigenvalues, Eigenvectors, Eigenspaces.** Eigen properties are important to understanding vector direction in local pixel region matrices. When a matrix acts on a vector, and the vector orientation is preserved, and when the sign or direction is simply reversed, the vector is considered to be an eigenvector, and the matrix factor is considered to be the eigenvalue. An eigenspace is therefore all eigenvectors within the space with the same eigenvalue. Eigen properties are valuable for interest point detection, orientation, and feature detection. For example, Turk and Petland [150] use eigenvectors reduced into a smaller set of vectors via PCA for face recognition, in a method they call Eigenfaces.

Interest Point Method Survey

We will now look briefly at algorithms and computational methods for some common interest point detector methods including:

- Laplacian of Gaussian (LOG)
- Moravec corner detector
- Harris and Stephens corner detection
- Shi and Tomasi corner detector (improvement on Harris method)
- Difference of Gaussians (DoG; an approximation of LOG)
- Harris methods, Harris–/Hessian–Laplace, Harris/Hessian Affine
- Determinant of Hessian (DoH)
- Salient regions
- SUSAN
- FAST, FASTER, AGAST
- Local curvature
- Morphological interest points
- MSER (discussed in the section on polygon shape descriptors)
- *NOTE: many feature descriptors, such as SIFT, SURF, BRISK, and others, provide their own detector method along with the descriptor method, see Appendix A.

Laplacian and Laplacian of Gaussian

The Laplacian operator, as used in image processing, is a method of finding the derivative or maximum rate of change in a pixel area. Commonly, the Laplacian is approximated using standard convolution kernels that add up to zero, such as:

$$\begin{aligned} L1 &= \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \\ L2 &= \begin{pmatrix} -1 & 0 & -1 \\ 0 & 4 & 0 \\ -1 & 0 & -1 \end{pmatrix} \end{aligned}$$

The Laplacian of Gaussian (LOG) is simply the Laplacian performed over a region that has been processed using a Gaussian smoothing kernel to focus edge energy; see Gun [147].

Moravac Corner Detector

The Moravac corner detection algorithm is an early method of corner detection whereby each pixel in the image is tested by correlating overlapping patches surrounding each neighboring pixel. The strength of the correlation in any direction reveals information about the point: a corner is found when there is change in all directions, and an edge is found when there is no change along the edge direction. A flat region yields no change in any direction. The correlation difference is calculated using the SSD between the two overlapping patches. Similarity is measured by the near-zero difference in the SSD. This method is compute intensive; see Moravac [322].

Harris Methods, Harris–Stephens, Shi–Tomasi, and Hessian Type Detectors

The Harris or Harris–Stephens corner detector family [148, 357] provides improvements over the Moravac method. The goal of the Harris method is to find the direction of fastest and lowest change for feature orientation, using a covariance matrix of local directional derivatives. The directional derivative values are compared with a scoring factor to identify which features are corners, which are edges, and which are likely noise. Depending on the formulation of the algorithm, the Harris method can provide high rotational invariance, limited intensity invariance, and in some of the formulations of the algorithm, scale invariance is provided such as the Harris–Laplace method using scale space [204, 501]. Many Harris family algorithms can be implemented in a compute-efficient manner.

Note that corners have an ill-defined gradient, since two edges converge at the corner, but near the corner the gradient can be detected with two different values with respect to x and y —this is a basic idea behind the Harris corner detector.

Variations on the Harris method include:

- The Shi, Tomasi, and Kanade corner detector [149] is an optimization on the Harris method, using only the minimum eigenvalues for discrimination, thus streamlining the computation considerably.

- The Hessian (Hessian affine) corner detector [145] is designed to be affine invariant, and it uses the basic Harris corner detection method but combines interest points from several scales in a pyramid, with some iterative selection criteria and a Hessian matrix.
- Many other variations on the basic Harris operator exist, such as the Harris–Hessian–Laplace [323], which provides improved scale invariance using a scale selection method, and the Harris/Hessian affine method [145, 298].

Hessian Matrix Detector and Hessian–Laplace

The Hessian Matrix method, also referred to as Determinant of Hessian (DoH) method, is used in the popular SURF algorithm [152]. It detects interest objects from a multi-scale image set where the determinant of the Hessian matrix is at a maxima and the Hessian matrix operator is calculated using the convolution of the second-order partial derivative of the Gaussian to yield a gradient maxima.

The DoH method uses integral images to calculate the Gaussian partial derivatives very quickly. Performance for calculating the Hessian Matrix is therefore very good, and accuracy is better than many methods. The related Hessian–Laplace method [298, 323] also operates on local extrema, using the determinant of the Hessian at multiple scales for spatial localization, and the Laplacian at multiple scales for scale localization.

Difference of Gaussians

The Difference of Gaussians (DoG) is an approximation of the Laplacian of Gaussians, but computed in a simpler and faster manner using the difference of two smoothed or Gaussian filtered images to detect local extrema features. The idea with Gaussian smoothing is to remove noise artifacts that are not relevant at the given scale, which would otherwise be amplified and result in false DoG features. The DoG features are used in the popular SIFT method [153], and as shown in Fig. 6.15, the simple difference of Gaussian filtered images is taken to identify maxima regions.

Salient Regions

Salient regions [154, 155] are based on the notion that interest points over a range of scales should exhibit local attributes or entropy that are “unpredictable” or “surprising” compared to the surrounding region. The method proceeds as follows:

1. The Shannon entropy E of pixel attributes such as intensity or color are computed over a scale space, where Shannon entropy is used the measure of unpredictability.
2. The entropy values are located over the scale space with maxima or peak values M . At this stage, the optimal scales are determined as well.
3. The probability density function (PDF) is computed for magnitude deltas at each peak within each scale, where the PDF is computed using a histogram of pixel values taken from a circular window of desired radius from the peak.
4. Saliency is the product of E and M at each peak, and is also related to scale. So the final detector is salient and robust to scale.

SUSAN, and Trajkovic and Hedly

The SUSAN method [156, 157] is dependent on segmenting image features based on local areas of similar brightness, which yields a bimodal valued feature. No noise filtering and no gradients are used. As shown in Fig. 6.3, the method works by using a center nucleus pixel value as a comparison reference against which neighbor pixels within a given radius region are compared, yielding a set of pixels with similar brightness, called a Univalue Segment Assimilating Nucleus (USAN).

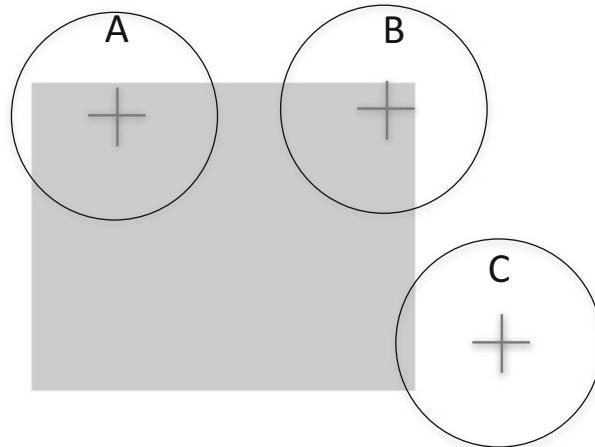


Figure 6.3 SUSAN method of computing interest points. The *dark region* of the image is a *rectangle* intersecting USANs A, B, and C. USAN A will be labeled as an edge, USAN B will be labeled as a corner, and USAN C will be labeled as neither an edge nor a corner

Each USAN contains structural information about the image in the local region, and the size, centroid, and second-order moments of each USAN can be computed. The SUSAN method can be used for both edge and corner detection. Corners are determined by the ratio of pixels similar to the center pixel in the circular region: a low ratio around 25 % indicates a corner, and a higher ratio around 50 % indicates an edge. SUSAN is very robust to noise.

The Trajkovic and Hedly method [206] is similar to SUSAN, and discriminates among points in USAN regions, edge points, and corner points.

SUSAN is also useful for noise suppression, and the bilateral filter [294], discussed in Chap. 2, is closely related to SUSAN. SUSAN uses fairly large circular windows; several implementations use 37 pixel radius windows. The FAST [130] detector is also similar to SUSAN, but uses a smaller 7×7 or 9×9 window and only some of the pixels in the region instead of all of them; FAST yields a local binary descriptor.

Fast, Faster, AGHAST

The FAST methods [130] are derived from SUSAN with respect to a bimodal segmentation goal. However, FAST relies on a connected set of pixels in a circular pattern to determine a corner. The connected region size is commonly 9 or 10 out of a possible 16; either number may be chosen, referred to as FAST9 and FAST10. FAST is known to be efficient to compute and fast to match; accuracy is also quite good. FAST can be considered a relative of the local binary pattern LBP.

FAST is not a scale-space detector, and therefore it may produce many more edge detections at the given scale than a scale-space method such as used in SIFT.

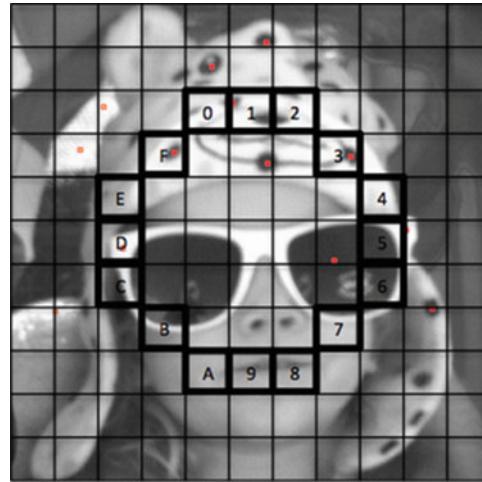


Figure 6.4 The FAST detector with a 16-element circular sampling pattern grid. Note that each pixel in the grid is compared against the center pixel to yield a binary value, and each binary value is stored in a bit vector

As shown in Fig. 6.4, FAST uses binary comparison with each pixel in a circular pattern against the center pixel using a threshold to determine if a pixel is less than or greater than the center pixel. The resulting descriptor is stored as a contiguous bit vector in order from 0 to 15. Also, due to the circular nature of the pixel compare pattern, it is possible to retrofit FAST and store the bit vector in a rotational-invariant representation, as demonstrated by the RILBP descriptor discussed later in this chapter; see Fig. 6.11.

Local Curvature Methods

Local curvature methods [200–204] are among the early means of detecting corners, and some local curvature methods are the first known to be reliable and accurate in tracking corners over scale variations [202]. Local curvature detects points where the gradient magnitude and the local surface curvature are both high. One approach taken is a differential method, computing the product of the gradient magnitude and the level curve curvature together over scale space, and then selecting the maxima and minima absolute values in scale and space. One formulation of the method is shown here.

$$\tilde{\alpha}(x, y; t) = L_x^2 L_{yy} + L_y^2 L_{xx} - 2L_x L_y L_{xy}$$

Various formulations of the basic algorithm can be taken depending on the curvature equation used. To improve scale invariance and noise sensitivity, the method can be modified using a normalized formulation of the equation over scale space, as follows:

$$\tilde{\alpha}_{\text{norm}}(x, y; t) = t^{2\gamma} (L_x^2 L_{yy} + L_y^2 L_{xx} - 2L_x L_y L_{xy})$$

where

$$\gamma = 0.875$$

At larger scales, corners can be detected with less sharp and more rounded features, while at lower scales or at unity scale sharper corners over smaller areas are detected. The Wang and Brady method [205] also computes interest points using local curvature on the 2D surface, looking for inflection points where the surface curvature changes rapidly.

Morphological Interest Regions

Interest points can be determined from a pipeline of morphological operations, such as thresholding followed by combinations or erosion and dilation to smooth, thin, grown, and shrink pixel groups. If done correctly for a given application, such morphological features can be scale and rotation invariant. Note that the simple morphological operations alone are not enough; for example, erode left unconstrained will shrink regions until they disappear. So intelligence must be added to the morphology pipeline to control the final region size and shape. For polygon shape descriptors, morphological interest points define the feature, and various image moments are computed over the feature, as described in Chap. 3 and also in the section on polygon shape descriptors later in this chapter.

Morphological operations can be used to create interest regions on binary, gray scale, or color channel images. To prepare gray scale or color channel images for morphology, typically some sort of preprocessing is used, such as pixel remapping, LUT transforms, or histogram equalization. (These methods were discussed in Chap. 2.) For binary images and binary morphology approaches, binary thresholding is a key preprocessing step. Many binary thresholding methods have been devised, ranging from simple global thresholds to statistical and structural kernel-based local methods.

Note that the morphological interest region approach is similar to the maximally stable extrema region (MSER) feature descriptor method discussed later in the section on polygon shape descriptors, since both methods look for connected groups of pixels at maxima or minima. However, MSER does not use morphology operators.

A few examples of morphological and related operation sequences for interest region detection are shown in Fig. 6.5, and many more can be devised.



Figure 6.5 Morphological methods to find interest regions. (*Left to right*) Original image, binary thresholded and segmented image using Chan Vese method, skeleton transform, pruned skeleton transform, and distance transform image. Note that binary thresholding requires quite a bit of work to set parameters correctly for a given application

Feature Descriptor Survey

This section provides a survey and observations about a few representative feature descriptor methods, with no intention to directly compare descriptors to each other. For more detailed information on analytical methods for comparing feature descriptors, see also Lempitsky [844], and Huang [889]. In practice, the feature descriptor methods are often modified and customized, and often several descriptors are used together as a multivariate descriptor to increase confidence, see Varma

[770], Vedaldi [887], and Gehler [792] for more details about multivariate descriptors, and applying boosting to weight the descriptors in the classifier (i.e. a multi-stage classifier). The goal of this survey is to examine a range of feature descriptor approaches from each feature descriptor family from the taxonomy that was presented in Chap. 5:

- Local binary descriptors
- Spectra descriptors
- Basis space descriptors
- Polygon shape descriptors
- 3D, 4D, and volumetric descriptors

For key feature descriptor methods, we provide here a summary analysis:

- **General Vision Taxonomy and FME:** covering feature attributes including spectra, shape, and pattern, single or multivariate, compute complexity criteria, data types, memory criteria, matching method, robustness attributes, and accuracy.
- **General Robustness Attributes:** covering invariance attributes such as illumination, scale, perspective, and many others.

No direct comparisons are made between feature descriptors here, but ample references are provided to the literature for detailed comparisons and performance information on each method. See Table 8.2 for a comparison of the memory footprints for various feature descriptor methods in this survey, which is useful for performance analysis.

Local Binary Descriptors

This family of descriptors represents features as binary bit vectors. To compute the features, image pixel point-pairs are compared and the results are stored as binary values in a vector. Local binary descriptors are efficient to compute, efficient to store, and efficient to match using Hamming distance. In general, local binary pattern methods achieve very good accuracy and robustness compared to other methods.

A variety of local sampling patterns are used with local binary descriptors to set the pairwise point comparisons; see the section in Chap. 4 on local binary descriptor point-pair patterns for a discussion on local binary sampling patterns. We start this section on local binary descriptors by analyzing the local binary pattern (LBP) and some LBP variants, since the LBP is a powerful metric all by itself and is well known.

Local Binary Patterns

Local binary patterns (LBP) were developed in 1994 by Ojala et al. [165] as a novel method of encoding both pattern and contrast to define texture [161–165]. LBPs can be used as an image processing operator. The LBP creates a descriptor or texture model using a set of histograms of the local texture neighborhood surrounding each pixel. In this case, local texture is the feature descriptor.

The LBP metric is simple yet powerful; see Fig. 6.6. We cover some level of detail on LBPs, since there are so many applications for this powerful texture metric as a feature descriptor as well. Also, hundreds of researchers have added to the LBP literature [165] in the areas of theoretical foundations, generalizations into 2D and 3D, applied as a descriptor for face detection, and also applied to spatiotemporal applications such as motion analysis. LBP research remains quite active at this

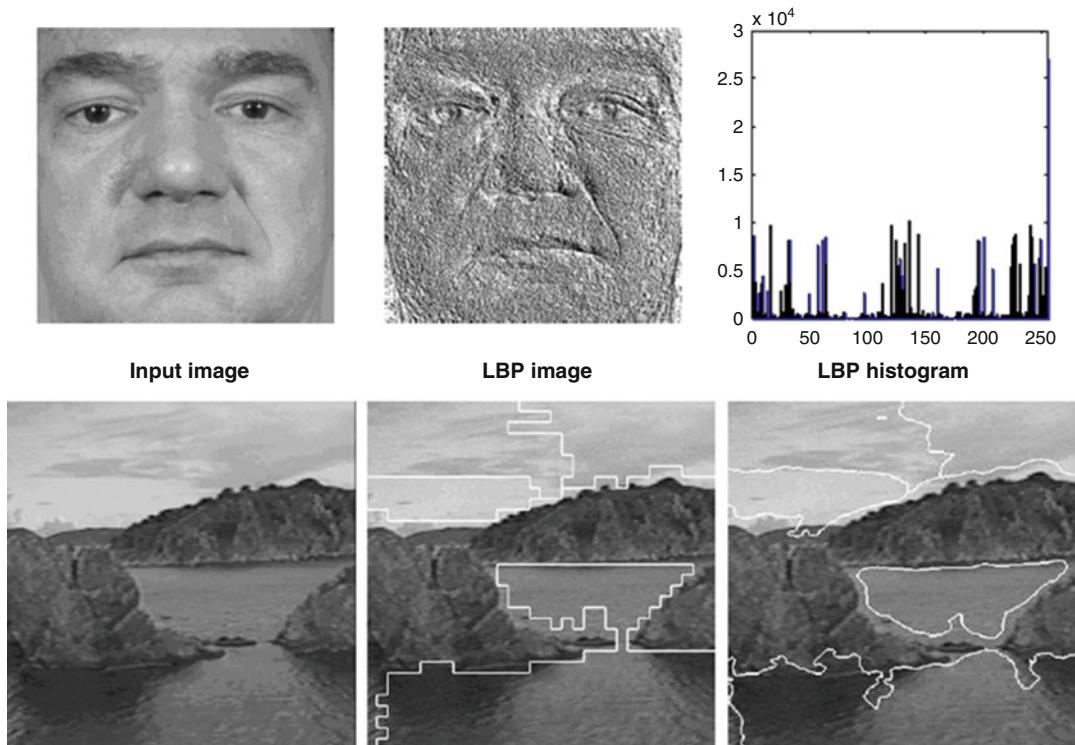


Figure 6.6 (Above) A local binary pattern representation of an image where the LBP is used as an image processing operator, and the corresponding histogram of cumulative LBP features. (Bottom) Segmentation results using LBP texture metrics. (Images courtesy and © Springer Press, from Computer Vision Using Local Binary Patterns, by Matti Pietikäinen and Janne Heikkilä [165])

time. In addition, the LBP is used as an image processing operator, and has been used as a feature descriptor retrofit in SIFT with excellent results, described in this chapter.

In its simplest embodiment, LBP has the goal of creating a binary coded neighborhood descriptor for a pixel. It does this by comparing each pixel against its neighbors using the $>$ operator and encoding the compare results ($1, 0$) into a binary number, as shown in Fig. 6.8. LBP histograms from larger image regions can even be used as signals and passed into a 1D FFT to create a feature descriptor. The Fourier spectrum of the LBP histogram is rotational invariant; see Fig. 6.6. The FFT spectrum can then be concatenated onto the LBP histogram to form a multivariate descriptor, see Varma [770], Vedaldi [887], and Gehler [792] for more details about multivariate descriptors, and applying boosting to weight the features.

As shown in Fig. 6.6, the LBP is used as an image processing operator, region segmentation method, and histogram feature descriptor. The LBP has many applications. An LBP may be calculated over various sizes and shapes using various sizes of forming kernels. A simple 3×3 neighborhood provides basic coverage for local features, while wider areas and kernel shapes are used as well.

Assuming a 3×3 LBP kernel pattern is chosen, this means that there will be 8 pixel compares and up to 2^8 combinations of results for a 256-bin histogram possible. However, it has been shown [18] that reducing the 8-bit 256-bin histogram to use only 58 LBP bins based on *uniform patterns* is the optimal number. The 58 bins or uniform patterns are chosen to represent only two contiguous LBP patterns around the circle, which consists of two connected contiguous segments rather than all

256 possible pattern combinations [15, 165]. The same uniform pattern logic applies to LBPs of dimension larger than 8 bits. So uniform patterns provide both histogram space savings and feature compare-space optimization, since fewer features need be matched (58 instead of all 256).

LBP feature recognition may follow the steps shown in Fig. 6.7.

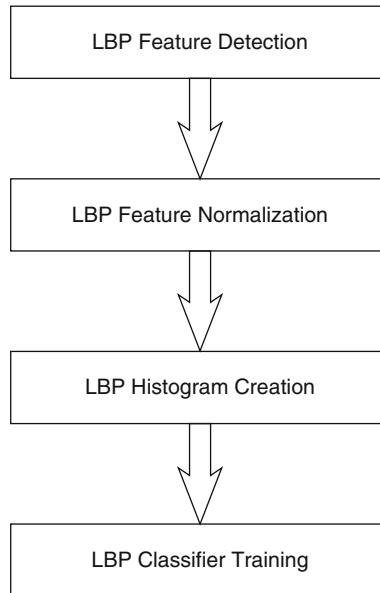


Figure 6.7 LBP feature flow for feature detection. (Image used by permission, © Intel Press, from Building Intelligent Systems)

The LBP is calculated by assigning a binary weighting value to each pixel in the local neighborhood and summing up the pixel compare results as binary values to create a composite LBP value. The LBP contains region information encoded in a compact binary pattern, as shown in Fig. 6.8, so the LBP is thus a binary coded neighborhood texture descriptor.

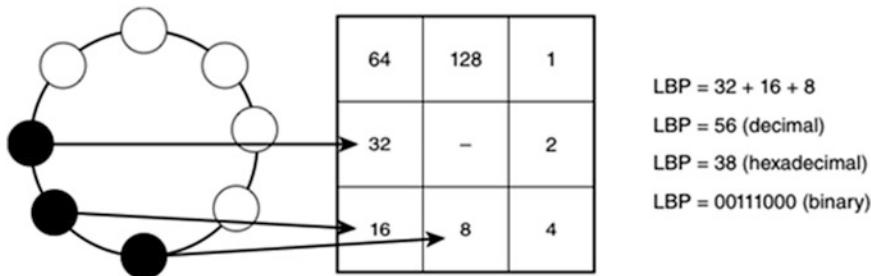


Figure 6.8 Assigned LBP weighting values. (Image used by permission, © Intel Press, from Building Intelligent Systems)

Assuming a 3×3 neighborhood is used to describe the LBP patterns, one may compare the 3×3 rectangular region to a circular region, suggesting 360° directionality at 45° increments, as shown in Fig. 6.9.

The steps involved in calculating a 3×3 LBP are illustrated in Fig. 6.10.

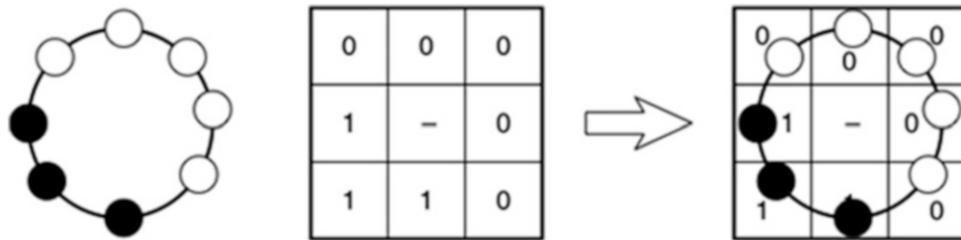
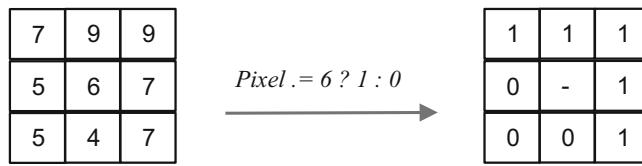


Figure 6.9 The concept of LBP directionality. (Image used by permission, © Intel Press, from Building Intelligent Systems)



```

Pixel[0,0](7) >= 6 ? 1 : 0 = 00000001
Pixel[1,0](9) >= 6 ? 1 : 0 = 00000010
Pixel[2,0](9) >= 6 ? 1 : 0 = 00000100
Pixel[2,1](7) >= 6 ? 1 : 0 = 00001000
Pixel[2,2](7) >= 6 ? 1 : 0 = 00010000
Pixel[1,2](4) >= 6 ? 1 : 0 = 00000000
Pixel[0,2](5) >= 6 ? 1 : 0 = 00000000
Pixel[0,1](5) >= 6 ? 1 : 0 = 00000000
LBP          00011111

```

Figure 6.10 LBP neighborhood comparison

Neighborhood Comparison

Each pixel in the 3×3 region is compared to the center pixel. If the pixel \geq the center pixel, then the LBP records a bit value of 1 for that position, and a bit value of 0 otherwise. See Fig. 6.10

Histogram Composition

Each LBP descriptor over an image region is recorded in a histogram to describe the cumulative texture feature. Uniform LBP histograms would have 56 bins, since only single-connected regions are histogrammed.

Optionally Normalization

The final histogram can be reduced to a smaller number of bins using binary decimation for powers of two or some similar algorithm, such as $256 \rightarrow 32$. In addition, the histograms can be reduced in size by thresholding the range of contiguous bins used for the histogram—for example, by ignoring bins 1–64 if little or no information is binned in them.

Descriptor Concatenation

Multiple LBPs taken over overlapping regions may be concatenated together into a larger histogram feature descriptor to provide better discrimination.

LBP SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Local binary</i>
<i>Feature shape:</i>	<i>Square</i>
<i>Feature pattern:</i>	<i>Pixel region compares with center pixel</i>
<i>Feature density:</i>	<i>Local 3×3 at each pixel</i>
<i>Search method:</i>	<i>Sliding window</i>
<i>Distance function:</i>	<i>Hamming distance</i>
<i>Robustness:</i>	<i>3 (brightness, contrast, *rotation for RILBP)</i>

Rotation Invariant LBP (RILBP)

To achieve rotational invariance, the rotation invariant LBP (RILBP) [165] is calculated by circular bitwise rotation of the local LBP to find the minimum binary value. The minimum value LBP is used as a rotation invariant signature and is recorded in the histogram bins. The RILBP is computationally very efficient.

To illustrate the method, Fig. 6.11 shows a pattern of three consecutive LBP bits; in order to make this descriptor rotation invariant, the value is *left-shifted* until a minimum value is reached.

original	<< 1	<< 2	<< 3	<< 4	<< 5	<< 6	<< 7
							
00010110	00101100	01011000	10110000	01100001	11000010	10000101	000001011

Figure 6.11 Method of calculating the minimum LBP by using circular bit shifting of the binary value to find the minimum value. The LBP descriptor is then rotation invariant

Note that many researchers [163, 164] are extending the methods used for LBP calculation to use refinements such as local derivatives, local median or mean values, trinary or quinary compare functions, and many other methods, rather than the simple binary compare function, as originally proposed.

Dynamic Texture Metric Using 3D LBPs

Dynamic textures are visual features that morph and change as they move from frame to frame; examples include waves, clouds, wind, smoke, foliage, and ripples. Two extensions of the basic LBP used for tracking such dynamic textures are discussed here: VLBP and LBP-TOP.

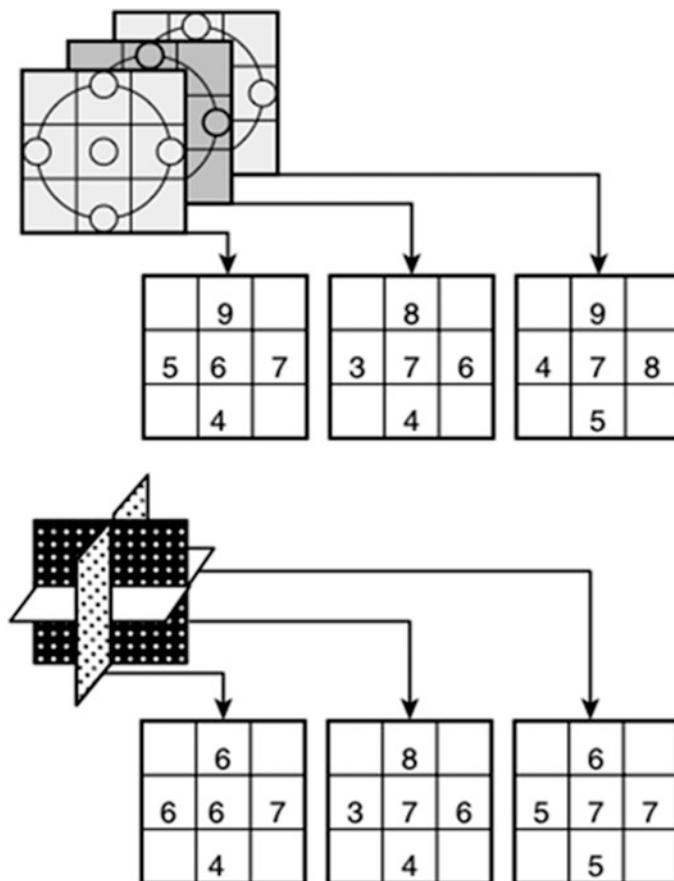


Figure 6.12 (Top) VLBP method [167] of calculating LBPs from parallel planes. (Bottom) LBP-TOP method [168] of calculating LBPs from orthogonal planes. (Image used by permission, © Intel Press, from Building Intelligent Systems)

Volume LBP (VLBP)

To create the VLBP [167] descriptor, first an image volume is created by stacking together at least three consecutive video frames into a volume 3D dataset. Next, three LBPs are taken centered on the selected interest point, one LBP from each parallel plane in the volume, into a summary volume LBP or VLBP, and the histogram of each orthogonal LBP is concatenated into a single dynamic descriptor vector, the VLBP. The VLBP can then be tracked from frame to frame and recalculated to account for dynamic changes in the texture from frame to frame. See Fig. 6.12.

LBP-TOP

The LBP-TOP [168] is created like the VLBP, except that instead of calculating the three individual LBPs from parallel planes, they are calculated from orthogonal planes in the volume (x, y, z) intersecting the interest point, as shown in Fig. 6.12. The 3D composite descriptor is the same size as the VLBP and contains three planes' worth of data. The histograms for each LBP plane are also concatenated for the LBP-TOP like the VLBP.

Other LBP Variants

As shown in Table 6.1, there are many variants of the LBP [165]. Note that the LBP has been successfully used as a replacement for SIFT, SURF, and also as a texture metric.

Table 6.1 LBP variants (from reference [165])

ULBP (Uniform LBP) Uses only 56 uniform bins instead of the full 256 bins possible with 8-bit pixels to create the histogram. The uniform patterns consist of contiguous segments of connected TRUE values.

RLBP (ROBUST LBP) Adds + scale factor to eliminate transitions due to noise ($p_1 - p_2 + \text{SCALE}$)

CS-LBP Circle-symmetric, half as many vectors as LBP, comparison of opposite pixel pairs vs. w/center pixel, useful to reduce LBP bin counts

LBP-HF Fourier spectrum descriptor + LBP

MLBP Median LBP Uses area median value instead of center pixel value for comparison

M-LBP Multiscale LBP combining multiple radii LBPs concatenated

MB-LBP Multiscale Block LBP; compare average pixel values in small blocks

SEMB-LBP: Statistically Effective MB-LBP (SEMB-LBP) uses the percentage in distributions, instead of the number of 0-1 and 1-0 transitions in the LBP and redefines the uniform patterns in the standard LBP. Used effectively in face recognition using GENTLE ADA-BOOSTing [531]

VLBP Volume LBP over adjacent video frames OR within a volume - concatenate histograms together to form a longer vector

LGBP (Local Gabor Binary Pattern) 40 or so Gabor filters are computed over a feature, LBPs are extracted and concatenated to form a long feature vector that is invariant over more scales and orientations

LEP Local Edge Patterns: Edge enhancement (Sobel) prior to standard LBP

EBP Elliptic Binary Pattern Standard LBP but over elliptical area instead of circular

EQP Elliptical Quinary Patterns - LBP extended from binary (2) level resolution to quinary (5) level resolution (-2,-1,0,1,2)

LTP - LBP extended over Ternary range to deal with near constant areas (-1, 0, 1)

LLBP Local line Binary Pattern - calculates LBP over line patterns (cross shape) and then calculates a magnitude metrics using SQRT of SQUARES of each X/Y dimension

TPLBP- [x5]three LBPs are calculated together: the basic LBP for the center pixel, plus two others around adjacent pixels so the total descriptor is a set of overlapping LBP's,

FPLBP- [x5]four LBPs are calculated together: the basic LBP for the center pixel, plus two others around adjacent pixels so the total descriptor is a set of overlapping LBP's, XPLBP –

**NOTE: The TPLBP and FPLBP method can be extended to 3,4,n dimensions in feature space. LARGE VECTORS.*

TBP - Ternary (3) Binary pattern, like LBP, but uses three levels of encoding (1,0,-1) to effectively deal with areas of equal or near equal intensity, uses twobinary patterns (one for + and one for -) concatenated together

ETLP - Elongated Ternary Local Patterns (elliptical + ternary[3] levels

FLBP - Fuzzy LBP where each pixel contributes to more than one bin

PLBP - Probabilistic LBP computes magnitude of difference between each pixel & center pixel (more compute, more storage)

SILTP - Scale invariant LBP using a 3 part piece-wise comparison function to compensate and support intensity scale invariance to deal with image noise

tLBP - Transition Coded LBP, where the encoding is clockwise between adjacent pixels in the LBP

(continued)

Table 6.1 (continued)

dLBP - Direction Coded LBP - similar to CSLBP, but stores both maxima and comparison info (is this pixel greater, less than, or maxima)

CBP - Centralized Binary pattern - center pixel compared to average of all nine kernel neighbors

S-LBP Semantic LBP done in a colorimetric-accurate space (like CIE LAB etc.) over uniform connected LBP circular patterns to find principal direction + arc length used to form a 2D histogram as the descriptor.

F-LBP - Fourier Spectrum of color distance from center pixel to adjacent pixels

LDP - Local Derivate Patterns (higher order derivatives) - basic

LBP is the first order directional derivative, which is combined with additional nth order directional derivatives concatenated into a histogram, more sensitive to noise of course

BLBP - Bayesian LBP - combination of LBP and LTP together using Bayesian methods to optimize towards a more robust pattern

FLS - Filtering, Labeling and Statistical Framework for LBP comparison, translates LBP's or any type of histogram descriptor into vector space allowing efficient comparison "A Bayesian Local Binary Pattern Texture Descriptor"

MB-LBP Multiscale Block LBP - compare average pixel values in small blocks instead of individual pixels, thus a 3x3 pixel PBL will become a 9x9 block LBP where each block is a 3x3 region. The histogram is calculated by scaling the image and creating a rendering at each scale and creating a histogram of each scaled image and concatenating the histograms together.

PM-LBP Pyramid Based MultiStructured LBP - used 5 templates to extract different structural info at varying levels 1) Gaussian filters, 4 anisotropic filters to detect gradient directions

MSLBF - Multiscale Selected Local Binary Features

RILBP - Rotation Invariant LBP rotates the bins (binary LBP value) until minimum value is achieved, the max value is considered rotational invariant. This is the most widely used method for LBP rotational invariance.

ALBP - Adaptive LBP for rotational invariance, instead of shifting to a maximal value as in the standard LBP method, find the dominant vector orientation and shift the vector to the dominant vector orientation

LBPV - Local binary pattern variance - uses local area variance to weight pixel contribution to the LBP, align features to principal orientations, determine non-dominant patterns and reduce their contribution.

OCLBP - Opponent Color LBP - describes color and texture together - each color channel LBP is converted, then opposing color channel LBP's are converted by using one color as the center pixel and another color as the neighborhood, so a total of 9 RGB combination LBP patterns are considered.

SDMCLBP - SDM (co -LBP images for each color are used as the basis for generating occurrence matrices, and then Haralick features are extracted from the images to form a multi dimensional feature space.

MSCLBP - Multi Scale Color Local Binary Patterns (concatenate 6 histograms together)- USES COLOR SPACE COMPONENTS

HUE-LBP OPPONENT-LBP (ALL 3 CHANNELS) nOPPONENT-LBP (COMPUTED OVER 2 CHANNELS), light intensity change, intensity shift, intensity change+shift, color-change color-shift, DEFINE SIX NEW OPERATORS: transformed color LBP (RGB)[subtract mean, divide by STD DEV], opponent LBP, nOpponent LBP, Hue LBP, RGB-LBP, nRGB-LBP [x8] "Multi-scale Color Local Binary Patterns for Visual Object Classes Recognition", Chao ZHU, Charles-Edmond BICHOT, Liming CHEN

3D histograms - 3DRGBLBP [best performance, high memory footprint] - 3D histogram computed over RGB-LBP color image space using uniform pattern minimization to yield 10 levels or patterns per color yielding a large descriptor: $10 \times 10 \times 10 = 1000$ descriptors.

LATCH - *LATCH: Learned Arrangements of Three Patch Codes* [871]

Census

The Census transform [169] is basically an LBP, and like a population census, it uses simple greater-than and less-than queries to count and compare results. Census records pixel comparison results made between the center pixel in the kernel and the other pixels in the kernel region. It employs comparisons and possibly a threshold, and stores the results in a binary vector. The Census transform also uses a feature called the *rank value scalar*, which is the number of pixel values less than the center pixel. The Census descriptor thus uses both a bit vector and a rank scalar.

CENSUS SUMMARY VISION TAXONOMY

<i>Spectra:</i>	<i>Local binary + scalar ranking</i>
<i>Feature shape:</i>	<i>Square</i>
<i>Feature pattern:</i>	<i>Pixel region compares with center pixel</i>
<i>Feature density:</i>	<i>Local 3×3 at each pixel</i>
<i>Search method:</i>	<i>Sliding window</i>
<i>Distance function:</i>	<i>Hamming distance</i>
<i>Robustness:</i>	<i>2 (brightness, contrast)</i>

Modified Census Transform

The Modified Census trasform (MCT) [197] seeks to improve the local binary pattern robustness of the original Census transform. The method uses an ordered comparison of each pixel in the 3×3 neighborhood against the mean intensity of all the pixels of the 3×3 neighborhood, generating a binary descriptor bit vector with bit values set to an intensity lower than the mean intensity of all the pixels. The bit vector can be used to create an MCT image using the MCT value for each pixel. See Fig. 6.13.

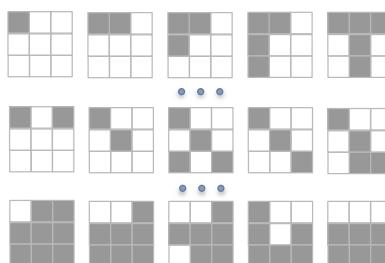


Figure 6.13 Abbreviated set of 15 out of a possible 511 possible binary patterns for a 3×3 MCT. The structure kernels in the pattern set are the basis set of the MCT feature space comparison. The structure kernels form a pattern basis set which can represent lines, edges, corners, saddle points, semicircles, and other patterns

As shown in Fig. 6.13, the MCT relies on the full set of possible 3×3 binary patterns ($2^9 - 1$ or 511 variations) and uses these as a kernel index into the binary patterns as the MCT output, since each binary pattern is a unique signature by itself and highly discriminative. The end result of the MCT is analogous to a nonlinear filter that assigns the output to any of the $2^9 - 1$ patterns in the kernel index. Results show that the MCT results are better than the basic CT for some types of object recognition [197].

BRIEF

As described in Chap. 4, in the section on local binary descriptor point-pair patterns, and illustrated in Fig. 4.11, the BRIEF [124, 125] descriptor uses a random distribution pattern of 256 point-pairs in a local 31×31 region for the binary comparison to create the descriptor. One key idea with BRIEF is to select random pairs of points within the local region for comparison.

BRIEF is a local binary descriptor and has achieved very good accuracy and performance in robotics applications [195]. BRIEF and ORB are closely related; ORB is an oriented version of BRIEF, and the ORB descriptor point-pair pattern is also built differently than BRIEF. BRIEF is known to be not very tolerant of rotation.

BRIEF SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Local binary</i>
<i>Feature shape:</i>	<i>Square centered at interest point</i>
<i>Feature pattern:</i>	<i>Random local pixel point-pair compares</i>
<i>Feature density:</i>	<i>Local 31×31 at interest points</i>
<i>Search method:</i>	<i>Sliding window</i>
<i>Distance function:</i>	<i>Hamming distance</i>
<i>Robustness:</i>	<i>2 (brightness, contrast)</i>

ORB

ORB [126] is an acronym for Oriented BRIEF, and as the name suggests, ORB is based on BRIEF and adds rotational invariance to BRIEF by determining corner orientation using FAST9, followed by a Harris corner metric to sort the keypoints; the corner orientation is refined by intensity centroids using Rosin's method [53]. The FAST, Harris, and Rosin processing are done at each level of an image pyramid scaled with a factor of 1.4, rather than the common octave pyramid scale methods. ORB is discussed in some detail in Chap. 4, in the section on local binary descriptor point-pair patterns, and is illustrated in Fig. 4.11.

It should be noted that ORB is a highly optimized and very well engineered descriptor, since the ORB authors were keenly interested in compute speed, memory footprint, and accuracy. Many of the descriptors surveyed in this section are primarily research projects, with less priority given to practical issues, but ORB focuses on optimizing and practical issues.

Compared to BRIEF, ORB provides an improved training method for creating the local binary patterns for pairwise pixel point sampling. While BRIEF uses random point pairs in a 31×31 window, ORB goes through a training step to find uncorrelated point pairs in the window with high variance and means 0.5, which is demonstrated to work better. For details on visualizing the ORB patterns, see Fig. 4.11.

For correspondence search, ORB uses multi-probe locally sensitive hashing (MP-LSH), which searches for matches in neighboring buckets when a match fails, rather than renavigating the hash tree. The authors report that MP-LSH requires fewer hash tables, resulting in a lower memory footprint. MP-LSH also produces more uniform hash bucket sizes than BRIEF. Since ORB is a binary descriptor based on point-pair comparisons, Hamming distance is used for correspondence.

ORB is reported to be an order of magnitude faster than SURF, and two orders of magnitude faster than SIFT, with comparable accuracy. The authors provide impressive performance results in a test of over 24 NTSC resolution images on the Pascal dataset [126].

ORB*	SURF	SIFT
15.3ms	217.3ms	5228.7ms

*Results reported as measured in reference [126].

ORB SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Local binary + orientation vector</i>
<i>Feature shape:</i>	<i>Square</i>
<i>Feature pattern:</i>	<i>Trained local pixel point-pair compares</i>
<i>Feature density:</i>	<i>Local 31×31 at interest points</i>
<i>Search method:</i>	<i>Sliding window</i>
<i>Distance function:</i>	<i>Hamming distance</i>
<i>Robustness:</i>	<i>3 (brightness, contrast, rotation, limited scale)</i>

BRISK

BRISK [123, 135] is a local binary method using a circular-symmetric pattern region shape and a total of 60 point-pairs as line segments arranged in four concentric rings, as shown in Fig. 4.10 and described in detail in Chap. 4. The method uses point-pairs of both short segments and long segments, and this provides a measure of scale invariance, since short segments may map better for fine resolution and long segments may map better at coarse resolution.

The brisk algorithm is unique, using a novel FAST detector adapted to use scale space, reportedly achieving an order of magnitude performance increase over SURF with comparable accuracy. Here are the main computational steps in the algorithm:

- Detects keypoints using FAST or AGHAST based selection in scale space.
- Performs Gaussian smoothing at each pixel sample point to get the point value.
- Makes three sets of pairs: long pairs, short pairs, and unused pairs (the unused pairs are not in the long pair or the short pair set; see Fig. 4.10).
- Computes gradient between long pairs, sums gradients to determine orientation.
- Uses gradient orientation to adjust and rotate short pairs.
- Creates binary descriptor from short pair point-wise comparisons.

BRISK SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Local binary + orientation vector</i>
<i>Feature shape:</i>	<i>Square</i>
<i>Feature pattern:</i>	<i>Trained local pixel point-pair compares</i>
<i>Feature density:</i>	<i>Local 31×31 at FAST interest points</i>
<i>Search method:</i>	<i>Sliding window</i>
<i>Distance function:</i>	<i>Hamming distance</i>
<i>Robustness:</i>	<i>4 (brightness, contrast, rotation, scale)</i>

FREAK

FREAK [122] uses a novel foveal-inspired multiresolution pixel pair sampling shape with trained pixel pairs to mimic the design of the human eye as a coarse-to-fine descriptor, with resolution highest in the center and decreasing further into the periphery, as shown in Fig. 4.9. In the opinion of this author, FREAK demonstrates many of the better design approaches to feature description; it combines performance, accuracy, and robustness. Note that FREAK is fast to compute, has good discrimination compared to other local binary descriptors such as LBP, Census, BRISK, BRIEF, and ORB, and compares favorably with SIFT.

The FREAK feature training process involves determining the point-pairs for the binary comparisons based on the training data, as shown in Fig. 4.9. The training method allows for a range of descriptor sampling patterns and shapes to be built by weighting and choosing sample points with high variance and low correlation. Each sampling point is taken from the overlapping circular regions, where the value of each sampling point is the Gaussian average of the values in each region. The circular regions are designed in concentric circles of 6 regions in each circle, with small regions in the center, and larger regions towards the edge, similar to the biological retinal distribution of receptor cells with some overlap to adjacent regions, which improves accuracy.

The feature descriptor is thus designed in a coarse-to-fine cascade of four groups of 16 byte coarse-to-fine descriptors containing pixel-pair binary comparisons stored in a vector. The first 16 bytes, the coarse resolution set in the cascade, is normally sufficient to find 90 % of the matching features and to discard nonmatching features. FREAK uses 45 point pairs for the descriptor from a 31×31 pixel patch sampling region.

By storing the point-pair comparisons in four cascades of decreasing resolution pattern vectors, the matching process proceeds from coarse to fine, mimicking the human visual system's saccadic search mechanism, allowing for accelerated matching performance when there is early success or rejection in the matching phase. In summary, the FREAK approach works very well.

FREAK SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Local binary coarse-to-fine + orientation vector</i>
<i>Feature shape:</i>	<i>Square</i>
<i>Feature pattern:</i>	<i>31×31 region pixel point-pair compares</i>
<i>Feature density:</i>	<i>Sparse local at AGAST interest points</i>
<i>Search method:</i>	<i>Sliding window over scale space</i>
<i>Distance function:</i>	<i>Hamming distance</i>
<i>Robustness:</i>	<i>6 (brightness, contrast, rotation, scale, viewpoint, blur)</i>

Spectra Descriptors

Compared to the local binary descriptor group, the spectra group of descriptors typically involves more intense computations and algorithms, often requiring floating point calculations, and may consume considerable memory. In this taxonomy and discussion, *spectra* is simply a quantity that can be measured or computed, such as light intensity, color, local area gradients, local area statistical features and moments, surface normals, and sorted data such 2D or 3D histograms of any spectral

type, such as histograms of local gradient direction. Many of the methods discussed in this section use local gradient information.

Local binary descriptors, as discussed in the previous section, are an attempt to move away from more costly spectral methods to reduce power and increase performance. Local binary descriptors in many cases offer similar accuracy and robustness to the more compute-intensive spectra methods.

SIFT

The Scale Invariant Feature Transform (SIFT) developed by Lowe [153, 170] is the most well-known method for finding interest points and feature descriptors, providing invariance to scale, rotation, illumination, affine distortion, perspective and similarity transforms, and noise. Lowe demonstrates that by using several SIFT descriptors together to describe an object, there is additional invariance to occlusion and clutter, since if a few descriptors are occluded, others will be found [153]. We provide some detail here on SIFT since it is well designed and well known.

SIFT is commonly used as a benchmark against which other vision methods are compared. The original SIFT research paper by author David Lowe was initially rejected several times for publication by the major computer vision journals, and as a result Lowe filed for a patent and took a different direction. According to Lowe, “By then I had decided the computer vision community was not interested, so I applied for a patent and intended to promote it just for industrial applications.”¹ Eventually, the SIFT paper was published and went on to become the most widely cited article in computer vision history!

SIFT is a complete algorithm and processing pipeline, including both an interest point and a feature descriptor method. SIFT includes stages for selecting center-surrounding circular weighted Difference of Gaussian (DoG) maxima interest points in scale space to create scale-invariant keypoints (a major innovation), as illustrated in Fig. 6.14. Feature descriptors are computed surrounding the scale-invariant keypoints. The feature extraction step involves calculating a binned Histogram Of Gradients (HOG) structure from local gradient magnitudes into Cartesian rectangular bins, or into log polar bins using the GLOH variation, at selected locations centered around the maximal response interest points derived over several scales.

The descriptors are fed into a matching pipeline to find the nearest distance ratio metric between closest match and second closest match, which considers a primary match and a secondary match together and rejects both matches if they are too similar, assuming that one or the other may be a false match. The local gradient magnitudes are weighted by a strength value proportional to the pyramid scale level, and then binned into the local histograms. In summary, SIFT is a very well thought out and carefully designed multi-scale localized feature descriptor.

A variation of SIFT for color images is known as CSIFT [171].

Here is the basic SIFT descriptor processing flow (note: the matching stage is omitted since this chapter is concerned with feature descriptors and related metrics):

1. Create a Scale Space Pyramid

¹ <http://yann.lecun.com/ex/pamphlets/publishing-models.html>.

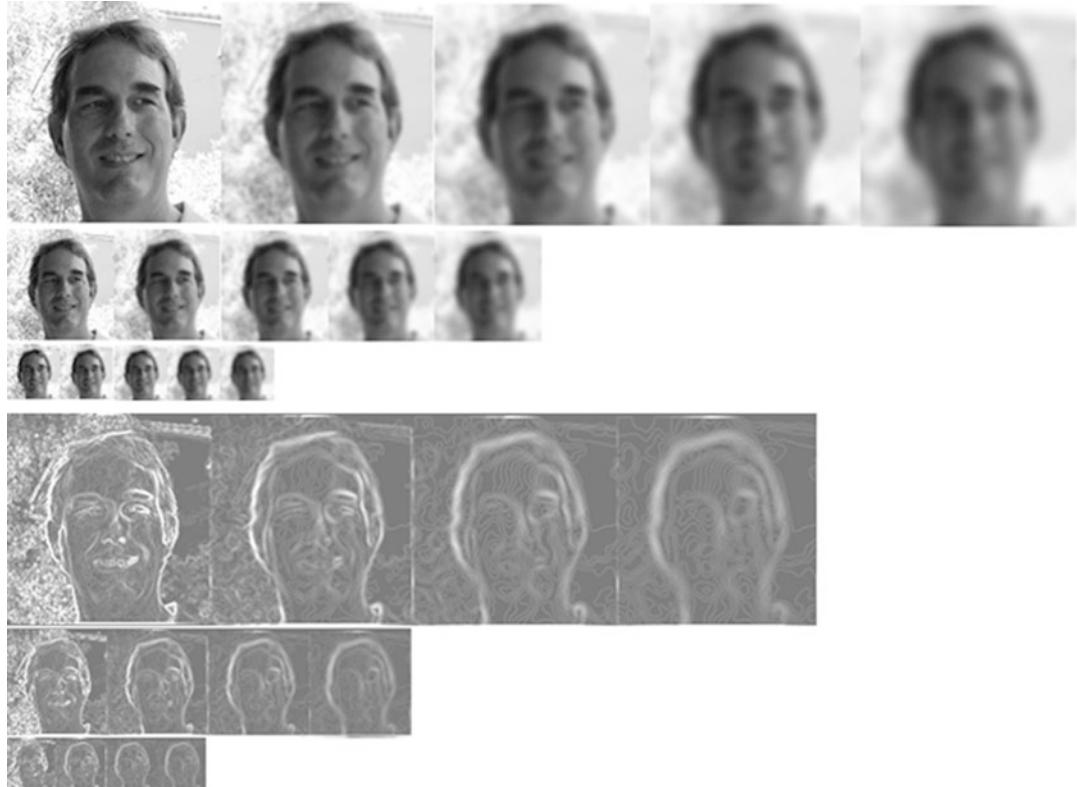


Figure 6.14 (Top) Set of Gaussian Images obtained by convolution with a Gaussian kernel and the corresponding set of DoG images. (Bottom) In octave sets. The DOG function approximates a LOG gradient, or tunable bypass filter. Matching features against the various images in the scaled octave sets yields scale invariant features

An octave scale $n/2$ image pyramid is used with Gaussian filtered images in a scale space. The amount of Gaussian blur is proportional to the scale, and then the Difference of Gaussians (DoG) method is used to capture the interest point extrema maxima and minima in adjacent images in the pyramid. The image pyramid contains five levels. SIFT also uses a double-scale first pyramid level using pixels at two times the original magnification to help preserve fine details. This technique increases the number of stable keypoints by about four times, which is quite significant. Otherwise, computing the Gaussian blur across the original image would have the effect of throwing away the high-frequency details. See Figs. 6.15 and 6.16.

2. Identify Scale-Invariant Interest Points

As shown in Fig. 6.16, the candidate interest points are chosen from local maxima or minima as compared between the 26 adjacent pixels in the DOG images from the three adjacent octaves in the pyramid. In other words, the interest points are scale invariant.

The selected interest points are further qualified to achieve invariance by analyzing local contrast, local noise, and local edge presence within the local 26 pixel neighborhood. Various methods may be used beyond those in the original method, and several techniques are used together to select the best interest points, including local curvature interpolation over small regions, and balancing edge responses to include primary and secondary edges. The keypoints are localized to sub-pixel precision over scale and space. The complete interest points are thus invariant to scale.

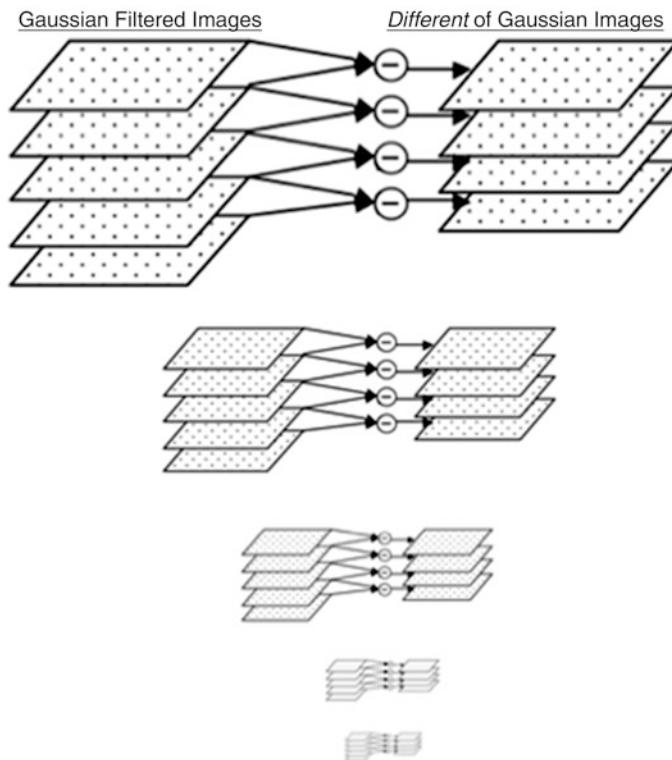


Figure 6.15 SIFT DoG as the simple arithmetic difference between the Gaussian filtered images in the pyramid scale

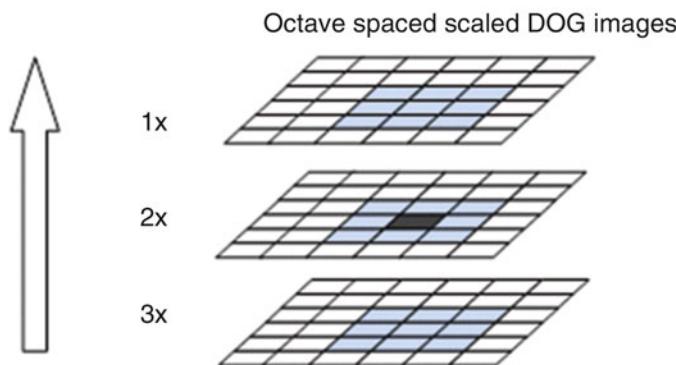


Figure 6.16 SIFT interest point or keypoint detection using scale invariant extrema detection, where the dark pixel in the middle octave is compared within a $3 \times 3 \times 3$ area against its 26 neighbors in adjacent DoG octaves, which includes the eight neighbors at the local scale plus the nine neighbors at adjacent octave scales (up or down)

3. Create Feature Descriptors

A local region or patch of size 16×16 pixels surrounding the chosen interest points is the basis of the feature vector. The magnitude of the local gradients in the 16×16 patch and the gradient orientations are calculated and stored in a HOG (Histogram of Gradients) feature vector, which is

weighted in a circularly symmetric fashion to downweight points farther away from the center interest point around which the HOG is calculated using a Gaussian weighting function.

As shown in Fig. 6.17, the 4×4 gradient binning method allows for gradients to move around in the descriptor and be combined together, thus contributing invariance to various geometric distortions that may change the position of local gradients, similar to the human visual system treatment of the 3D position of gradients across the retina [240]. The SIFT HOG is reasonably invariant to scale, contrast, and rotation. The histogram bins are populated with gradient information using trilinear interpolation, and normalized to provide illumination and contrast invariance.

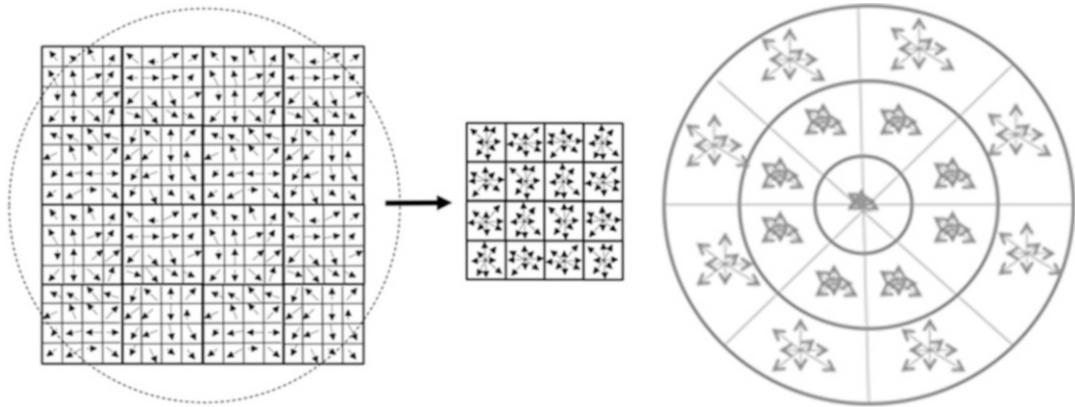


Figure 6.17 (*Left and center*) Gradient magnitude and direction binned into histograms for the SIFT HOG, note the circle over the bin region on the left image suggests how SIFT weights bins farther from center less than bins closer to the center, (*Right*) GLOH descriptors

SIFT can also be performed using a variant of the HOG descriptor called the Gradient Location and Orientation Histogram (GLOH), which uses a log polar histogram format instead of the Cartesian HOG format; see Fig. 6.17. The calculations for the GLOH log polar histogram are straightforward, as shown below from the Cartesian coordinates used for the Cartesian HOG histogram, where the vector magnitude is the hypotenuse and the angle is the arctangent.

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - (x, y-1))^2}$$

$$\theta(x, y) = \text{TAN}^{-1}(L(x, y+1) - L(x, y-1)) / (L(x+1, y) - (x-1, y))$$

As shown in Fig. 6.17, SIFT HOG and GLOH are essentially 3D histograms, and in this case the histogram bin values are gradient magnitude and direction. The descriptor vector size is thus $4 \times 4 \times 8 = 128$ bytes. The 4×4 descriptor (center image) is a set of histograms of the combined eight-way gradient direction and magnitude of each 4×4 group in the left image, in Cartesian coordinates, while the GLOH gradient magnitude and direction are binned in polar coordinate spaced into 17 bins over a greater binning region. SIFT-HOG (left image) also uses a weighting factor to smoothly reduce the contribution of gradient information in a circularly symmetric fashion with increasing distance from the center.

Table 6.2 SIFT compute complexity (from Vinukonda [172])

SIFT Pipeline Step	Complexity	Number of Operations
Gaussian blurring pyramid	$\ominus N^2 U^2 s$	$4N^2 W^2 s$
Difference of Gaussian pyramid	$\ominus sN^2$	$4N^2 s$
Scale-space extrema detection	$\ominus sN^2$	$104sN^2$
Keypoint detection	$\ominus \alpha sN^2$	$100s\alpha N^2$
Orientation assignment	$\ominus sN^2(1 - \alpha\beta)$	$48sN^2$
Descriptor generation	$\ominus(x^2 N^2(\alpha\beta + \gamma))$	$\ominus 1520x^2 N^2(\alpha\beta + \gamma) N^2$

Overall compute complexity for SIFT is high [172], as shown in Table 6.2. Note that feature description is most compute-intensive owing to all the local area gradient calculations for orientation assignment and descriptor generation including histogram binning with trilinear interpolation. The gradient orientation histogram developed in SIFT is a key innovation that provides substantial robustness.

The resulting feature vector for SIFT is 128 bytes. However, methods exist to reduce the dimensionality and vary the descriptor, which are discussed next.

SIFT SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Local gradient magnitude + orientation</i>
<i>Feature shape:</i>	<i>Square, with circular weighting</i>
<i>Feature pattern:</i>	<i>Square with circular-symmetric weighting</i>
<i>Feature density:</i>	<i>Sparse at local 16×16 DoG interest points</i>
<i>Search method:</i>	<i>Sliding window over scale space</i>
<i>Distance function:</i>	<i>Euclidean distance (*or Hellinger distance with RootSIFT retrofit)</i>
<i>Robustness:</i>	<i>6 (brightness, contrast, rotation, scale, affine transforms, noise)</i>

SIFT-PCA

The SIFT-PCA method developed by Ke and Suthankar [175] uses an alternative feature vector derived using principal component analysis (PCA), based on the normalized gradient patches rather than the weighted and smoothed histograms of gradients, as used in SIFT. In addition, SIFT-PCA reduces the dimensionality of the SIFT descriptor to a smaller set of elements. SIFT originally was reported using 128 vectors, but using SIFT-PCA the vector is reduced to a smaller number such as 20 or 36.

The basic steps for SIFT-PCA are as follows:

1. Construct an eigenspace based on the gradients from the local 41×41 image patches resulting in a 3042 element vector; this vector is the result of the normal SIFT pipeline.
2. Compute local image gradients for the patches.
3. Create the reduced-size feature vector from the eigenspace using PCA on the covariance matrix of each feature vector.

SIFT-PCA is shown to provide some improvements over SIFT in the area of robustness to image warping, and the smaller size of the feature vector results in faster matching speed. The authors note that while PCA in general is not optimal as applied to image patch features, the method works well for the SIFT style gradient patches that are oriented and localized in scale space [175].

SIFT-GLOH

The Gradient Location and Orientation Histogram (GLOH) [136] method uses polar coordinates and radially distributed bins rather than the Cartesian coordinate style histogram binning method used by SIFT. It is reported to provide greater accuracy and robustness over SIFT and other descriptors for some ground truth datasets [136]. As shown in Fig. 6.17, GLOH uses a set of 17 radially distributed bins to sum the gradient information in polar coordinates, yielding a 272-bin histogram. The center bin is not direction oriented. The size of the descriptor is reduced using PCA. GLOH has been used to retrofit SIFT.

SIFT-SIFER Retrofit

The Scale Invariant Feature Detector with Error Resilience (SIFER) [216] method provides alternatives to the standard SIFT pipeline, yielding measurable accuracy improvements reported to be as high as 20 % for some criteria. However, the accuracy comes at a cost, since the performance is about twice as slow as SIFT. The major contributions of SIFER include improved scale-space treatment using a higher granularity image pyramid representation, and better scale-tuned filtering using a cosine modulated Gaussian filter.

The major steps in the method are shown in Table 6.3. The scale-space pyramid is blurred using a cosine modulated Gaussian (CMG) filter, which allows each scale of the octave to be subdivided into six scales, so the result is better scale accuracy.

Table 6.3 Comparison of SIFT, SURF, and SIFER pipelines (adapted from [216])

	SIFT	SURF	SIFER
Scale Space Filtering	Gaussian 2nd derivative	Gaussian 2nd derivative	Cosine Modulated Gaussian
Detector	LoG	Hessian	Wavelet Modulus Maxima
Filter approximation level	OK accuracy	OK accuracy	Good accuracy
Optimizations	DoG for gradient	Integral images, constant time	Convolution, constant time
Image up-sampling	2x	2x	Not used
Sub-sampling	Yes	Yes	Not used

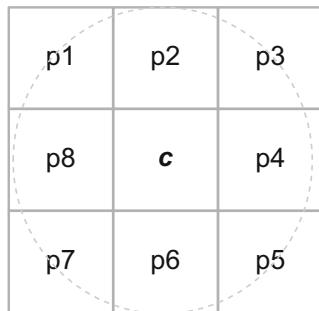
Since the performance of the CMG is not good, SIFER provides a fast approximation method that provides reasonable accuracy. Special care is given to the image scale and the filter scale to increase accuracy of detection, thus the cosine is used as a bandpass filter for the Gaussian filter to match the scale as well as possible, tuning the filter in a filter bank over scale space with well-matched filters for each of the six scales per octave. The CMG provides more error resilience than the SIFT Gaussian second derivative method.

SIFT CS-LBP Retrofit

The SIFT-CSLBP retrofit method [165, 194] combines the best attributes of SIFT and the center symmetric LBP (CS-LBP) by replacing the SIFT gradient calculations with much more compute-

efficient LBP operators, and by creating similar histogram-binned orientation feature vectors. LBP is computationally simpler both to create and to match than the SIFT descriptor.

The CS-LBP descriptor begins by applying an adaptive noise-removal filter (a Weiner filter is the variety used in this work) to the local patch for adaptive noise removal, which preserves local contrast. Rather than computing all 256 possible 8-bit local binary patterns, the CS-LBP only computes 16 center symmetric patterns for reduced dimensionality, as shown in Fig. 6.18.



$$\begin{aligned}
 \text{LPB} = & \\
 & s(p1 - c)^0 + \\
 & s(p2 - c)^1 + \\
 & s(p3 - c)^2 + \\
 & s(p4 - c)^3 + \\
 & s(p5 - c)^4 + \\
 & s(p6 - c)^5 + \\
 & s(p7 - c)^6 + \\
 & s(p8 - c)^7
 \end{aligned}
 \quad
 \begin{aligned}
 \text{CS-LPB} = & \\
 & s(p1 - p5)^0 + \\
 & s(p2 - p6)^1 + \\
 & s(p3 - p7)^2 + \\
 & s(p4 - p8)^3
 \end{aligned}$$

Figure 6.18 CS-LBP sampling pattern for reduced dimensionality

Instead of weighting the histogram bins using the SIFT circular weighting function, no weighting is used, which reduces compute. Like SIFT, the CS-LBP binning method uses a 3×3 region Cartesian grid; simpler bilinear interpolation for binning is used, rather than trilinear, as in SIFT. Overall, the CS-LCP retrofit method simplifies the SIFT compute pipeline and increases performance with comparable accuracy; greater accuracy is reported for some datasets. See Table 6.4.

Table 6.4 SIFT and CSLBP retrofit performance (as per reference [194])

	Feature extraction	Descriptor construction	Descriptor normalization	Total ms time
CS-LBP 256	0.1609	0.0961	0.007	0.264
CS-LBP 128	0.1148	0.0749	0.0022	0.1919
SIFT 128	0.4387	0.1654	0.0025	0.6066

RootSIFT Retrofit

The RootSift method [166] provides a set of simple, key enhancements to the SIFT pipeline, resulting in better compute performance and slight improvements in accuracy, as follows:

- **Hellinger distance:** RootSIFT uses a simple performance optimization of the SIFT object retrieval pipeline using Hellinger distance instead of Euclidean distance for correspondence. All other portions of the SIFT pipeline remain the same; K-means is still employed to build the feature vector set, and other approximate nearest neighbor methods may still be used as well for larger feature vector sets. The authors claim a simple modification to SIFT code to perform the Hellinger distance optimization instead of Euclidean distance can be a simple set of one-line changes to the code. Other enhancements in RootSIFT are optional, discussed next.

- **Feature augmentation:** This method increases total recall. Developed by Turcot and Lowe [324], it is applied to the features. Feature vectors or visual words from similar views of the same object in the database are associated into a graph used for finding correspondence among similar features, instead of just relying on a single feature.
- **Discriminative query expansion (DQE):** This method increases query expansion during training. Feature vectors within a region of proximity are associated by averaging into a new feature vector useful for requeries into the database, using both positive and negative training data in a linear SVM; better correspondence is reported in reference [166].

By combining the three innovations described above into the SIFT pipeline, performance, accuracy, and robustness are shown to be significantly improved.

Table 6.5 Major differences between CenSurE and SIFT and SURF (adapted from reference [177])

	CenSurE	SIFT	SURF
Resolution	Every pixel	Pyramid sub-sampled	Pyramid sub-sampled
Edge filter method	Harris	Hessian	Hessian
Scale space extrema method	Laplace, Center Surround	Laplace, DOG	Hessian, DOB
Rotational invariance	Approximate	yes	no
Spatial resolution in scale	Full	subsampled	Subsampled

CenSurE and STAR

The Center Surround Extrema or CenSurE [177] method provides a true multi-scale descriptor, creating a feature vector using full spatial resolution at all scales in the pyramid, in contrast to SIFT and SURF, which find extrema at subsampled pixels that compromises accuracy at larger scales. CenSurE is similar to SIFT and SURF, but some key differences are summarized in Table 6.5. Modifications have been made to the original CenSurE algorithm in OpenCV, which goes by the name of STAR descriptor.

The authors have paid careful attention to creating methods which are computationally efficient, memory efficient, with high performance and accuracy [177]. CenSurE defines an optimized approach to find extrema by first using the Laplacian at all scales, followed by a filtering step using the Harris method to discard corners with weak responses.

The major innovations of CenSurE over SIFT and SURF are as follows:

1. Use of bilevel center-surround filters, as shown in Fig. 6.19, including Difference of Boxes (DoB), Difference of Octagons (DoO) and Difference of Hexagons (DoH) filters, octagons and hexagons are more rotationally invariant than boxes. DoB is computationally simple and may be computed with integral images vs. the Gaussian scale space method of SIFT. The DoO and DoH filters are also computed quickly using a modified integral image method. Circle is the desired shape, but more computationally expensive.
2. To find the extrema, the DoB filter is computed using a seven-level scale space of filters at each pixel, using a $3 \times 3 \times 3$ neighborhood. The scale space search is composed using center-surround Haar-like features on non-octave boundaries with filter block sizes [1,2,3,4,5,6,7] covering 2.5 octaves between [1 and 7] yielding five filters. This scale arrangement provides more

discrimination than an octave scale. A threshold is applied to eliminate weak filter responses at each level, since the weak responses are likely not to be repeated at other scales.

3. Nonrectangular filter shapes, such as octagons and hexagons, are computed quickly using combinations of overlapping integral image regions; note that octagons and hexagons avoid artifacts caused by rectangular regions and increase rotational invariance; see Fig. 6.19.
4. CenSurE filters are applied using a fast, modified version of the SURF method called Modified Upright SURF (MU-SURF) [180, 181], discussed later with other SURF variants, which pays special attention to boundary effects of boxes in the descriptor by using an expanded set of overlapping subregions for the HAAR responses.



Figure 6.19 CenSurE bilevel center surround filter shape approximations to the Laplacian using binary kernel values of 1 and -1 , which can be efficiently implemented using signed addition rather than multiplication. Note that the circular shape is the desired shape, but the other shapes are easier to compute using integral images, especially the rectangular method

CENSURE SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Center-surround shaped bi-level filters</i>
<i>Feature shape:</i>	<i>Octagons, circles, boxes, hexagons</i>
<i>Feature pattern:</i>	<i>Filter shape masks, 24×24 largest region</i>
<i>Feature density:</i>	<i>Sparse at Local interest points</i>
<i>Search method:</i>	<i>Dense sliding window over scale space</i>
<i>Distance function:</i>	<i>Euclidean distance</i>
<i>Robustness:</i>	<i>5 (brightness, contrast, rotation, scale, affine transforms)</i>

Correlation Templates

One of the most well known and obvious methods for feature description and detection, as used as the primary feature in basic deep learning architectures discussed in Chaps. 9 and 10, takes an image of the complete feature and searches for it by direct pixel comparison—this is known as *correlation*. Correlation involves stepping a sliding window containing a first pixel region template across a second image region template and performing a simple pixel-by-pixel region comparison using a method such as sum of differences (SAD); the resulting score is the correlation.

Since image illumination may vary, typically the correlation template and the target image are first intensity normalized, typically by subtracting the mean and dividing by the standard deviation; however, contrast leveling and LUT transform may also be used. Correlation is commonly implemented in the spatial domain on rectangular windows, but can be used with frequency domain methods as well [4, 9].

Correlation is used in video-based target tracking applications where translation as orthogonal motion from frame-to-frame over small adjacent regions predominates. For example, video motion encoders find the displacement of regions or blocks within the image using correlation, since usually

small block motion in video is orthogonal to the Cartesian axis and maps well to simple displacements found using correlation. Correlation can provide sub-pixel accuracy between 1/4 and 1/20 of a pixel, depending on the images and methods used; see reference [143]. For video encoding applications, correlation allows for the motion vector displacements of corresponding blocks to be efficiently encoded and accurately computed. Correlation is amenable to fixed function hardware acceleration.

Variations on correlation include cross-correlation (sliding dot product), normalized cross-correlation (NCC), zero-mean normalized cross-correlation (ZNCC), and texture auto correlation (TAC).

In general, correlation is a good detector for orthogonal motion of a constant-sized mono-space pattern region. It provides sub-pixel accuracy, has limited robustness and accuracy over illumination, but little to no robustness over rotation or scale. However, to overcome these robustness problems, it is possible to accelerate correlation over a scale space, as well as various geometric translations, using multiple texture samplers in a graphics processor in parallel to rapidly scale and rotate the correlation templates. Then, the correlation matching can be done either via SIMD SAD instructions or else using the fast fixed function correlators in the video encoding engines.

Correlation is illustrated in Fig. 6.20.

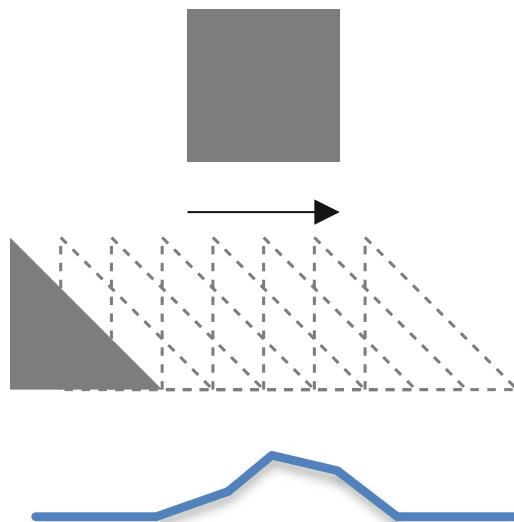


Figure 6.20 Simplified model of digital correlation using a triangular template region swept past a rectangular region. The best correlation is shown at the location of the highest point

CORRELATION SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Correlation</i>
<i>Feature shape:</i>	<i>Square, rectangle</i>
<i>Feature pattern:</i>	<i>Dense</i>
<i>Feature density:</i>	<i>Variable sized kernels</i>
<i>Search method:</i>	<i>Dense sliding window</i>
<i>Distance function:</i>	<i>SSD typical, others possible</i>
<i>Robustness:</i>	<i>I (illumination, sub-pixel accuracy)</i>

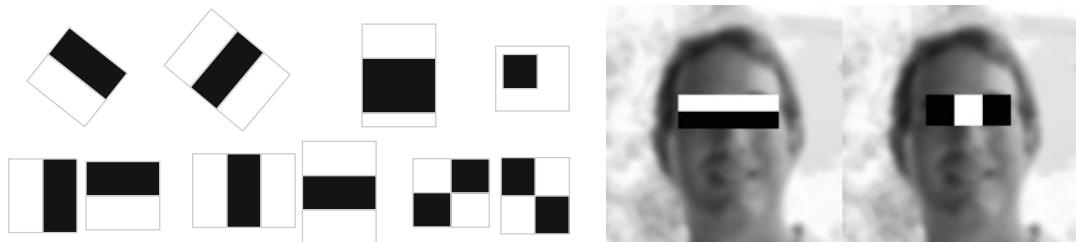


Figure 6.21 Example HAAR-like features

HAAR Features

HAAR-like features [4, 9] were popularized in the field of computer vision by the Viola–Jones [178] algorithm. HAAR features are based on specific sets of rectangle patterns, as shown in Fig. 6.21, which approximate the basic HAAR wavelets, where each HAAR feature is composed of the average pixel value of pixels within the rectangle. This is efficiently computed using integral images.

By using the average pixel value in the rectangular feature, the intent is to find a set of small patterns in adjacent areas where brighter or darker region adjacency may reveal a feature—for example, a bright cheek next to a darker eye socket. However, HAAR features have drawbacks, since rectangles by nature are not rotation invariant much beyond 15° . Also, the integration of pixel values within the rectangle destroys fine detail.

Depending on the type of feature to be detected, such as eyes, a specific set of HAAR feature is chosen to reveal eye/cheek details and eye/nose details. For example, HAAR patterns with two rectangles are useful for detecting edges, while patterns with three rectangles can be used for lines, and patterns with an inset rectangle or four rectangles can be used for single-object features. Note that HAAR features may be a rotated set.

Of course, the scale of the HAAR patterns is an issue, and since a given HAAR feature only works with an image of appropriate scale. Image pyramids are used for HAAR feature detection, along with other techniques for stepping the search window across the image in optimal grid sizes for a given application. Another method to address feature scale is to use a wider set of scaled HAAR features to perform the pyramiding in the feature space rather than the image space. One method to address HAAR feature granularity and rectangular shape is to use overlapping HAAR features to approximate octagons and hexagons; see the CenSurE and STAR methods in Fig. 6.19.

HAAR features are closely related to wavelets [219, 326]. Wavelets can be considered as an extension of the earlier concept of Gabor functions [179, 325]. We provide only a short discussion of wavelets and Gabor functions here; more discussion was provided in Chap. 2. Wavelets are an *orthonormal* set of small duration functions. Each set of wavelets is designed to meet various goals to locate short-term signal phenomenon. There is no single wavelet function; rather, when designing wavelets, a mother wavelet is first designed as the basis of the wavelet family, and then daughter wavelets are derived using translation and compression of the mother wavelet into a basis set. Wavelets are used as a set of nonlinear basis functions, where each basis function can be designed as needed to optimally match a desired feature in the input function. So, unlike transforms which use a uniform set of basis functions like the Fourier transform, composed of SIN and COS functions, wavelets use a dynamic set of basis functions that are complex and nonuniform in nature. Wavelets can be used to describe very complex short-term features, and this may be an advantage in some feature detection applications.

However, compared to integral images and HAAR features, wavelets are computationally expensive, since they represent complex functions in a complex domain. HAAR 2D basis functions are commonly used owing to the simple rectangular shape and computational simplicity, especially when HAAR features are derived from integral images.

HAAR SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Integral box filter</i>
<i>Feature shape:</i>	<i>Square, rectangle</i>
<i>Feature pattern:</i>	<i>Dense</i>
<i>Feature density:</i>	<i>Variable-sized kernels</i>
<i>Search method:</i>	<i>Grid search typical</i>
<i>Distance function:</i>	<i>Simple difference</i>
<i>Robustness:</i>	<i>I (illumination)</i>

Viola–Jones with HAAR-Like Features

The Viola–Jones method [178] is a feature detection pipeline framework based on HAAR-like features using a perceptron learning algorithm to train a detector matching network that consists of three major parts:

1. Integral images used to rapidly compute HAAR-like features.
2. The ADA-BOOST learning algorithm to create a strong pattern matching and classifier network by combining strong classifiers with good matching performance with weak classifiers that have been “boosted” by adjusting weighting factors during the training process.
3. Combining classifiers into a detector cascade or funnel to quickly discard unwanted features at early stages in the cascade.

Since thousands of HAAR pattern matches may be found in a single image, the feature calculations must be done quickly. To make the HAAR pattern match calculation rapidly, the entire image is first processed into an integral image. Each region of the image is searched for known HAAR features using a sliding window method stepped at some chosen interval, such as every n pixels, and the detected features are fed into a classification funnel known as a *HAAR Cascade Classifier*. The top of the funnel consists of feature sets which yield low false positives and false negatives, so the first-order results of the cascade contain high-probability regions of the image for further analysis. The HAAR features become more complex progressing deeper into the funnel of the cascade. With this arrangement, images regions are rejected as soon as possible if the desired HAAR features are not found, minimizing processing overhead.

A complete HAAR feature detector may combine hundreds or thousands of HAAR features together into a final classifier, where not only the feature itself may be important but also the spatial arrangements of features—for example, the distance and angular relationships between features could be used in the classifier.

SURF

The Speeded-up Robust Features Method (SURF) [152] operates in a scale space and uses a fast Hessian detector based on the determinant maxima points of the Hessian matrix. SURF uses a scale space over a $3 \times 3 \times 3$ neighborhood to localize bloblike interest point features. To find feature orientation, a set of HAAR-like feature responses are computed in the local region surrounding each interest point within a circular radius, computed at the matching pyramid scale for the interest point.

The dominant orientation assignment for the local set of HAAR features is found, as shown in Fig. 6.22, using a sliding sector window of size $\frac{\pi}{3}$. This sliding sector window is rotated around the interest point at intervals. Within the sliding sector region, all HAAR features are summed. This includes both the horizontal and vertical responses, which yield a set of orientation vectors; the largest vector is chosen to represent dominant feature orientation. By way of comparison, SIFT uses a histogram of gradient directions to record orientation.

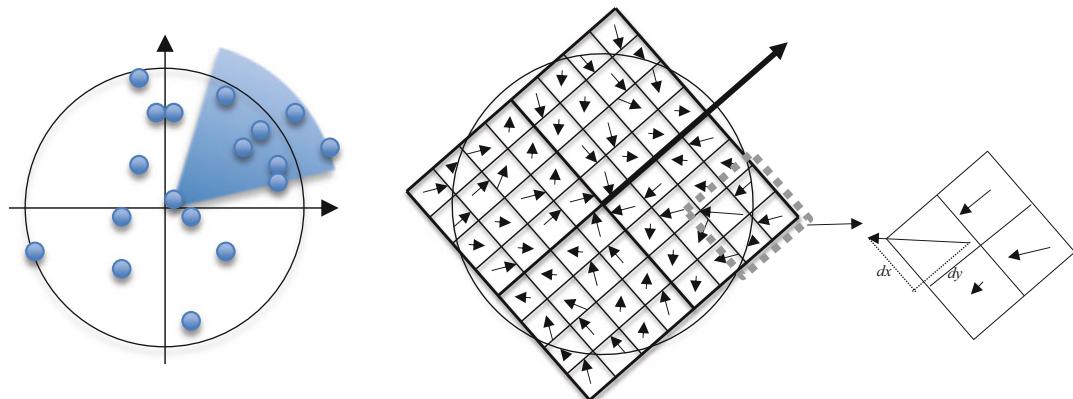


Figure 6.22 (Left) The sliding sector window used in SURF to compute the dominant orientation of the HAAR features to add rotational invariance to the SURF features. (Right) The feature vector construction process, showing a grid containing a 4×4 region subdivided into 4×4 subregions and 2×2 subdivisions

To create the SURF descriptor vector, a rectangular grid of 4×4 regions is established surrounding the interest point, similar to SIFT, and each region of this grid is split into 4×4 subregions. Within each subregion, the HAAR wavelet response is computed over 5×5 sample points. Each HAAR response is weighted using a circularly symmetric Gaussian weighting factor, where the weighting factor decreases with distance from the center interest point, which is similar to SIFT. Each feature vector contains four parts:

$$v = \left(\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y| \right)$$

The wavelet responses d_x and d_y for each subregion are summed, and the absolute value of the responses $|d_x|$ and $|d_y|$ provide polarity of the change in intensity. The final descriptor vector is $4 \times 4 \times 4$: 4×4 regions with four parts per region, for a total vector length of 64. Of course, other vector lengths can be devised by modifying the basic method.

As shown in Fig. 6.22, the SURF gradient grid is rotated according to the dominant orientation, computed during the sliding sector window process, and then the wavelet response is computed in

each square region relative to orientation for binning into the feature vector. Each of the wavelet directional sums d_x , d_y , $|d_x|$, $|d_y|$ is recorded in the feature vector.

The SURF and SIFT pipeline methods are generally comparable in implementation steps and final accuracy, but SURF is one order of magnitude faster to compute than SIFT, as compared in an ORB benchmarking test [126]. However, the local binary descriptors, such as ORB, are another order of magnitude faster than SURF, with comparable accuracy for many applications [126]. For more information, see the section earlier in this chapter on local binary descriptors.

SURF SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Integral box filter + orientation vector</i>
<i>Feature shape:</i>	<i>HAAR rectangles</i>
<i>Feature pattern:</i>	<i>Dense</i>
<i>Feature density:</i>	<i>Sparse at Hessian interest points</i>
<i>Search method:</i>	<i>Dense sliding window over scale space</i>
<i>Distance function:</i>	<i>Mahalanobis or Euclidean</i>
<i>Robustness:</i>	<i>4 (scale, rotation, illumination, noise)</i>

Variations on SURF

A few variations on the SURF descriptor [180, 181] are worth discussing, as shown in Table 6.6. Of particular interest are the G-SURF methods [180], which use a differential geometry concept [182] of a local region gauge coordinate system to compute the features. Since gauge coordinates are not global but, rather, local to the image feature, gauge space features carry advantages for geometrical accuracy.

Table 6.6 SURF variants (as discussed in Alcantarilla et al. [180])

SURF	Circular Symmetric Gaussian Weighting Scheme, 20x20 grid
U-SURF [181]	Faster version of SURF, only upright features are used; no orientation. Like M-SURF except calculated upright “U” with no rotation of the grid, uses a 20x20 grid, no overlapping HAAR features, modified Gaussian weighting scheme, bilinear interpolation between histogram bins.
M-SURF MU-SURF [181]	Circular symmetric Gaussian weighting scheme computed in two steps instead of one as for normal SURF, 24x24 grid using overlapping HAAR features, rotation orientation left out in MU-SURF version.
G-SURF, GU-SURF [180]	Instead of HAAR features, substitutes 2 nd order gauge derivatives in Gauge coordinate space, no Gaussian weighting, 20x20 grid. Gauge derivatives are rotation and translation invariant, while the HAAR features are simple rectangles, and rectangles have poor rotational invariance, maybe +/-15 degrees at best.
MG-SURF [180]	Same as M-SURF, but uses gauge derivatives.
NG-SURF [180]	N = No Gaussian weighting as in SURF; same as SURF but no Gaussian weighting applied, allows for comparison between gauge derivative features and HAAR features.

Histogram of Gradients (HOG) and Variants

The Histogram of Gradients (HOG) method [98] is intended for image classification, and relies on computing local region gradients over a dense grid of overlapping blocks, rather than at interest points. HOG is appropriate for some applications, such as person detection, where the feature in the image is quite large.

HOG operates on raw data; while many methods rely on Gaussian smoothing and other filtering methods to prepare the data, HOG is designed specifically to use all the raw data without introducing filtering artifacts that remove fine details. The authors show clear benefits using this approach. It is a trade-off: *filtering artifacts* such as smoothing vs. *image artifacts* such as fine details. The HOG method shows preferential results for the raw data. See Fig. 4.12, showing a visualization of a HOG descriptor.

Major aspects in the HOG method are as follows:

- Raw RGB image is used with no color correction or noise filtering, using other color spaces and color gamma adjustment provided little advantage for the added cost.
- Prefers a 64×128 sliding detector window; 56×120 and 48×112 sized windows were also tested. Within this detector window, a total of 8×16 8×8 pixel block regions are defined for computation of gradients. Block sizes are tunable.
- For each 8×8 pixel block, a total of 64 local gradient magnitudes are computed. The preferred method is simple line and column derivatives $[-1,0,1]$ in x/y ; other gradient filter methods are tried, but larger filters with or without Gaussian filtering degrade accuracy and performance. Separate gradients are calculated for each color channel.
- Local gradient magnitudes are binned into a 9-bin histogram of edge orientations, quantizing dimensionality from 64 to 9, using bilinear interpolation; <9 bins produce poorer accuracy, >9 bins does not seem to matter. Note that either rectangular R-HOG or circular log polar C-HOG binning regions can be used.
- Normalization of gradient magnitude histogram values to unit length to provide illumination invariance. Normalization is performed in groups, rather than on single histograms. Overlapping 2×2 blocks of histograms are used within the detector window; the block overlapping method reduces sharp artifacts, and the 2×2 region size seems to work best.
- For the 64×128 pixel detector window method, a total of 128 8×8 pixel blocks are defined. Each 8×8 block has four cells for computing separate 9-bin histograms. The total descriptor size is then $8 \times 16 \times 4 \times 9 = 4608$.

Note that various formulations of the sliding window and block sizes are used for dealing with specific application domains. See Fig. 4.12, showing a visualization of HOG descriptor computed using 7×15 8×8 pixel cells. Key findings from the HOG [98] design approach include:

- The abrupt edges at fine scales in the raw data are required for accuracy in the gradient calculations, and post-processing and normalizing the gradient bins later works well.
- L2 style block normalization of local contrast is preferred and provides better accuracy over global normalization; note that the local region blocks are overlapped to assist in the normalization.
- Dropping the L2 block normalization stage during histogram binning reduces accuracy by 27 %.
- HOG features perform much better than HAAR-style detectors, and this makes sense when we consider that a HAAR wavelet is an integrated directionless value, while gradient magnitude and direction over the local HOG region provides a richer spectra.

HOG SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Local region gradient histograms</i>
<i>Feature shape:</i>	<i>Rectangle or circle</i>
<i>Feature pattern:</i>	<i>Dense 64×128 typical rectangle</i>
<i>Feature density:</i>	<i>Dense overlapping blocks</i>
<i>Search method:</i>	<i>Grid over scale space</i>
<i>Distance function:</i>	<i>Euclidean</i>
<i>Robustness:</i>	<i>4 (illumination, viewpoint, scale, noise)</i>

PHOG and Related Methods

The Pyramid Histogram of Oriented Gradients (PHOG) [183] method is designed for global or regional image classification, rather than local feature detection. PHOG combines regional HOG features with whole image area features using spatial relationships between features spread across the entire image in an octave grid region subdivision; see Fig. 6.23.

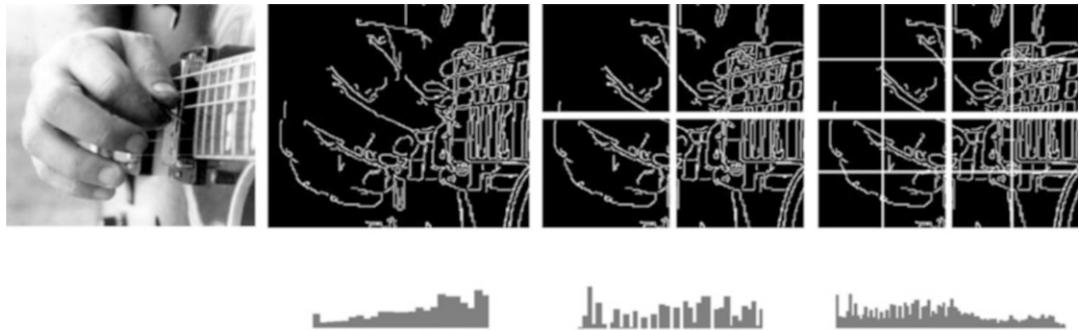


Figure 6.23 Set of PHOG descriptors computed over the whole image, using octave grid cells to bound the edge information. (Center Left) A single histogram. (Center right) Four histograms shown concatenated together. (Right) Sixteen histograms shown concatenated

PHOG is similar to related work using a coarse-to-fine grid of region histograms called Spatial Pyramid Matching by Lazebni, Schmid, and Ponce [516], using histograms of oriented edges and SIFT features to provide multi-class classification. It is also similar to earlier work on pyramids of concatenated histogram features taken over a progressively finer grid, called Pyramid Match Kernel and developed by Grauman and Darrell [517], which computes correspondence using weighted, multi-resolution histogram intersection. Other related earlier work using multi-resolution histograms for texture classification are described in reference [47].

The PHOG descriptor captures several feature variables, including:

- **Shape features**, derived from local distribution of edges based on gradient features inspired by the HOG method [98].
- **Spatial relationships**, across the entire image by computing histogram features over a set of octave grid cells with blocks of increasingly finer size over the image.

- **Appearance features**, using a dense set of SIFT descriptors calculated across a regularly spaced dense grid. PHOG is demonstrated to compute SIFT vectors for color images; results are provided in [183] for the HSV color space.

A set of training images is used to generate a set of PHOG descriptor variables for a class of images, such as cars or people. This training set of PHOG features is reduced using K-means clustering to a set of several hundred visual words to use for feature matching and image classification.

Some key concepts of the PHOG are illustrated in Fig. 6.23. For the feature shape, the edges are computed using the Canny edge detector, and the gradient orientation is computed using the Sobel operator. The gradient orientation binning is linearly interpolated across adjacent histogram bins by gradient orientation (HOG), each bin represents the angle of the edge. A HOG vector is computed for each size of grid cell across the entire image. The final PHOG descriptor is composed of a weighted concatenation of all the individual HOG histograms from each grid level. There is no scale-space smoothing between the octave grid cell regions to reduce fine detail.

As shown in Fig. 6.23, the final PHOG contains all the HOGs concatenated. Note that for the center left image, the full grid size cell produces 1 HOG, for the center right, the half octave grid produces 4 HOGs, and for the right image, the fine grid produces 16 HOG vectors. The final PHOG is normalized to unity to reduce biasing due to concentration of edges or texture.

PHOG SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Global and regional gradient orientation histograms</i>
<i>Feature shape:</i>	<i>Rectangle</i>
<i>Feature pattern:</i>	<i>Dense grid of tiles</i>
<i>Feature density:</i>	<i>Dense tiles</i>
<i>Search method:</i>	<i>Grid regions, no searching</i>
<i>Distance function:</i>	<i>l_2 norm</i>
<i>Robustness:</i>	<i>3 (image classification under some invariance to illumination, viewpoint, noise)</i>

Daisy and O-Daisy

The Daisy Descriptor [206, 308] is inspired by SIFT and GLOH-like descriptors, and is devised for dense-matching applications such as stereo mapping and tracking, reported to be about 40 % faster than SIFT. See Fig. 6.24. Daisy relies on a set of radially distributed and increasing size Gaussian convolution kernels that overlap and resemble a flower-like shape (Daisy).

Daisy does not need local interest points, and instead computes a descriptor densely at each pixel, since the intended application is stereo mapping and tracking. Rather than using gradient magnitude and direction calculations like SIFT and GLOH, Daisy computes a set of convolved orientation maps based on a set of oriented derivatives of Gaussian filters to create eight orientation maps spaced at equal angles.

As shown in Fig. 6.24, the size of each filter region and the amount of blur in each Gaussian filter increase with distance away from the center, mimicking the human visual system by maintaining a sharpness and focus in the center of the field of view and decreasing focus and resolution farther away from the center. Like SIFT, Daisy also uses histogram binning of the local orientation to form the descriptor.

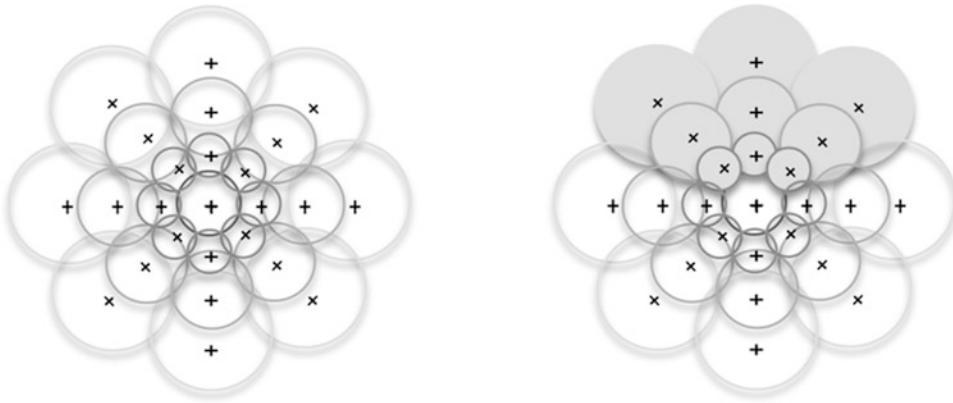


Figure 6.24 (Left) Daisy pattern region, which is composed of four sets of eight overlapping concentric circles, with increasing Gaussian blur in the outer circles, where the radius of each circle is proportional to the Gaussian kernel region standard deviation. The overlapping circular regions provide a degree of filtering against adjacent region transition artifacts. (Right) A hypothetical binary occlusion mask; *darker regions* indicate points that may be occluded and “turned off” in the descriptor during matching

Daisy is designed with optimizations in mind. The convolution orientation map approach consumes fewer compute cycles than the gradient magnitude and direction approach of SIFT and GLOH, yet yields similar results. The Daisy method also includes optimizations for computing larger Gaussian kernels by using a sequential set of smaller kernels, and also by computing certain convolution kernels recursively. Another optimization is gained using a circular grid pattern instead of the rectangular grid used in SIFT, which allows Daisy to vary the rotation by rotating the sampling grid rather than recomputing the convolution maps.

As shown in Fig. 6.24 (right image), Daisy also uses binary occlusion masks to identify portions of the descriptor pattern to use or ignore in the feature matching distance functions. This is a novel feature and provides for invariance to occlusion.

An FPGA optimized version of Daisy, called O-Daisy [209], provides enhancements for increased rotational invariance.

DAISY SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Gaussian convolution values</i>
<i>Feature shape:</i>	<i>Circular</i>
<i>Feature pattern:</i>	<i>Overlapping concentric circular</i>
<i>Feature density:</i>	<i>Dense at each pixel</i>
<i>Search method:</i>	<i>Dense sliding window</i>
<i>Distance function:</i>	<i>Euclidean</i>
<i>Robustness:</i>	<i>3 (illumination, occlusion, noise)</i>

CARD

The Compact and Realtime Descriptor (CARD) method [210] is designed with performance optimizations in mind, using learning-based sparse hashing to convert descriptors into binary codes

supporting fast Hamming distance matching. A novel concept from CARD is the lookup-table descriptor extraction of histograms of oriented gradients from local pixel patches, as well as the lookup-table binning into Cartesian or log polar bins. CARD is reported to achieve significantly better rotation and scale robustness compared to SIFT and SURF, with performance at least ten times better than SIFT and slightly better than SURF.

CARD follows the method of RIFF [211, 214] for feature detection, using FAST features located over octave levels in the image pyramid. The complete CARD pyramid includes intermediate levels between octaves for increased resolution. The pyramid levels are computed at intervals of $1/\sqrt{2}$, with level 0 being the full image. Keypoints are found using a Shi–Tomasi [149] optimized Harris corner detector.

Like SIFT, CARD computes the gradient at each pixel, and can use either Cartesian coordinate binning, or log polar coordinate binning like GLOH; see Fig. 6.17. To avoid the costly bilinear interpolation of gradient information into the histogram bins, CARD instead optimizes this step by rotating the binning pattern before binning, as shown in Fig. 6.25. Note that the binning is further optimized using lookup tables, which contain function values based on principal orientations of the gradients in the patch.

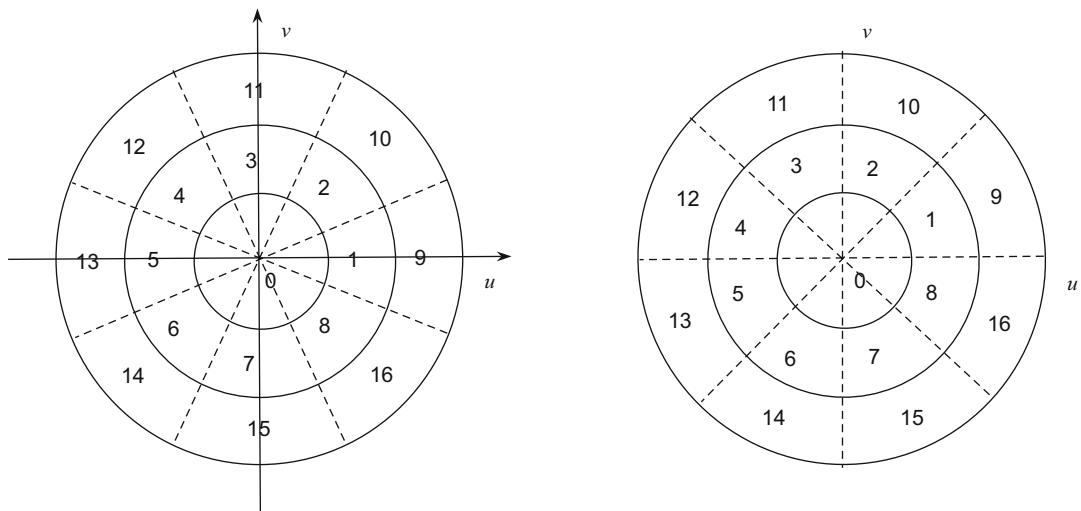


Figure 6.25 CARD patch pattern containing 17 log polar coordinate bins, with image on *left* rotated to optimize binning

As shown in Fig. 6.25, to speed up binning, instead of rotating the patch based on the estimated gradient direction to extract and bin a rotationally invariant descriptor, as done in SIFT and other methods, CARD rotates the binning pattern over the patch based on the gradient direction and then performs binning, which is much faster. Figure 6.25 shows the binning pattern unrotated on the right, and rotated by $\pi/8$ on the left. All binned values are concatenated and normalized to form the descriptor, which is 128 bits long in the most accurate form reported [210].

CARD SUMMARY TAXONOMY

Spectra: Gradient magnitude and direction

Feature shape: Circular, variable sized based on pyramid scale and principal orientation

Feature pattern: Dense

Feature density: Sparse at FAST interest points over image pyramid

Search method: Sliding window

Distance function: Hamming

Robustness: 3 (illumination, scale, rotation)

Robust Fast Feature Matching

Robust Feature Matching in 2.3us developed by Taylor, Rosten and Drummond [212] (RFM2.3) (this acronym is coined here by the author) is a novel, fast method of feature description and matching, optimized for both compute speed and memory footprint. RFM2.3 stands alone among the feature descriptors surveyed here with regard to the combination of methods and optimizations employed, including sparse region histograms and binary feature codes. One of the key ideas developed in RFM2.3 is to compute a descriptor for multiple views of the same patch by creating a set of scaled, rotated, and affine warped views of the original feature, which provides invariance under affine transforms such as rotation and scaling, as well as perspective.

In addition to warping, some noise and blurring is added to the warped patch set to provide robustness to the descriptor. RFM2.3 is one of few methods in the class of deformable descriptors [336–338]. FAST keypoints in a scale space pyramid are used to locate candidate features, and the warped patch set is computed for each keypoint. After the warped patch set has been computed, FAST corners are again generated over each new patch in the set to determine which patches are most distinct and detectable, and the best patches are selected and quantized into binary feature descriptors and saved in the pattern database.

As shown in Fig. 6.26, RFM2.3 uses a sparse 8×8 sampling pattern within a 16×16 region to capture the patch. A sparse set of 13 pixels in the 8×8 sampling pattern is chosen to form the index into the pattern database for the sparse pattern. The index is formed as a 13-bit integer, where each bit

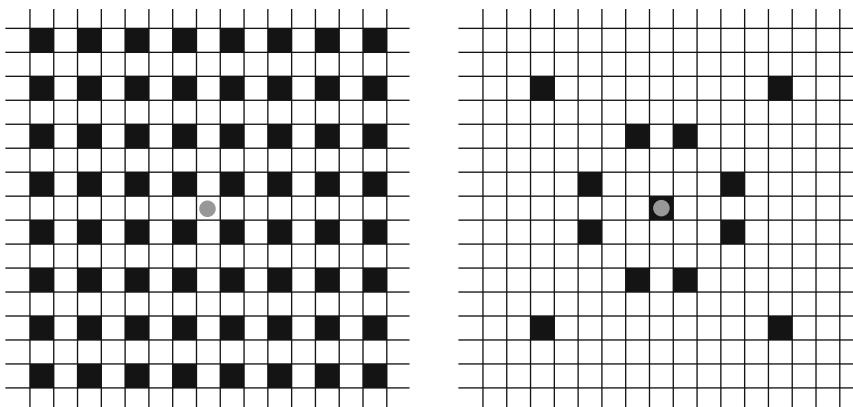


Figure 6.26 RFM2.3 (*Left*) Descriptor sparse sampling pattern. (*Right*) Sparse descriptor using 13 samples used to build the feature index into the database

is set to 1 if the pixel value is greater than the patch mean value, limiting the index to 2^{13} or 8192 entries, so several features in the database may share the same index. However, feature differences can be computed very quickly using Hamming distance, so the index serves mostly as a database key for organizing like-patches. A training phase determines the optimal set of index values to include in the feature database, and the optimal patterns to save, since some patterns are more distinct than others. Initially, features are captured at full resolution, but if few good features are found at full resolution, additional features are extracted at the next level of the image pyramid.

The descriptor is modeled during training as a 64-value normalized intensity distribution function, which is reduced in size to compute the final descriptor vector in two passes: first, the 64 values are reduced to a five-bin histogram of pixel intensity distribution; second, when training is complete, each histogram bin is binary encoded with a 1 bit if the bin is used, and a 0 bit if the bin is rarely used. The resulting descriptor is a compressed, binary encoded bit vector suitable for Hamming distance.

RFM2.3 SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Normalized histogram patch intensity encoded into binary patch index code</i>
<i>Feature shape:</i>	<i>Rectangular, multiple viewpoints</i>
<i>Feature pattern:</i>	<i>Sparse patterns in 15×15 pixel patch</i>
<i>Feature density:</i>	<i>Sparse at FAST9 interest points</i>
<i>Search method:</i>	<i>Sliding window over image pyramid</i>
<i>Distance function:</i>	<i>Hamming</i>
<i>Robustness:</i>	<i>4 (illumination, scale, rotation, viewpoint)</i>

RIFF, CHOG

The Rotation Invariant Fast Features (RIFF) [211, 214] method is motivated by tracking and mapping applications in mobile augmented reality. The basis of the RIFF method includes the development of a radial gradient transform (RGT), which expresses gradient orientation and magnitude in a compute-efficient and rotationally invariant fashion. Another contribution of RIFF is a tracking method, which is reported to be more accurate than KLT with $26\times$ better performance. RIFF is reported to be $15\times$ faster than SURF.

RIFF uses a HOG descriptor computed at FAST interest points located in scale space, and generally follows the method of the author's previous work in CHOG [215] (compressed HOG) for reduced dimensionality, low bitrate binning. Prior to binning the HOG gradients, a radial gradient transform (RGT) is used to create a rotationally invariant gradient format. As shown in Fig. 6.27 (left image), the RGT uses two orthogonal basis vectors (r,t) to form the radial coordinate system that surrounds the patch center point c , and the HOG gradient g is projected onto (r,t) to express as the rotationally invariant vector $(g^T r, g^T t)$. A vector quantizer and a scalar quantizer are both suggested and used for binning, illustrated in Fig. 6.27.

As shown in Fig. 6.27 (right image) the basis vectors can be optimized by using gradient direction approximations in the approximated radial gradient transform (ARGT), which is optimized to be easily computed using a simple differences between adjacent, normalized pixels along the same gradient line, and simple 45° quantization. Also note in Fig. 6.27 (center left image), that the histogramming is optimized by sampling every other pixel within the annuli regions, and four annuli regions are used for practical reasons as a trade-off between discrimination and performance. To meet real-time system performance goals for quantizing the gradient histogram bins, RIFF uses a 5×5 scalar quantizer rather than a vector quantizer.

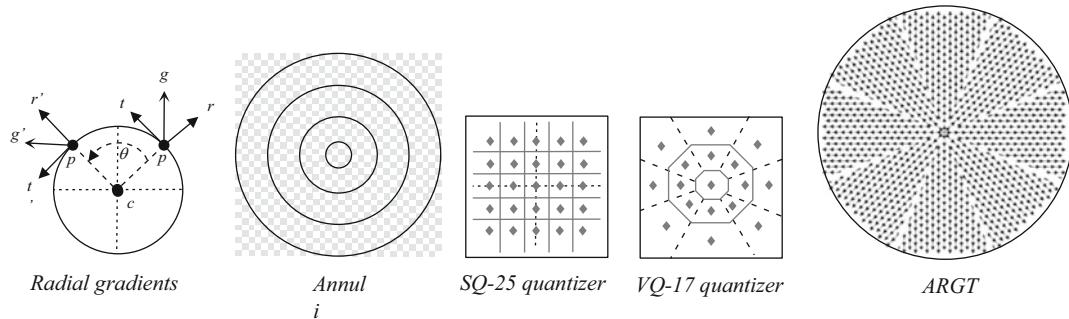


Figure 6.27 Concepts behind the RIFF descriptor [211, 214], based partially on CHOG [215]

In Fig. 6.27 (left image), the gradient projection of \mathbf{g} at point c onto a radial coordinate system (\mathbf{r}, \mathbf{t}) is used for a rotationally invariant gradient expression, and the descriptor patch is centered at c . The center left image (Annuli) illustrates the method of binning, using four annuli rings, which reduces dimensionality, and sampling only the gray pixels provides a $2\times$ speedup. The center and center right images illustrate the bin centering mechanism for histogram quantization: (1) the more flexible scalar quantizer SQ-25 and (2) the faster vector quantizer VQ-17. And the right image illustrates the radial coordinate system basis vectors for gradient orientation radiating from the center outwards, showing the more compute efficient ARGT, or approximated radial gradient transform (RGT), which does not use floating point math (RGT not shown, see [214]).

RIFF SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Local region histogram of approximated radial gradients</i>
<i>Feature shape:</i>	<i>Circular</i>
<i>Feature pattern:</i>	<i>Sparse every other pixel</i>
<i>Feature density:</i>	<i>Sparse at FAST interest points over image pyramid</i>
<i>Search method:</i>	<i>Sliding window</i>
<i>Distance function:</i>	<i>Symmetric KL-divergence</i>
<i>Robustness:</i>	<i>4 (illumination, scale, rotation, viewpoint)</i>

Chain Code Histograms

A Chain Code Histogram (CCH) [198] descriptor records the shape of the perimeter as a histogram by binning the direction of the connected components—connected perimeter pixels in this case. As the perimeter is traversed pixel by pixel, the direction of the traversal is recorded as a number, as shown in Fig. 6.28, and recorded in a histogram feature. To match the CCH features, SSD or SAD distance metrics can be used.

Chain code histograms are covered by U.S. Patent US4783828. CCH was invented in 1961 [198] and is also known as the Freeman chain code. A variant of the CCH is the Vertex chain code [199], which allows for descriptor size reduction and is reported to have better accuracy.

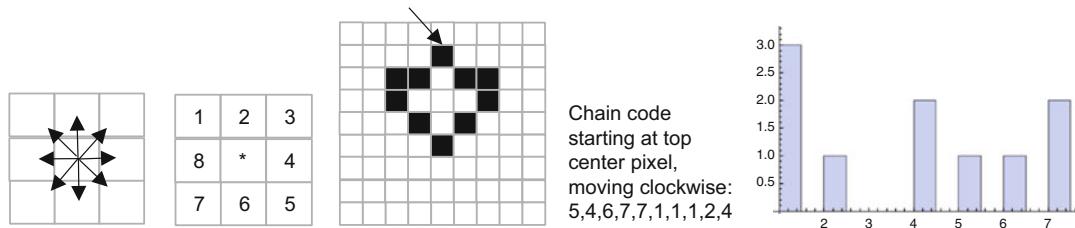


Figure 6.28 Chain code process for making a histogram. (Left to right) 1. The eight possible directions that the connected perimeter may change. 2. Chain code values for each connected perimeter direction change; direction for determining the chain code value is starting from the *center* pixel. 3. An object with a connected perimeter highlighted by black pixels. 4. Chain code for the object following the connected perimeter starting at the *top* pixel. 5. Histogram of all the chain code values

D-NETS

The D-NETS (Descriptor-NETS) [127] approach developed by Hundelshausen and Sukthankar abandons patch or rectangular descriptor regions in favor of a set of strips connected at endpoints. D-NETS allows for a family of strip patterns composed of directed graphs between a set of endpoints; it does not specifically limit the types of endpoints or strip patterns that may be used. The D-NETS paper provides a discussion of results from three types of patterns:

- **Clique D-NETS:** A fully connected network of strips linking all the interest points. While the type of interest point used may vary within the method, the initial work reports results using SIFT keypoints.
- **Iterative D-NETS:** Dynamically creates the network using a subset of the interest points, increasing the connectivity using a stopping criterion to optimize the connection density for obtaining desired matching performance and accuracy.
- **Densely sampled D-NETS:** This variant does not use interest points, and instead densely samples the nets over a regularly spaced grid, a 10-pixel grid being empirically chosen and preferred, with some hysteresis or noise added to the grid positions to reduce pathological sampling artifacts. The dense method is suitable for highly parallel implementations for increased performance.

For an illustration of the three D-NETS patterns and some discussion, see Fig. 4.8.

Each strip is an array of raw pixel values sampled between two points. The descriptor itself is referred to as a *d-token*, and various methods for computing the d-token are suggested, such as binary comparisons among pixel values in the strip similar to FERNS or ORB, as well as comparing the 1D Fourier transforms of strip arrays, or using wavelets. The best results reported are a type of empirically engineered d-token, created as follows:

- **Strip vector sampling**, where each pixel strip vector is sampled at equally spaced locations between 10 and 80 % of the length of the pixel strip vector; this sampling arrangement was determined empirically to ignore pixels near the endpoints.
- **Quantize** the pixel strip vector by integrating the values into a set of uniform chunks, s , to reduce noise.
- **Normalize** the strip vector for scaling and translation.
- **Discretize** the vector values into a limited bit range, b .
- **Concatenate** all uniform chunks into the d-token, which is a bit string of length $s \times b$.

Descriptor matching makes use of an efficient and novel hashing and hypothesis correspondence voting method. D-NETS results are reported to be higher in precision and recall than ORB or SIFT.

D-NETS SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Normalized, averaged linear pixel intensity chunks</i>
<i>Feature shape:</i>	<i>Line segment connected networks</i>
<i>Feature pattern:</i>	<i>Sparse line segments between chosen points</i>
<i>Feature density:</i>	<i>Sparse along lines</i>
<i>Search method:</i>	<i>Sliding window</i>
<i>Distance function:</i>	<i>Hashing and voting</i>
<i>Robustness:</i>	<i>5 (illumination, scale, rotation, viewpoint, occlusion)</i>

Local Gradient Pattern

A variation of the LBP approach, the local gradient pattern (LGP) [196] uses local region gradients instead of local image intensity pair comparison to form the binary descriptor. The 3×3 gradient of each pixel in the local region is computed, then each gradient magnitude is compared to the mean value of all the local region gradients, and the binary bit value of 1 is assigned if the value is greater, and 0 otherwise. The authors claim accuracy and discrimination improvements over the basic LBP in face-recognition algorithms, including a reduction in false positives. However, the compute requirements are greatly increased due to the local region gradient computations.

LGP SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Local region gradient comparisons between center pixel and local region gradients</i>
<i>Feature shape:</i>	<i>Square</i>
<i>Feature pattern:</i>	<i>Every pixel 3×3 kernel region</i>
<i>Feature density:</i>	<i>Dense in 3×3 region</i>
<i>Search method:</i>	<i>Sliding window</i>
<i>Distance function:</i>	<i>Hamming</i>
<i>Robustness:</i>	<i>3 (illumination, scale, rotation)</i>

Local Phase Quantization

The local phase quantization (LPQ) descriptor [158–160] was designed to be robust to image blur, and it leverages the blur insensitive property of Fourier phase information. Since the Fourier transform is required to compute phase, there is some compute overhead; however, integer DFT methods can be used for acceleration. LPQ is reported to provide robustness for uniform blur, as well as uniform illumination changes. LPQ is reported to provide equal or slightly better accuracy on nonblurred images than LBP and Gabor filter bank methods. While mainly used for texture description, LPQ can also be used for local feature description to add blur invariance by combining LPQ with another descriptor method such as SIFT.

To compute, first a DFT is computed at each pixel over small regions of the image, such as 8×8 blocks. The low four frequency components from the phase spectrum are used in the descriptor. The authors note that the kernel size affects the blur invariance, so a larger kernel block may provide more invariance at the price of increased compute overhead.

Before quantization, the coefficients are de-correlated using a whitening transform, resulting in a uniform phase shift and 8-degree rotation, which preserves blur invariance. De-correlating the coefficients helps to create samples that are statistically independent for better quantization.

For each pixel, the resulting vectors are quantized into an 8-dimensional space, using an 8-bit binary encoded bit vector like the LBP and a simple scalar quantizer to yield 1 and 0 values. Binning into the feature vector is performed using 256 hypercubes derived from the 8-dimensional space. The resulting feature vector is a 256-dimensional 8-bit code.

LPQ SUMMARY TAXONOMY

<i>Spectra:</i>	<i>Local region whitened phase using DFT → an 8-bit binary code</i>
<i>Feature shape:</i>	<i>Square</i>
<i>Feature pattern:</i>	<i>8×8 kernel region</i>
<i>Feature density:</i>	<i>Dense every pixel</i>
<i>Search method:</i>	<i>Sliding window</i>
<i>Distance function:</i>	<i>Hamming</i>
<i>Robustness:</i>	<i>3 (contrast, brightness, blur)</i>

Basis Space Descriptors

This section covers the use of basis spaces to describe image features for computer vision applications. A *basis space* is composed of a set of functions, the *basis functions*, which are composed together as a set, such as a series like the Fourier series (discussed in Chap. 3). A complex signal can be decomposed into a chosen basis space as a descriptor.

Basis functions can be designed and used to describe, reconstruct, or synthesize a signal. They require a forward transform to project values into the basis set, and an inverse transform to move data back to the original values. A simple example is transforming numbers between the base 2 number system and the base 10 number system; each basis had advantages.

Sometimes it is useful to transform a dataset from one basis space to another to gain insight into the data, or to process and filter the data. For example, images captured in the time domain as sets of pixels in a Cartesian coordinate system can be transformed into other basis spaces, such as the Fourier basis space in the frequency domain, for processing and statistical analysis. A good basis space for computer vision applications will provide forward and inverse transforms. Again, the Fourier transform meets these criteria, as well as several other basis spaces.

Basis spaces are similar to coordinate systems, since both have invertible transforms to related spaces. In some cases, simply transforming a feature spectra into another coordinate system makes analysis and representation simpler and more efficient. (Chapter 4 discusses coordinates systems used for feature representation.) Several of the descriptors surveyed in this chapter use non-Cartesian coordinate systems, including GLOH, which uses polar coordinate binning, and RIFF, which uses radial coordinate descriptors.

Fourier Descriptors

Fourier descriptors [219] represent feature data as sine and cosine terms, which can be observed in a Fourier Power Spectrum. The Fourier series, Fourier transform, and Fast Fourier transform are used for a wide range of signal analysis, including 1D, 2D, and 3D problems. No discussion of image processing or computer vision is complete without Fourier methods, so we will explore Fourier methods here with applications to feature description.

Instead of developing the mathematics and theory behind the Fourier series and Fourier transform, which has been done very well in the standard text by Bracewell [219], we discuss applications of the Fourier Power Spectrum to feature description and provide minimal treatment of the fundamentals here to frame the discussion; see also Chap. 3. The basic idea behind the Fourier series is to define a series of sine and cosine basis functions in terms of magnitude and phase, which can be summed to approximate any complex periodic signal. Conversely, the Fourier transform is used to decompose a complex periodic signal into the Fourier series set of sine and cosine basis terms. The Fourier series components of a signal, such as a line or 2D image area, are used as a Fourier descriptor of the region.

For this discussion, a *Fourier descriptor* is the selected components from a Fourier Power Spectrum—typically, we select the lower-frequency components, which carry most of the power. Here are a few examples using Fourier descriptors; note that either or both the Fourier magnitude and phase may be used.

- **Fourier Spectrum of LBP Histograms.** As shown in Fig. 3.10, an LBP histogram set can be represented as a Fourier Spectrum magnitude, which makes the histogram descriptor invariant to rotation.
- **Fourier Descriptor of Shape Perimeter.** As shown in Fig. 6.29, the shape of a polygon object can be described by Fourier methods using an array of perimeter to centroid line segments taken at intervals, such as 10° . The array is fed into an FFT to produce a shape descriptor, which is scale and rotation invariant.
- **Fourier Descriptor of Gradient Histograms.** Many descriptors use gradients to represent features, and use gradient magnitude or direction histograms to bin the results. Fourier Spectrum magnitudes may be used to create a descriptor from gradient information to add invariance.
- **Fourier Spectrum of Radial Line Samples.** As used in the RFAN descriptor [128], radial line samples of pixel values from local regions can be represented as a Fourier descriptor of Fourier magnitudes.

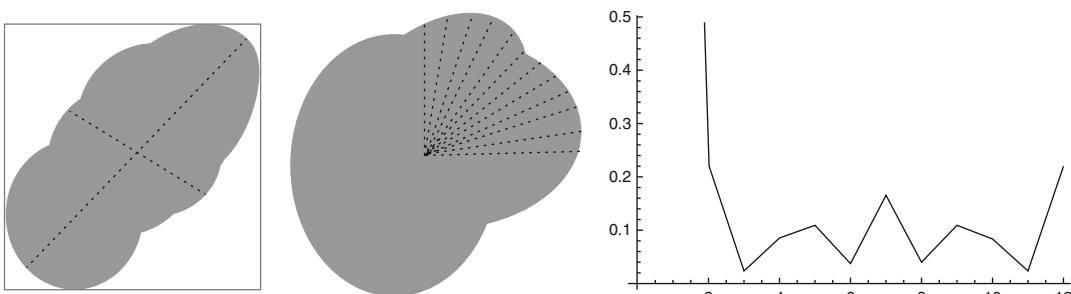


Figure 6.29 (Left) Polygon shape major and minor axis and bounding box. (Center) Object with radial sample length taken from the centroid to the perimeter, each sample length saved in an array, normalized. (Right) Image fed into the Fourier Spectrum to yield a Fourier descriptor

- **Fourier Spectrum Phase.** The LPQ descriptor, described in this chapter, makes use of the Fourier Spectrum phase information in the descriptor, and the LPQ is reported to be insensitive to blur owing to the phase information.

Other Basis Functions for Descriptor Building

Besides the Fourier basis series, other function series and basis sets are used for descriptor building, pattern recognition, and image coding. However, such methods are usually applied over a global or regional area. See Chap. 3 for details on several other methods.

Sparse Coding Methods

Any of the local feature descriptor methods discussed in this chapter may be used as the basis for a sparse codebook, which is a collection of descriptors boiled down to a representative set. Sparse coding and related methods are discussed in more detail in Chap. 10. Interesting examples are found in the work by Aharon, Alad, and Bruckstein [518] as well as Fei-Fei, Fergus, and Torralba [519]. See Fig. 6.30.

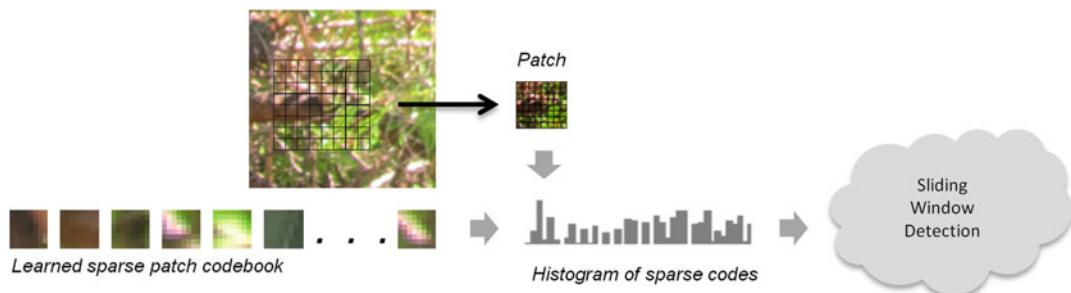


Figure 6.30 One method of feature learning using sparse coding, showing how Histograms of Sparse Codes (HSC) are constructed from a set of learned sparse codes. The HSC method [117] is reported to outperform HOG in many cases

Polygon Shape Descriptors

Polygon shape descriptors compute a set of *shape features* for an arbitrary polygon or blob, and the shape is described using statistical moments or image moments (as discussed in Chap. 3). These shape features are based on the perimeter of the polygon shape. The methods used to delineate image perimeters to highlight shapes prior to measurement and description are often complex, empirically tuned pipelines of image preprocessing operations, like thresholding, segmentation, and morphology (as discussed in Chap. 2). Once the polygon shapes are delineated, the shape descriptors are computed; see Fig. 6.31. Typically, polygon shape methods are applicable to larger region-size features. In the literature, this topic may also be discussed as *image moments*. For a deep dive into the topic of image moments, see Flusser et al. [500].

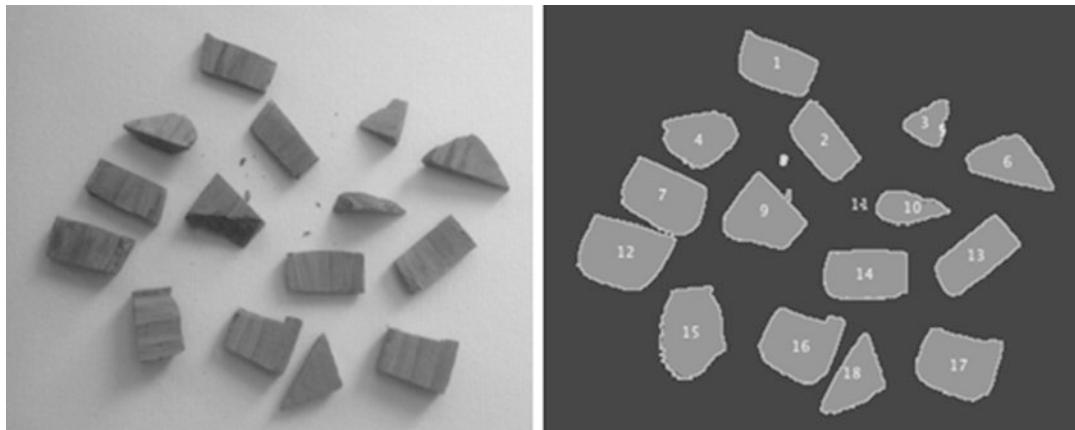


Figure 6.31 Polygon shape descriptors. (*Left*) Malachite pieces. (*Right*) Polygon shapes defined and labeled after binary thresholding, perimeter tracing, and feature labeling. (Image processing and particle analysis performed using ImageJ Fiji)

Polygon shape methods are commonly used in medical and industrial applications, such as automated microscopy for cell biology, and also for industrial inspection; see Fig. 6.31. Commercial software libraries are available for polygon shape description, commonly referred to as *particle analysis* or *blob analysis*. See Appendix C.

MSER Method

The Maximally Stable Extremal Regions (MSER) method [186] is usually discussed in the literature as an interest region detector, and in fact it is. However, we include MSER in the shape descriptor section because MSER regions can be much larger than other interest point methods, such as HARRIS or FAST.

The MSER detector was developed for solving disparity correspondence in a wide baseline stereo system. Stereo systems create a warped and complex geometric depth field, and depending on the baseline between cameras and the distance of the subject to the camera, various geometric effects must be compensated for. In a wide baseline stereo system, features nearer the camera are more distorted under affine transforms, making it harder to find exact matches between the left/right image pair. The MSER approach attempts to overcome this problem by matching on blob-like features. MSER regions are similar to morphological blobs and are fairly robust to skewing and lighting. MSER is essentially an efficient variant of the watershed algorithm, except that the goal of MSER is to find a range of thresholds that leave the watershed basin unchanged in size.

The MSER method involves sorting pixels into a set of regions based on binary intensity thresholding; regions with similar pixel value over a range of threshold values in a connected component pattern are considered maximally stable. To compute a MSER, pixels are sorted in a binary intensity thresholding loop, which sweeps the intensity value from min to max. First, the binary threshold is set to a low value such as zero on a single image channel—luminance, for example. Pixels $<$ the threshold value are black, pixels \geq are white. At each threshold level, a list of connected components or pixels is kept. The intensity threshold value is incremented from 0 to the max pixel value. Regions that do not grow or shrink or change as the intensity varies are

considered maximally stable, and the MSER descriptor records the position of the maximal regions and the corresponding thresholds.

In stereo applications, smaller MSER regions are preferred and correlation is used for the final correspondence, and similarity is measured inside a set of circular MSER regions at chosen rotation intervals. Some interesting advantages of the MSER include:

- Multi-scale features and multi-scale detection. Since the MSER features do not require any image smoothing or scale space, both coarse features and fine-edge features can be detected.
- Variable-size features computed globally across an entire region, not limited to patch size or search window size.
- Affine transform invariance, which is a specific goal.
- General invariance to shape change, and stability of detection, since the extremal regions tend to be detected across a wide range of image transformations.

The MSER can also be considered as the basis for a shape descriptor, and as an alternative to morphological methods of segmentation. Each MSER region can be analyzed and described using shape metrics, as discussed later in this chapter.

Object Shape Metrics for Blobs and Polygons

Object shape metrics are powerful and yield many degrees of freedom with respect to invariance and robustness. Object shape metrics are not like local feature metrics, since object shape metrics can describe much larger features. This is advantageous for tracking from frame to frame. For example, a large object described by just a few simple object shape metrics such as area, perimeter, and centroid can be tracked from frame to frame under a wide range of conditions and invariance. For more information, see references [120, 121] for a survey of 2D shape description methods.

Shape can be described by several methods, including:

- **Object shape moments and metrics:** the focus of this section.
- **Image moments:** see Chap. 3 under “Image Moments.”
- **Fourier descriptors:** discussed in this chapter and Chap. 3.
- **Shape Context feature descriptor:** discussed in this section.
- **Chain code descriptor for perimeter description:** discussed in this section.

Object shape is closely related to the field of morphology, and computer methods for morphological processing are discussed in detail in Chap. 2. Also see the discussion about morphological interest points earlier in this chapter.

In many areas of computer vision research, local features seem to be favored over object shape-based features. The lack of popularity of shape analysis methods may be a reaction to the effort involved in creating preprocessing pipelines of filtering, morphology, and segmentation to prepare the image for shape analysis. If the image is not preprocessed and prepared correctly, shape analysis is not possible. (See Chap. 8 for a discussion of a hypothetical shape analysis preprocessing pipeline.)

Polygon shape metrics can be used for virtually any scene analysis application to find common objects and take accurate measurements of their size and shape; typical applications include biology and manufacturing. In general, most of the polygon shape metrics are rotational and scale invariant. Table 6.7 provides a sampling of some of the common metrics that can be derived from region shapes, both binary shapes and gray scale shapes.

Table 6.7 Various common object shape and blob object metrics

Object Binary Shape Metrics	Description
Perimeter	Length of all points around the edge of the object, including the sum of diagonal lengths ≈ 1.4 and adjacent lengths = 1
Area	Total area of object in pixels
Convex hull	Polygon shape or set of line segments enclosing all perimeter points
Centroid	Center of object mass, average value of all pixel coordinates or average value of all perimeter coordinates
Fourier descriptor	Fourier spectrum result from an array containing the length of a set of radial line segments passing from centroid to perimeter at regular angles used to model a 1D signal function, the 1D signal function is fed into a 1D FFT and the set of FFT magnitude data is used as a metric for a chosen set of octave frequencies
Major/minor axis	Longest and shortest line segments passing through centroid contained within and touching the perimeter
Feret	Largest caliper diameter of object
Breadth	Shortest caliper diameter
Aspect ratio	Feret/Breadth
Circularity	$4 \times \pi \times \text{Area}/\text{Perimeter}^2$
Roundness	$4 \times \text{Area}/(\pi \times \text{Feret}^2)$ (Can also be calculated from the Fourier descriptors)
Area equivalent diameter	$\sqrt{(4/\pi) \times \text{Area}}$
Perimeter equivalent diameter	Area/π
Equivalent ellipse	$(\pi \times \text{Feret} \times \text{Breadth})/4$
Compactness	$\sqrt{(4/\pi) \times \text{Area}}/\text{Feret}$
Solidity	$\text{Area}/\text{Convex_Area}$
Concavity	$\text{Convex_Area}-\text{Area}$
Convexity	$\text{Convex_Hull}/\text{Perimeter}$
Shape	$\text{Perimeter}^2/\text{Area}$
Modification ratio	$(2 \times \text{MinR})/\text{Feret}$
Shape matrix	A 2D matrix representation or plot of a polygon shape (may use Cartesian or polar coordinates; see Figure 6-32)
Grayscale Object Shape Metrics	
SDM plots	*See Chapter 3, "Texture Metrics" section.
Scatter plots	*See Chapter 3, "Texture Metrics" section.
Statistical moments of grayscale pixel values	Minimum Maximum Median Average Average deviation Standard deviation Variance Skewness Kurtosis Entropy

*Note: some of binary object metrics also apply to grayscale objects.

Shape is considered to be binary; however, shape can be computed around intensity channel objects as well, using gray scale morphology. Perimeter is considered as a set of connected components. The shape is defined by a single pixel wide perimeter at a binary threshold or within an intensity band, and pixels are either on, inside, or outside of the perimeter. The perimeter edge may be computed by scanning the image, pixel by pixel, and examining the adjacent touching pixel neighbors for connectivity. Or, the perimeter may be computed from the shape matrix [327] or chain code discussed earlier in this chapter. Perimeter length is computed for each segment (pixel), where segment length = 1 for horizontal and vertical neighbors, and $\sqrt{2}$ otherwise for diagonal neighbors.

The perimeter may be used as a mask, and gray scale or color channel statistical metrics may be computed within the region. The object area is the count of all the pixels inside the perimeter. The centroid may be computed either from the average of all (x,y) coordinates of all points contained within the perimeter area, or from the average of all perimeter (x,y) coordinates.

Shape metrics are powerful. For example, shape metrics may be used to remove or exclude objects from a scene prior to measurement. For example, objects can be removed from the scene when the area is smaller than a given size, or if the centroid coordinates are outside a given range.

As shown in Fig. 6.29 and Table 2, the Fourier descriptor provides a rotation and scale invariant shape metric, with some occlusion invariance also. The method for determining the Fourier descriptor is to take a set of equally angular-spaced radius measurements, such as every 10° , from the centroid out to points on the perimeter, and then to assemble the radius measurements into a 1D array that is run through a 1D FFT to yield the Fourier moments of the object. Or radial pixel spokes can be used as a descriptor.

Other examples of useful shape metrics, shown in Fig. 6.29, include the bounding box with major and minor axis, which has longest and shortest diameter segments passing through the centroid to the perimeter; this can be used to determine rotational orientation of an object.

The SNAKES method [522] uses a spline model to fit a collection of interest points, such as selected perimeter points, into a region contour. The interest points are the spline points. The SNAKE can be used to track contoured features from frame to frame, deforming around the interest point locations.

In general, the 2D object shape methods can be extended to 3D data; however, we do not explore 3D object shape metrics here, see reference [192, 193] for a survey of 3D shape descriptors.

Shape Context

The shape context method developed by Belongie, Malik, and Puzicha [231–233], describes local feature shape using a reference point on the perimeter as the Cartesian axis origin, and binning selected perimeter point coordinates relative to the reference point origin. The relative coordinates of each point are binned into a log polar histogram. Shape context is related to the earlier shape matrix descriptor [327] developed in 1985 as shown in Fig. 6.32, which describes the perimeter of an object using log polar coordinates also. The shape context method provides for variations, described in several papers by the authors [231–233]. Here, we look at a few key concepts.

To begin, the perimeter edge of the object is sparsely sampled at uniform intervals, typically keeping about 100 edge sample points for coarse binning. Sparse perimeter edge points are typically distinct from interest points, and found using perimeter tracing. Next, a reference point is chosen on the perimeter of the object as the origin of a Cartesian space, and the vector angle and magnitude (r, θ) from the origin point to each other perimeter point are computed. The magnitude or distance is normalized to fit the histogram. Each sparse perimeter edge point is used to compute a tangent with

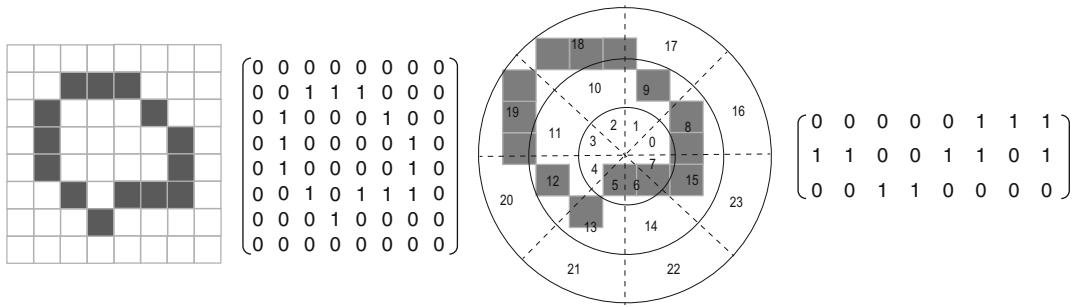


Figure 6.32 A shape matrix descriptor [327] for the perimeter of an object. (Left two images) Cartesian coordinate shape matrix. (Right two images) polar coordinate shape matrix using three rows of eight numbered bin regions, gray boxes represent pixels to be binned. Note that multiple shape matrices can be used together. Values in matrix are set if the pixel fills at least half of the bin region, no interpolation is used

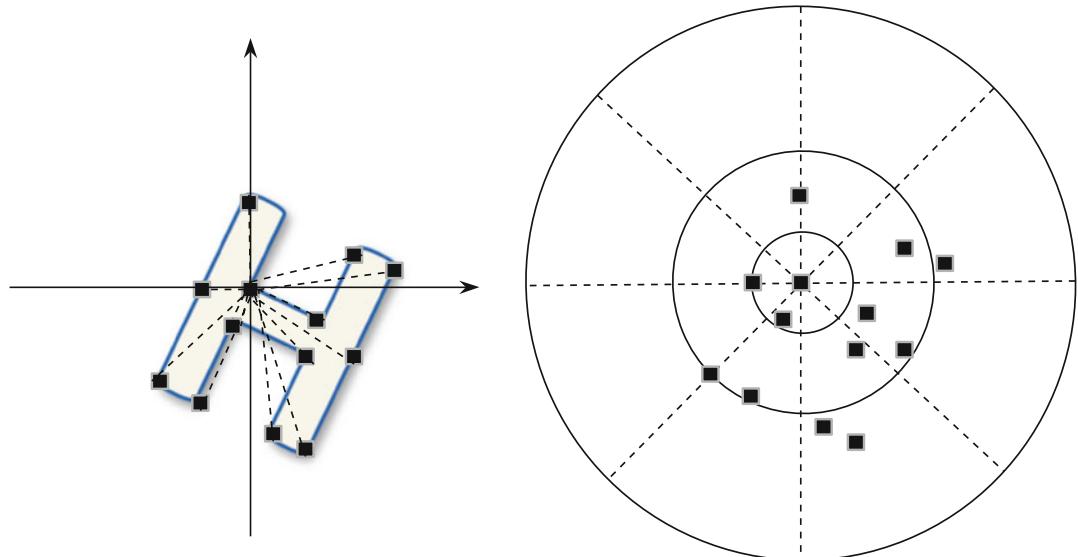


Figure 6.33 Shape context method. (Left) Perimeter points are measured as a shape vector, both angle and distance, with respect to a chosen perimeter point as the reference Cartesian origin. (Right) Shape vectors are binned into a log polar histogram feature descriptor

the origin. Finally, each normalized vector is binned using (r, θ) into a log polar histogram, which is called the *shape context*.

An alignment transform is generated between descriptor pairs during matching, which yields the difference between targets and chosen patterns, and could be used for reconstruction. The alignment transform can be chosen as desired from affine, Euclidean, spline-based, and other methods. Correspondence uses the Hungarian method, which includes histogram similarity, and is weighted by the alignment transform strength using the tangent angle dissimilarity. Matching may also employ a local appearance similarity measure, such as normalized correlation between patches or color histograms.

The shape context method provides a measure of invariance over scale, translation, rotation, occlusion, and noise. See Fig. 6.33.

3D, 4D, Volumetric and Multimodal Descriptors

With the advent of more and more 3D sensors, such as stereo cameras and other depth-sensing methods, as well as the ubiquitous accelerometers and other sensors built into inexpensive mobile devices, the realm of 3D feature description and multimodal feature description is beginning to blossom.

Many 3D descriptors are associated with robotics research and 3D localization. Since the field of 3D feature description is early in the development cycle, it is not yet clear which methods will be widely adopted, so we present only a small sampling of 3D descriptor methods here. These include 3D HOG [188], 3D SIFT [187], and HON 4D [190], which are based on familiar 2D methods. We refer the interested reader to references [192, 193, 208] for a survey of 3D shape descriptors. Several interesting 3D descriptor metrics are available as open source in the Point Cloud Library,² including Radius-Based Surface Descriptors (RSD) [521], Principal Curvature Descriptors (PCD), Signatures of Histogram Orientations (SHOT) [523], Viewpoint Feature Histogram (VFH) [381], and Spin Images [520].

Some noteworthy 3D descriptors we do not survey include 3D Shapenets by Wu [863], 3D voxel patterns [864], triangular surface patches [865], 3D surface patch features [865], see also [866–870]. Applications driving the research into 3D descriptors include robotics and activity recognition, where features are tracked frame to frame as they morph and deform. The goals are to localize position and recognize human actions, such as walking, waving a hand, turning around, or jumping. See also the LBP variants for 3D: V-LBP and LBP-TOP, which are surveyed earlier in this chapter as illustrated in Fig. 6.12, which are also used for activity recognition. Since the 2D features are moving during activity recognition, time is the third dimension incorporated into the descriptors. We survey some notable 3D activity-recognition research here.

One of the key concepts in the action-recognition work is to extend familiar 2D features into a 3D space that is spatiotemporal, where the 3D space is composed of 2D x,y video image sequences over time t into a volumetric representation with the form $v(x,y,t)$. In addition, the 3D surface normal, 3D gradient magnitude, and 3D gradient direction are used in many of the action-recognition descriptor methods.

Development of 3D descriptors is continuing, which is beyond the scope of this brief introduction. However, for the interested reader, we mention recent work in the areas of volumetric shape descriptors, depth image surface shape descriptors, and 3D reconstruction using depth-based landmark detectors, which can be found in the references [863–870].

3D HOG

The 3D HOG [188] is partially based on some earlier work in volumetric features [191]. The general idea is to employ the familiar HOG descriptor [98] in a 3D HOG descriptor formulation, using a stack of sequential 2D video frames or slices as a 3D volume, and to compute spatiotemporal gradient orientation on adjacent frames within the volume. For efficiency, a novel integral video approach is developed as an alternative to image pyramids based on the same line of thinking as the integral image approach used in the Viola–Jones method.

A similar approach using the integral video concept was also developed in [191] using a subsampled space of 64×64 over 4–40 video frames in the volume, using pixel intensity instead

² <http://pointclouds.org>.

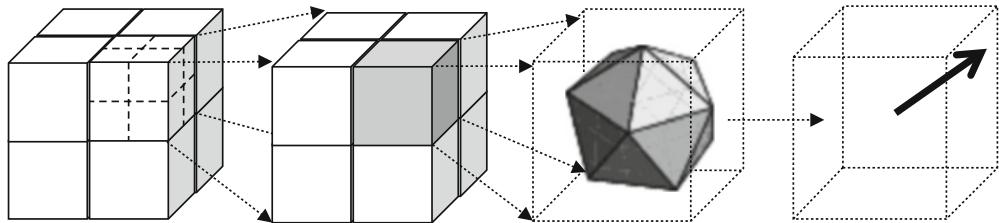


Figure 6.34 HOG 3D descriptor computation. (*Left*) $2 \times 2 \times 2$ descriptor cell block. (*Left center*) Gradient orientation histogram computed over $2 \times 2 \times 2$ cell sub-blocks. (*Right center*) Gradient orientations quantized by projecting the vector intersection to the faces of a 20-faceted icosahedron. (*Right*) Mean gradient orientation computed over integral video blocks (volume vector integral)

of the gradient direction. The integral video method, which can also be considered an *integral volume* method, allows for arbitrary cuboid regions from stacked sequential video frames to be integrated together to compute the local gradient orientation over arbitrary scales. This is space efficient and time efficient compared to using precomputed image pyramids. In fact, this integral video integration method is a novel contribution of the work, and may be applied to other spectra such as intensity, color, and gradient magnitude in either 2D or 3D to eliminate the need for image pyramids—providing more choices in terms of image scale besides just octaves.

The 3D HOG descriptor computations are illustrated in Fig. 6.34. To find feature keypoints to anchor the descriptors, a space-time extension of the Harris operator [189] is used, then a histogram descriptor is computed from the mean of the oriented gradients in a cubic region at the keypoint. Since gradient magnitude is sensitive to illumination changes, gradient orientation is used instead to provide invariance to illumination, and it is computed over 3D cuboid regions using simple x , y , z derivatives. The mean gradient orientation of any 3D cuboid is computed quickly using the integral video method. Gradient orientations are quantized into histogram bins via projection of each vector onto the faces of a regular icosahedron 20-sided shape to combine all vectors, as shown in Fig. 6.34. The 20 icosahedron faces act as the histogram bins. The sparse set of spatiotemporal features is combined into a bag of features or bag of words in a visual vocabulary.

HON 4D

A similar approach to the 3D HOG is called HON 4D [190], which computes descriptors as Histogram of Oriented 4D Normals, where the 3D surface normal + time add up to four dimensions (4D). HON 4D uses sequences of depth images or 3D depth maps as the basis for computing the descriptor, rather than 2D image frames, as in the 3D HOG method. So a depth camera is needed. In this respect, HON 4D is similar to some volume rendering methods which compute 3D surface normals, and may be accelerated using similar methods [434–436].

In the HON 4D method, the surface normals capture the surface shape cues of each object, and changes in normal orientation over time can be used to determine motion and pose. Only the orientation of the surface normal is significant in this method, so the normal lengths are all normalized to unity length. As a result, the binning into histograms acts differently from the HOG style binning, so that the fourth dimension of time encodes differences in the gradient from frame to frame. The HON 4D descriptor is binned and quantized using 4D projector functions, which quantize local surface normal orientation into a 600-cell polychoron, which is a geometric extension of a 2D polygon into 4-space.

Consider the discrimination of the HON 4D method using gradient orientation vs. the HOG method using gradient magnitude. If two surfaces are the same or similar with respect to gradient magnitude, the HOG style descriptor cannot differentiate; however, the HON 4D style descriptor can differentiate owing to the orientation of the surface normal used in the descriptor. Of course, computing 3D normals is compute-intensive without special optimizations considering the noncontiguous memory access patterns required to access each component of the volume.

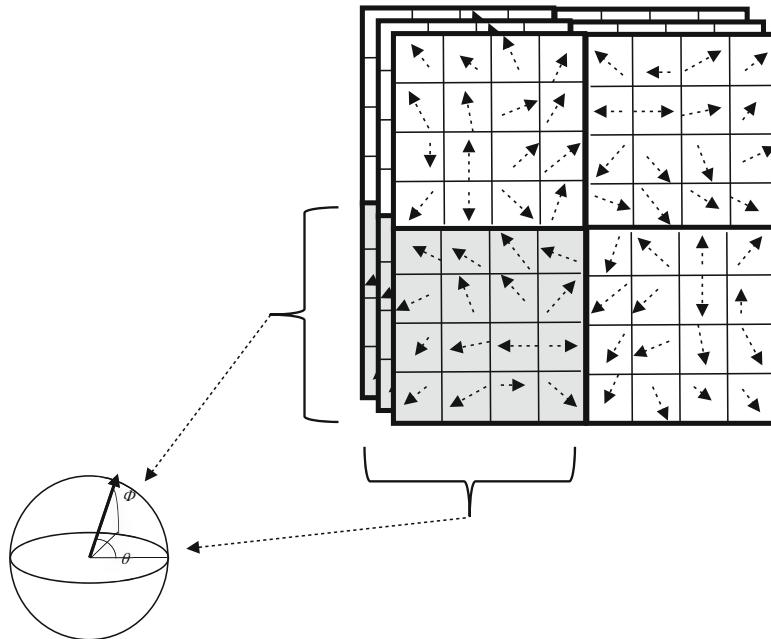


Figure 6.35 Computation of the 3D SIFT [187] vector histogram bins as a combination of the combined gradient orientation of the sub-volumes in a volume space or 3D spatiotemporal region of three consecutive 2D image frames

3D SIFT

The 3D SIFT method [187] starts with the 2D SIFT feature method and reformulates the feature binning to use a volumetric spatiotemporal area $v(x,y,t)$, as shown in Fig. 6.35.

The 3D orientation of the gradient pair orientation is computed as follows:

$$m3D(x, y, t) = \sqrt{L_x^2 + L_y^2 + L_t^2}$$

$$\theta(x, y, t) = \tan^{-1} \left(\frac{L_y}{L_x} \right)$$

$$\phi(x, y, t) = \tan^{-1} \left(\frac{L_{yt}}{\sqrt{L_x^2 + L_y^2}} \right)$$

This method provides a unique two-valued (ϕ, θ) representation for each angle of the gradient orientation in 3-space at each keypoint. The binning stage is handled differently from SIFT, and instead uses orthogonal bins defined by meridians and parallels in a spherical coordinate space. This is simpler to compute, but requires normalization of each value to account for the spherical difference in the apparent size ranging from the poles to the equator.

To compute the SIFT descriptor, the 3D gradient orientation of each sub-histogram is used to guide rotation of the 3D region at the descriptor keypoint to point to 0, which provides a measure of rotational invariance to the descriptor. Each point will be represented as a single gradient magnitude and two orientation vectors (ϕ, θ) instead of one, as in 2D SIFT. The descriptor binning is computed over three dimensions into adjacent cubes instead of over two dimensions in the 2D SIFT descriptor.

Once the feature vectors are binned, the feature vector set is clustered into groups of like features, or words, using hierarchical K-means clustering into a spatiotemporal word vocabulary. Another step beyond the clustering could be to reduce the feature set using sparse coding methods [107–109], but the sparse coding step is not attempted.

Results using 3D SIFT for action recognition are reported to be quite good compared to other similar methods; see reference [187].

Summary

In this chapter we survey a wide range of local interest point detectors and feature descriptor methods to learn “what” practitioners are doing, including both 2D and 3D methods. The vision taxonomy from Chap. 5 is used to divide the feature descriptor survey along the lines of descriptor families, such as local binary methods, spectra methods, and polygon shape methods. There is some overlap between local and regional descriptors; however, this chapter tries to focus on local descriptor methods, leaving regional methods to Chap. 3. Local interest point detectors are discussed in a simple taxonomy including intensity-based regions methods, edge-based region methods, and shape-based region methods, including background on key concepts and mathematics used by many interest point detector methods. Some of the difficulties in choosing an appropriate interest point detector are discussed and several detector methods are surveyed.

This chapter also highlights retrofits to common descriptor methods. For example, many descriptors are retrofitted by changing the descriptor spectra used, such as LBP vs. gradient methods, or by swapping out the interest point detector for a different method. Summary information is provided for feature descriptors following the taxonomy attributes developed in Chap. 5 to enable limited comparisons, using concepts from the analysis of local feature description design concepts presented in Chap. 4.

Chapter 6: Learning Assignments

1. Interest points, or Keypoints, are located in images at locations such as maxima and minima. Describe the types of maxima and minima features found in images.
2. Interest point detectors must be selected and parametrically tuned to give best results. Describe various approaches to select and tune interest point detectors for a range of different types of images.
3. Describe your favorite interest point detector, discuss the advantages compared to other detectors, and describe the basic algorithm.
4. Describe and summarize the names of as many interest point detectors as you can remember, and describe the basic concepts and goals of each algorithm.
5. An interest point adapter function can be devised to help tune interest point parameters to automatically find better interest points. Select an interest point detector of your choice, describe how the interest point detector algorithm works using pseudo code, describe each parameter to the interest point function, describe the image search pattern the adapter could use, and describe parameters to control region size and iterations. (See also assignment 6 below).
6. Write an interest point adapter function using your favorite interest point detector in your favorite programming language, and provide test results.
7. Describe how the local binary pattern (LBP) algorithm works using pseudo code.
8. List a few applications for the local binary pattern.
9. Describe how the local binary pattern can be stored in a rotationally invariant format.
10. Compare local binary pattern algorithms including Brief, Brisk, Orb, and Freak, and highlight the differences in the pixel region sampling patterns.
11. List the distance function most applicable to local binary descriptors, and how it can be optimized.
12. Describe the basic SIFT algorithm, highlighting the scales over which the pixel regions are sampled, the algorithm for detecting interest points, the algorithm for computing the feature descriptor, and the summary information stored in the descriptor.
13. Describe at least one enhancement to the basic SIFT algorithm, such as SIFT-PCA and SIFT-GLOH, SIFT-SIFER, or RootSIFT, and highlight the major improvements provided by the enhancement.
14. Describe the pixel patch region shape and sizes used in the SIFT algorithm, and describe how the pixel samples are weighted within the region.
15. Discuss the SURF feature descriptor algorithm.
16. Compare the local binary feature descriptors ORB, FREAK, and BRISK.
17. Describe how HAAR-like features are used in feature description, draw or describe a few example HAAR-like features, and discuss how HAAR features are related to Wavelets.
18. Describe integral images, how they are built, and discuss why integral images can be used to optimize working with HAAR filters.
19. Describe the Viola-Jones feature classification funnel and pipeline.
20. Design an algorithm to compute *gradient histograms* from a local region, describe how to create a useful feature descriptor from the *gradient histograms*, and select a specific distance function that could be applied to measure correspondence between gradient histograms, and discuss the strengths and weaknesses of your algorithm.
21. Describe the algorithm for your favorite feature descriptor, discuss the advantages, and provide simple comparisons to a few other feature descriptors.

22. Describe how a chain code histogram is computed.
23. Describe how a polygon feature shape can be refined (for example using morphology operations, thresholding operations), and then describe the types of feature metrics that can be computed over polygon shapes.
24. List at least five polygon shape feature metrics and describe how they are computed, including perimeter and centroid.

Buy the truth and do not sell it.

—Proverbs 23:23

This chapter discusses several topics pertaining to *ground truth data*, the basis for computer vision metric analysis. We look at examples to illustrate the importance of ground truth data design and use, including manual and automated methods. We then illustrate ground truth data by developing a method and corresponding ground truth dataset for measuring interest point detector response as compared to human visual system response and human expectations. Also included here are example applications of the general robustness criteria and the general vision taxonomy developed in Chap. 5 as applied to the preparation of hypothetical ground truth data. Lastly, we look at the current state of the art, its best practices, and a survey of available ground truth datasets.

Key topics include:

- Creating and collecting ground truth data: manual vs. synthetic methods
- Labeling and describing ground truth data: automated vs. human annotated
- Selected ground truth datasets
- Metrics paired with ground truth data
- Over-fitting, under-fitting, and measuring quality
- Publicly available datasets
- An example scenario that compares the human visual system to machine vision detectors, using a synthetic ground truth dataset

Ground truth data may not be a cutting-edge research area, however it is as important as the algorithms for machine vision. Let us explore some of the best-known methods and consider some open questions.

What Is Ground Truth Data?

In the context of computer vision, ground truth data includes a set of images, and a set of labels on the images, and defining a model for object recognition as discussed in Chap. 4, including the count, location, and relationships of key features. The labels are added either by a human or automatically by image analysis, depending on the complexity of the problem. The collection of labels, such as interest points, corners, feature descriptors, shapes, and histograms, form a model.

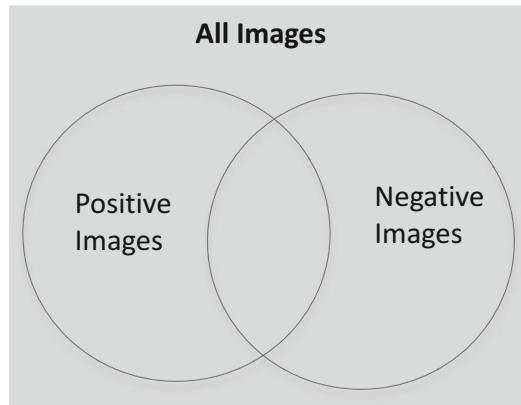


Figure 7.1 Set of all ground truth data, composed of both positive and negative training examples

For deep learning systems (surveyed in Chap. 10), the training protocols often involve expanding the training set by including geometric transformations and contrast enhancements to each original image to create more variations to the training set.

A model may be trained using a variety of machine learning methods. At run-time, the detected features are fed into a classifier to measure the correspondence between detected features and modeled features. Modeling, classification, and training are covered elsewhere in this book. Instead, we are concerned here with the content and design of the ground truth images.

Creating a ground truth dataset, then, may include consideration of the following major tasks:

- **Model design.** The model defines the composition of the objects—for example, the count, strength, and location relationship of a set of SIFT features. The model should be correctly fitted to the problem and image data so as to yield meaningful results.
- **Training set.** This set is collected and labeled to work with the model, and it contains both positive and negative images and features. Negatives contain images and features intended to generate false matches; see Fig. 7.1.
- **Test set.** A set of images is collected for testing against the training set to verify the accuracy of the model to predict the correct matches.
- **Classifier design.** This is constructed to meet the application goals for speed and accuracy, including data organization and searching optimizations for the model.
- **Training and testing.** This work is done using several sets of images to check against ground truth.

Unless the ground truth data contains carefully selected and prepared image content, the algorithms cannot be measured effectively. Thus, *ground-truthing* is closely related to root-causing: there is no way to improve what we cannot measure and do not understand. Being able to root-cause algorithm problems and understand performance and accuracy are primary purposes for establishing ground truth data. Better ground truth data will enable better analysis.

Ground truth data varies by task. For example, in 3D image reconstruction or face recognition, different attributes of the ground truth data must be recognized for each task. Some tasks, such as face recognition, require segmentation and labeling to define the known objects, such as face locations, position and orientation of faces, size of faces, and attributes of the face, such as emotion, gender, and age. Other tasks, such as 3D reconstruction, need the raw pixels in the images and a reference 3D mesh or point cloud as their ground truth.

Ground truth datasets fall into several categories:

- **Synthetic produced:** images are generated from computer models or renderings.
- **Real produced:** a video or image sequence is designed and produced.
- **Real selected:** real images are selected from existing sources.
- **Machine-automated annotation:** feature analysis and learning method are used to extract features from the data.
- **Human annotated:** an expert defines the location of features and objects.
- **Combined:** any mixture of the above.

Many practitioners are firmly against using synthetic datasets and insist on using real datasets. However, deep-learning systems often permute the original data by rotating, scaling, adding noise, and changing contrast to hopefully add more robustness to the training process. See Chap. 10 for details on the training protocols for various DNNs and CNNs. In some cases, random ground truth images are required; in other cases, carefully scripted and designed ground truth images need to be produced, similar to creating a movie with scenes and actors.

Random and natural ground truth data with unpredictable artifacts, such as poor lighting, motion blur, and geometric transformation, is often preferred. Many computer problems demand real images for ground truth, and random variations in the images are important. Real images are often easy to obtain and/or easy to generate using a video camera or even a cell phone camera. But creating synthetic datasets is not as clear; it requires knowledge of appropriate computer graphics rendering systems and tools, so the time investment to learn and use those tools may outweigh their benefits.

However, synthetic computer-generated datasets can be a way to avoid legal and privacy issues concerning the use of real images.

Previous Work on Ground Truth Data: Art vs. Science

In this section, we survey some literature on ground truth data. We also highlight several examples of automatic ground truth data labeling, as well as other research on metrics for establishing if, in fact, the ground truth data is effective. Other research surveyed here includes how closely ground truth features agree with human perception and expectations, for example, whether or not the edges that humans detect in the ground truth data are, in fact, found by the chosen detector algorithms.

General Measures of Quality Performance

Compared to other topics in computer vision, little formal or analytic work has been published to guide the *creation* of ground truth data. However, the machine learning community provides a wealth of guidance for measuring the *quality* of visual recognition between ground truth data used for training and test datasets. In general, the size of the training set or ground truth data is key to its accuracy [329–331] and the larger the better, assuming the right data is used.

Key journals to dig deeper into machine learning and testing against ground truth data include the journal IEEE PAMI for Pattern Analysis and Machine Intelligence, whose articles on the subject go back to 1979. While the majority of ground truth datasets contain real images and video sequences, some practitioners have chosen to create synthetic ground truth datasets for various application domains, such as the standard Middlebury dataset with synthetic 3D images. See Appendix B for available real ground truth datasets, along with a few synthetic datasets.

One noteworthy example framework for ground truth data, detector, and descriptor evaluation is the Mikolajczyk and Schmidt methodology (M&S), discussed later in this chapter. Many computer vision research projects follow the M&S methodology using a variety of datasets.

Measures of Algorithm Performance

Ericsson and Karlsson [94] developed a ground truth correspondence measure (GCM) for benchmarking and ranking algorithm performance across seven real datasets and one synthetic dataset. Their work focused on statistical shape models and boundaries, referred to as *polygon shape descriptors* in the vision taxonomy in Chap. 5. The goal was to automate the correspondence between shape models in the database and detected shapes from the ground truth data using their GCM. Since shape models can be fairly complex, the goal of automating model comparisons and generating quality metrics specific to shape description is novel.

Dutagaci et al. [83] developed a framework and method, including ground truth data, to measure the *perceptual* agreement between humans and 3D interest point detectors—in other words, do the 3D interest point detectors find the same interest points as the humans expect? The ground truth data includes a known set of human-labeled interest points within a set of images, which were collected automatically by an Internet scraper application. The human-labeled interest points were sorted toward a consensus set, and outliers were rejected. The consensus criterion was a radius region counting the number of humans who labeled interest points within the radius. A set of 3D interest point detectors was ran against the data and compared using simple metrics such as false positives, false negatives, and a weighted miss error. The ground truth data was used to test the agreement between humans and machine vision algorithms for 3D interest point detectors. The conclusions included observations that humans are indecisive and widely divergent about choosing interest points, and also that interest point detection algorithms are a fuzzy problem in computer vision.

Hamarneh et al. [80] develop a method of automatically generating ground truth data for medical applications from a reference dataset with known landmarks, such as segmentation boundaries and interest points. The lack of experts trained to annotate the medical images and generate the ground truth data motivated the research. In this work, the data was created by generating synthetic images simulating object motion, vibrations, and other considerations, such as noise. Prastawa et al. [81] developed a similar approach for medical ground truth generation. Haltakov et al. [492] developed synthetic ground truth data from an automobile-driving simulator for testing driver assistance algorithms, which provided situation awareness using computer vision methods.

Vedaldi et al. [82] devised a framework for characterizing affine co-variant detectors, using synthetically generated ground truth as 3D scenes employing ray tracing, including simulated natural and man-made environments; a depth map was provided with each scene. The goal was to characterize co-variant detector performance under affine deformations, and to design better covariant detectors as a result. A set of parameterized features were defined for modeling the detectors, including points, disks and oriented disks, and various ellipses and oriented ellipses. A large number of 3D scenes were generated, with up to 1000 perspective views, including depth maps and camera calibration information. In this work, the metrics and ground truth data were designed together to focus on the analysis of geometric variations. Feature region shapes were analyzed with emphasis on disks and warped elliptical disks to discover any correspondence and robustness over different orientations, occlusion, folding, translation, and scaling (The source code developed for this work is available.¹).

¹ See the “VLFeat” open-source project online (<http://www.vlfeat.org>).

Rosin's Work on Corners

Research by Rosin [53, 84] involved the development of an analytical taxonomy for ground truth data pertaining to gray scale corner properties, as illustrated in Fig. 7.2. Rosin developed a methodology and case study to generate both the ground truth dataset and the metric basis for evaluating the performance and accuracy of a few well-known corner detectors. The metric is based on the receiver operating characteristic (ROC) to measure the accuracy of detectors to assess corners vs. noncorners. The work was carried out over 13,000 synthetic corner images with variations on the synthetic corners to span different orientations, subtended angles, noise, and scale. The synthetic ground truth dataset was specifically designed to enable the detection and analysis of a set of chosen corner properties, including bluntness or shape of apex, boundary shape of cusps, contrast, orientation, and subtended angle of the corner.

A novel aspect of Rosin's work was the generation of explicit types of *synthetic interest points* such as corners, nonobvious corners, and noncorners into the dataset, with the goal of creating a statistically interesting set of features for evaluation that diverged from idealized features. The synthetic corners were created and generated in a simulated optical system for realistic rendering to produce corners with parameterized variations including affine transformations, diffraction, subsampling, and in some cases, adding noise. Rosin's ground truth dataset is available for research use, and has been used for corner detector evaluation of methods from Kitchen and Rosenfeld, Paler, Foglein, and Illingworth, as well as the Kittler Detector and the Harris and Stephens Detector.

Similar to Rosin, a set of synthetic interest point alphabets are developed later in this chapter and tested in Appendix A, including edge and corner alphabets, with the goal of comparing human perception of interest points against machine vision methods. The synthetic interest points and

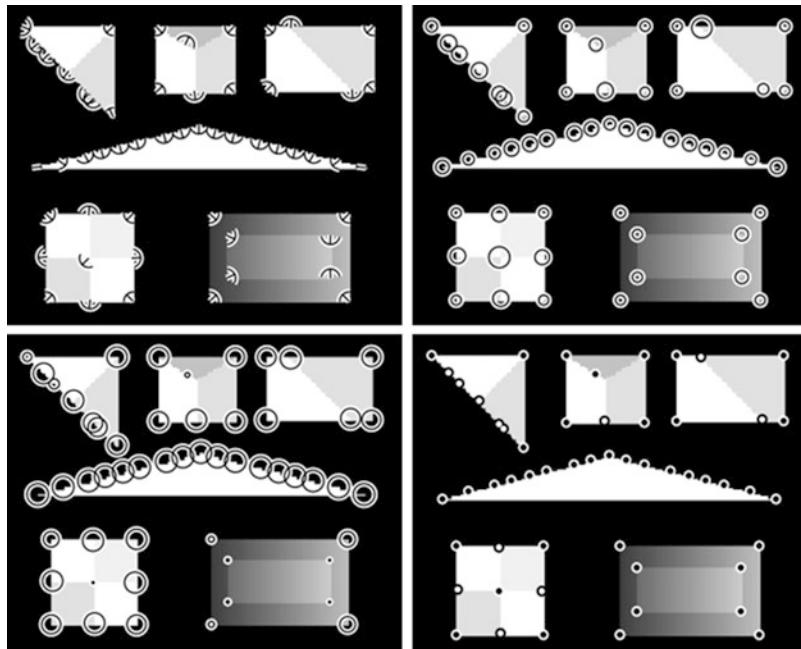


Figure 7.2 Images illustrating the Rosin corner metrics: (Top left) Corner orientation and subtended angle. (Top right) Bluntness. (Bottom left) Contrast. (Bottom right) Black/white corner color. (Images © Paul Rosin and used by permission [53])

corners are designed to test pixel thickness, edge intersections, shape, and complexity. The set diverges significantly from those of Rosin and others, and attempts to fill a void in the analysis of interest point detectors. The alphabets are placed on a regular grid, allowing for determining position detection count.

Key Questions for Constructing Ground Truth Data

In this section we identify some key questions to answer for creating ground truth data, rather than providing much specific guidance or answers. The type of work undertaken will dictate the type of guidance, for example, published research usually requires widely accepted ground truth data to allow for peer review and duplication of results. In medical or automobile industries, there may be government regulations, and also legal issues if competitors publish measurement or performance data. For example, if a company publishes any type of benchmark results against a ground truth data set comparing the results with those of competitor systems, all such data and claims should be reviewed by an attorney to avoid the complexities and penalties of commerce regulations, which can be daunting and severe.

For real products and real systems, perhaps the best guidance comes from the requirements, expectations and goals for performance and accuracy. Once a clear set of requirements are in place, then the ground truth selection process can begin.

Content: Adopt, Modify, or Create

It is useful to become familiar with existing ground truth datasets prior to creating a new one. The choices are obvious:

- Adopt an existing dataset.
- Adopt-And-Modify an existing data set.
- Create a new dataset.

Survey of Available Ground Truth Data

[Appendix B](#) has information on several existing ground truth datasets. Take some time to get to know what is already available, and study the research papers coming out of SIGGRAPH, CVPR, IJCV, NIPS in [Appendix C](#), and other research conferences to learn more about new datasets and how they are being used. The available datasets come from a variety of sources, including:

- Academic research organizations, usually available free of charge for academic research.
- Government datasets, sometimes with restricted use.
- Industry datasets, available from major corporations like Microsoft, sometimes can be licensed for commercial use.

Fitting Ground Truth Data to Algorithms

Perhaps the biggest challenge is to determine whether a dataset is a correct fit for the problem at hand. Is the detail in the ground truth data sufficient to find the boundaries and limits of the chosen algorithms and systems? “Fitting” applies to key variables such as the ground truth data, the algorithms used, the object models, classifier, and the intended use-cases. See Fig. 7.3, which illustrates how ground truth data, image preprocessing, detector and descriptor algorithms, and model metrics should be fitted.

Ground truth data should be carefully selected to fit and measure the accuracy of the statistical model of the classifier and machine vision algorithms. Overfitting happens when the model captures the noise in the training data, or in other words, the model fits the training data too well and does not generalize to related data. Overfitting may be caused by a complex model. Underfitting happens when the model fails to capture the underlying data trend. Underfitting may be caused by a simplistic model. The sweet spot between overfitting and underfitting can be found with the right ground truth data. Usually, ground truth data is divided into a larger training set and one or more smaller test sets as needed.

The training results can be evaluated against the following criteria: *overfitting* captures noise, *underfitting* misses the trend, and *good fitting* captures the trend. Both training data and the training algorithms are related.



Figure 7.3 (Top left) Image preprocessing for edges shown using Shen-Castan edge detection against ground truth data. (Top right) Over-fitting detection parameters yield too many small edges. (Bottom left) Under fitting parameters yield too few edges. (Bottom right) Relaxed parameters yield reasonable edges

Here are a few examples to illustrate the variables.

- **Training Data fitting:** If the dataset does not provide enough pixel resolution or bit depth, or there are insufficient unique samples in the training set, the model will be incomplete, the matching may suffer, and the data is *under-fitted* to the problem. Or, if the ground truth contains too many different types of features that will never be encountered in the test set or in real applications. If the model resolution is 16 bits per RGB channel when only 8 bits per color channel are provided in real data, the data and model are *over-fitted* to the problem.
- **Training Algorithm fitting:** If scale invariance is included in the ground truth data, and the LBP operator being tested is not claimed to be scale invariant, then the algorithm is *under-fitted* to the data. If the SIFT method is used on data with no scale or rotation variations, then the SIFT algorithm is *over-fitted* to the data.
- **Use-case fitting:** If the use-cases are not represented in the data and model, the data and model are *under-fitted* to the problem.

Scene Composition and Labeling

Ground truth data is composed of labeled features such as foreground, background, and objects or features to recognize. The labels define exactly what features are present in the images, and these labels may be a combination of on-screen labels, associated label files, or databases. Sometimes a randomly composed scene from the wild is preferred as ground truth data, and then only the required items in the scene are labeled. Other times, ground truth data is scripted and composed the way a scene for a movie would be.

In any case, the appropriate objects and actors in the scene must be labeled, and perhaps the positions of each must be known and recorded as well. A database or file containing the labels must therefore be created and associated with each ground truth image to allow for testing. See Fig. 7.4, which shows annotated or labeled ground truth dataset images for a scene analysis of cuboids [54]. See also the Labelme database described in Appendix B, which allows contributors to provide labeled databases.



Figure 7.4 Annotated or labeled ground-truth dataset images for scene analysis of cuboids (*left and center*). The labels are annotated manually into the ground-truth dataset, in *yellow* (*light gray* in B&W version) marking the cuboid edges and corners. (*Right*) Ground-truth data contains precomputed 3D corner HOG descriptor sets, which are matched against live detected cuboid HOG feature sets. Successful matches shown in *green* (*dark gray* in B&W version). (Images used by permission © Bryan Russel, Jianxiong Xiao, and Antonio Torralba)

Composition

Establishing the right set of ground truth data is like assembling a composition; several variables are involved, including:

- **Scene Content:** Designing the visual content, including fixed objects (those that do not move), dynamic objects (those that enter and leave the scene), and dynamic variables (such as position and movement of objects in the scene).
- **Lighting:** Casting appropriate lighting onto the scene.
- **Distance:** Setting and labeling the correct distance for each object to get the pixel resolution needed—too far away means not enough pixels.
- **Motion Scripting:** Determining the appropriate motion of objects in the scene for each frame; for example, how many people are in the scene, what are their positions and distances, number of frames where each person appears, and where each person enters and exits. Also, scripting scenes to enable invariance testing for changes in perspective, scale, affine geometry, occlusion.
- **Labeling:** Creating a formatted file, database, or spreadsheet to describe each labeled ground truth object in the scene for each frame.
- **Intended Algorithms:** Deciding which algorithms for interest point and feature detection will be used, what metrics are to be produced, and which invariance attributes are expected from each algorithm; for example, an LBP by itself does not provide scale invariance, but SIFT does.
- **Intended Use-Cases:** Determining the problem domain or application. Does the ground truth data represent enough real use-cases?
- **Image Channel Bit Depth, Resolution:** Setting these to match requirements.
- **Metrics:** Defining the group of metrics to measure—for example, false positives and false negatives. Creating a test fixture to run the algorithms against the dataset, measuring and recording all necessary results.
- **Analysis:** Interpreting the metrics by understanding the limitations of both the ground truth data and the algorithms, defining the success criteria.
- **Open Rating Systems:** Exploring whether there is an open rating system that can be used to report the results. For example, the Middlebury Dataset provides an open rating system for 3D stereo algorithms, and is described in [Appendix B](#); other rating systems are published as a part of grand challenge contests held by computer vision organizations and governments, and some are reviewed in [Appendix B](#). Open rating systems allow existing and new algorithms to be compared on a uniform scale.

Labeling

Ground truth data may simply be images returned from a search engine, and the label may just be the search engine word or phrase. Fig. 7.5 shows a graph of photo connectivity for photo tourism [55–57] that is created from pseudo-random images of a well-known location, the Trevi Fountain in Rome. It is likely that in 5–10 years, photo tourism applications will provide high-quality image reconstruction including textures, 3D surfaces, and rerenderings of the same location, rivaling real photographs.

For some applications, labels and markers are inserted into the ground truth datasets to enable analysis of results, as shown in the 3D scene understanding database for cuboids in Fig. 7.4. Another example later in this chapter composes scenes using *synthetic alphabets* of interest points and corners that are superimposed on the images of a regularly spaced grid to enable position verification (see also [Appendix A](#)). In some visual tracking applications, *markers* are attached to physical objects (a wrist band, for example) to establish ground truth features.

Another example is ground truth data composed to measure *gaze detection*, using a video sequence containing labels for two human male subjects entering and leaving the scene at a known location and time, walking from left to right at a known speed and depth in the scene. The object they are gazing at would be at a known location and be labeled as well.

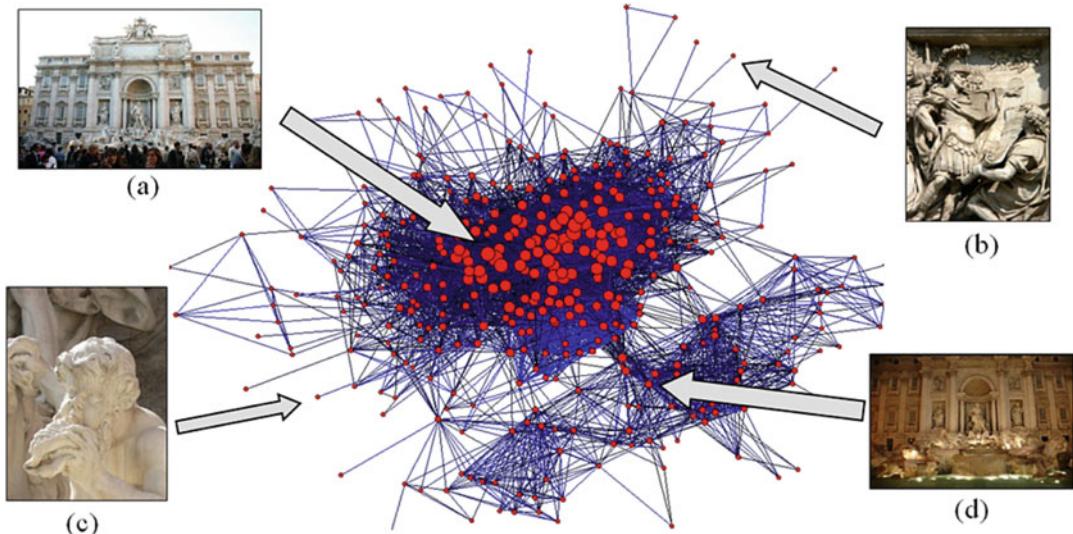


Figure 7.5 Graph of photo connectivity (*center*) created from analyzing multiple public images from a search engine of the Trevi Fountain (a). Edges show photos matched and connected to features in the 3D scene, including daytime and nighttime lighting (b–d). (Images © Noah Snavely [56] and used by permission)

Defining the Goals and Expectations

To establish goals for the ground truth data, questions must be asked. For instance, what is the intended use of the application requiring the ground truth data? What decisions must be made from the ground truth data in terms of accuracy and performance? How is quality and success measured? The goals of academic research and commercial systems are quite different.

Mikolajczyk and Schmid Methodology

A set of well-regarded papers by Mikolajczyk, Schmid and others [37, 71, 74, 83, 298] provides a good methodology to start with for measuring local interest points and feature detector quality. Of particular interest is the methodology used to measure scale and affine invariant interest point detectors [298] which uses natural images to start, then applies a set of known affine transformations to those images, such as homography, rotation, and scale. Interest point detectors are run against the images, followed by feature extractors, and then the matching recall and precision are measured across the transformed images to yield quality metrics.

Open Rating Systems

The computer vision community is, little by little, developing various open rating systems, which encourage algorithm comparisons and improvements to increase quality. In areas where such open databases exist, there is rapid growth in quality for specific algorithms. [Appendix B](#) lists open rating

systems such as the Pascal VOC Challenge for object detection. Pascal VOC uses an open ground truth database with associated grand challenge competition problems for measuring the accuracy of the latest algorithms against the dataset.

Another example is the Middlebury Dataset, which provides ground truth datasets covering the 3D stereo algorithm domain, allowing for open comparison of key metrics between new and old algorithms, with the results published online.

Corner Cases and Limits

Finding out where the algorithms fail is valuable. Academic research is often not interested in the rigor required by industry in defining failure modes. One way to find the corner cases and limits is to run the same tests on a wide range of ground truth data, perhaps even data that is outside the scope of the problem at hand. Given the availability of publicly available ground truth databases, using several databases is realistic.

However, once the key ground truth data is gathered, it can also be useful to devise a range of corner cases—for example, by providing noisy data, intensity filtered data, or blurry data to test the limits of performance and accuracy.

Interest Points and Features

Interest points and features are not always detected as expected or predicted. Machine vision algorithms detect a different set of interest points than those humans expect. For example, Fig. 7.6 shows obvious interest points missed by the SURF algorithm with a given set of parameters, which uses a method based on determinant of Hessian blob detection. Note that some interest points obvious to humans are not detected at all, some false positives occur, and some identical interest points are not detected consistently.

Also, real interest points change over time—for example, as objects move and rotate—which is a strong argument for using real ground truth data vs. synthetic data to test a wide range of potential interest points for false positives and false negatives.

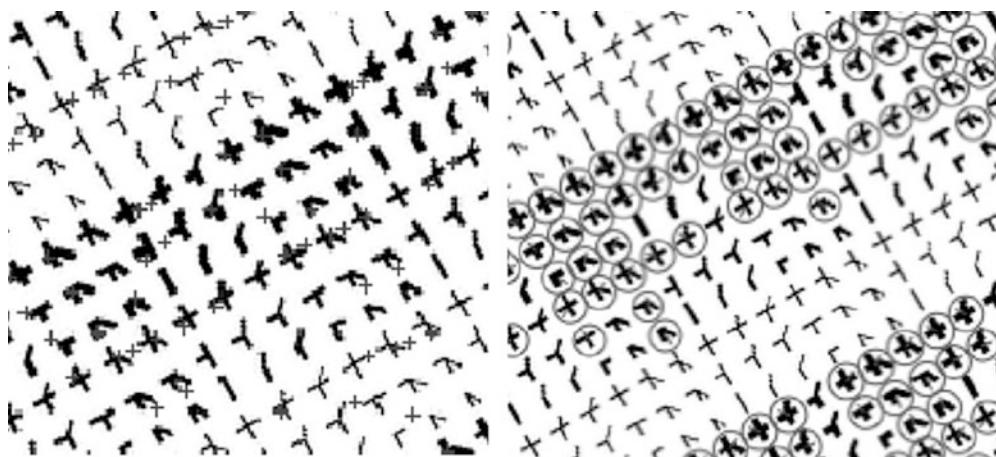


Figure 7.6 Interest points detected on the same image using different methods: (*Left*) Shi-Tomasi corners marked with *crosses*. (*Right*) SURF interest points marked with *circles*. Results are not consistent or deterministic

Robustness Criteria for Ground Truth Data

In Chap. 5, a robustness criteria was developed listing various invariance attributes, such as rotation and scale. Here, we apply the robustness criteria to the development of ground truth data.

Illustrated Robustness Criteria

Table 7.1 discusses various robustness criteria attributes, not all attributes are needed for a given application. For example, if radial distortion might be present in an optical system, then the best algorithms and corresponding metrics will be devised that are robust to radial distortion, or as mitigation, the vision pipeline must be designed with a preprocessing section to remove or compensate for the radial distortion prior to determining the metrics.

Table 7.1 Robustness criteria for ground truth data

Attribute	Discussion
Uneven illumination	Define range of acceptable illumination for the application; uneven illumination may degrade certain algorithms, some algorithms are more tolerant.
Brightness	Define expected brightness range of key features, and prepare ground-truth data accordingly.
Contrast	Define range of acceptable contrast for the application; some algorithms are more tolerant.
Vignette	Optical systems may degrade light and manifest as dim illumination at the edges. Smaller the features are localized better and may be able to overcome this situation; large features that span areas of uneven light are affected more.
Color accuracy	Inaccurate color space treatment may result in poor color performance. Colorimetry is important; consider choosing the right color space (RGB, YIQ, Lab, Jab, etc.) and use the right level of bit precision for each color, whether 8/16 bits is best.
Clutter	Some algorithms are not tolerant of clutter in images and rely on the scene to be constructed with a minimal number of subjects. Descriptor pixel size may be an issue for block search methods—too much extraneous detail in a region may be a problem for the algorithm.
Occlusion and clipping	Objects may be occluded or hidden or clipped. Algorithms may or may not tolerate such occlusion. Some occlusion artifacts can be eliminated or compensated for using image pre-processing and segmentation methods.
Outliers and proximity	Sometimes groups of objects within a region are the subject, and outliers are to be ignored. Also, proximity of objects or features may guide classification, so varying the arrangement of features or objects in the scene may be critical.
Noise	Noise may take on regular or random patterns, such as snow, rain, single-pixel spot noise, line noise, random electrical noise affecting pixel bit resolution, etc.
Motion blur	Motion blur is an important problem for almost all real-time applications. This can be overcome by using faster frame rates and employing image pre-processing to remove the motion blur, if possible.
Jitter and judder	Common problem in video images taken from moving cameras, where each scan line may be offset from the regular 2D grid.
Focal plane or depth	If the application or use-case for the algorithm assumes all depths of the image to be in focus, then using ground truth data with out-of-focus depth planes may be a good way to test the limits.

(continued)

Table 7.1 (continued)

Attribute	Discussion
Pixel depth Resolution	If features are matched based on the value of pixels, such as gray scale intensity or color intensity, pixel resolution is an issue. For example, if a feature descriptor uses 16 bits of effective gray scale intensity but the actual use-case and ground truth data provide only 8 bits of resolution, the descriptor may be over-fitted to the data, or the data may be unrealistic for the application.
Geometric distortion	Complex warping may occur due to combinations of geometric errors from optics or distance to subject. On deformable surfaces such as the human face, surface and feature shape may change in ways difficult to geometrically describe.
Scale, projection	Near and far objects will be represented by more or less pixels, thus a multi-scale dataset may be required for a given application, as well as multi-scale feature descriptors. Algorithm sensitivity to feature scale and intended use case also dictate ground truth data scale.
Affine transforms and rotation	In some applications like panoramic image stitching, very little rotation is expected between adjacent frames—perhaps up to 15 degrees may be tolerated. However, in other applications like object analysis and tracking of parts on an industrial conveyor belt, rotation between 0 and 360 degrees is expected.
Feature mirroring, translation	In stereo correspondence, L/R pair matching is done using the assumption that features can be matched within a limited range of translation difference between L/R pairs. If the translation is extreme between points, the stereo algorithm may fail, resulting in holes in the depth map, which must be filled.
Reflection	Some applications, like recognizing automobiles in traffic, require a feature model, which incorporates a reflective representation and a corresponding ground truth dataset. Automobiles may come and go from different directions, and have a reflected right/left feature pair.
Radial distortion	Optics may introduce radial distortion around the fringes; usually this is corrected by a camera system using digital signal processors or fixed-function hardware prior to delivering the image.

Using Robustness Criteria for Real Applications

Each application requires a different set of robustness criteria to be developed into the ground truth data. Table 7.2 illustrates how the robustness criteria may be applied to a few real and diverse applications.

Table 7.2 Robustness criteria applied to sample applications (each application with different requirements for robustness)

General Objective Criteria Attributes	Industrial inspection of apples on a conveyor belt, fixed distance, fixed speed, fixed illumination	Automobile identification on roadway, day and night, all road conditions	Multi-view stereo reconstruction bundle adjustment
Uneven illumination	-	Important	Useful
Brightness	Useful	Important	Useful
Contrast	Useful	Important	Useful
Vignette	Important	Useful	Useful

(continued)

Table 7.2 (continued)

General Objective Criteria Attributes	Industrial inspection of apples on a conveyor belt, fixed distance, fixed speed, fixed illumination	Automobile identification on roadway, day and night, all road conditions	Multi-view stereo reconstruction bundle adjustment
Color accuracy	Important	Important	Useful
Clutter	-	Important	Important
Occlusion	-	Important	Important
Outliers	-	Important	Important
Noise	-	Important	Useful
Motion blur	Useful	Important	Useful
Focal plane or depth	-	Important	Useful
Pixel depth resolution	Useful	Important	important
Subpixel resolution	-	-	important
Geometric distortion (warp)	-	Useful	Important
Affine transforms	-	Important	Important
Scale	-	Important	Important
Skew	-	-	-
Rotation	Important	Useful	Useful
Translation	Important	Useful	Useful
Projective transformations	Important	Important	-
Reflection	Important	Important	-
Radial distortion	-	-	Important
Polar distortion	-	-	Important
Discrimination or uniqueness	-	Useful	-
Location accuracy	-	Useful	-
Shape and thickness distortion	-	Useful	-

As illustrated in Table 7.2, a multi-view stereo (MVS) application will hold certain geometric criteria as very important, since accurate depth maps require accurate geometry assumptions as a basis for disparity calculations. For algorithm accuracy tuning, corresponding ground truth data should be created using a well-calibrated camera system for positional accuracy of the 3D scene to allow for effective comparisons.

Another example in Table 7.2 with many variables in an uncontrolled environment is that of automobile identification on roadways—which may be concerned with distance, shape, color, and noise. For example, identifying automobiles may require ground truth images of several vehicles from a wide range of natural conditions, such as dawn, dusk, cloudy day, and full sun, and including conditions such as rainfall and snowfall, motion blur, occlusion, and perspective views. An example automobile recognition pipeline is developed in Chap. 8.

Also shown Table 7.2 is an example with a controlled environment: industrial inspection. In industrial settings, the environment can be carefully controlled using known lighting, controlling the speed of a conveyor belt, and limiting the set of objects in the scenes. Accurate models and metrics for each object can be devised, perhaps taking color samples and so forth—all of which can be done a priori. Ground truth data could be easily created from the actual factory location.

Pairing Metrics with Ground Truth

Metrics and ground truth data should go together. Each application will have design goals for robustness and accuracy, and each algorithm will also have different intended uses and capabilities. For example, the SUSAN detector discussed in Chap. 6 is often applied to wide baseline stereo applications, and stereo applications typically are not concerned much with rotational invariance because the image features are computed on corresponding stereo pair frames that have been affine rectified to align line by line. Feature correspondence between image pairs is expected within a small window, with some minor translation on the x axis.

Pairing and Tuning Interest Points, Features, and Ground Truth

Pairing the right interest point detectors and feature descriptors can enhance results, and many interest point methods are available and were discussed in Chap. 6. When preparing ground truth data, the method used for interest point detection should be considered for guidance.

For example, interest point methods using derivatives, such as the Laplace and Hessian style detectors, will not do very well without sufficient contrast in the local pixel regions of the images, since contrast accentuates maxima, minima and local region changes. However, a method such as FAST9 is much more suited to low-contrast images, uses local binary patterns, and is simple to tune the compare threshold and region size to detect corners and edges; but the trade-off in using FAST9 is that scale invariance is sacrificed.

A method using edge gradients and direction, such as eigen methods, would require ground truth containing sufficient oriented edges at the right contrast levels. A method using morphological interest points would likewise require image data that can be properly thresholded and processed to yield the desired shapes.

Interest point methods also must be tuned for various parameters like strength of thresholds for accepting and rejecting candidate interest points, as well as and region size. Choosing the right interest point detector, tuning, and pairing with appropriate ground truth data are critical. The effect of tuning interest point detector parameters is illustrated in Fig. 7.6.

Examples Using the General Vision Taxonomy

As a guideline for pairing metrics and ground truth data, we use the vision taxonomy developed in Chap. 5 to illustrate how feature metrics and ground truth data can be considered together.

Table 7.3 presents a sample taxonomy and classification for SIFT and FREAK descriptors, which can be used to guide selection of ground truth data and also show several similarities in algorithm capabilities. In this example, the invariance attributes built into the data can be about the same—namely scale and rotation invariance. Note that the compute performance claimed by FREAK is orders of magnitude faster than SIFT, so perhaps the ground truth data should contain a sufficient minimum and maximum number of features per frame for good performance measurements.

Table 7.3 General vision taxonomy for describing FREAK and SIFT

Visual Metric Taxonomy Comparison		
Attribute	SIFT	FREAK
Feature Category Family	Spectra Descriptor	Local Binary Descriptor
Spectra Dimensions	Multivariate	Single Variate
Spectra Value	Orientation Vector Gradient Magnitude Gradient Direction HOG, Cartesian Bins	Orientation Vector Bit Vector Of values Cascade of 4 Saccadic Descriptors
Interest Point	SIFT DOG over 3D Scale Pyramid	Multi-scale AGAST
Storage Format	Spectra Vector	Bit Vector Orientation Vector
Data Types	Float	Integer
Descriptor Memory	512 bytes, 128 floats	64 Bytes, 4 16-byte Cascades
Feature Shape	Rectangle	Circular
Feature Search Method	Coarse to Fine Image Pyramid Scale Space Image Pyramid Double-scale First Pyramid Level Sparse at Interest Points	Sparse at interest points
Pattern Pair Sampling	n.a.	Foveal Centered Trained Pairs
Pattern Region Size	41x41 Bounding Box	31x31 Bounding Box (may vary)
Distance Function	Euclidean Distance	Hamming Distance
Run-Time Compute	100% (SIFT is the baseline)	.1% of SIFT
Feature Density	Sparse	Sparse
Feature Pattern	Rectangular kernel Sample Weighting Pattern	Binary compare pattern
Claimed Robustness	Scale Rotation Noise Affine Distortion Illumination	Scale Rotation Noise
*Final robustness is a combination of interest point method, descriptor method, and classifier		

Synthetic Feature Alphabets

In this section, we create synthetic ground truth datasets for interest point algorithm analysis. We create alphabets of *synthetic interest points* and *synthetic corner points*. The alphabets are *synthetic*, meaning that each element is designed to perfectly represent chosen binary patterns, including points, lines, contours, and edges.

Various pixel widths or thicknesses are used for the alphabet characters to measure fine and coarse feature detection. Each pattern is registered at known pixel coordinates on a grid in the images to

allow for detection accuracy to be measured. The datasets are designed to enable comparison between human interest point perception and machine vision interest point detectors.

Here is a high-level description of each synthetic alphabet dataset:

- **Synthetic Interest Point Alphabet.** Contains points such as boxes, triangles, circle, half boxes, half triangles, half circles, edges, and contours.
- **Synthetic Corner Point Alphabet.** Contains several types of corners and multi-corners at different pixel thickness.
- **Natural images overlaid with synthetic alphabets.** Contains both black and white versions of the interest points and corners overlaid on natural images.

Note The complete set of ground truth data is available in [Appendix A](#).

Analysis is provided in [Appendix A](#), which includes running ten detectors against the datasets. The detectors are implemented in OpenCV, including SIFT, SURF, ORB, BRISK, HARRIS, GFFT, FAST9, SIMPLE BLOB, MSER, and STAR. Note that the methods such as SIFT, SURF, and ORB provide both an interest point detector and a feature descriptor implementation. We are only concerned with the interest point detector portion of each method for the analysis, not the feature descriptor.

The idea of using synthetic image alphabets is not new. As shown in Fig. 7.2, Rosin [53] devised a synthetic set of gray corner points and corresponding measurement methods for the purpose of quantifying corner properties via attributes such as bluntness or shape of apex, boundary shape of cusps, contrast, orientation, and subtended angle of the corner. However, the synthetic interest point and corner alphabets in this work are developed to address a different set of goals, discussed next.

Goals for the Synthetic Dataset

The goals and expectations for this synthetic dataset are listed in Table 7.4. They center on enabling analysis to determine which synthetic interest points and corners are found, so the exact count and position of each interest point is a key requirement.

Table 7.4 Goals and expectations for the ground truth data examples: comparison of human expectations with machine vision results

Goals	Approach
Interest point and corner detectors, stress testing	Provide synthetic features easily recognized by a human; measure how well various detectors perform.
Human recognizable synthetic interest point sets	Synthetic features recognized by humans are developed spanning shapes and sizes of edges and line segments, contours and curved lines, and corners and multi-corners.
Grid positioning of interest points	Each interest point will be placed on a regular grid at a known position for detection accuracy checking.
Scale invariance	Synthetic interest points to be created with the same general shape but using different pixel thickness for scale.
Rotation invariance	Interest points will be created, then rotated in subsequent frames.
Noise invariance	Noise will be added to some interest point sets.
Duplicate interest points, known count	Interest points will be created and duplicated in each frame for determining detection and performance.
Hybrid synthetic interest points overlaid on real images	Synthetic interest points on a grid are overlaid onto real images to allow for hybrid testing.
Interest point detectors, determinism and repeatability	Detectors will include SIFT, SURF, ORB, BRISK, HARRIS, GFFT, FAST9, SIMPLE BLOB, MSER, and STAR. By locating synthetic interest points on a grid, we can compute detection counts.

The human visual system does not work like an interest point detector, since detectors can accept features which humans may not recognize. The human visual system discriminates and responds to gradient information [240] in a scale and rotationally invariant manner across the retina, and tends to look for learned features relationships among gradients and color.

Humans learn about features by observations and experience, so learned expectations play a key role interpreting visual features. *People see what they believe and what they are looking for, and may not believe what they see if they are not looking for it.* For example, Fig. 7.7 shows examples of machine corner detection; a human would likely not choose all the same corner features. Note that the results are not what a human might expect, and also the algorithm parameters must be tuned to the ground truth data to get the best results.



Figure 7.7 Machine corner detection using the Shi-Tomasi method marked with *crosses*; results are shown using different parameter settings and thresholds for the strength and pixel size of the corners

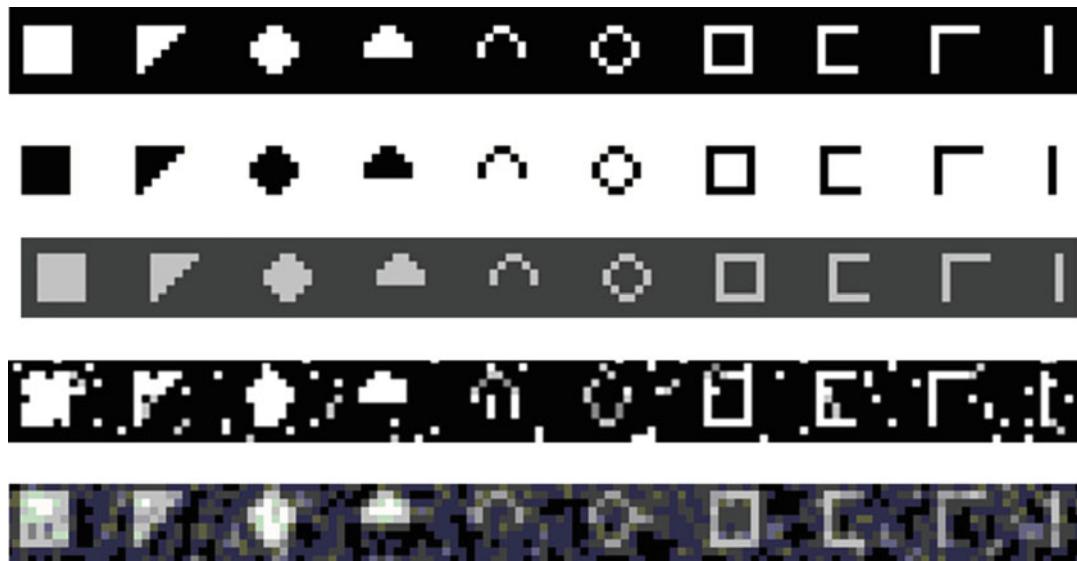


Figure 7.8 Portion of the synthetic interest point alphabet: points, edges, edges, and contours. (*Top to bottom*) White on black, black on white, light gray on dark gray, added salt and pepper noise, added Gaussian noise

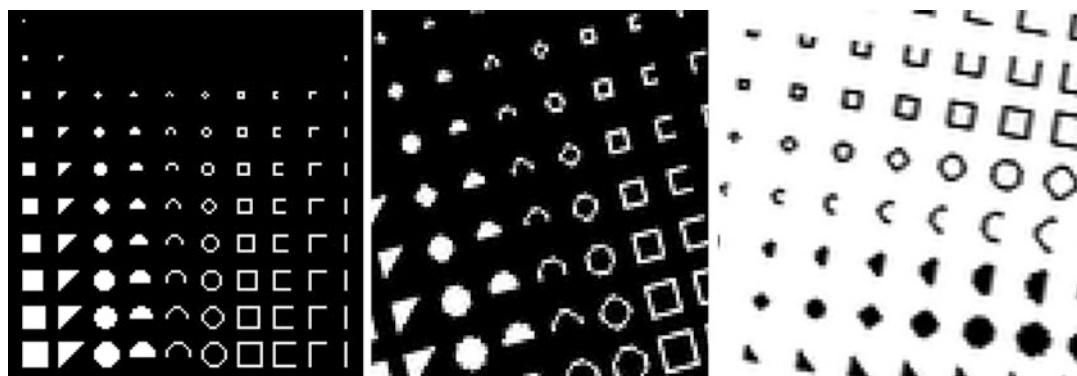


Figure 7.9 Scaled and rotated examples of the synthetic interest point alphabet. Notice the artifacts introduced by the affine rotation, which distorts the synthetic binary patterns via anti-aliasing and subsampling artifacts

Accuracy of Feature Detection via Location Grid

The goal of detector accuracy for this synthetic ground truth is addressed by placing synthetic features at a known position on a regular spaced grid, then after detection, the count and position are analyzed. Some of the detectors will find multiple features for a single synthetic interest point or corner. The feature grid size chosen is 14×14 pixels, and the grid extends across the entire image. See Figs. 7.8 and 7.9.

Rotational Invariance via Rotated Image Set

For each ground truth set, rotated versions of each image are created in the range $0\text{--}90^\circ$ at 10° increments. Since the synthetic features are placed on a regularly spaced grid at known positions, the

new positions under rotation are easily computed. The detected synthetic features can be counted and analyzed. See [Appendix A](#) for results.

Scale Invariance via Thickness and Bounding Box Size

The synthetic corner point features are rendered into the ground truth data with feature edge thickness ranging from 1 to 3 pixels for simulated scale variation. Some of the interest point features, such as boxes, triangles, and circles, are scaled in a bounding box ranging from 1×1 pixels to 10×10 pixels to allow for scale invariance testing.

Noise and Blur Invariance

A set of synthetic alphabets is rendered using Gaussian noise, and another set using salt-and-pepper noise to add distortion and uncertainty to the images. In addition, by rotating the interest point alphabet at varying angles between 0° and 90° , digital blur is introduced to the synthetic patterns as they are rendered, owing to the anti-aliasing interpolations introduced in the affine transform algorithms.

Repeatability

Each ground truth set contains a known count of synthetic features to enable detection rates to be analyzed. To enable measurement of the repeatability of each detector, there are multiple duplicate copies of each interest point feature in each image. A human would expect identical features to be detected in an identical manner; however, results in [Appendix A](#) show that some interest point detectors do not behave in a predictable manner, and some are more predictable than others.

As shown in Fig. 7.6, detectors do not always find the same identical features. For example, the synthetic alphabets are provided in three versions—black on white, white on black, and light gray on dark gray—for the purpose of testing each detector on the same pattern with different gray levels and polarity. See [Appendix A](#) showing the how the detectors provide different results based on the polarity and gray level factors.

Real Image Overlays of Synthetic Features

A set of images composed of synthetic interest points and corners overlayed on top of real images is provided, sort of like markers. Why overlay interest point markers, since the state of the art has moved beyond markers to markerless tracking? The goal is to understand the limitations and behavior of the detectors themselves, so that analyzing their performance in the presence of natural and synthetic features will provide some insight.

Synthetic Interest Point Alphabet

As shown in Figs. 7.7 and 7.8, an alphabet of synthetic interest points is defined across a range of pixel resolutions or thicknesses to include the following features:

- POINT/SQUARE, 1–10 PIXELS SIZE
- POINT/TRIANGLE HALF-SQUARE, 3–1 PIXELS SIZE
- CIRCLE, 3–10 PIXELS SIZE
- CIRCLE/HALF-CIRCLE, 3–10 PIXELS SIZE
- CONTOUR, 3–10 PIXELS SIZE
- CONTOUR/HALF-CONTOUR, 3–10 PIXELS SIZE
- CONNECTED EDGES

- DOUBLE CORNER, 3–10 PIXELS SIZE
- CORNER, 3–10 PIXELS SIZE
- EDGE, 3–10 PIXELS SIZE

The synthetic interest point alphabet contains 83 unique elements composed on a 14×14 grid, as shown in Fig. 7.8. A total of seven rows and seven columns of the complete alphabet can fit inside a 1024×1024 image, yielding a total of $7 \times 7 \times 83 = 4067$ total interest points.

Synthetic Corner Alphabet

The synthetic corner alphabet is shown in Fig. 7.9. The alphabet contains the following types of corners and attributes:

- 2-SEGMENT CORNERS, 1,2,3 PIXELS WIDE
- 3-SEGMENT CORNERS, 1,2,3 PIXELS WIDE
- 4-SEGMENT CORNERS, 1,2,3 PIXELS WIDE

As shown in Fig. 7.10, the corner alphabet contains patterns with multiple types of corners composed of two-line segments, three-line segments, and four-line segments, with pixel widths of 1, 2, and 3. The synthetic corner alphabet contains 54 unique elements composed on a 14×14 pixel grid.

Each 1024×1024 pixel image contains 8×12 complete alphabets composed of 6×9 unique elements each, yielding $6 \times 9 \times 12 \times 8 = 5184$ total corner points per image. The full dataset includes rotated versions of each image from 0° to 90° at 10° intervals.



Figure 7.10 Portion of the synthetic corner alphabet, features include 2-, 3-, and 4-segment corners. (*Top to bottom*) White on black, black on white, light gray on dark gray, added salt and pepper noise, added Gaussian noise

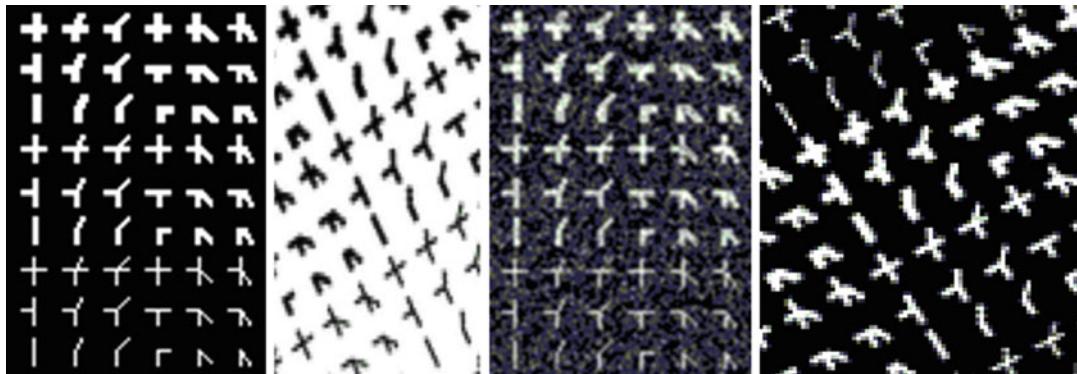


Figure 7.11 Synthetic corner points image portions

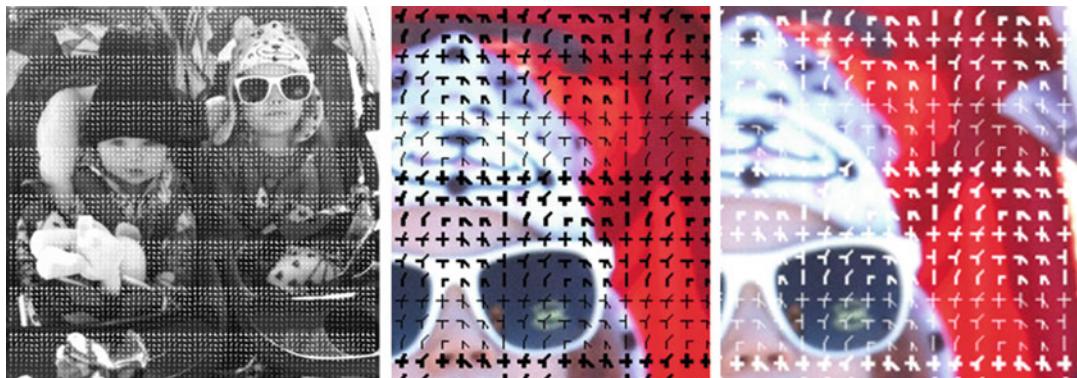


Figure 7.12 Synthetic interest points combined with real images, used for stress testing interest point and corner detectors with unusual pixel patterns

Hybrid Synthetic Overlays on Real Images

We combine the synthetic interest points and corners as overlays with real images to develop a *hybrid ground truth dataset* as a more complex case.

The merging of synthetic interest points over real data will provide new challenges for the interest point algorithms and corner detectors, as well as illustrate how each detector works. Using hybrid synthetic feature overlays on real images is a new approach for ground truth data (as far as the author is aware), and the benefits are not obvious outside of curiosity. One reason the synthetic overlay approach was chosen here is to fill the gap in the literature and research, since synthetic features overlays are not normally used. See Fig. 7.12.

The hybrid synthetic and real ground truth datasets are designed with the following goals:

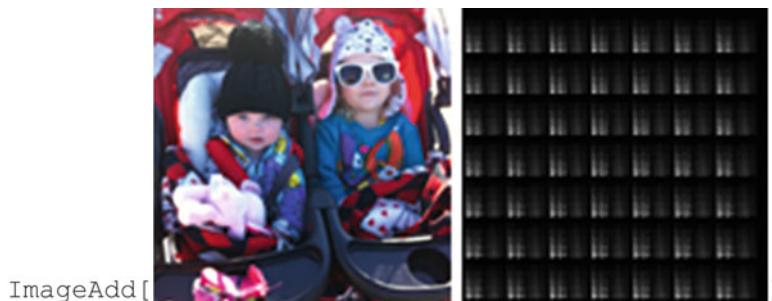
- Separate ground truth sets for interest points and corners, using the full synthetic alphabets overlaid on real images, to provide a range of pixel detail surrounding each interest point and corner.
- Display known positions and counts of interest points on a 14×14 grid.
- Provide color and gray scale images of the same data.
- Provide rotated versions of the same data $0\text{--}90^\circ$ at 10° intervals.

Method for Creating the Overlays

The alphabet can be used as a *binary mask* of 8-bit pixel values of black 0 × 00 and white 0 × ff for composing the image overlays. The following Boolean masking example is performed using Mathematica code `ImageMultiply` and `ImageAdd` operators.



`ImageMultiply` is used to get the negatives, and then followed by `ImageAdd` to get the positives. Note that in other image processing tool systems, a Boolean `ImageAND`, `ImageOR`, and `ImageNOT` may be provided as alternatives.



Summary

We survey manual and automated approaches to creating ground truth data, identify some best practices and guidelines, apply the robustness criteria and vision taxonomy developed in Chap. 5, and work through examples to create a ground truth dataset for evaluation of human perceptions compared to machine vision methods for keypoint detectors.

Here are some final thoughts and key questions for preparing ground truth data:

- **Appropriateness:** How appropriate is the ground truth dataset for the analysis and intended application? Are the use-cases and application goals built into the ground truth data and model? Is the dataset under-fitted or over-fitted to the algorithms and use-cases?
- **Public vs. proprietary:** Proprietary ground truth data is a barrier to independent evaluation of metrics and algorithms. It must be possible for interested parties to duplicate the metrics produced

by various types of algorithms so they can be compared against the ground truth data. Open rating systems may be preferred, if they exist for the problem domain. But there are credibility and legal hurdles for open-sourcing any proprietary ground truth data.

- **Privacy and legal concerns:** There are privacy concerns for individuals in any images chosen to be used; images of people should not be used without their permission, and prohibitions against the taking of pictures at restricted locations should be observed. Legal concerns are very real.
- **Real data vs. synthetic data:** In some cases it is possible to use computer graphics and animations to create synthetic ground datasets. Synthetic datasets should be considered especially when privacy and legal concerns are involved, as well as be viewed as a way of gaining more control over the data itself.

Chapter 7: Learning Assignments

1. Describe specific goals for ground truth data with respect to robustness and invariance attributes for a face emotion recognition application.
2. Discuss the classes of image emotions required for collecting a ground truth data set. Derive a statistical method to determine how many images are needed for each class, and how the images should be processed.
3. Provide a high-level design or write some script code (php, ruby, etc.) to automatically collect and evaluate the ground truth data against the selected goals. Include statistical metrics and other image analysis methods in the code for automatically evaluating the images, and describe what is possible to automate in the code, and what must be done by humans.
4. Name a few publicly available ground truth data sets, and where they can be obtained.
5. Discuss positive and negative training samples.
6. Discuss when synthetic ground truth images might be useful, and how synthetic images might be designed, and prepare a plan including the names of software tools needed.
7. Discuss when a ground truth data set should be adopted vs. created from scratch.
8. Discuss when a labeled ground truth data set is appropriate vs. an unlabeled ground truth data set.
9. Discuss how to select interest point detectors that work well with specific ground truth data, describe the process you would follow to find the best interest point detectors, and devise some pseudo-code to evaluate and cull interest point detectors.

“More speed, less haste . . .”

—Treebeard, Lord of the Rings

This chapter explores some hypothetical computer vision pipeline designs to understand HW/SW design alternatives and optimizations. Instead of looking at isolated computer vision algorithms, this chapter ties together many concepts into complete vision pipelines. Vision pipelines are sketched out for a few example applications to illustrate the use of different methods. Example applications include object recognition using shape and color for automobiles, face detection and emotion detection using local features, image classification using global features, and augmented reality. The examples have been chosen to illustrate the use of different families of feature description metrics within the *Vision Metrics Taxonomy* presented in Chap. 5. Alternative optimizations at each stage of the vision pipeline are explored. For example, we consider which vision algorithms run better on a CPU versus a GPU, and discuss how data transfer time between compute units and memory affects performance.

Note This chapter does not address optimizations for the training stage or the classification stage. Instead, we focus here on the vision pipeline stages prior to classification. Hypothetical examples in this chapter are sometimes sketchy, not intended to be complete. Rather, the intention is to explore design alternatives. Design choices are made in the examples *for illustration only*; other, equally valid or even better design choices could be made to build working systems. The reader is encouraged to analyze the examples to find weaknesses and alternatives. If the reader can improve the examples, we have succeeded.

This chapter addresses the following major topics, in this order:

1. General design concepts for optimization across the SOC (CPU, GPU, memory).
2. Four hypothetical vision pipeline designs using different descriptor methods.
3. Overview of SW optimization resources and specific optimization techniques.

**NOTE: we do not discuss DNN-specific optimizations here (see Chaps. 9 and 10), and we do not discuss special-purpose vision processors here. For more information on the latest vision processors, contact the Embedded Vision Alliance.*

Stages, Operations, and Resources

A computer vision solution can be implemented into a pipeline of *stages*, as shown in Fig. 8.1. In a pipeline, both parallel and sequential operations take place simultaneously. By using all available compute resources in the optimal manner, performance can be maximized for speed, power, and memory efficiency.

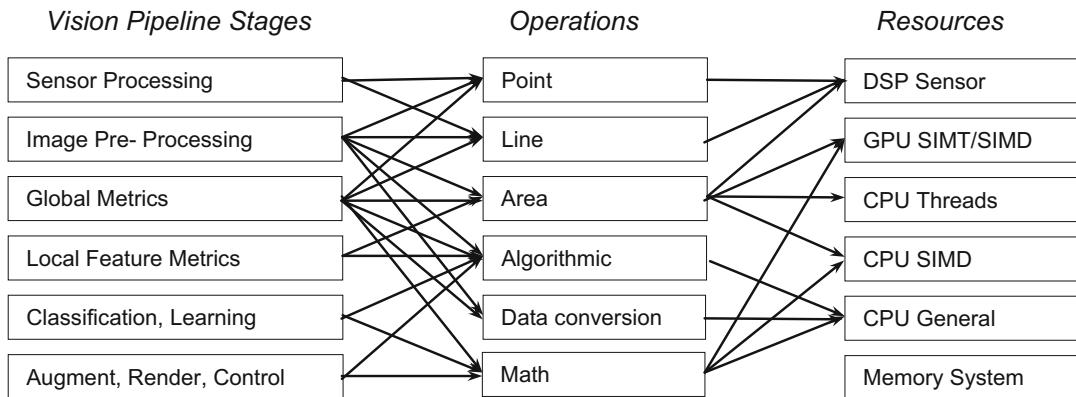


Figure 8.1 Hypothetical assignment of vision pipeline stages to operations and to compute resources. Depending on the actual resource capabilities and optimization targets for power and performance, the assignments will vary

Optimization approaches vary by system. For example, a low-power system for a mobile phone may not have a rich CPU SIMD instruction set, and the GPU may have a very limited thread count and low memory bandwidth, unsuitable to generic GPGPU processing for vision pipelines. However, a larger compute device, such as a rack-mounted compute server, may have several CPUs and GPUs, and each CPU and GPU will have powerful SIMD instructions and high memory bandwidth.

Table 8.1 provides more details on possible assignment of operations to resources based on data types and processor capabilities. For example, in the sensor processing stage, point line and area operations dominate the workload, as sensor data is assembled into pixels and corrections are applied. Most sensor processors are based on a digital signal processor (DSP) with wide SIMD instruction words, and the DSP may also contain a fixed-function geometric correction unit or warp unit for correcting optics problems like lens distortion. The Sensor DSP and the GPU listed in Table 8.1 typically contain a dedicated texture sampler unit, which is capable of rapid pixel interpolation, geometric warps, and affine and perspective transforms. If code is straight line with lots of branching and not much parallel operations, the CPU is the best choice.

Table 8.1 Hypothetical assignment of basic operations to compute resources guided by data type and parallelism (see also Zinner [477])

Operations	Hypothetical Resources and Data Types					
	DSP	GPU SIMT/SIMD	CPU Threads	CPU SIMD	CPU General	Memory System DMA
	<i>uint16</i> <i>int16</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>	
	<i>WarpUnit</i>	<i>TextureUnit</i>				
Point	x	x		x		
Line	x	x		x		
Area	x	x	x (tiles)	x		
Algorithmic Branching					x	
General Math					x	
Data Copy & Conversions						x (<i>DMA preferred</i>)

As illustrated in Table 8.1, the data type and data layout normally guides the selection of the best compute resource for a given task, along with the type of parallelism in the algorithm and data. Also, the programming language is chosen based on the parallelism, such as using OpenCL vs. C++. For example, a CPU may support float and double data types, but if the underlying code is SIMT and SIMD parallel oriented, calling for many concurrent thread-parallel kernel operations, then a GPU with a high thread count may be a better choice than a single CPU. However, running a language like OpenCL on multiple CPUs may provide performance as good as a smaller GPU; for performance information, see reference [526] and vendor information on OpenCL compilers. See also the section later in this chapter, “SIMD, SIMT, and SPMD Fundamentals.”

For an excellent discussion of how to optimize fundamental image processing operations across different compute units and memory, see the PfeLib work by Zinner et al. [477], which provides a deep dive into the types of optimizations that can be made based on data types and intelligent memory usage.

To make the assignments from vision processing stages to operations and compute resources concrete, we look at specific vision pipelines examples later in this chapter.

Compute Resource Budgets

Prior to implementing a vision pipeline, a reasonable attempt should be made to count the cost in terms of the compute platform resources available, and determine if the application is matched to the resources. For example, a system intended for a military battlefield may place a priority on compute speed and accuracy, while an application for a mobile device will prioritize power in terms of battery life and make trade-offs with performance and accuracy.

Since most computer vision research is concerned with breaking ground in handling relatively narrow and well-defined problems, there is limited research available to guide a general engineering discussion on vision pipeline analysis and optimizations. Instead, we follow a line of thinking that starts with the hardware resources themselves, and we discuss performance, power, memory, and I/O requirements, with some references to the literature for parallel programming and other

code-optimization methods. Future research into automated tools to measure algorithm intensity, such as the number of integer and float operations, the bit precision of data types, and the number of memory transfers for each algorithm in terms of read/write, would be welcomed by engineers for vision pipeline analysis and optimizations.

As shown in Fig. 8.2, the main elements of a computer system are composed of I/O, compute, and memory.

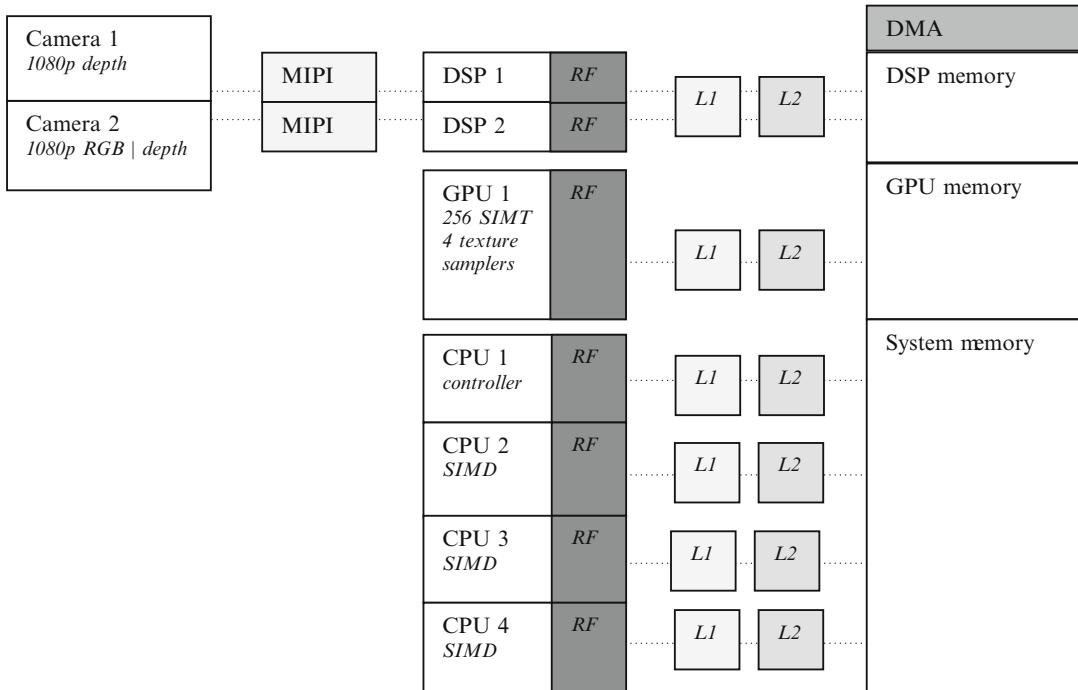


Figure 8.2 Hypothetical computer system, highlighting compute elements in the form of a DSP, GPU, 4 CPU cores, DMA, and memory architecture using L1 and L2 cache and register files RF within each compute unit

We assume suitable high bandwidth I/O busses and cache lines interconnecting the various compute units to memory; in this case, we call out the MIPI camera interface in particular, which connects directly to the DSP in our hypothetical SOC. In the case of a simple computer vision system of the near future, we assume that the price, performance, and power curves continue in the right direction to enable a *system-on-a-chip* (SOC) sufficient for most computer vision applications to be built at a low price point, approaching throw-away computing cost—similar in price to any small portable electronic gadget. This would thereby enable low-power and high-performance ubiquitous vision applications without resorting to special-purpose hardware accelerators built for any specific computer vision algorithms.

Here is a summary description of the SOC components shown in Fig. 8.2:

- **Two 1080p cameras**, one for RGB and the other for a self-contained depth camera, such as a TOF sensor (as discussed in Chap. 1).
- **One small low-power controller CPU** with a reduced instruction set and no floating point, used for handling simple things like the keyboard, accelerometer updates, servicing interrupts from the DSP, and other periodic tasks, such as network interrupt handlers.

- **Three full SIMD capable CPUs** with floating point, used for heavy compute, typically thread parallel algorithms such as tiling, but also for SIMD parallel algorithms.
- **A GPU** capable of running >256 threads with full integer and floating point, and four texture samplers. A wide range of area algorithms map well to the GPU, but the programming model is SIMT kernels such as compute shaders for DirectX and OpenGL, or OpenCL.
- **A DSP** with a limited instruction set and VLIW processing capabilities well suited to pixel processing and sensor processing in general.
- **A DMA unit for fast memory transfers**; although obvious, DMA is a simple and effective method to increase memory bandwidth and reduce power.

Compute Units, ALUs, and Accelerators

There are several types of compute units in a typical system, including CPUs, GPUs, DSPs, and special-purpose hardware accelerators such as cryptography units, texture samplers, and DMA engines. Each ALU has a different instruction set tuned to the intended use, so understanding each compute unit's ALU instruction set is very helpful.

Generally speaking, computer architecture has not advanced to the point of providing any standard vision pipeline methods or hardware accelerators. That is because there are so many algorithm refinements for computer vision emerging; choosing to implement any vision accelerators in silicon is an obsolescence risk. Also, creating computer vision hardware accelerators is difficult, since applications must be portable. So developers typically choose high-level language implementations that are good enough and portable, with minimal dependencies on special purpose hardware or APIs.

Instead, reliance on general-purpose languages like C++ and optimizing the software is a good path to follow to start, as is leveraging existing pixel-processing acceleration methods in a GPU as needed, such as pixel shaders and texture samplers. The standard C++ language path offers flexibility to change and portability across platforms, without relying on any vendor-specific hardware acceleration features.

In the example vision pipelines developed in this section, we make two basic assumptions. First, the DSP is dedicated to sensor processing and light image preprocessing to load-balance the system. Second, the CPUs and the GPUs are used downstream for subsequent sections of the vision pipeline, so the choice of CPU vs. GPU depends on the algorithm used.

Since the compute units with programmable ALUs are typically where all the tools and attention for developers are focused, we dedicate some attention to programming acceleration alternatives later in this chapter in the “Vision Algorithm Optimizations and Tuning” section; there is also a survey of selected optimization resources and software building blocks.

In the hypothetical system shown in Fig. 8.2, the compute units include general-purpose CPUs, a GPU intended primarily for graphics and media acceleration and some GPGPU acceleration, and a DSP for sensor processing. Each compute unit is programmable and contains a general-purpose ALU with a tuned instruction set. For example, a CPU contains all necessary instructions for general programming, and may also contain SIMD instructions discussed later in this chapter. A GPU contains transcendental instructions such as square root, arctangent, and related instructions to accelerate graphics processing. The DSP likewise has an instruction set tuned for sensor processing, likely a VLIW instruction set.

Hardware accelerators are usually built for operations that are common, such as a geometric correction unit for sensor processing in the DSP and texture samplers for warping surface patches in the GPU. There are no standards yet for computer vision, and new algorithm refinements are being

developed constantly, so there is little incentive to add any dedicated silicon for computer vision accelerators, except for embedded and special-purpose systems. Instead, finding creative methods of using existing accelerators may prove beneficial.

Later in this chapter we discuss methods for optimizing software on various compute units, taking advantage of the strengths and intended use of each ALU and instruction set.

Power Use

It is difficult to quantify the amount of power used for a particular algorithm on an SOC or a single compute device without very detailed power analysis; likely simulation is the best method. Typically, systems engineers developing vision pipelines for an SOC do not have accurate methods of measuring power, except crude means such as running the actual finished application and measuring wall power or battery drain.

The question of power is sometimes related to which compute device is used, such as CPU vs. GPU, since each device has a different gate count and clock rate, therefore is burning power at a different rate. Since silicon architects for both GPU and CPU designs are striving to deliver the most *performance per watt per square millimeter*, (and we assume that each set of silicon architects is equally efficient), there is no clear winner in the CPU vs. GPU power/performance race. The search to save power by using the GPU vs. the CPU might not even be worth the effort compared to other places to look, such as data organization and memory architecture.

One approach for making the power and performance trade-off in the case of SIMD and SIMT parallel code is to use a language such as OpenCL, which supports running the same code on either a CPU or a GPU. The performance and power would then need to be measured on each compute device to quantify actual power and performance; there is more discussion on this topic later, in the “Vision Algorithm Optimizations and Tuning” section.

For detailed performance analysis using the same OpenCL code running on a specific CPU vs. a GPU, as well as clusters, see the excellent research by the National Center for Super Computing Applications [526]. Also, see the technical computing resources provided by major OpenCL vendors, such as INTEL, NVIDIA, and AMD, for details on their OpenCL compilers running the same code across the CPU vs. GPU. Sometimes the results are surprising, especially for multi-core CPU systems vs. smaller GPUs.

In general, the compute portion of the vision pipeline is not where the power is burned anyway; most power is burned in the memory subsystem and the I/O fabric, where high data bandwidth is required to keep the compute pipeline elements full and moving along. In fact, all the register files, caches, I/O busses, and main memory consume the lion’s share of power and lots of silicon real estate. So memory use and bandwidth are high-value targets to attack in any attempt to reduce power. The fewer the memory copies, the higher the cache hit rates; the more reuse of the same data in local register files, the better.

Memory Use

Memory is the most important resource to manage as far as power and performance are concerned. Most of the attention on developing a vision pipeline is with the algorithms and processing flow, which is challenging enough. However, vision applications are highly demanding of the memory system. The size of the images alone is not so great, but when we consider the frame rates and number of times a pixel is read or written for kernel operations through the vision pipeline, the memory

transfer bandwidth activity becomes clearer. The memory system is complex, consisting of local register files next to each compute unit, caches, I/O fabric interconnects, and system memory. We look at several memory issues in this section, including:

- Pixel resolution, bit precision, and total image size
- Memory transfer bandwidth in the vision pipeline
- Image formats, including gray scale and color spaces
- Feature descriptor size and type
- Accuracy required for matching and localization
- Feature descriptor database size

To explore memory usage, we go into some detail on a local interest point and feature extraction scenario, assuming that we locate interest points first, filter the interest points against some criteria to select a smaller set, calculate descriptors around the chosen interest points, and then match features against a database.

A reasonable first estimate is that between a lower bound and upper bound of 0.05–1 % of the pixels in an image can generate decent interest points. Of course, this depends entirely on: (1) the complexity of the image texture, and (2) the interest point method used. For example, an image with rich texture and high contrast will generate more interest points than an image of a far away mountain surrounded by clouds with little texture and contrast. Also, interest point detector methods yield different results—for example, the FAST corner method may detect more corners than a SIFT scale invariant DoG feature, see Appendix A.

Descriptor size may be an important variable, see Table 8.2. A 640×480 image will contain 307,200 pixels. We estimate that the upper bound of 1 %, or 3072 pixels, may have decent interests points; and we assume that the lower bound of 0.05 % is 153. We provide a second estimate that interest points may be further filtered to sort out the best ones for a given application. So if we assume perhaps only as few as 33 % of the interest points are actually kept, then we can say that between 153×0.33 and 3072×0.33 interest points are good candidates for feature description. This estimate varies widely out of bounds, depending of course on the image texture, interest point method

Table 8.2 Descriptor bytes per frame (1 % interest points), adapted from [133]

Descriptor	Size in bytes	480p NTSC	1080p HD	2160p 4kUHD	4320p 8kUHD
Resolution		640 x 480	1920 x 1080	3840 x 2160	7680 x 4320
Pixels		307200	2073600	8294400	33177600
BRIEF	32	98304	663552	2654208	10616832
ORB	32	98304	663552	2654208	10616832
BRISK	64	196608	1327104	5308416	21233664
FREAK (4 cascades)	64	196608	1327104	5308416	21233664
SURF	64	196608	1327104	5308416	21233664
SIFT	128	393216	2654208	10616832	42467328
LIOP	144	442368	2985984	11943936	47775744
MROGH	192	589824	3981312	15925248	63700992
MRRID	256	786432	5308416	21233664	84934656
HOG (64x128 block)	3780	n.a.	n.a.	n.a.	n.a.

used, and interest point filtering criteria. Assuming a feature descriptor size is 256 bytes, the total descriptor size per frame is $3072 \times 256 \times 0.33 = 259,523$ bytes maximum—that is not extreme. However, when we consider the feature match stage, the feature descriptor count and memory size will be an issue, since each extracted feature must be matched against each trained feature set in the database.

In general, local binary descriptors offer the advantage of a low memory footprint. For example, Table 8.2 provides the byte count of several descriptors for comparison, as described in Miksik and Mikolajczyk [133]. The data is annotated here to add the descriptor working memory size in bytes per frame for various resolutions.

In Table 8.2, image frame resolutions are in row 1, pixel count per frame is in row 2, and typical descriptor sizes in bytes are in subsequent rows. Total bytes for selected descriptors are in column 1, and the remaining columns show total descriptor size per frame assuming an estimated 1 % of the pixels in each frame are used to calculate an interest point and descriptor. In practice, we estimate that 1 % is an upper-bound estimate for a descriptor count per frame and 0.05 % is a lower-bound estimate. Note that descriptor sizes in bytes do vary from those in the table, based on design optimizations.

Memory bandwidth is often a hidden cost, and often ignored until the very end of the optimization cycle, since developing the algorithms is usually challenging enough without also worrying about the memory access patterns and memory traffic. Table 8.2 includes a summary of several memory variables for various image frame sizes and feature descriptor sizes. For example, using the 1080p image pixel count in row 2 as a base, we see that an RGB image with 16 bits per color channel will consume:

$$2,073,600_{\text{pixels}} \times 3_{\text{channels/RGB}} \times 2_{\text{bytes/pixel}} = 12,441,600 \text{ bytes/frame}$$

And if we include the need to keep a gray scale channel I around, computed from the RGB, the total size for RGBI increases to:

$$2,073,600_{\text{pixels}} \times 4_{\text{channels/RGBI}} \times 2_{\text{bytes/pixel}} = 16,588,800 \text{ bytes/frame}$$

If we then assume 30 frames per second and two RGB cameras for depth processing + the I channel, the memory bandwidth required to move the complete 4-channel RGBI image pair out of the DSP is nearly 1 GB/s:

$$16,588,800_{\text{pixels}} \times 30_{\text{fps}} \times 2_{\text{stereo}} = 995,328,000 \text{ mb/s}$$

So we assume in this example a baseline memory bandwidth of about ~1 GB/s just to move the image pair downstream from the ISP. We are ignoring the ISP memory read/write requirements for sensor processing for now, assuming that clever DSP memory caching, register file design, and loop-unrolling methods in assembler can reduce the memory bandwidth.

Typically, memory coming from a register file in a compute unit transfers in a single clock cycle; memory coming from various cache layers can take maybe tens of clock cycles; and memory coming from system memory can take hundreds of clock cycles. During memory transfers, the ALU in the CPU or GPU may be sitting idle, waiting on memory.

Memory bandwidth is spread across the fast register files next to the ALU processors, and through the memory caches and even system memory, so actual memory bandwidth is quite complex to analyze. Even though some memory bandwidth numbers are provided here, it is only to illustrate the activity.

And the memory bandwidth only increases downstream from the DSP, since each image frame will be read, and possibly rewritten, several times during image preprocessing, then also read again during interest point generation and feature extraction. For example, if we assume only one image preprocessing operation using 5×5 kernels on the I channel, each I pixel is read another 25 times, hopefully from memory cache lines and fast registers.

This memory traffic is not all coming from slow-system memory, and it is mostly occurring inside the faster-memory cache system and faster register files until there is a cache miss or reload of the fast-register files. Then, performance drops by an order of magnitude waiting for the buffer fetch and register reloading. If we add a FAST9 interest point detector on the I channel, each pixel is read another 81 times (9×9), maybe from memory cache lines or registers. And if we add a FREAK feature descriptor over maybe 0.05 % of the detected interest points, we add 41×41 pixel reads per descriptor to get the region (plus 45×2 reads for point-pair comparisons within the 41×41 region), hopefully from memory cache lines or registers.

Often the image will be processed in a variety of formats, such as image preprocessing the RGB colors to enhance the image, and conversion to gray scale intensity I for computing interest points and feature descriptors. The color conversions to and from RGB are a hidden memory cost that requires data copy operations and temporary storage for the color conversion, which is often done in floating point for best accuracy. So several more GB/s of memory bandwidth can be consumed for color conversions. With all the memory activity, there may be cache evictions of all or part of the required images into a slower system memory, degrading into nonlinear performance.

Memory size of the descriptor, therefore, is a consideration throughout the vision pipeline. First, we consider when the features are extracted; and second, we look at when the features are matched and retrieved from the feature database. In many cases, the size of the feature database is by far the critical issue in the area of memory, since the total size of all the descriptors to match against affects the static memory storage size, memory bandwidth, and pattern match rate. Reducing the feature space into a quickly searchable format during classification and training is often of paramount importance. Besides the optimized classification methods discussed in Chap. 4, the data organization problems may be primarily in the areas of standard computer science searching, sorting, and data structures; some discussion and references were provided in Chap. 4.

When we look at the feature database or training set, memory size can be the dominant issue to contend with. Should the entire feature database be kept on a cloud server for matching? Or should the entire feature database be kept on the local device? Should a method of caching portions of the feature database on the local device from the server be used? All of the above methods are currently employed in real systems.

In summary, memory, caches, and register files exceed the silicon area of the ALU processors in the compute units by a large margin. Memory bandwidth across the SOC fabric through the vision pipeline is key to power and performance, demanding fast memory architecture and memory cache arrangement, and careful software design. Memory storage size alone is not the entire picture, though, since each byte needs to be moved around between compute units. So careful consideration of memory footprint and memory bandwidth is critical for anything but small applications.

Often, performance and power can be dramatically improved by careful attention to memory issues alone. Later in the chapter we cover several design methods to help reduce memory bandwidth and increase memory performance, such as locking pages in memory, pipelining code, loop unrolling, and SIMD methods. Future research into minimizing memory traffic in a vision pipeline is a worthwhile field.

I/O Performance

We lump I/O topics together here as a general performance issue, including data bandwidth on the SOC I/O fabric between compute units, image input from the camera, and feature descriptor matching database traffic to a storage device. We touched on I/O issues above the discussion on memory, since pixel data is moved between various compute devices along the vision pipeline on I/O busses. One of the major I/O considerations is feature descriptor data moving out of the database at feature match time, so using smaller descriptors and optimizing the feature space using effective machine learning and classification methods is valuable.

Another type of I/O to consider is the camera input itself, which is typically accomplished via the standard MIPI interface. However, any bus or I/O fabric can be used, such as USB. If the vision pipeline design includes a complete HW/SW system design rather than software only on a standard SOC, special attention to HW I/O subsystem design for the camera and possibly special fast busses for image memory transfers to and from a HW-assisted database may be worthwhile. When considering power, I/O fabric silicon area and power exceed the area and power for the ALU processors by a large margin.

The Vision Pipeline Examples

In this section we look at four hypothetical examples of vision pipelines. Each is chosen to illustrate separate descriptor families from the Vision Metrics Taxonomy presented in Chap. 5, including global methods such as histograms and color matching, local feature methods such as FAST interest points combined with FREAK descriptors, basis space methods such as Fourier descriptors, and shape-based methods using morphology and whole object shape metrics. The examples are broken down into *stages*, *operations*, and *resources*, as shown in Fig. 8.1, for the following applications:

- **Automobile recognition**, using shape and color
- **Face recognition**, using sparse local features
- **Image classification**, using global features
- **Augmented reality**, using depth information and tracking

None of these examples includes classification, training, and machine learning details, which are outside the scope of this book (machine learning references are provided in Chap. 4). A simple database storing the feature descriptors is assumed to be adequate for this discussion, since the focus here is on the image preprocessing and feature description stages. After working through the examples and exploring alternative types of compute resource assignments, such as GPU vs. CPU, this chapter finishes with a discussion on optimization resources and techniques for each type of compute resource.

Automobile Recognition

Here we devised a vision pipeline to recognize objects such as automobiles or machine parts by using *polygon shape descriptors* and *accurate color matching*. For example, polygon shape metrics can be used to measure the length and width of a car, while color matching can be used to measure paint color. In some cases, such as custom car paint jobs, color alone is not sufficient for identification.

For this automobile example, the main design challenges include segmentation of automobiles from the roadway, matching of paint color, and measurement of automobile size and shape. The overall system includes an RGB-D camera system, accurate color and illumination models, and several feature descriptors used in concert. See Fig. 8.3. We work through this example in some detail as a way of exploring the challenges and possible solutions for a complete vision pipeline design of this type.

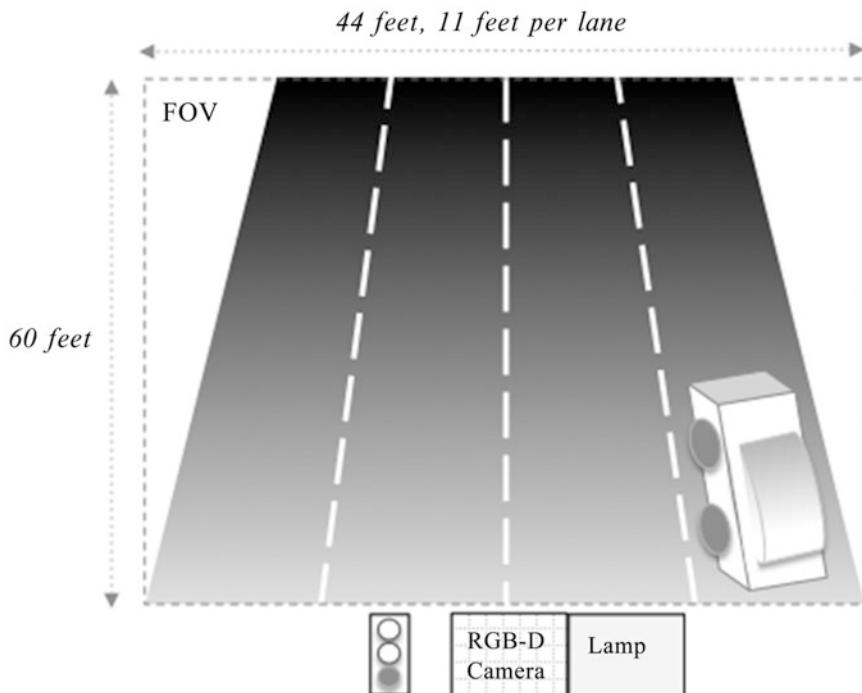


Figure 8.3 Setting for an automobile identification application using a shape-based and color-based vision pipeline. The RGB and D cameras are mounted above the road surface, looking directly down

We define the system with the following requirements:

- 1080p RGB color video (1920×1080 pixels) at 120 fps, horizontally mounted to provide highest resolution in length, 12 bits per color, 65° FOV.
- 1080p stereo depth camera with 8 bits Z resolution at 120 fps, 65° FOV.
- Image FOV covering 44 ft in width and 60 ft in length over four traffic lanes of oncoming traffic, enough for about three normal car lengths in each lane when traffic is stopped.
- Speed limit of 25 mph, which equals ~ 37 ft/s.
- Camera mounted next to overhead stoplight, with a street lamp for night illumination.
- Embedded PC with 4 CPU cores having SIMD instruction sets, 1 GPU, 8 GB memory, 80 GB disk; assumes high-end PC equivalent performance (not specified for brevity).
- Identification of automobiles in real time to determine make and model; also count of occurrences of each, with time stamp and confidence score.
- Automobile ground truth training dataset provided by major manufacturers to include geometry, and accurate color samples of all body colors used for stock models; custom colors and after-market colors not possible to identify.

- Average car sizes ranging from 5 to 6 ft wide and 12 to 16 ft long.
- Accuracy of 99 % or better.
- Simplified robustness criteria to include noise, illumination, and motion blur.

Segmenting the Automobiles

To segment the automobiles from the roadway surface, a stereo depth camera operating at 1080p 120 fps (frames per second) is used, which makes isolating each automobile from the roadway simple using depth. To make this work, a method for calibrating the depth camera to the baseline road surface is developed, allowing automobiles to be identified as being higher than the roadway surface. We sketch out the depth calibration method here for illustration.

Spherical depth differences are observed across the depth map, mostly affecting the edges of the FOV. To correct for the spherical field distortion, each image is rectified using a suitable calibrated depth function (to be determined on-site and analytically), then each horizontal line is processed, taking into consideration the curvilinear true depth distance, which is greater at the edges, to set the depth equal across each line.

Since the speed limit is 25 mph, or 37 ft/s, imaging at 120 FPS yields maximum motion blur of about 0.3 ft, or 4 in. per frame. Since the length of a pixel is determined to be 0.37 inches, as developed in a subsequent section below “Measuring the Automobile Size and Shape”, the ability to compute car length from pixels is accurate within about 4 in./0.37 in. = 11 pixels, or about 3 % of a 12-ft-long car at 25 mph including motion blur. However, motion blur compensation can be applied during image preprocessing to each RGB and depth image to effectively reduce the motion blur further; several methods exist based on using convolution or compensating over multiple sequential images [297, 474].

Matching the Paint Color

We assume that it is possible to identify a vehicle using paint color alone in many cases, since each manufacturer uses proprietary colors, therefore accurate colorimetry can be employed. For matching paint color, 12 bits per color channel should provide adequate resolution, which is determined in the color match stage using the CIECAM02 model and the *Jch* color space [245]. This requires development of several calibrated device models of the camera with the scene under different illumination conditions, such as full sunlight at different times of day, cloud cover, low light conditions in early morning and at dusk, and nighttime using the illuminator lamp mounted above traffic along with the camera and stop light.

The key to colorimetric accuracy is the device models’ accounting for various lighting conditions. A light sensor to measure color temperature, along with the knowledge of time of day and season of the year, is used to select the correct device models for proper illumination for times of day and seasons of the year. However, dirty cars present problems for color matching; for now we ignore this detail (also custom paint jobs are a problem). In some cases, the color descriptor may not be useful or reliable; in other cases, color alone may be sufficient to identify the automobile. See the discussion of color management in Chap. 2.

Measuring the Automobile Size and Shape

For automobile size and shape, the best measurements are taken looking directly down on the car to reduce perspective distortion. As shown in Fig. 8.4, the car is segmented into C (cargo), T (top), and H (hood) regions using depth information from the stereo camera, in combination with a polygon shape segmentation of the auto shape. To compute shape, some weighted combination of RGB and D images into a single image will be used, based on best results during testing. We assume the camera is mounted in the best possible location centered above all lanes, but that some perspective distortion

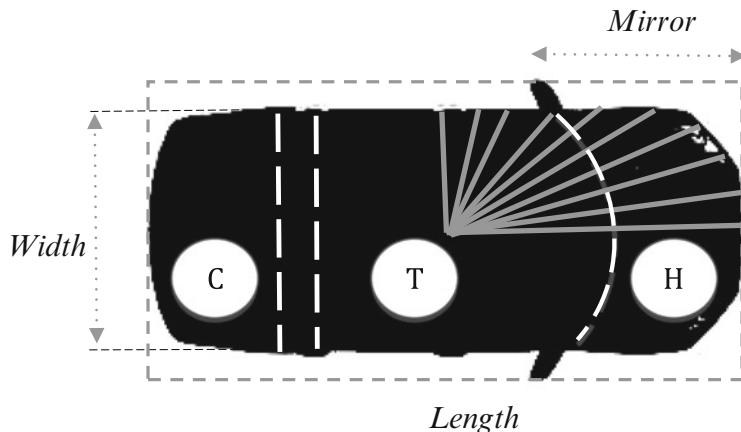


Figure 8.4 Features used for automobile identification

will exist at the far ends of the FOV. We also assume that a geometric correction is applied to rectify the images into Cartesian alignment. Assuming errors introduced by the geometric corrections to rectify the FOV are negligible, the following approximate dimensional precision is expected for length and width, using the minimum car size of $5' \times 12'$ as an example:

FOV Pixel Width: $1080_{\text{pixels}}/(44' \times 12'')_{\text{inches}}$ = each pixel is ~ 0.49 in. wide

FOV Pixel Length: $1920_{\text{pixels}}/(60' \times 12'')_{\text{inches}}$ = each pixel is ~ 0.37 in. long

Automobile Width: $(5' \times 12'')/0.49 = \sim 122$ pixels

Automobile Length: $(12' \times 12'')/0.37 = \sim 389$ pixels

This example uses the following shape features:

- Bounding box containing all features; width and length are used
- Centroid computed in the middle of the automobile region
- Separate width computed from the shortest diameter passing through the centroid to the perimeter
- Mirror feature measured as the distance from the front of the car; mirror locations are the smallest and largest perimeter width points within the bounding box
- Shape segmented into three regions using depth; color is measured in each region: cargo compartment (C), top (T), and hood (H)
- Fourier descriptor of the perimeter shape computed by measuring the line segments from centroid to perimeter points at intervals of 5°

Feature Descriptors

Several feature descriptors are used together for identification, and the confidence of the automobile identification is based on a combined score from all descriptors. The key feature descriptors to be extracted are as follows:

- **Automobile shape factors:** Depth-based segmentation of each automobile above the roadway is used for the coarse shape outline. Some morphological processing follows to clean up the edges and remove noise. For each segmented automobile, object shape factors are computed for area,

perimeter, centroid, bounding box, and Fourier descriptors of perimeter shape. The bounding box measures overall width and height, the Fourier descriptor measures the roundness and shape factors; some automobiles are more boxy, some are more curvy. (See Chap. 6 for more information on shape descriptors. See Chap. 1 for more information on depth sensors.) In addition, the distance of the mirrors from the front of the automobile is computed; mirrors are located at width extrema around the object perimeter, corresponding to the width of the bounding box.

- **Automobile region segmentation:** Further segmentation uses a few individual regions of the automobile based on depth, namely the hood, roof, and trunk. A simple histogram is created to gather the depth statistical moments, a clustering algorithm such as K-means is performed to form three major clusters of depth: the roof will be highest, hood and trunk will be next highest, windows will be in between (top region is missing for convertibles, not covered here). The pixel areas of the hood, top, trunk, and windows are used as a descriptor.
- **Automobile color:** The predominant colors of the segmented hood, roof, and trunk regions are used as a color descriptor. The colors are processed in the *Jch* color space, which is part of the CIECAM system yielding high accuracy. The dominant color information is extracted from the color samples and normalized against the illumination model. In the event of multiple paint colors, separate color normalization occurs for each. (See Chap. 3 for more information on colorimetry.)

Calibration, setup, and Ground Truth Data

Several key assumptions are made regarding scene setup, camera calibration, and other corrections; we summarize them here:

- **Roadway depth surface:** Depth camera is calibrated to the road surface as a reference to segment autos above the road surface; a baseline depth map with only the road is calibrated as a reference and used for real-time segmentation.
- **Device models:** Models for each car are created from manufacturer's information, with accurate body shape geometry and color for each make and model. Cars with custom paint confuse this approach; however, the shape descriptor and the car region depth segmentation provide a failsafe option that may be enough to give a good match—only testing will tell for sure.
- **Illumination models:** Models are created for various conditions, such as morning light, daylight, and evening light, for sunny and cloudy days; illumination models are selected based on time of day and year and weather conditions for best matching.
- **Geometric model for correction:** Models of the entire FOV for both the RGB and depth camera are devised, to be applied at each new frame to rectify the image.

Pipeline Stages and Operations

Assuming the system is fully calibrated in advance, the basic real-time processing flow for the complete pipeline is shown in Fig. 8.5, divided into three primary stages of operations. Note that the complete pipeline includes an image preprocessing stage to align the image in the FOV and segment features, a feature description stage to compute shape and color descriptors, and a correspondence stage for feature matching to develop the final automobile label composed of a weighted combination of shape and color features. We assume that a separate database table for each feature in some standard database is fine.

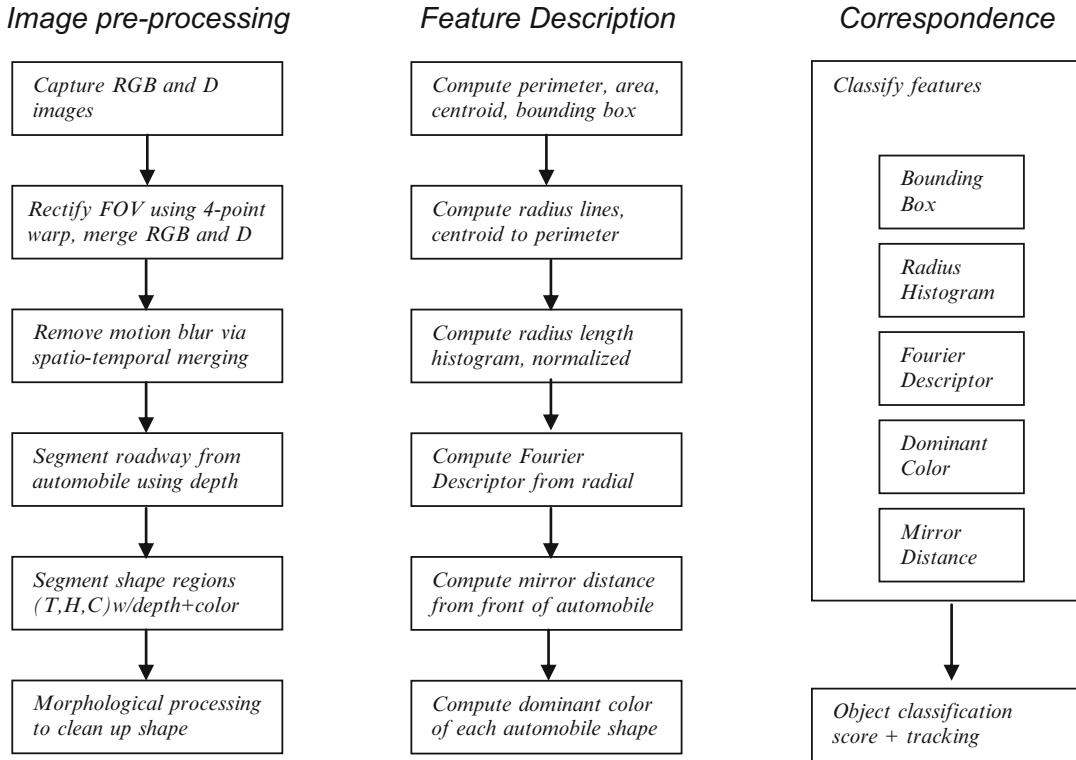


Figure 8.5 Operations in hypothetical vision pipeline for automobile identification using polygon shape features and color

No attempt is made to create an optimized classifier or matching stage here; instead, we assume, without proving or testing, that a brute-force search using a standard database through a few thousand makes and models of automobile objects works fine for the ALPHA version.

Note in Fig. 8.5 (bottom right) that each auto is tracked from frame to frame, we do not define the tracking method here.

Operations and Compute Resources

For each operation in the pipeline stages, we now explore possible mappings to the available compute resources. First, we review the major resources available in our example system, which contains 8 GB of fast memory, we assume sufficient free space to map and lock the entire database in memory to avoid paging. Our system contains four CPU cores, each with SIMD instruction sets, and a GPU capable of running 128 SIMT threads simultaneously with 128 GB/s memory bandwidth to shared memory for the GPU and CPU, considered powerful enough. Let us assume that, overall, the compute and memory resources are fine for our application and no special memory optimizations need to be considered. Next, we look at the coarse-grain optimizations to assign operations to compute resources. Table 8.3 provides an evaluation of possible resource assignments.

Criteria for Resource Assignments

In our simple example, as shown in Table 8.3, the main criteria for assigning algorithms to compute units are processor suitability and load balancing among the processors; power is not an issue for this

Table 8.3 Assignment of operations to compute resources

Operations	Resources and Predominant Data Types				
	DSP sensor VLIW	GPU SIMT/SIMD	CPU Threads	CPU SIMD	CPU General
	<i>uint16</i> <i>int16</i> <i>WarpUnit</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i> <i>TextureUnit</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>
1. Capture RGB-D images	x				
2. 4-point warp image rectify		x		x	
3. Remove motion blur		x			
4. Segment auto, roadway			x		
5. Segment auto shape regions			x		
6. Morphology to clean up shapes		x			
7. Area, perimeter, centroid					x
8. Radius line segments					x
9. Radius histograms			x		
10. Fourier descriptors			x		
11. Mirror distance			x		
12. Dominant region colors			x		
13. Classify features			x		
14. Object classification score					x

application. The operation to resource assignments provided in Table 8.3 are a starting point in this hypothetical design exercise; actual optimizations would be different, adjusted based on performance profiling. However, assuming what is obvious about the memory access patterns used for each algorithm, we can make a good guess at resource assignments based on memory access patterns. In a second-order analysis, we could also look at load balancing across the pipeline to maximize parallel uses of compute units; however, this requires actual performance measurements.

Here we will tentatively assign the tasks from Table 8.3 to resources. If we look at memory access patterns, using the GPU for the sequential tasks 2 and 3 makes sense, since we can map the images into GPU memory space first and then follow with the three sequential operations using the GPU. The GPU has a texture sampler to which we assign task 2, the geometric corrections using the four-point warp. Some DSPs or camera sensor processors also have a texture sampler capable of geometric corrections, but not in our example. In addition to geometric corrections, motion blur is a good candidate for the GPU as well, which can be implemented as an area operation efficiently in a shader. For higher-end GPUs, there may even be hardware acceleration for motion blur compensation in the media section.

Later in the pipeline, after the image has been segmented in tasks 4 and 5, the morphology stage in task 6 can be performed rapidly using a GPU shader; however, the cost of moving the image to and from the GPU for the morphology may actually be slower than performing the morphology on the CPU, so performance analysis is required for making the final design decision regarding CPU vs. GPU implementation.

In the case of stages 7–11, shown in Table 8.3, the algorithm for area, perimeter, centroid, and other measurements span a nonlocalized data access pattern. For example, perimeter tracing follows the edge of the car. So we will make one pass using a single CPU through the image to track the perimeter and compute the area, centroid, and bounding box for each automobile. Then, we assign

each bounding box as an image tile to a separate CPU thread for computation of the remaining measurements: radial line segment length, Fourier descriptor, and mirror distance. Each bounding box is then assigned to a separate CPU thread for computation of the colorimetry of each region, including cargo, roof, and hood, as shown in Table 8.3. Each CPU thread uses C++ for the color conversions and attempts to use compiler flags to force SIMD instruction optimizations.

Tracking the automobile from frame to frame is possible using shape and color features; however, we do not develop the tracking algorithm here. For correspondence and matching, we rely on a generic database from a third party, running in a separate thread on a CPU that is executing in parallel with the earlier stages of the pipeline. We assume that the database can split its own work into parallel threads. However, an optimization phase could rewrite and create a better database and classifier, using parallel threads to match feature descriptors.

Face, Emotion, and Age Recognition

In this example, we design a face, emotion, and age recognition pipeline that uses local feature descriptors and interest points. Face recognition is concerned with identifying the unique face of a unique person, while face detection is concerned with determining only where a face is located and interesting characteristics such as emotion, age, and gender. Our example is for face detection, and finding the emotions and age of the subject.

For simplicity, this example uses mugshots of single faces taken with a stationary camera for biometric identification to access a secure area. Using mugshots simplifies the example considerably, since there is no requirement to pick out faces in a crowd from many angles and distances. Key design challenges include finding a reliable interest point and feature descriptor method to identify the key facial landmarks, determining emotion and age, and modeling the landmarks in a normalized, relative coordinate system to allow for distance ratios and angles to be computed.

Excellent facial recognition systems for biometric identification have been deployed for several decades that use a wide range of methods, achieving accuracies of close to 100 %. In this exercise, no attempt is made to prove performance or accuracy. We define the system with the following requirements:

- 1080p RGB color video (1920×1080 pixels) at 30 fps, horizontally mounted to provide highest resolution in length, 12 bits per color, 65° FOV, 30 FPS
- Image FOV covers 2 ft in height and 1.5 ft in width, enough for a complete head and top of the shoulder
- Background is a white drop screen for ease of segmentation
- Illumination is positioned in front of and slightly above the subject, to cast faint shadows across the entire face that highlight corners around eyes, lips, and nose
- For each face, the system identifies the following landmarks:
 - Eyes: two eye corners and one center of eye
 - Dominant eye color: in CIECAM02 JCH color coordinates
 - Dominant face color: in CIECAM02 JCH color coordinates
 - Eyebrows: two eyebrow endpoints and one center of eyebrow arc, used for determining emotions
 - Nose: one point on nose tip and two widest points by nostrils, used for determining emotions and gender
 - Lips: two endpoints of lips, two center ridges on upper lip
 - Cheeks: one point for each cheek center

- Chin: one point, bottom point of chin, may be unreliable due to facial hair
- Top of head: one point; may be unreliable due to hairstyle
- Unique facial markings: these could include birthmarks, moles, or scars, and must fall within a bounding box computed around the face region
- A FREAK feature is computed at each detected landmark on the original image
- Accuracy is 99 % or better
- Simplified robustness criteria to include scale only

Note that emotion, age, and gender can all be estimated from selected relative distances and proportional ratios of facial features, and we assume that an expert in human face anatomy provides the correct positions and ratios to use for a real system. See Fig. 8.6.

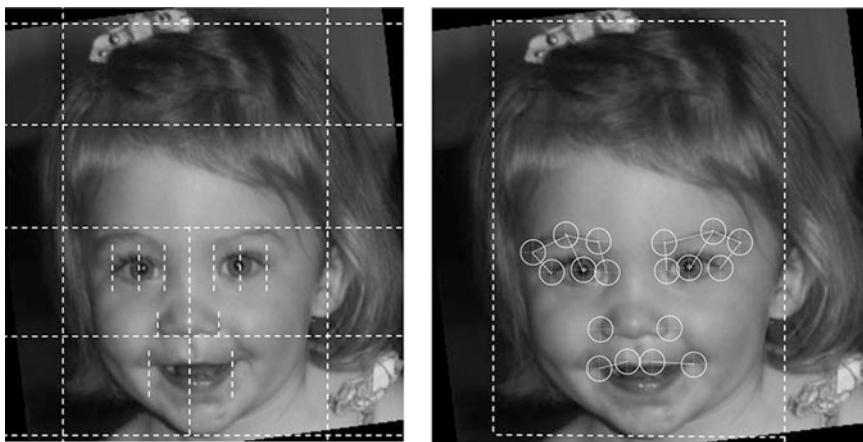


Figure 8.6 (Left) Proportional ratios based on a bounding box of the head and face regions as guidelines to predict the location of facial landmarks. (Right) Annotated image with detected facial landmark positions and relative angles and distances measured between landmarks. The relative measurements are used to determine emotion, age, and gender

The set of features computed for this example system includes:

1. Relative positions of facial landmarks such as eyes, eyebrows, nose, and mouth
2. Relative proportions and ratios between landmarks to determine age, sex, and emotion
3. FREAK descriptor at each landmark
4. Eye color

Calibration and Ground Truth Data

The calibration is simple: a white backdrop is used in back of the subject, who stands about 4 ft away from the camera, enabling a shot of the head and upper shoulders. (We discuss the operations used to segment the head from the background region later in this section.) Given that we have a 1080p image, we allocate the 1920 pixels to the vertical direction and the 1080 pixels to the horizontal.

Assuming the cameraman is good enough to center the head in the image so that the head occupies about 50 % of the horizontal pixels, and about 50 % of the vertical pixels, we have pixel resolution for the head of ~540 pixels horizontal and ~960 pixels vertical, which is good enough for our application and corresponds to the ratio of head height to width. Since we assume that average

head height is about 9 in. and width as 6 in. across for male and female adults, using our assumptions for a four-foot distance from the camera, we have plenty of pixel accuracy and resolution:

$$9''/(1920_{\text{pixels}} \times 0.5) = 0.009'' \text{ vertical pixel size}$$

$$6''/(1080_{\text{pixels}} \times 0.5) = 0.01'' \text{ horizontal pixel size}$$

The ground truth data consists of: (1) mugshots of known people, and (2) a set of canonical eye landmark features in the form of correlation templates used to assist in locating face landmarks (a sparse codebook of correlation templates). There are two sets of correlation templates: one for *fine features* based on a position found using a Hessian detector, and one for *coarse features* based on a position found using a steerable filter based detector (the fine and coarse detectors are described in more detail later in this example).

Since facial features like eyes and lips are very similar among people, the canonical landmark feature correlation templates provide only rough identification of landmarks and their location. Several templates are provided covering a range of ages and genders for all landmarks, such as eye corners, eyebrow corners, eyebrow peaks, nose corners, nose bottom, lip corners, and lip center region shapes. For sake of brevity, we do not develop the ground truth dataset for correlation templates here, but we assume the process is accomplished using synthetic features created by warping or changing real features and testing them against several real human faces to arrive at the best canonical feature set. The correlation templates are used in the face landmark identification stage, discussed later.

Interest Point Position Prediction

To find the facial landmarks, such as eyes, nose, and mouth, this example application is simplified by using mugshots, making the position of facial features predictable and enabling intelligent search for each feature at the predicted locations. Rather than resort to scientific studies of head sizes and shapes, for this example we use basic proportional assumptions from human anatomy (used for centuries by artists) to predict facial feature locations and enable search for facial features at predicted locations. Facial feature ratios differ primarily by age, gender, and race; for example, typical adult male ratios are shown in Table 8.4.

Table 8.4 Basic approximate face and head feature proportions

Head height	head width X 1.25
Head width	head height X .75
Face height	head height X.75
Face width	head height X.75
Eye position	eye center located 30% in from edges, 50% from top of head
Eye length	head width X .25
Eye spacing	head width X .5 (center to center)
Nose length	head height X .25
Lip corners	about eye center x, about 15% higher than chin y

Note The information in Table 8.4 is synthesized for illustration purposes from elementary artists' materials and is not guaranteed to be accurate.

The most basic coordinates to establish are the bounding box for the head. From the bounding box, other landmark facial feature positions can be predicted.

Segmenting the Head and Face Using the Bounding Box

As stated earlier, the mugshots are taken from a distance of about 4 ft against a white drop background, allowing simple segmentation of the head. We use thresholding on simple color intensity as $RGBI-I$, where $I = (R + G + B)/3$ and the white drop background is identified as the highest intensity.

The segmented head and shoulder region is used to create a bounding box of the head and face, discussed next. (Note: wild hairstyles will require another method, perhaps based on relative sizes and positions of facial features compared to head shape and proportions.) After segmenting the bounding box for the head, we proceed to segment the facial region and then find each landmark. The rough size of the bounding box for head is computed in two steps:

1. Find the top and left, right sides of the head— Top_{xy} , Left_{xy} , Right_{xy} —which we assume can be directly found by making a pass through the image line by line and recording the rows and columns where the background is segmented to meet the foreground of head, to establish the coordinates. All leftmost and rightmost coordinates for each line can be saved in a vector, and sorted to find the median values to use as $\text{Right}_x/\text{Left}_x$ coordinates. We compute head width as:

$$H_w = \text{Right}_x - \text{Left}_x$$

2. Find the chin to assist in computing the head height H_h . The chin is found by first predicting the location of the chin, then performing edge detection and some filtering around the predicted location to establish the chin feature, which we assume is simple to find based on gradient magnitude of the chin perimeter. The chin location prediction is made by using the head top coordinates Top_{xy} and the normal anatomical ratio of the head height H_h to head width H_w , which is known to be about 0.75. Since we know both Top_{xy} and H_w from step 1, we can predict the x and y coordinates of the chin as follows:

$$\text{Chin}_y = (0.25 \times H_w) + \text{Top}_y$$

$$\text{Chin}_x = \text{Top}_x$$

Actually, hair style makes the segmentation of the head difficult in some cases, since the hair may be piled high on top or extend widely on the sides and cover the ears. However, we can either iterate the chin detection method a few times to find the best chin, or else assume that our segmentation method will solve this problem somehow via a hair filter module, so we move on with this example for the sake of brevity.

To locate the chin position, a horizontal edge detection mask is used around the predicted location, since the chin is predominantly a horizontal edge. The coordinates of the connected horizontal edge maxima are filtered to find the lowest y coordinates of the horizontal edge set, and the median of the lowest x/y coordinates is used as the initial guess at the chin center location. Later, when the eye positions are known, the chin x position can be sanity-checked with the position of the midpoint between the eyes and recomputed, if needed. See Fig. 8.7.



Figure 8.7 Location of facial landmarks. (*Left*) Facial landmarks enhanced using largest eigenvalues of Hessian tensor [475] in FeatureJ; note the fine edges that provide extra detail. (*Center*) Template-based feature detector using steerable filters with additional filtering along the lines of the Canny detector [382] to provide coarse detail. (*Right*) Steerable filter pattern used to compute center image. Both images are enhanced using contrast window remapping to highlight the edges. FeatureJ plug-in for ImageJ used to generate eigenvalues of Hessian (FeatureJ developed by Erik Meijering)

The head bounding box, containing the face, is assumed to be:

```
BoundingBoxTopLeftx = Leftx
BoundingBoxTopLefty = Topy
BoundingBoxBottomRightx = Rightx
BoundingBoxBottomRighty = Chiny
```

Face Landmark Identification and Compute Features

Now that the head bounding box is computed, the locations of the face landmark feature set can be predicted using the basic proportional estimates from Table 8.4. A search is made around each predicted location to find the features; see Fig. 8.6. For example, the eye center locations are ~30 % in from the sides and about 50 % down from the top of the head.

In our system we use an image pyramid with two levels for feature searching, a coarse-level search down-sampled by four times, and a fine-level search at full resolution to relocate the interest points, compute the feature descriptors, and take the measurements. The coarse-to-fine approach allows for wide variation in the relative size of the head to account for mild scale invariance owing to distance from the camera and/or differences in head size owing to age.

We do not add a step here to rotate the head orthogonal to the Cartesian coordinates in case the head is tilted; however, this could be done easily. For example, an iterative procedure can be used to minimize the width of the orthogonal bounding box, using several rotations of the image taken every 2° from -10 to +10°. The bounding box is computed for each rotation, and the smallest bounding box width is taken to find the angle used to correct the image for head tilt.

In addition, we do not add a step here to compute the surface texture of the skin, useful for age detection to find wrinkles, which is easily accomplished by segmenting several skin regions, such as forehead, eye corners, and the region around mouth, and computing the surface texture (wrinkles) using an edge or texture metric.

The landmark detection steps include feature detection, feature description, and computing relative measurements of the positions and angles between landmarks, as follows:

1. Compute interest points: Prior to searching for the facial features, interest point detectors are used to compute likely candidate positions around predicted locations. Here we use a combination of two detectors: (1) the largest eigenvalue of the Hessian tensor [475], and (2) steerable filters [370] processed with an edge detection filter criteria similar to the Canny method [382], as illustrated in Fig. 8.7. Both the Hessian and the Canny-like edge detectors images are followed by contrast windowing to enhance the edge detail. The Hessian style and Canny-style images are used together to vote on the actual location of best interest points during the correlation stage next.
2. Compute landmark positions using correlation: The final position of each facial landmark feature is determined using a canonical set of correlation templates, described earlier, including eye corners, eyebrow corners, eyebrow peaks, nose corners, nose bottom, lip corners, and lip center region shapes. The predicted location to start the correlation search is the average position of both detectors from step 1: (1) The Hessian approach provides fine-feature details, (2) while the steerable filter approach provides coarse-feature details. Testing will determine if correlation alone is sufficient without needing interest points from step 1.
3. Describe landmarks using FREAK descriptors: For each landmark location found in step 2, we compute a FREAK descriptor. SIFT may work just as well.
4. Measure dominant eye color using CIECAM02 JCH: We use a super-pixel method [249, 250] to segment out the regions of color around the center of the eye, and make a histogram of the colors of the super-pixel cells. The black pupil and the white of the eye should cluster as peaks in the histogram, and the dominant color of the eye should cluster in the histogram also. Even multicolored eyes will be recognized using our approach using histogram correspondence.
5. Compute relative positions and angles between landmarks: In step 2 above, correlation was used to find the location of each feature (to sub-pixel accuracy if desired [450]). As illustrated in Fig. 8.6, we use the landmark positions as the basis for measuring the relative distances of several features, such as:
 - (a) Eye distance, center to center, useful for age and gender
 - (b) Eye size, corner to corner
 - (c) Eyebrow angle, end to center, useful for emotion
 - (d) Eyebrow to eye angle, ends to center positions, useful for emotion
 - (e) Eyebrow distance to eye center, useful for emotion
 - (f) Lip or mouth width
 - (g) Center lip ridges angle with lip corners, useful for emotion

Pipeline Stages and Operations

The pipeline stages and operations are shown in Fig. 8.8. For correspondence, we assume a separate database table for each feature. We are not interested in creating an optimized classifier to speed up pattern matching; brute-force searching is fine.

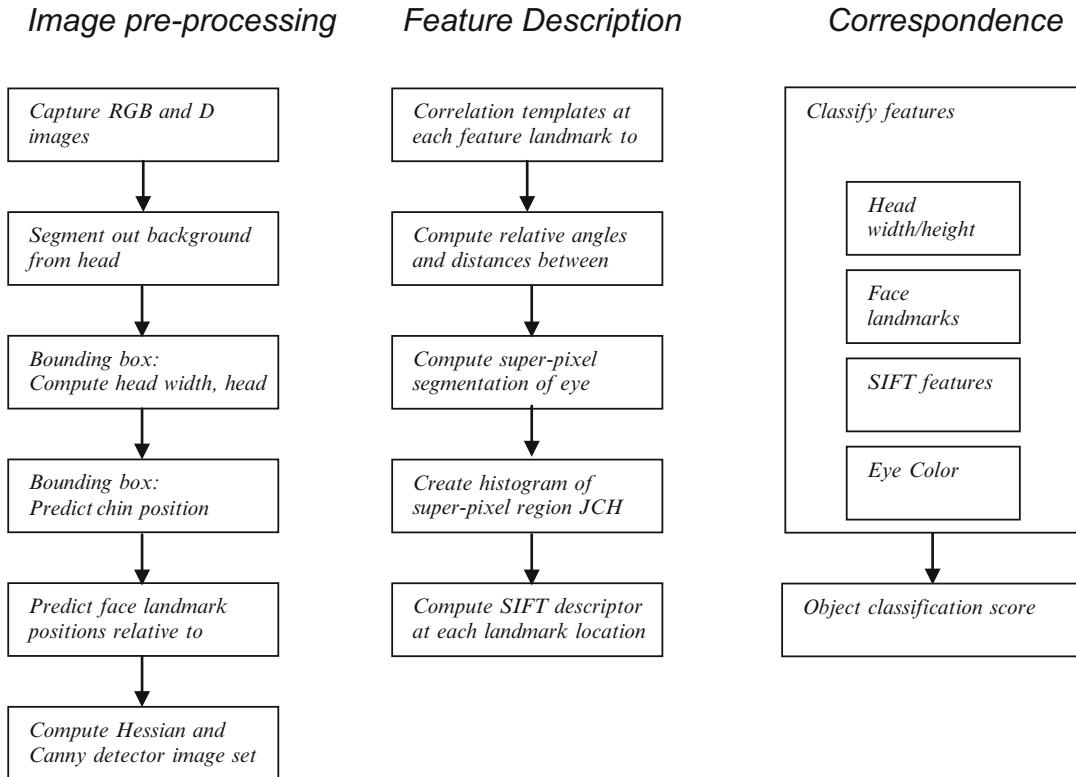


Figure 8.8 Operations in hypothetical vision pipeline for face, emotion, and age detection using local features

Operations and Compute Resources

For this example, there is mostly straight-line code best suited for the CPU. Following the data access patterns as a guide, the bounding box, relative distances and ratios, FREAK descriptors and correspondence are good candidates for the CPU. In some cases, separate CPU threads can be used, such as computing the FREAK descriptors at each landmark in separate threads (threads are likely overkill for this simple application). We assume feature matching using a standard database. Our application is assumed to have plenty of time to wait for correspondence.

Some operations are suited for a GPU; for example the area operations, including the Hessian and Canny-like interest point detectors. These methods could be combined and optimized into a single shader program using a single common data read loop and combined processing loop, which produce output into two images, one for each detector. In addition, we assume that the GPU provides an API to a fast, HW accelerated correlation block matcher in the media section, so we take advantage of the HW accelerated correlation.

Criteria for Resource Assignments

In this example, performance is not a problem, so the criteria for using computer resources are relaxed. In fact, all the code could be written to run in a single thread on a single CPU, and the performance would likely be fast enough with our target system assumptions. However, the resource assignments shown in Table 8.5 are intended to illustrate reasonable use of the resources for each operation to spread the workload around the SOC.

Table 8.5 Assignments of operations to compute resources

Operations	Resources and Predominant Data Types				
	DSP sensor VLIW	GPU SIMT/SIMD	CPU Threads	CPU SIMD	CPU General
	<i>uint16</i> <i>int16</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>	<i>uint16/32</i> <i>int16/32</i> <i>float/double</i>
1. Capture RGB-D images	x				
2. Segment background from head					x
3. Bounding box					x
4. Compute Hessian and Canny		x			
5. Correlation		x			
6. Compute relative angles, distance			x		
7. Super-pixel eye segmentation					x
8. Eye segment color histogram					x
9. FREAK descriptors			x		
10. Correspondence					x
11. Object classification score					x

Image Classification

For our next example, we design a simple image classification system intended for mobile phone use, with the goal of identifying the main objects in the camera's field of view, such as buildings, automobiles, and people. For image classification applications, the entire image is of interest, rather than specific local features. The user will have a simple app which allows them to point the camera at an object, and wave the camera from side to side to establish the stereo baseline for MVS depth sensing, discussed later. A wide range of global metrics can be applied (as discussed in Chap. 3), computed over the entire image, such as texture, histograms of color or intensity, and methods for connected component labeling. Also, local features (as discussed in Chap. 6) can be applied to describe key parts of the images. This hypothetical application uses both global and local features.

We define the system with the following requirements:

- 1080p RGB color video (1920×1080 pixels) at 30 fps, 12 bits per color, 65° FOV, 30 FPS
- Image FOV covers infinite focus view from a mobile phone camera
- Unlimited lighting conditions (bad and good)
- Accuracy of 90 % or better
- Simplified robustness criteria, including scale, perspective, occlusion
- For each image, the system computes the following features:
 - *Global RGBI histogram*, in RGB-I color space
 - *GPS coordinates*, since the phone has a GPS
 - *Camera pose via MVS depth sensing*, using the accelerometer data for geometric rectification to an orthogonal FOV plane (the user is asked to wave the camera while pointed at the subject, the camera pose vector is computed from the accelerometer data and relative to the main objects in the FOV using ICP)

- *SIFT features*, ideally between 20 and 30 features stored for each image
- *Depth map via monocular dense depth sensing*, used to segment out objects in the FOV, depth range target 0.3–30 m, accuracy within 1 % at 1 m, and within 10 % at 30 m
- *Scene labeling and pixel labeling*, based on attributes of segmented regions, including RGB-I color and LBP texture

Scene recognition is a well-researched field, and several grand challenge competitions are held annually to find methods for increased accuracy using established ground truth datasets, as shown in Appendix B. The best accuracy achieved for various categories of images in the challenges ranges from 50 to over 90 %. In this exercise, no attempt is made to prove performance or accuracy.

Segmenting Images and Feature Descriptors

For this hypothetical vision pipeline, several methods for segmenting the scene into objects will be used together, instead of relying on a single method, as follows:

1. **Dense segmentation, scene parsing, and object labeling:** A depth map generated using monocular MVS is used to segment common items in the scene, including the ground or floor, sky or ceiling, left and right walls, background, and subjects in the scene. To compute monocular depth from the mobile phone device, the user is prompted by the application to move the camera from left to right over a range of arm's length covering 3 ft or so, to create a series of wide baseline stereo images for computing depth using MVS methods (as discussed in Chap. 1). MVS provides a dense depth map. Even though MVS computation is compute-intensive, this is not a problem, since our application does not require continuous real-time depth map generation—just a single depth map; 3–4 s to acquire the baseline images and generate the depth map is assumed possible for our hypothetical mobile device.
2. **Color segmentation and component labeling using super-pixels:** The color segmentation using super-pixels should correspond roughly with portions of the depth segmentation.
3. **LBP region segmentation:** This method is fairly fast to compute and compact to represent, as discussed in Chap. 6.
4. **Fused segmentation:** The depth, color, and LBP segmentation regions are combined using Boolean masks and morphology and some logic into a fused segmentation. The method uses an iterative loop to minimize the differences between color, depth, and LBP segmentation methods into a new fused segmentation map. The fused segmentation map is one of the global image descriptors.
5. **Shape features for each segmented region:** basic shape features, such as area and centroid, are computed for each fused segmentation region. Relative distance and angle between region centroids is also computed into a composite descriptor.

In this hypothetical example, we use several feature descriptor methods together for additional robustness and invariance, and some preprocessing, summarized as follows:

1. SIFT interest points across the entire image are used as additional clues. We follow the SIFT method exactly, since SIFT is known to recognize larger objects using as few as three or four SIFT features [153]. However, we expect to limit the SIFT feature count to 20 or 30 strong candidate features per scene, based on training results.
2. In addition, since we have an accelerometer and GPS sensor data on the mobile phone, we can use sensor data as hints for identifying objects based on location and camera pose alone, for example assuming a server exists to look up the GPS coordinates of landmarks in an area.

3. Since illumination invariance is required, we perform RGBI contrast remapping in an attempt to normalize contrast and color prior to the SIFT feature computations, color histograms, and LBP computations. We assume a statistical method for computing the best intensity remapping limits is used to spread out the total range of color to mitigate dark and oversaturated images, based on ground truth data testing, but we do not take time to develop the algorithm here; however, some discussion on candidate algorithms is provided in Chap. 2. For example, computing SIFT descriptors on dark images may not provide sufficient edge gradient information to compute a good SIFT descriptor, since SIFT requires gradients. Oversaturated images will have washed-out color, preventing good color histograms.
4. The fused segmentation combines the best of all the color, LBP, and depth segmentation methods, minimizing the segmentation differences by fusing all segmentations into a fused segmentation map. LBP is used also, which is less sensitive to both low light and oversaturated conditions, providing some balance.

Again, in the spirit of a hypothetical exercise, we do not take time here to develop the algorithm beyond the basic descriptions given above.

Pipeline Stages and Operations

The pipeline stages are shown in Fig. 8.9. They include an image preprocessing stage primarily to correct image contrast, compute depth maps and segmentation maps. The feature description stage computes the RGBI color histograms, SIFT features, a fused segmentation map combining the best of depth, color, and LBP methods, and then labels the pixels as connected components. For correspondence, we assume a separate database table for each feature, using brute-force search; no optimization attempted.

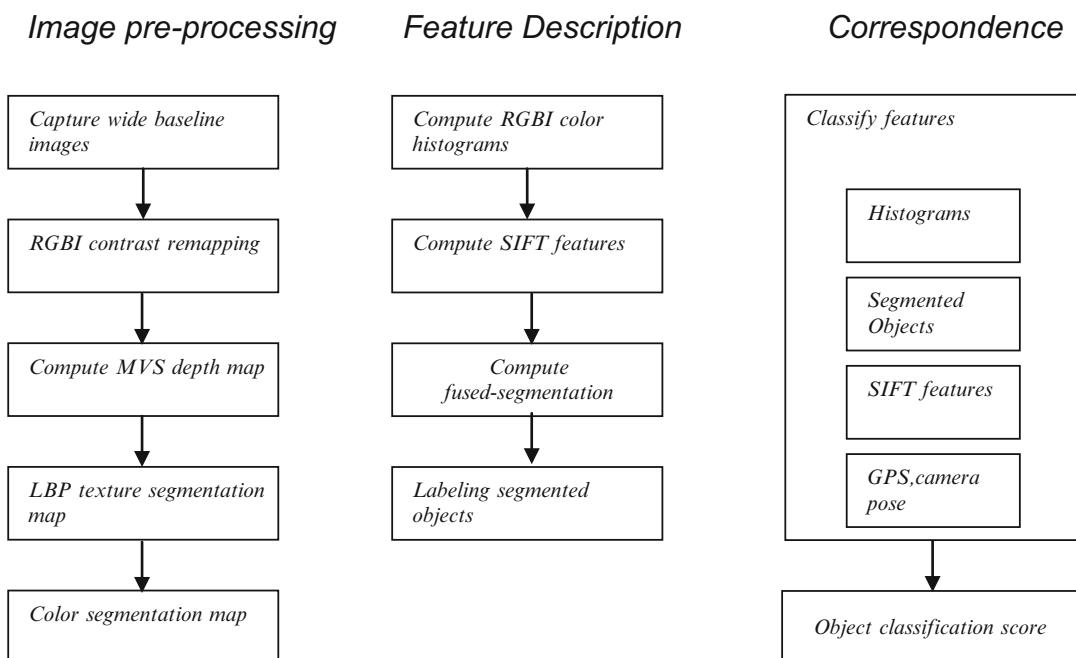


Figure 8.9 Operations in hypothetical image classification pipeline using global features

Mapping Operations to Resources

We assume that the DSP provides an API for contrast remapping, and since the DSP is already processing all the pixels from the sensor anyway and the pixel data is already there, contrast remapping is a good match for the DSP.

The MVS depth map computations follow a data pattern of line and area operations. We use the GPU for the heavy-lifting portions of the MVS algorithm, like left/right image pair pattern matching. Our algorithm follows the basic stereo algorithms, as discussed in Chap. 1. The stereo baseline is estimated initially from the accelerometer, then some bundle adjustment iterations over the baseline image set are used to improve the baseline estimates. We assume that the MVS stereo workload is the heaviest in this pipeline and consumes most of the GPU for a second or two. A dense depth map is produced in the end to use for depth segmentation.

The color segmentation is performed on RGBI components using a super-pixel method [249, 250]. A histogram of the color components is also computed in RGBI for each superpixel cell. The LBP texture computation is a good match for the GPU since it is an area operation amenable to shader programming style. So it is possible to combine the color segmentation and the LBP texture segmentation into the same shader to leverage data sharing in register files and avoid data swapping and data copies.

The SIFT feature description can be assigned to CPU threads, and the data can be tiled and divided among the CPU threads for parallel feature description. Likewise, the fused segmentation can be assigned to CPU threads and the data tiled also. Note that tiled data can include overlapping boundary regions or buffers, see Fig. 8.12 for an illustration of overlapped data tiling. Labeling can also be assigned to parallel CPU threads in a similar manner, using tiled data regions. Finally, we assume a brute-force matching stage using database tables for each descriptor to develop the final score, and we weight some features more than others in the final scoring, based on training against ground truth data.

Criteria for Resource Assignments

The basic criterion for the resource assignments is to perform the early point processing on the DSP, since the data is already resident, and then to use the GPU SIMD model to compute the area operations as shaders to create the depth maps, color segmentation maps, and LBP texture maps. The last stages of the pipeline map nicely to thread parallel methods and data tiling. Given the chosen operation to resource assignments shown in Table 8.6, this application seems cleanly amenable to workload balancing and parallelization across the CPU cores in threads and the GPU.

Augmented Reality

In this fourth example, we design an augmented reality application for equipment maintenance using a wearable display device such as glasses or goggles and wearable cameras. The complete system consists of a portable, wearable device with camera and display connected to a server via wireless. Processing is distributed between the wearable device and the server (*Note:* this example is especially high level and leaves out a lot of detail, since the actual system would be complex to design, train and test.).

The server system contains all the CAD models of the machine and provides on-demand graphics models or renderings of any machine part from any viewpoint. The wearable cameras track the eye gaze and the position of the machine. The wearable display allows a service technician to look at a

Table 8.6 Assignments of operations to compute resources

Operations	Resources and Predominant Data Types				
	DSP sensor VLIW <i>uint16 int16 WarpUnit</i>	GPU SIMT/SIMD <i>uint16/32 int16/32 float/double TextureUnit</i>	CPU Threads <i>uint16/32 int16/32 float/double</i>	CPU SIMD <i>uint16/32 int16/32 float/double</i>	CPU General <i>uint16/32 int16/32 float/double</i>
1. Capture RGB wide baseline images	x				
2. RGBI contrast remapping	x				
3. MVS depth map		x			
4. LBP texture segmentation map		x			
5. Color segmentation map		x			
6. RGBI color histograms			x		
7. SIFT features			x		
8. Fused segmentation			x		
9. Labeling segmented objects			x		
10. Correspondence			x		
11. Object classification score					x

machine and view augmented reality overlays on the display, illustrating how to service the machine. As the user looks at a given machine, the augmented reality features identify the machine parts and provide overlays and animations for assisting in troubleshooting and repair. The system uses a combination of RGB images as textures on 3D depth surfaces and a database of 3D CAD models of the machine and all the component machine parts.

The system will have the following requirements:

- 1080p RGB color video camera (1920×1080 pixels) at 30 fps, 12 bits per color, 65° FOV, 30 FPS
- 1080p stereo depth camera with 8 bits Z resolution at 60 fps, 65° FOV; all stereo processing performed in silicon in the camera ASIC with a depth map as output
- 480p near infra-red camera pointed at eyes of technician, used for gaze detection; the near-infrared camera images better in the low-light environment around the head-mounted display
- 1080p wearable RGB display
- A wearable PC to drive the cameras and display, descriptor generation, and wireless communications with the server; the system is battery powered for mobile use with an 8-h battery life
- A server to contain the CAD models of the machines and parts; each part will have associated descriptors precomputed into the data base; the server can provide either graphics models or complete renderings to the wearable device via wireless
- Server to contain ground truth data consisting of feature descriptors computed on CAD model renderings of each part + normalized 3D coordinates for each descriptor for machine parts
- Simplified robustness criteria include perspective, scale, and rotation

Calibration and Ground Truth Data

We assume that the RGB camera and the stereo camera system are calibrated with correct optics to precisely image the same FOV, since the RGB camera and 3D depth map must correspond at each pixel location to enable 2D features to be accurately associated with the corresponding 3D depth location. However, the eye gaze camera will require some independent calibration, and we assume a simple calibration application is developed to learn the technician's eye positions by using the stereo and RGB cameras to locate a feature in the FOV, and then overlay an eye gaze vector on a monitor to confirm the eye gaze vector accuracy. We do not develop the calibration process here.

However, the ground truth data takes some time to develop and train, and requires experts in repair and design of the machine to work together during training. The ground truth data includes feature sets for each part, consisting of 2D SIFT features along corners, edges, and other locations such as knobs. To create the SIFT features, first a set of graphics renderings of each CAD part model is made from representative viewpoints the technician is likely to see, and then the 2D SIFT features are computed on the graphics renderings, and the geometry of the model is used to create relative 3D coordinates for each SIFT feature for correspondence.

The 2D SIFT feature locations are recorded in the database along with relative 3D coordinates, and associated into objects using suitable constraints such as angles and relative distances, see Fig. 8.10. An expert selects a minimum set of features for each part during training—primarily strongest features from corners and edges of surfaces. The relative angles and distances in three dimensions between the 2D SIFT features are recorded in the database to provide for perspective, scale, and rotation invariance. The 3D coordinates for all the parts are normalized to the size of the machine. In addition, the dominant color and texture of each part surface is computed from the renderings and stored as texture and color features. This system would require considerable training and testing.

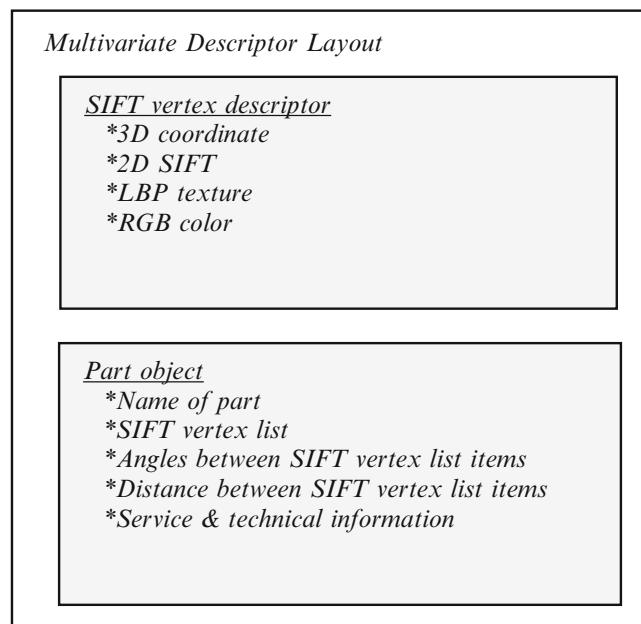


Figure 8.10 SIFT vertex descriptor is similar to a computer graphics vertex using 3D location, color, and texture. The SIFT vertex descriptor contains the 2D SIFT descriptor from the RGB camera, the 3D coordinate of the 2D SIFT descriptor generated from the depth camera, the RGB color at the SIFT vertex, and the LBP texture at the SIFT vertex. The Part object contains a list of SIFT vertex descriptors, along with relative angles and distances between each 3D coordinate in the SIFT vertex list

Feature and Object Description

In actual use in the field, the RGB camera is used to find the 2D SIFT, LBP and color features, and the stereo camera is used to create the depth map. Since the RGB image and depth map are pixel-aligned, each feature has 3D coordinates taken from the depth map, which means that a 3D coordinate can be assigned to a 2D SIFT feature location. The 3D angles and 3D distances between 2D SIFT feature locations are computed as constraints, and the combined LBP, color and 2D SIFT features with 3D location constraints are stored as SIFT vertex features and sent to the server for correspondence. See Fig. 8.10 for an illustration of the layout of the SIFT vertex descriptors and parts objects. Note that the 3D coordinate is associated with several descriptors, including SIFT, LBP texture, and RGB color, similar to the way a 3D vertex is represented in computer graphics by 3D location, color, and texture. During training, several SIFT vertex descriptors are created from various views of the parts, each view associated by 3D coordinates in the database, allowing for simplified searching and matching based on 3D coordinates along with the features.

Overlays and Tracking

In the server, SIFT vertex descriptors in the scene are compared against the database to find parts object. The 3D coordinates, angles, and distances of each feature are normalized relative to the size of the machine prior to searching. As shown in Fig. 8.10, the SIFT features are composed at a 3D coordinate into a SIFT vertex descriptor, with an associated 2D SIFT feature, LBP texture, and color. The SIFT vertex descriptors are associated into part objects, which contain the list of vertex coordinates describing each part, along with the relative angles and distances between SIFT vertex features.

Assuming that the machine part objects can be defined using a small set of SIFT vertex features, sizes and distance can be determined in real time, and the relative 3D information such as size and position of each part and the whole machine can be continually computed. Using 3D coordinates of recognized parts and features, augmented reality renderings can be displayed in the head-mounted display, highlighting part locations and using overlaying animations illustrating the parts to remove, as well as the path for the hand to follow in the repair process.

The near infrared camera tracks the eyes of the technician to create a 3D gaze vector onto the scene. The gaze vector can be used for augmented reality “help” overlays in the head-mounted display, allowing for gaze-directed zoom or information, with more detailed renderings and overlay information displayed for the parts the technician is looking at.

Pipeline Stages and Operations

The pipeline stages are shown in Fig. 8.11. Note that the processing is divided between the wearable device (primarily for image capture, feature description, and display), and a server for heavy workloads, such as correspondence and augmented reality renderings. In this example, the wearable device is used in combination with the server, relying on a wireless network to transfer images and data. We assume that data bandwidth and data compression methods are adequate on the wireless network for all necessary data communications.

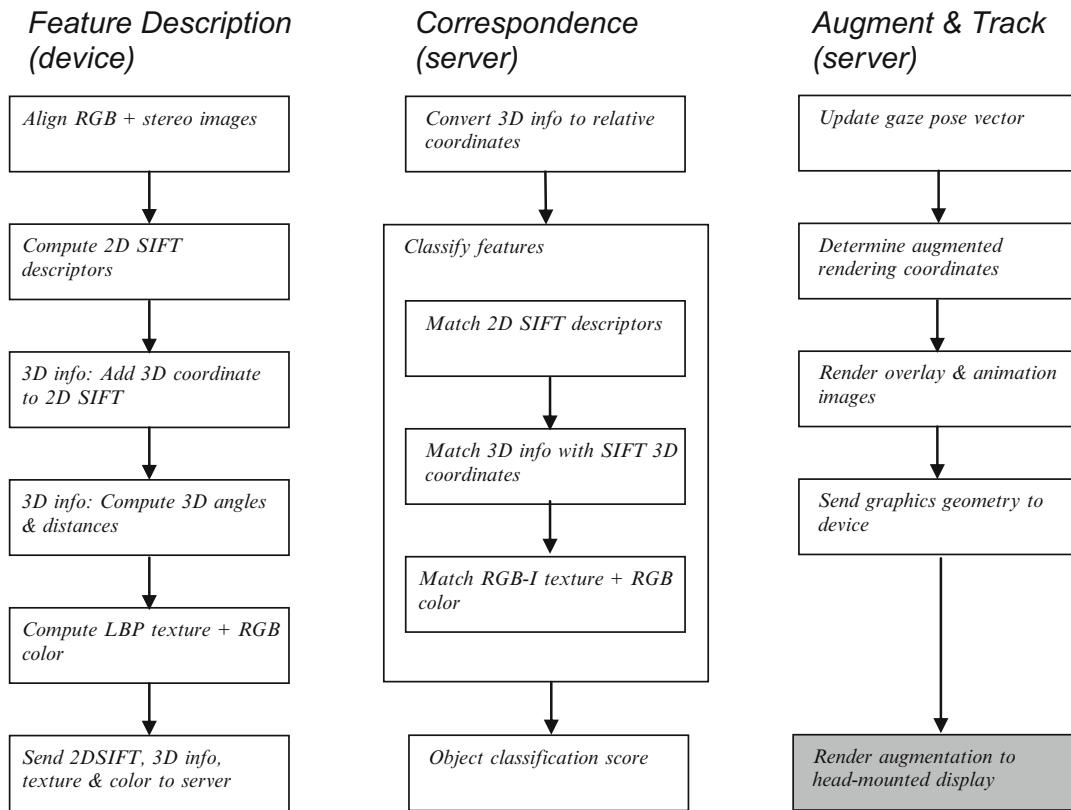


Figure 8.11 Operations in hypothetical augmented reality pipeline

Table 8.7 Assignments of operations to compute resources

Operations	Resources and Predominant Data Types				
	DSP sensor VLIW	GPU SIMT/SIMD	CPU Threads	CPU SIMD	CPU General
1. Capture RGB & stereo images	Device				
2. Align RGB and stereo images		Device			
5. Compute 2D SIFT			Device		
3. Compute LBP texture			Device		
4. Compute color			Device		
5. Compute 2D SIFT			Device		
6. Compute 3D angles/distances			Device		
7. Normalize 3D coordinates					Server
8. Match 2D SIFT descriptors			Server		
9. Match SIFT vertex coordinates			Server		
10. Match SIFT vertex color & LBP			Server		
11. Object classification score					Server
12. Update gaze pose vector					Server
13. Render overlay & animation images		Server			
14. Display overlays & animations		Device *GFX pipe			

Mapping Operations to Resources

We make minimal use of the GPU for GPGPU processing and assume the server has many CPUs available, and we use the GPU for graphics rendering at the end of the pipeline. Most of the operations map well into separate CPU threads using data tiling. Note that a server commonly has many high-power and fast CPUs, so using CPU threads is a good match. See Table 8.7.

Criteria for Resource Assignments

On the mobile device, the depth map is computed in silicon on the depth camera. We use the GPU to perform the RGB and depth map alignment using the texture sampler, then perform SIFT computations on the CPU, since the SIFT computations must be done first to have the vertex to anchor and compute the multivariate descriptor information. We continue and follow data locality and perform the LBP and color computations for each 2D SIFT point in separate CPU threads using data tiling and overlapped regions. See Fig. 8.12 for an illustration of overlapped data tiling.

On the server, we have assigned the CAD database and most of the heavy portions of the workload, including feature matching and database access, since the server is expected to have large storage and memory capacity and many CPUs available. In addition, we wish to preserve battery life and minimize heat on the mobile device, so the server is preferred for the majority of this workload.

Acceleration Alternatives

There are a variety of common acceleration methods that can be applied to the vision pipeline, including attention to memory management, coarse-grained parallelism using threads, data-level parallelism using SIMD and SIMT methods, multi-core parallelism, advanced CPU and GPU assembler language instructions, and hardware accelerators.

There are two fundamental approaches for acceleration:

1. Follow the data
2. Follow the algorithm

Optimizing algorithms for compute devices, such as SIMD instruction sets or SIMT GPGPU methods, also referred to as *stream processing*, is oftentimes the obvious choice designers consider. However, optimizing for data flow and data residency can yield better results. For example, bouncing data back and forth between compute resources and data formats is not a good idea; it eats up time and power consumed by the copy and format conversion operations. Data copying in slow-system memory is much slower than data access in fast-register files within the compute units. Considering the memory architecture hierarchy of memory speeds, as was illustrated in Fig. 8.2, and considering the image-intensive character of computer vision, it is better to find ways to follow the data and keep the data resident in fast registers and cache memory as long as possible, local to the compute unit.

Memory Optimizations

Attention to memory footprint and memory transfer bandwidth are the most often overlooked areas when optimizing an imaging or vision application, yet memory issues are the most critical in terms of

power, bandwidth, silicon area, and overall performance. As shown in Table 8.2 and the memory discussion following, a very basic vision pipeline moves several GB/s of descriptor through the system between compute units and system memory, and DNN's may be an order of magnitude more data intensive. In addition, area processes like interest point detection and image preprocessing move even more data in complex routes through the register files of each compute unit, caches, and system memory.

Why optimize for memory? By optimizing memory use, data transfers are reduced, performance is improved, power costs are reduced, and battery life is increased. Power is costly; in fact, a large Internet search company has built server farms very close to the Columbia River's hydroelectric systems to guarantee clean power and reduce power transmission costs.

For mobile devices, battery life is a top concern. Governments are also beginning to issue carbon taxes and credits to encourage power reductions. Memory use, thus, is a cost that is often overlooked. Memory optimization APIs and approaches will be different for each compute platform and operating system. A good discussion on memory optimization methods for Linux is found in reference [476].

Minimizing Memory Transfers Between Compute Units

Data transfers between compute units should be avoided, if possible. Workload consolidation should be considered during the optimization and tuning stage in order to perform as much processing as possible on the same data while it is resident in register files and the local cache of a given compute unit. That is, follow the data.

For example, using a GPGPU shader for a single-area operation, then processing the same data on the CPU will likely be slower than performing all the processing on the CPU. That is because GPGPU kernels require device driver intervention to set up the memory for each kernel and launch each kernel, while a CPU program accesses code and data directly, with no driver setup required other than initial program loading. One method to reduce the back-and-forth between compute units is to use loop coalescing and task chaining, discussed later in this section.

Memory Tiling

When dividing workloads for coarse-grained parallelism into several threads, the image can be broken into tiled regions and each tile assigned to a thread. Tiling works well for point, line, and area processing, where each thread performs the same operation on the tiled region. By allowing for an overlapped read region between tiles, the hard boundaries are eliminated and area operations like convolution can read into adjacent tiles for kernel processing, as well as write finished results into their tile.

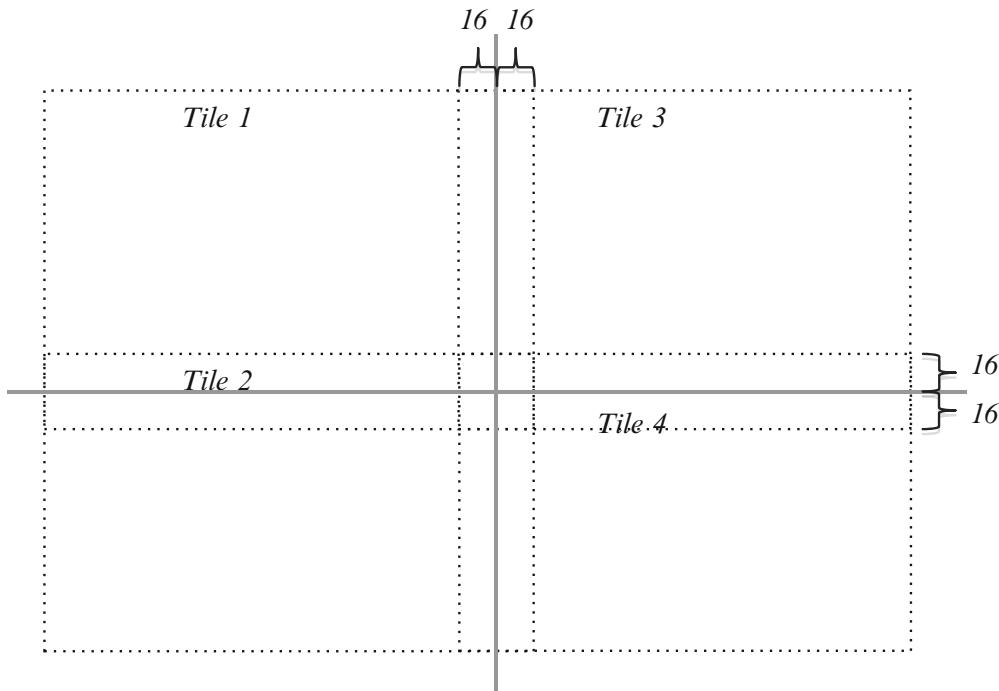


Figure 8.12 Data tiling into four overlapping tiles. The tiles overlap a specific amount, 16 pixels in this case, allowing for area operations such as convolutions to read, not write, into the overlapped region for assembling convolution kernel data from adjacent regions. However, each thread only writes into the nonoverlapped region within its tile. Each tile can be assigned to a separate thread or CPU core for processing

DMA, Data Copy, and Conversions

Often, multiple copies of an image are needed in the vision pipeline, and in some cases, the data must be converted from one type to another. Converting 12-bit unsigned color channel data stored in a 16-bit integer to a 32-bit integer allowing for more accurate numerical precision downstream in computations is one example. Also, the color channels might be converted into a chosen color space, such as RGBI, for color processing in the I component space ($R \times G \times B)/3 = I$; then, the new I value is mixed and copied back into the RGB components. Careful attention to data layout and data residency will allow more efficient forward and backward color conversions.

When copying data, it is good to try using the direct memory access (DMA) unit for the fastest possible data copies. The DMA unit is implemented in hardware to directly optimize and control the I/O interconnect traffic in and out of memory. Operating systems provide APIs to access the DMA unit [476]. There are variations for optimizing the DMA methods, and some interesting reading comparing cache performance against DMA in vision applications are found in references [477, 479].

Register Files, Memory Caching, and Pinning

The memory system is a hierarchy of virtual and physical memories for each processor, composed of slow fixed storage such as file systems, page files, and swap files for managing virtual memory, system memory, caches, and fast-register files inside compute units, and with memory interconnects in between. If the data to process is resident in the register files, it is processed by the ALU at processor clock rates. Best-case memory access is via the register files close to each ALU, so keeping the data in registers and performing all possible processing before copying the data is optimal, but this may require some code changes (discussed later in this section).

If the cache must be accessed to get the data, more clock cycles are burned (power is burned, performance is lost) compared to accessing the register files. And if there is a cache miss and much slower system memory must be accessed, typically many hundreds of clock cycles are required to move the memory to register files through the caches for ALU processing.

Operating systems provide APIs to lock or pin the data in memory, which usually increases the amount of data in cache, decreasing paging and swapping. (Swapping is a hidden copy operation carried out by the operating system automatically to make more room in system memory). When data is accessed often, the data will be resident in the faster cache memories, as was illustrated in Fig. 8.2.

Data Structures, Packing, and Vector vs. Scatter-Gather Data Organization

The data structures used contribute to memory traffic. Data organization should allow serial access in contiguous blocks as much as possible to provide best performance. From the programming perspective, data structures are often designed with convenience in mind, and no attention is given to how the compiler will arrange the data or the resulting performance.

For example, consider a data structure with several fields composed of bytes, integers, and floating point data items; compilers may attempt to rearrange the positions of data items in the data structures, and even pack the data in a different order for various optimizations. Compilers usually provide a set of compiler directives, such as in-line pragmas and compiler switches, to control the data packing behavior; these are worth looking into.

For point processing, vectors of data are the natural structure, and the memory system will operate at peak performance in accessing and processing contiguous vectors. For area operations, rectangles spanning several lines are used, and the rectangles cause memory access patterns that can generate cache misses. Using scatter-gather operations for gathering convolution kernel data allows a large data structure to be split apart into vectors of data, increasing performance. Often, CPU and GPU memory architectures pay special attention to data-access patterns and provide hidden methods for optimizations.

Scatter-gather operations, also referred to as *vectored I/O* or *strided* memory access, can be implemented in the GPU or CPU silicon to allow for rapid read/write access to noncontiguous data structure patterns. Typically, a scatter operation writes multiple input buffers into a contiguous pattern in a single output buffer, and a gather operation analogously reads multiple input buffers into a contiguous pattern in the output buffer.

Operating systems and compute languages provide APIs for scatter-gather operations. For Linux-style operating systems, see the *readv* and *writev* function specified in the POSIX 1003.1-2001 specification. The *async_work_group_strided_copy* function is provided by OpenCL for scatter-gather.

Coarse-Grain Parallelism

A vision pipeline can be implemented using coarse-grain parallelism by breaking up the work into threads, and also by assigning work to multiple processor cores. Coarse-grained parallelism can be achieved by breaking up the compute workload into pipelines of threads, or by breaking up the memory into tiles assigned to multiple threads.

Compute-Centric vs. Data-Centric

Coarse-grain parallelism can be employed via compute-centric and data-centric approaches. For example, in a *compute-centric* approach, vision pipeline stages can be split among independent execution threads and compute units along the lines of pipeline stages, and data is fed into the next stage a little at a time via queues and FIFOs. In a *data-centric* approach, an image can be split into tiles, as was shown in Fig. 8.12, and each thread processes an independent tile region.

Threads and Multiple Cores

Several methods exist to spread threads across multiple CPU cores, including reliance on the operating system scheduler to make optimum use of each CPU core and perform load balancing. Another is by assigning specific tasks to specific CPU cores. Each operating system has different controls available to tune the process scheduler for each thread, and also may provide the capability to assign specific threads to specific processors. (We discuss programming resources, languages and tools for coarse-grained threading later in this chapter.) Each operating system will provide an API for threading, such as *pthreads*. See Fig. 8.13.

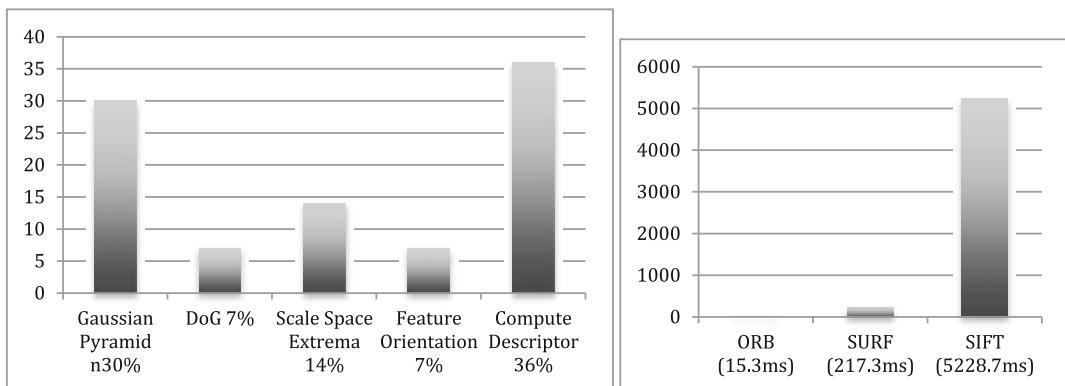


Figure 8.13 (Left) Typical SIFT descriptor pipeline compute allocation [172]. (Right) Reported compute times [112] for ORB, SURF, and SIFT, averaged over twenty-four 640×480 images containing about 1000 features per image. Retrofitting ORB for SIFT may be a good choice in some applications

Fine-Grain Data Parallelism

Fine-grain parallelism refers to the data organization and the corresponding processor architectures exploiting parallelism, traditionally referred to as *array processors or vector processors*. Not all applications are data parallel. Deploying non-data-parallel code to run on a data-parallel machine is counterproductive; it is better to use the CPU and straight-line code to start.

A data-parallel operation should exhibit common memory patterns, such as large arrays of regular data like lines of pixels or tiles of pixels, which are processed in the same way. Referring back to Fig. 8.1, note that some algorithms operate on vectors of points, lines, and pixel regions. These data patterns and corresponding processing operations are inherently data-parallel. Examples of point operations are color corrections and data-type conversions, and examples of area operations are convolution and morphology. Some algorithms are straight-line code, with lots of branching and little parallelism. Fine-grained data parallelism is supported directly via SIMD and SIMT methods.

SIMD, SIMT, and SPMD Fundamentals

The supercomputers of yesterday are now equivalent to the GPUs and multi-core CPUs of today. The performance of SIMD, SIMT, and SPMD machines, and their parallel programming languages, is of great interest to the scientific community. It has been developed over decades, and many good resources are available that can be applied to inexpensive SOCs today; see the National Center for Supercomputing Applications [526] for a starting point.

SIMD instructions and multiple threads can be applied when fine-grained parallelism exists in the data layout in memory and the algorithm itself, such as with point, line, and area operations on vectors. Single Instruction Multiple Data (SIMD) instructions process several data items in a vector simultaneously. To exploit fine-grained parallelism at the SIMD level, both the computer language and the corresponding ALUs should provide direct support for a rich set of vector data types and vector instructions. Vector-oriented programming languages are required to exploit data-parallelism, as shown in Table 8.8; however, sometimes compiler switches are available to exploit SIMD. Note that languages like C++ do not directly support vector data types and vector instructions, while data-parallel languages do, as shown in Table 8.8.

Table 8.8 Common data-parallel language choices

Language Name	Standard or Proprietary	OS Platform Support
Pixel Shader GLSL	Standard OpenGL	Several OS platforms
Pixel Shader HLSL	Direct3D	Microsoft OS
Compute Shader	Direct3D	Microsoft OS
Compute Shader	Standard OpenGL	Several OS platforms
RenderScript	Android	Google OS
OpenCL	Standard	Several OS platforms
C++ AMP	Microsoft	Microsoft OS platforms
CUDA	Only for NVIDIA GPUs	Several OS platforms
OpenMP	Standard	Several OS platforms

In some cases, the cost of SIMT outweighs its benefit, especially considering run-time overhead for data setup and tear-down, thread management, code portability problems, and scalability across large and small CPUs and GPUs.

In addition to SIMD instructions, a method for launching and managing large groups of threads running the same identical code must be provided to exploit data-parallelism, referred to as Single Instruction Multiple Threading (SIMT), also known as Single Program Multiple Data (SPMD). The SIMT programs are referred to as *shaders*, since historically the pixel shaders and vertex shaders used in computer graphics were the first programs widely used to exploit fine-grained data parallelism. Shaders are also referred to as *kernels*.

Both CPUs and GPUs support SIMD instructions and SIMT methods—for example, using languages like OpenCL. The CPU uses the operating system scheduler for managing threads; however, GPUs use hardware schedulers, dispatchers, and scoreboard logic to track thread execution and blocking status, allowing several threads running an identical kernel on different data to share the same ALU. For the GPU, each shader runs on the ALU until it is blocked on a memory transfer, a function call, or is swapped out by the GPU shader scheduler when its time slice expires.

Note that both C++ AMP and CUDA seem to provide language environments closest to C++. The programming model and language for SIMT programming contains a run-time execution component to marshal data for each thread, launch threads, and manage communications and completion status for groups of threads. Common SIMT languages are shown in Table 8.8.

Note that CPU and GPU execution environments differ significantly at the hardware and software level. The GPU relies on device drivers for setup and tear-down, and fixed-function hardware scheduling, while CPUs rely on the operating system scheduler and perhaps micro-schedulers.

A CPU is typically programmed in C or C++, and the program executes directly from memory and is scheduled by the operating system, while a GPU requires a shader or kernel program to be written in a SIMD/SIMT-friendly language such as a compute shader or pixel shader in DirectX or OpenGL, or a GPGPU language such as CUDA or OpenCL.

Furthermore, a shader/kernel must be launched via a run-time system through a device driver to the GPU, and an execution context is created within the GPU prior to execution. A GPU may also use a dedicated system memory partition where the data must reside, and in some cases the GPU will also provide a dedicated fast-memory unit.

GPGPU programming has both memory data setup and program setup overhead through the run-time system, and unless several kernels are executed sequentially in the GPU to hide the overhead, the setup and tear-down overhead for a single kernel can exceed any benefit gained via the GPU SIMD/SIMT processing.

The decision to use a data parallelism SIMD/SIMT programming model affects program design and portability. The use of SIMD/SIMT is not necessary, and in any case a standard programming language like C++ must be used to control the SIMD/SIMT run-time environment, as well as the entire vision pipeline. However, the performance advantages of a data-parallel SIMD/SIMT model are in some cases dramatically compelling and the best choice. Note, however, that GPGPU SIMD/SIMT programming may actually be slower than using multiple CPU cores with SIMD instructions, coarse-grained threading, and data tiling, especially in cases where the GPU does not support enough parallel threads in hardware, which is the case for smaller GPUs.

Shader Kernel Languages and GPGPU

As shown in Table 8.8, there are several alternatives for creating SIMD/SIMT data-parallel code, sometimes referred to as GPGPU or stream processing. As mentioned above, the actual GPGPU programs are known as *shaders* or *kernels*. Historically, pixel shaders and vertex shaders were developed as data-parallel languages for graphics standards like OpenGL and DirectX. However, with the advent of CUDA built exclusively for NVIDIA GPUs, the idea of a standard, general-purpose compute capability within the GPU emerged. The concept was received in the industry, although no killer apps existed and pixel shaders could also be used to get equivalent results. In the end, each GPGPU programming language translates into machine language anyway, so the choice of high-level GPGPU language may not be significant in many cases.

However, the choice of GPGPU language is sometimes limited for a vendor operating system. For example, major vendors such as Google, Microsoft, and Apple do not agree on the same approach for GPGPU and they provide different languages, which means that industry-wide standardization is still a work in progress and portability of shader code is elusive. Perhaps the closest to a portable standard solution is OpenCL, but compute shaders for DirectX and OpenGL are viable alternatives.

Advanced Instruction Sets and Accelerators

Each processor has a set of advanced instructions for accelerating specific operations. The vendor processor and compiler documentation should be consulted for the latest information. A summary of advanced instructions is shown in Table 8.9.

APIs provided by operating system vendors may or may not use the special instructions. Compilers from each processor vendor will optimize all code to take best advantage of the advanced instructions; other compilers may or may not provide optimizations. However, each compiler will provide different flags to control optimizations, so code tuning and profiling are required. Using assembler language is the best way to get all the performance available from the advanced instruction sets.

Table 8.9 Advanced instruction set items

Instruction Type	Description
Trancendentals	GPU's have special assembler instructions to compute common transcendental math functions for graphics rendering math operations, such as dot product, square root, cosine, and logarithms. In some cases, CPUs also have transcendental functions.
Fused instructions	Common operations such as multiply and add are often implemented in single fused MADD instruction, where both multiply and add are performed in a single clock cycle; the instruction may have three or more operands.
SIMD instructions	CPUs have SIMD instruction sets, such as the Intel SSE and Intel AVX instructions, similar SIMD for AMD processors, and NEON for ARM processors.
Advanced data types	Some instruction sets, such as for GPU's, provide odd data types not supported by common language compilers, such as half-byte integers, 8-bit floating point numbers, and fixed-point numbers. Special data types may be supported by portions of the instruction set, but not all.
Memory access modifiers	Some processors provide strided memory access capability to support scatter-gather operations, bit-swizzling operations to allow for register contents to be moved and copied in programmable bit patterns, and permuted memory access patterns to support cross-lane patterns. Intel processors also provide MPX memory protection instructions for pointer checking.
Security	Cryptographic accelerators and special instructions may be provided for common ciphers such as SHA or AES ciphers; for example, INTEL AES-NI. In addition, Intel offers the INTEL SGX extensions to provide curtained memory regions to execute secure software; the curtained regions cannot be accessed by malware.
Hardware accelerators	Common accelerators include GPU texture samplers for image warping and sub-sampling, and DMA units for fast memory copies. Operating systems provide APIs to access the DMA unit [492]. Graphics programming languages such as OpenGL and DirectX provide access to the texture sampler, and GPGPU languages such as OpenCL and CUDA also provide texture sampler APIs.

Vision Algorithm Optimizations and Tuning

Optimizations can be based on intuition or on performance profiling, usually a combination of both. Assuming that the hot spots are identified, a variety of optimization methods can be applied as discussed in this section. Performance hotspots can be addressed from the data perspective, the algorithm perspective, or both. Most of the time memory access is a hidden cost, and not understood by the developer (the algorithms are hard enough). However memory optimizations alone can be the key to increasing performance. Table 8.11 summarizes various approaches for optimizations, which are discussed next.

Data access patterns for each algorithm can be described using the Zinner, Kubinger, and Isaac taxonomy [476] shown in Table 8.10. Note that usually the preferred data access pattern is in-place (IP) computations, which involve reading the data once into fast registers, processing and storing the results in the registers, and writing the final results back on top of the original image. This approach takes maximal advantage of the cache lines and the registers, avoiding slower memory until the data is processed.

Table 8.10 Image processing data access pattern taxonomy (from Zinner et al. [476])

Type	Description	Source Images	Destination Images	READ	WRITE
(1S)	1 source, 0 destination	1	0	Source image	no
(2S)	2 source, 0 destination	2	0	Source images	no
(IP)	<i>In-place*</i>	1	0	Source image	Source image
(1S1D)	1 source, 1 destination	1	1	Source image	Destination image
(2S1D)	2 source, 1 destination	2	1	Source images	Destination image

*IP processing is usually the simplest way to reduce memory read/write bandwidth and memory footprint

Compiler and Manual Optimizations

Usually a good compiler can automatically perform many of the optimizations listed in Table 8.11; however, check the compiler flags to understand the options. The goal of the optimizations is to keep the CPU instruction execution pipelines full, or to reduce memory traffic. However, many of the optimizations in Table 8.11 require hand coding to boil down the algorithm into tighter loops with more data sharing in fast registers and less data copying.

Table 8.11 Common optimization techniques, manual and compiler methods

Name	Description
Sub-function inlining	Eliminating function calls by copying the function code in-line
Task chaining	Feeding the output of a function into a waiting function piece by piece
Branch elimination	Re-coding to eliminate conditional branches, or reduce branches by combining multiple branch conditions together
Loop coalescing	Combining inner and outer loops into fewer loops using more straight line code
Packing data	Rearranging data alignment within structures and adding padding to certain data items for better data alignment to larger data word or page boundaries to allow for more efficient memory read and write
Loop unrolling	Reducing the loop iteration count by replicating code inside the loop; may be accomplished using straight line code replication or by packing multiple iterations into a VLIW
Function coalescing*	Rewriting serial functions into a single function, with a single outer loop to read and write data to system memory; passing small data items in fast registers between coalesced functions instead of passing large images buffers
ROS-DMA*	Double-buffering DMA overlapped with processing; DMA and processing occur in parallel, DMA the new data in during processing, DMA the results out

*Function coalescing and ROS-DMA are not compiler methods, and may be performed at the source code level

Note: See references [480, 481] for more information on compiler optimizations, and see each vendor's compiler documentation for information on available optimization controls

Tuning

After optimizing, tuning a working vision pipeline can be accomplished from several perspectives. The goal is to provide run-time *controls*. Table 8.12 provides some examples of tuning controls that may be implemented to allow for run-time or compile-time tuning.

Table 8.12 Run-time tuning controls for a vision pipeline

Image Resolution	Allowing variable resolution over an octave scale or other scale to reduce workload
Frames per second	Skipping frames to reduce the workload
Feature database size and accuracy	Finding ways to reduce the size of the database, for example have one data base with higher accuracy, and another database with lower accuracy, each built using a different classifier
Feature database organization and speed	Improving performance through better organization and searching, perhaps have more than one database, each using a different organization strategy and classifier

Feature Descriptor Retrofit, Detectors, Distance Functions

As discussed in Chap. 6, many feature descriptor methods such as SIFT can be retro-fitted to use other representations and feature descriptions. For example, the LBP-SIFT retrofit discussed in Chap. 6 uses a local binary pattern in place of the gradient methods used by SIFT for impressive speedup, while preserving the other aspects of the SIFT pipeline. The ROOT-SIFT method is another SIFT acceleration alternative discussed in Chap. 6. Detectors and descriptors can be mixed and matched to achieve different combinations of invariance and performance, see the REIN framework [379].

In addition to the descriptor extractor itself, the distance functions often consume considerable time in the feature matching stage. For example, local binary descriptors such as FREAK and ORB use fast Hamming distance, while SIFT uses the Euclidean distance, which is slower. Retro-fitting the vision pipeline to use a local binary descriptor is an example of how the distance function can have a significant performance impact.

It should be pointed out that the descriptors reviewed in Chap. 6 are often based on academic research, not on extensive engineering field trials and optimizations. Each method is just a starting point for further development and customization. We can be sure that military weapon systems have been using similar, but far more optimal feature description methods for decades within vision pipelines in deployed systems.

Boxlets and Convolution Acceleration

Convolution is one of the most common operations in feature description and image preprocessing, so convolution is a key target for optimizations and hardware acceleration. The boxlet method [374] approximates convolution and provides a speed vs. accuracy trade-off. Boxlets can be used to optimize any system that relies heavily on convolutions, such as the convolutional network approach used by LeCun and others [77, 328, 331]. The basic approach is to approximate a pair of 2D signals, the kernel and the image, as low-degree polynomials, which quantizes each signal and reduces the data size; and then differentiating the two signals to obtain the impulse functions and convolution approximation. The full convolution can be recovered by integrating the result of the differentiation.

Another convolution and general area processing acceleration method is to reuse as much overlapping data as possible while it exists in fast registers, instead of reading the entire region of data items for each operation. When performing area operations, it is possible to program to use sliding windows and pointers in an attempt to reuse data items from adjacent rectangles that are already in the register files, rather than copying complete new rectangles into registers for each area operation. This is another area suited for silicon acceleration.

Also, scatter-gather instructions can be used to gather the convolution data into memory for accelerated processing in some cases, and GPUs often optimize the memory architecture for fast area operations.

Data-Type Optimizations, Integer vs. Float

Software engineers usually use integers as the default data type, with little thought about memory and performance. Often, there is low-hanging fruit in most code in the area of data types. For example, conversion of data from int32 to int16, and conversion from double to float, are obvious space-savings items to consider when the extra bit precision is not needed.

In some cases, floating-point data types are used when an integer will do equally well. Floating-point computations in general require nearly four times more silicon area, which consumes correspondingly more power. The data types consume more memory and may require more clock cycles to compute. As an alternative to floating point, some processors provide fixed-point data types and instructions, which can be very efficient.

Optimization Resources

Several resources in the form of software libraries and tools are available for computer vision and image processing optimizations. Some are listed in Table 8.13.

Table 8.13 Vision optimization resources

Method	Acceleration Strategy	Examples
Threading libraries	Coarse-grained parallelism	Intel TBB, pthreads
Pipeline building tools	Connect functions into pipelines	PfeLib Vision Pipeline Library [477] Halide [525]*
Primitive acceleration libraries	Functions are pre-optimized	Intel IPP, NVIDIA NPP, Qualcomm FastCV
PGPUs languages	Develop SIMD SIMD code	CUDA, OpenCL, C++ AMP, INTEL CILK++, GLSL, HLSL, Compute Shaders for OpenGL and Direct3D, RenderScript
Compiler flags	Compiler optimizes for each processor; see Table 8.11	Vendor-specific
SIMD instructions	Directly code in assembler, or use compiler flags for standard languages, or use GPGPU languages.	Vendor-specific
Hardware accelerators	Silicon accelerators for complex functions	Texture Samplers; others provided selectively by vendors
Advanced instruction sets	Accelerate complex low-level operations, or fuse multiple instructions; see Table 8.9	INTEL AVX, ARM NEON, GPU instruction sets

*Open source available.

Summary

This chapter ties together the discussions from previous chapters into complete vision systems by developing four purely hypothetical high-level application designs. Design details such as compute resource assignments and optimization alternatives are discussed for each pipeline, intended to generate a discussion about how to design efficient systems (the examples are sketchy at times). The applications explored include automobile recognition using shape and color features, face and emotion detection using sparse local features, whole image classification using global features, and augmented reality. Each example illustrates the use of different feature descriptor families from the Vision Metrics Taxonomy presented in Chap. 5, such as polygon shape methods, color descriptors, sparse local features, global features, and depth information. A wide range of feature description methods are used in the examples to illustrate the challenges in the preprocessing stage.

In addition, a general discussion of design concepts for optimizations and load balancing across the compute resources in the SOC fabric (CPU, GPU, and memory) is provided to explore HW/SW system challenges, such as power reductions. Finally, an overview of SW optimization resources and specific optimization techniques is presented.

Chapter 8: Learning Assignments

1. Estimate the memory space and memory bandwidth required to process stereo RGB images of resolution 1920×1080 at 60 frames per second, and show how the estimates are derived.
2. A virtual memory system allows each running program to operate in a large virtual memory space, sharing physical memory with all other programs. Describe how a virtual memory system operates at a high level, including all layers and speeds of memory used between the fast registers, main memory, and the slowest page/swap file. Discuss the relative speeds of each memory layer in the memory architecture (HINT: registers operate at one processor clock cycle per read/write access).
3. Describe memory swapping and paging in a virtual memory system, and discuss the performance implications for computer vision applications.
4. Describe how DMA operates, and how memory regions can be locked into memory.
5. Discuss how data structure organization can influence memory performance in a computer vision application, and provide an example worst-case memory organization for a specific algorithm.
6. Compare the power use of a CPU and a GPU, and compare the silicon die area of each compute unit.
7. Name a few compiler flags that can be used to optimize code in a C++ compiler.
8. Describe sub-function in-lining and function coalescing.
9. Describe loop coalescing and loop unrolling.
10. Discuss the trade-off between using integer and floating point data types, and when each data type is appropriate.
11. List several methods to optimize memory access in computer vision applications by illustrating how a specific algorithm works, and how to optimize memory access for the specific algorithm. HINT: memory access can be optimized by the structure of memory items, and the speed of the memory used.
12. Describe at least three types of image processing and computer vision algorithms that can be optimized to use a multi-core CPU, and describe the algorithm optimization.
13. Name at least two types of assembler instructions available in high-end CPUs to accelerate computer vision and imaging applications.
14. Discuss multi-threading and how it can be applied to computer vision, and describe an algorithm that has been optimized for multi-threading.
15. Discuss SIMD instructions and how SIMD can be applied to computer vision.
16. Describe the major features of a GPU and describe how the GPU features can be applied to computer vision, and describe an algorithm that has been optimized for a GPU.
17. Name at least two programming languages that can be used to program a GPU.
18. Discuss SIMT processing and describe how SIMT can be applied to computer vision, and describe an algorithm that has been optimized for SIMT.
19. Discuss VLIW and instruction level parallelism, and how VLIW can be applied to computer vision, and describe an algorithm that has been optimized for VLIW.
20. Choose a computer vision application, then describe at a high level how to partition the compute workload to operate in parallel across a multi-core CPU and a GPU.
21. Name several image processing operations and describe specific optimization methods for each image processing operation.
22. List and describe the *facial landmark features*, such as eye corners, that should be detected to classify emotions, and describe the pixel characteristics of each facial landmark.

23. Define and code a face recognition algorithm and describe each *pipeline stage*. Provide an architecture document with requirements and a high level design, suitable for someone else to implement the system from the architecture document. Example pipeline stages may include (1) sensor processing, (2) global image metrics used to guide image preprocessing, (3) search strategies and feature detectors used to locate the face region in the image to know where to search for individual facial landmark features, (4) which feature detectors to use to locate the *face landmark features*, (5) how to design culling criteria for ignoring bad features (HINT: relative position of features is one possibility), how to measure correspondence between detected features and expected features, (6) define a visual vocabulary builder for the final classifier, using K-MEANS to cluster similar feature descriptors into the reduced vocabulary set, and a select distance function of your choice to measure correspondence between incoming detected features and learned vocabulary features in the dictionary. Select or create a face image database of your choice for ground truth data (some examples are in Appendix A, such as Faces In The Wild and CMU Multi-Pie Face). Create code to implement a training protocol to build the vocabulary dictionary. The code may run on the computer of your choice.

Feature Learning Architecture Taxonomy and Neuroscience Background

9

“... they have sought out many inventions.”

—Solomon, 848–796 BC

In many respects, computer vision practitioners are now being outpaced by neuroscientists, who are leading the way, *modeling computer vision systems directly after neurobiology*, and borrowing from computer vision and imaging to simulate the biology and theories of the human visual system. The state of the art in computer vision is rapidly moving towards synthetic brains and synthetic vision systems, similar to other biological sciences where we see synthetic biology such as prosthetics, robotics, and genomic engineering. Computer vision is becoming a subset of neuroscience and vision sciences, where researchers implement complete *synthetic vision* models, leveraging computer vision and imaging methods, leaving some computer vision and imaging methods in the wake of history.

And the *big-data* statisticians are also moving heavily into computer vision applications, treating video content as another form of big-data, borrowing feature description and learning tools from computer vision systems to build huge visual learning systems to classify and correlate visual information together with other forms of electronic information on a massive scale.



Figure 9.1 Connectome images, which are maps or wiring diagrams of connectivity pathways, captured *in vivo* using multiple neuroimaging modalities, “Courtesy of the Laboratory of Neuro Imaging and Martinos Center for Biomedical Imaging, Consortium of the Human Connectome Project—www.humanconnectomeproject.org”

Computer vision is becoming a commodity.

Here we provide a taxonomy of feature learning architecture concepts, a list of terminology, and a brief introduction to basic neuroscience concepts which have inspired many of the feature learning architectures surveyed in Chap. 10. We identify the basic architecture types, components, and structural elements here.

Key topics covered in this chapter include:

- Neuroscience inspiration
- Historical developments in machine learning for feature learning
- Terminology
- Feature learning architecture and component taxonomy

This chapter is recommended to understand key terminology and background concepts for the *Feature Learning Architecture and Deep Learning Survey* in Chap. 10. Expert computer vision practitioners may also benefit from a quick perusal.

Neuroscience Inspirations for Computer Vision

Neural networks can mimic portions of the visual pathway in the brain, resulting in a *deep learning approach* to computer vision consisting of a hierarchy of features that represent visual intelligence, such as low-level textures, object parts, entire objects, and scenes. While many computer vision and local feature descriptor methods, such as SIFT and FREAK, are heavily inspired by the anatomy of the human vision system closer to the retina, advances in compute power have made neurological models of the entire visual pathway in the brain attractive and practical for computer vision and feature learning systems. For example, we will survey various types of artificial neural networks and models of the entire visual pathway such as HMAX and HMO in Chap. 10.

Given the value of neuroscience, computer vision practitioners will benefit by following neurobiology and neuroscience journals as well as computer vision journals. Computer vision methods are feeding into neurobiological research. This chapter and Chap. 10 cover deep learning and feature learning architectures, complementing the other computer vision approaches using local feature descriptors covered in Chaps. 4 and 5.

Neural network architectures can implement complete, but primitive, computer vision systems. Even the recognition and classification logic can be entirely described and trained in the artificial neural architecture models, rather than relying on mathematical and statistical classification methods as discussed in Chap. 4. And hybrid methods combine neural networks with other methods.

The race is on to develop artificial brains [650–652] that provide a common framework architecture for vision and other forms of learning and reasoning. Up to the present time, we have seen the foundations for computer vision laid in basic understanding of color science, image processing science, and systems inspired by the human visual system at the retina. Now, with neuroscience investigations providing more insight into neurobiology, and sufficient compute power to implement artificial neural networks that mimic the human brain, we are witnessing the success of early *synthetic vision systems*, modeled after the combined principles of imaging science, biological visual science, and neuroscience.

In addition to neural networks, we introduce related *machine learning* topics which are applied across a wide range of application domains, such as data analytics for marketing and investment and government intelligence, speech recognition, and computer vision analytics of images and videos to understand scenes and find objects. We cannot provide a comprehensive treatment of machine learning, but only highlight a small subset of machine learning as applied to *feature learning*, and refer the interested reader to better references into machine learning topics outside our scope as we go along. An excellent introductory reference text to neural network design and training is provided by Hagan et al. [668], and another by Bishop [637]. See the machine learning text by Mitchel [809]. Good references on classification and learning include Duda and Hart [807], Alpaydin [808], Deng et al. [553] and LeCun [810]. To dig deeper into the field or ANNs in general, see Bengio et al. [554] and Schmidhuber [552]. A good survey text of statistical methods applied to machine learning is found in Hastie [347].

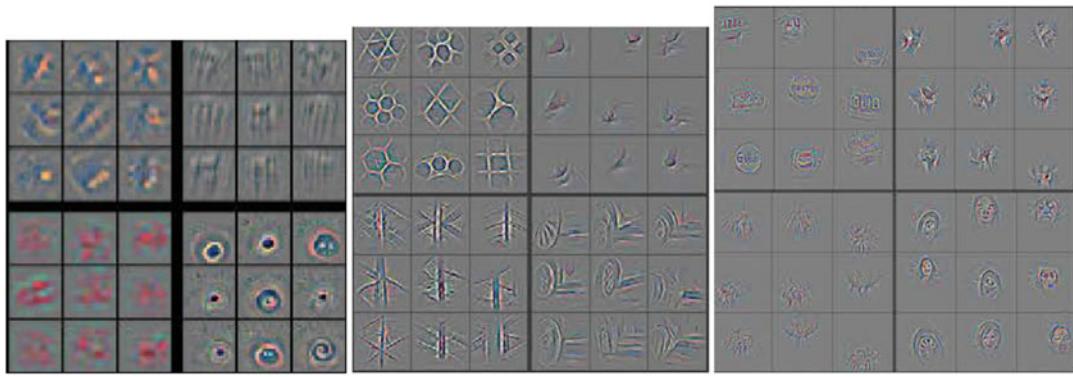


Figure 9.2 A hierarchy of learned features, *left* low-level, *center* mid-level, and *right* high-level. Feature visualizations from Zeiler and Fergus [641], © Springer-Verlag from CVPR, used by permission

Feature Generation vs. Feature Learning

What is termed feature learning in deep learning networks could also be termed *feature generation*. Deep learning networks typically use a *backpropagation method* analogous to a *tedious averaging procedure* to generate features *generically* over all training samples, but not specifically for any. The generated features are similar to the features in the images, but not exactly the same. In fact, by slightly changing the training data or weight initializations for a given deep learning architecture, different features are generated, so therefore nothing is really learned at all but rather *different features are generated*. It could be argued that features are latent within the image data, and that a good learning system would learn the features regardless of the weight initializations or slight variations in the training data, but this does not happen as one might expect in DNNs today. Thus, the deep learning systems following artificial neural models and deep-learning training mechanisms are still very primitive systems.

The deep learning training process, discussed in detail in Chap. 10, starts by taking training samples for input, and tuning a set of *feature weights* until each feature weight best represents the average of the features *attracted by a similarity measure* to the feature weights from the training set. In other words, the generated features are a *compressed representation*, not an exact representation, of a group of features. This is not how humans learn. Neuroscience suggests that the brain creates new

impressions of important items, rather than averaging impressions together, under the *view-based theories* surveyed in the HMAX section in Chap. 10.

However, in Appendix E we discuss a dense memory-based approach to feature representation based on stored hierarchical feature memory impressions, called *Visual Genomes* [534]. The basic idea is that features are unprocessed hierarchical memory impressions along the visual pathway, and that feature memory is virtually unlimited following view-based assumptions.

In summary, what is called feature learning in deep learning today is still a very primitive art form, and may be termed as feature generation instead.

Terminology of Neuroscience Applied to Computer Vision

Here we introduce some high-level terms for feature learning concepts, especially useful for understanding the artificial intelligence and deep learning approaches to feature learning. Reviewing this terminology section is encouraged, since there is some obscurity in the literature where practitioners (this author included) use many equivalent terms for the same concepts.

There are several independent research communities that have contributed to terminology in computer vision discussions on feature learning:

1. Neuroscience, Neurobiology
2. Artificial Neural Networks (ANNs)
3. Artificial Intelligence and machine learning
4. Computer Vision and Image Processing

As a result, the terminology used in computer vision literature ends up being a mixture of several dialects, often hard to understand, and sometimes confusing. In fact, the deep learning community has a dialect, which we elucidate and clarify as we can.

The neuroscientist takes a different approach than the computer vision practitioner, since neuroscience is concerned with understanding and modeling the biological structures and functions of the brain as neuroscientific information. The computer vision ANN researcher is mostly concerned with *mimicking* the neurobiology to create synthetic, or artificial models and architectures suitable for implementation in software and hardware, perhaps to solve real problems. The use of ANNs for computer vision has been mostly driven from outside the computer vision community, and introduced to computer vision as the ANN practitioners looked for new applications. ANN practitioners usually know less about existing methods for computer vision and image processing, and computer vision practitioners are usually well versed in image processing, signal processing, pattern recognition, and statistical methods for data classification, and know little or nothing of neuroscience or ANNs, which makes it challenging for computer vision practitioners to learn and apply ANN methods without a huge learning curve.

This section lists many common vocabulary terms intended to bridge the gap between computer vision and neuroscience research. The ANN community would benefit from more understanding of computer vision science, particularly local feature description methods surveyed in Chaps. 4, 5, and 6, to enable ANNs to be more effectively enhanced to incorporate the best thinking from computer vision.

Some of the terms used in deep learning discussions are *overloaded, meaning one thing in normal conversation, and quite another thing in deep learning research parlance*. For example, ambiguous terms include *deep learning* (what is deep?), *greedy learning* (greedy in what way?), *pooling* (is water involved?), and *stating that convolution is trice (1) a filter, (2) a feature, and (3) weights*, depending

on the context. Sometimes, practitioners introduce entirely new terminology to describe their own work, in spite of perfectly suitable words and existing terminology that would work equally well, if not better. So the author apologizes in advance for terminology confusion found herein, since terminology problems likely infect this author's work and chosen terminology as well.

Brief list of terms and definitions (not in any particular order):

- **Machine Learning (ML):** Systems that can be trained from input data to learn features or patterns, in order to make decisions and take actions.
- **Feature Learning (FL):** A method for generating, learning and tuning sets of features or patterns from specific ground-truth data (images in the computer vision case).
- **Hierarchical Learning (HL):** A model, like a pipeline of layers, that learns a hierarchy of feature sets for each level of the hierarchy, for example, the lowest level features may be oriented edges, contours, or blobs, the next higher level may be larger micro-textures and shapes, and higher levels may resemble motifs or parts of objects, or complete objects.
- **Artificial Neural Network (ANN):** One of many methods for implementing a NN, including FNN, RCN, and any method which is neurobiologically inspired.
- **Neural Computing (NC):** methods for machine learning and ANNs.
- **Neuron, neural function, artificial neuron, synthetic neuron, neural model:** Neurons are the processing units in the NN, taking inputs and producing outputs to feed to other layers. Many neural models are possible such as polynomial or correlational, or convolutional models. A convolutional *neural function* may model a neuron as follows: first, features are computed using a function such as $f() = (\text{inputs} \times \text{weights}) + \text{bias}$, second, an activation function is used to condition the result using a nonlinear function such as sigmoid to squash or spread the results (see Activation Function).
- **Activation Function, Transfer Function, Activation Function,** Biological neurons act as switches, and fire or *activate* in a binary *all-or-nothing* fashion when input values are sufficient. In this sense, a biological neuron activation function determines simply when to fire based on inputs and electrochemical bias. However, some artificial neural networks do not fire in a binary fashion, but rather as an analog numerical strength. The activation is the result of the neuron model (see Neuron) and may be a convolutional, or a polynomial, or some other function. See Nonlinearity, and Transfer Function. The output of the transfer function may be further conditioned (see Pooling and Rectification), and finally fed directly into the next layer of the network as inputs. Various transfer functions are used such as sigmoids, see Fig. 9.18. Transfer functions are chosen to be *differentiable* to support back propagation using gradient descent methods. Transfer functions are chosen to operate for example by centering the results around zero, which implies that the input data should also be centered around zero. Also, the transfer function is intended to reduce saturation effects at the extrema of the data range. One goal of nonlinearity is to project the purely linear convolution operation into a nonlinear solution space, which is believed to improve results. In addition, the nonlinearity may result in faster convergence during backpropagation training to move the gradient more quickly out of flat spots towards the local minima. Also, the nonlinearity is used to ensure that the value is differentiable for backpropagation.
- **Nonlinearity,** in convolutional style neural networks (CNNs), the Transfer Function (activation function) for each neuron may use a *nonlinear function*, such as a sigmoid, which distributes, or spreads, the actual output result from the feature match stage *within a range of values* (squashing) in a nonlinear fashion, which is believed to improve performance of ANNs in general, since neurobiology suggests that the neuron is a nonlinear function. In addition, the nonlinearity can move the data into a higher dimensional space, where features that were not separable become separable. However, nothing concrete is known about the algorithms used inside biological neural

activation functions, so practitioners guess. For example using sigmoid nonlinearity is believed to help solve problems of data saturation, perhaps caused by numeric overflow, poor lighting, or very strong lighting. For example, if the correlation output is saturated at 1 in range $-1 \dots +1$, a nonlinearity function may be chosen to ideally redistribute the value +1 value somewhere within the range, say -0.99 to $+0.99$, to overcome saturation.

- **Pooling and subsampling**, grouping features in a local region such as a 2×2 or 5×5 region, which may overlap adjacent pool regions, and then selecting a new feature to represent the entire pool to reduce the spatial resolution. See Fig. 9.3. Various methods are used to subsample the pool such as taking the average value or MAX value. Pooling also provides for some translational and deformation invariance. Pooling can help to produce more stable features from a related group of more unstable features.

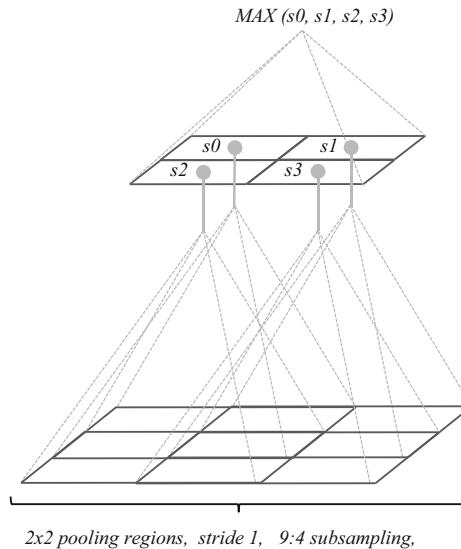


Figure 9.3 Pooling, with input as 4×4 regions from a 3×3 image, yielding an output of 4×4 pixels, which is then subsampled down to a single pixel using the MAX value of the pool

- **Multilayer pooling, cross channel pooling, cross channel parametric pooling, CCCP**, a pooling method that includes pixels from the current layer feature map and prior layer feature maps together (x,y,z regions), rather than limiting the pooling to spatial x,y regions only.
- **Rectification**, a method of dealing with poor feature matches by rectifying the value to account for feature polarity (positives and negatives of the feature). Rectification solves problems where, for example, negative matches occur where the feature is inverted. Various methods are used for rectification including ABSVAL().
- **Threshold, Bias**, in convolutional networks, various thresholds are used to scale the transfer functions or the convolutional filter in each neuron. In most neural networks we survey, the bias is ignored and used only as a convenience for matrix math operations in the neural model. In neurobiology, the bias may be a chemically or hormonally induced factor.
- **Feature, Filter, Weights**: all these terms may be used to refer to a single feature. For convolutional features, an input region is processed using a filter weight matrix as the dot product of the input and the matrix. The output is a filtered version of the input, and the output is placed in a feature map or output image. For a convolutional style network, weights are the features. Positive

weights excite; negative weights inhibit. Stronger weights have more influence. Typically, floating point numbers are used for weights. In *convolutional style networks*, the feature is used three ways: (1) as a filter on the input image to produce an output image for the next layer in the network, (2) as a feature descriptor, or *correlation template*, for pattern matching in the input, and (3) as a tunable feature, by tuning and adjusting the weights in the matrix during backpropagation to better match the input. The word *filter* is a misnomer here, since normally we refer to a filter as an operation that changes the image. However, mathematically it can be shown that convolution is equivalent to correlation under certain assumptions, but still this may be confusing since the filters are used for both filtering, and as features themselves. The correlation template weights resemble shapes in the hierarchy of features, such as edges in the low-level features, and higher level concepts such as motifs or object parts in the higher level features.

- **Feature Map**, This term is used in convolutional style networks to describe the output of applying the weight filter to the input image, which produces a filtered image called a feature map, which is passed to the next layer of the network as the input image. A series of feature maps are produced in each layer, one feature map per filter or feature weight matrix.
- **Subsampling**, a method to reduce the size of the image for the next layer, see Pooling.
- **Feature Set**: For a convolutional network, a feature set is a collection of weight matrices, one set for each layer in the hierarchy. A feature set may contain several hundred features per layer. Some networks use basis functions as feature descriptors for the lower level features, such as Gabor functions. See Basis Functions.
- **Basis Functions, Basis Features, Basis Set**: a function used to create and define a feature, rather than learning and tuning the features from scratch. Examples include Gabor Functions and Fourier Series. ANN models render the basis features into a weight matrix to use as convolutional features.
- **Codebook, dictionary, bag of visual words**: terms used to describe sets of learned features, perhaps local feature descriptors or ANN style features, for use in visual vocabulary analysis, similar to textual word analysis to determine the subject of an email. For example, a set of SIFT descriptors learned from a training set can be considered a codebook, and the codebook is useful to classify images.
- **Encoding, Sparse Coding**: various methods used to create, learn and represent condensed basis sets of features, rather than exhaustive sets, allowing for feature classification and reconstruction from linear combinations of basis features.
- **Layers**: levels or stages in a hierarchical network, which include numerical conditioning, convolutional filtering layers, fully connected classification layers, and post-process functions such as pooling and nonlinear activation functions.
- **Input unit**: input the network, such as pixel input.
- **Output unit**: output of the network, may be from a classifier or multi-stage classifier.
- **Hidden layers**: In DNN parlance, hidden means any layer of neurons in between the sensory inputs such as pixels, and the classification outputs of the network.
- **Hidden unit**: In DNN parlance, this is an artificial neuron in a hidden layer.
- **Deep Learning (DL)**: In DNN parlance, a DL model is a hierarchical learning model containing hidden units in hidden layers, typically three or more hidden layers.
- **Neural Network (NN)**: a network architecture and learning model inspired by neurobiology. Abbreviated as NN or ANN (artificial neural network). Several variants exist such as CNNs and RNNs. Typically ANNs use the convolutional filtering neural model with a nonlinear output conditioning function.
- **Convolutional Neural Network (CNN), Convnet**: the most common style of ANN used in computer vision presently, implemented using a feed-forward network with several hidden layers

of convolutional neural models. The major innovation in convnets is the use of a uniform sized local receptive field containing an $n \times n$ kernel of pixels from the input, which feeds into a single convolutional neural function. In this way, a single convolutional neuron can be sequentially fed with all the local receptive fields from the input image, rather than implementing a system with one neuron per receptive field.

- **Deep Neural Network (DNN)**: another term for NNs or HLs or CNNs, specifically networks where several layers of neural processing are used to produce hierarchical, or deeper sets of features, usually deep means several levels of features.
- **Feed-Forward Neural Network (FNN)**: an NN which does not have feedback loops, the hierarchy is a straight pipeline feed forward from input through the hierarchy of features to classifiers and outputs.
- **Recurrent Neural Network (RNN)**: an NN with some recurrent feedback loops, implementing a basic form of memory. Typically, RNNs are applied to spatiotemporal problems, or sequence learning.
- **Latent variables, features**: another term for features, or inferred variables in the statistical sense, features being latent in the image pixels until they are learned.
- **Hyperparameters**, DNN parameters tuned during learning and training to control learning rates, learning momentum, regularization terms to control weight decay, and other such variables.
- **Error Minimization, Cost Function**, an algorithm used during training to quantify the error between the trained pattern and the computed pattern. The error is computed using a suitable distance function.
- **Early stopping**, stopping the training process before the local minima are reached, in order to speed up training, especially when the convergence is very slow and proceeds in small steps.
- **Autoencoder (AE)**: a type of FNN that learns a sparse, compressed set of basis features to reconstruct its own input. One main difference between an AE and other ANNs is that the AE has the same number of outputs as inputs. Autoencoders can be useful for layer-wise training of DNNs.
- **Restricted Boltzmann Machine (RBM)**: a type of FNN with the restriction that all input units are connected directly to a hidden layer containing hidden units, rather than connecting inputs to output units. The input of an RBM may be pixels, or else the output of an RBM. The RBM architecture enables training protocols such as back propagation using gradient descent to tune feature weights.
- **Deep Belief Network (DBN)**: a type of FFN, such as a CNN, that is typically implemented using autoencoders or RBMs for the hidden layers.
- **Labeled data, unlabeled data**: training data, such as training images, which have been annotated to define the contents of the training image. In this way the system can be trained according to the known labels, as opposed to trying to learn image features and contents without labels.
- **Supervised learning**: learning which takes place using labeled data, and other preconditions which are set up to define a learning model.
- **Reinforcement Learning**: A close relative of supervised learning, which adds a *reinforcement function* for feedback, similar to a reward, to encourage the network to train itself according to the reinforcement feedback.
- **Unsupervised learning**: a learning method that attempts to create and tune features or patterns using unlabeled data and no learning preconditions.
- **Classifier**: an algorithm modeling a mapping between feature inputs and class outputs. Examples include SVMs, FC layers in CNNs, and regression models. Some classifiers (linear regression models, for example) rely on data groupings that can be separated by a line in 2D (linearly

separable), or by hyperplanes in higher dimensions, to find the largest margin between data groupings to delimit the classes. Other classifiers may use binary feature vectors representing parts models, where each bit in the vector represents the presence or absence of a part, and matching is performed using Hamming distance to compute matches. There is no limit to the clustering and group distance methods used for classification, see Chap. 4 for more details.

- **Fully Connected Layer (FC layer):** describes a connection topology where all outputs of a layer are each connected to all inputs of the next layer. Fully connected layers may implement the classifier of a CNN, modeled as a 1D vector of artificial neurons with a 1D vector of weights and bias factors.
- **Kernel Connected Layer:** instead of connecting all inputs to each convolutional processing layer node in the FC sense, kernel-connected layers gather $n \times n$ kernels of local regions using a sliding window over the input. This topology provides for parallelization of the convolutions by providing one $n \times n$ input kernel to each virtual artificial neuron to perform the convolutions. A single artificial neuron may be fed sequentially from all the kernels, or the kernels may be queued up for a group of artificial neurons to service in parallel.
- **Sparse Connected Layers** allow for random or sparse connection patterns from the input to subsequent layers. Sparse connections may be defined using variants of dropout methods to regularize the model, see Dropout.
- **Softmax:** a logistic or exponential function, typically used as the last classification layer, to squash and normalize class predictions into a range of probabilities 0.0..1, like a probability expressed as a percentage. For example, each node in the softmax layer will produce a result in the range 0.0..1 to rank the classification for the given sample.
- **Back Propagation:** A family of methods used during ANN training for tuning the accuracy of feature weights in NNs. Back propagation works by taking the results of the forward pass through the NN, finding the error between the current predicted result and the correct result, and distributing the error proportionally backwards through the network, to minimize the error at each layer by adjusting the feature weights. Backpropagation methods resemble a huge feature averaging process. Many methods are used, and some contain optimizations for various goals such as more rapid convergence. Examples include gradient descent and contrastive divergence.
- **Gradient Descent:** a backpropagation method based on modeling the classification error as a total gradient, and then working backwards through the network layer by layer, proportionally finding the contributing gradients for each feature weight, which can be understood from the chain rule from calculus. The gradients become smaller and smaller as they are propagated backwards.

Classes of Feature Learning

We take a wide view of feature learning in this work, and consider many approaches including artificial neural networks, deep learning, sparse coding, dictionary learning, and local feature descriptor learning. As described in Chaps. 4, 5 and 6, many local feature descriptor methods like SIFT, ORB and FREAK contain attributes that are *learned*, such as shape, pixel sampling pattern, and various weights. Some practitioners primarily refer to feature learning methods in the context of deep learning methods like CNNs, and then go on to claim that local feature descriptors, such as SIFT, ORB and FREAK, are not learned, but rather *handcrafted*. However, this distinction does not hold since many attributes of the better local features are learned and tuned. In fact, the best local features are inspired by the human vision system, and descriptors such as SIFT, ORB and FREAK actually

incorporate several features of the human visual system in their design. Actually, DNNs are heavily handcrafted, requiring extensive empirical work to get the architecture and parameters correct for training. In addition, DNNs typically use the most primitive and least invariant feature of all: correlation templates, limiting invariance.

Given this wide view of feature learning, we survey and discuss several classes of feature learning. Our criteria for dividing the various approaches is based on the feature descriptor used in each approach, so we find three primary categories of feature learning: (1) convolution feature weight learning, (2) local feature descriptor learning, and (3) basis feature composition and learning.

Convolutional Feature Weight Learning

For computer vision, a convolutional neural network (CNN) learns hierarchical sets of features represented as *weights* in a kernel or matrix shape, such as 3×3 . Features are created in a *feature hierarchy*, with low-level features representing micro-textures, and higher-level features representing part of objects. Some or all of the features in the hierarchy are fed to a classifier for matching larger objects composed from the features. Each weight is tuned during training, as discussed in Chap. 10. Some practitioners view CNN methods as a variant of parts models [549].

Local Feature Descriptor Learning

Many of the local feature descriptor methods discussed in Chap. 6, such as ORB, SIFT, FREAK and D-NETS, actually learn their shape, pixel sampling pattern, and weight thresholds during a training process. For example, FREAK and ORB are trained against ground truth data to learn how to build the feature sampling pattern, see Chap. 4. However, some of the more primitive local feature descriptor methods do not learn or tune themselves to the ground truth data, such as shape detectors like Gabor Functions, and other hard-coded descriptors.

Sparse coding is often employed with local feature descriptors to boil down the size of the descriptor set into a visual vocabulary, rather than an exhaustive vocabulary. The visual vocabulary of features is used to classify the image, similar to a word vocabulary to classify textual information. Sparse coding is analogous to JPEG image compression, which uses the selected level of frequency detail from local DCT transforms of image pixel blocks to create, or encode, a compressed representation of the image. Sparse coding is an encoding and compression method. Sparse coding can also be considered as a performance optimization method, since compute performance increases with fewer features to manage, but sparse coding in the extreme may affect accuracy if too much detail is compressed away.

The K-SVD method developed by Aharon et al. [779] is one example of a sparse feature encoding method, Aharon provides a good survey of various sparse coding methods. The Histogram of Sparse Codes (HSC) method [117] discussed in Chap. 6, learns a single-level representation of features in an unsupervised manner by using a sliding window edge detector, and then generates a histogram of gradients feature descriptor from the edges within the features. The K-SVD method is used for reducing the feature set to a sparse code set. Feng et al. [797] and Bo et al. [132] developed optimizations on the K-SVD method to shrink the codebook, and more uniformly distribute the values within the reconstruction space.

See Chap. 10 for a deeper dive into sparse coding and vocabulary methods.

Basis Feature Composition and Dictionary Learning

Basis features created from basis functions like Gabor functions can be used as *primitive base level features*, and then higher-level features can be composed from the basis features and further tuned into new higher-level features and collected into a *dictionary*, alternatively referred to as a *visual vocabulary*. The basis features may also be collected in a feature hierarchy. In this context, the higher-level vocabulary is learned, and is based on the lower-level basis features. Similar to textual analysis using histograms of word counts from the vocabulary dictionary, visual vocabularies can be used to discover the content of an image, for example by using a histogram format containing the weighted visual words detected in an image, fed into a classifier for matching, see Fig. 10.70. With parts models, for example the parts of a bicycle, either as patches or as local feature descriptors such as SIFT, may be learned and composed into a dictionary, and then at classification time, if enough parts of the bicycle are detected in an image, then the classifier can predict and match on a bicycle.

Summary Perspective on Feature Learning Methods

We can conclude that deep learning methods create a hierarchical set of *averaged, compressed* or a *sparse* set of features, which correspond to the dominant features in the training set. Local features such as SIFT and FREAK are trained to represent *complex* and *more invariant* features, which are individually more powerful than the single correlation template features used in deep learning. It seems that the power of deep learning style features arises from (1) the sheer number of features, and (2) the hierarchical nature of the features to represent low-, mid-, and higher-level concepts. This perspective suggests that sets of local features, such as SIFT and FREAK, can be created likewise in a hierarchical manner, to rival or exceed the performance of simple correlation template features as used in convolutional style deep learning methods.

Machine Learning Models for Computer Vision

Here we outline a very broad-brush picture of machine learning to set the stage for digging into feature learning architectures.

As Juergen Schmidhuber has noted, *machine learning is about compression*. The learned features are a compressed set of parameters, which represent a wide range of objects. In a convolutional neuron model such as a CNN or RNN, each filter is a *compressed representation* of many similar features. Perhaps deep learning methods compress millions of parameters, or features, into a few hundred.

In the early days of machine learning, the field of *Artificial Intelligence* (AI) was popularized as the field of study encompassing all methods for computerized learning and machine understanding. Initially, primitive compute and memory capabilities hampered development of practical AI systems, so many considered the field of AI to be over-hyped, and AI was relegated to obscure academic research programs for many years. Interest in AI methods has gradually renewed, keeping pace with technological advances in electronics, inexpensive compute power, and more memory, allowing AI methods to be applied to commercially viable systems such as data bases. Over time, machine learning and AI have branched off into several research segments to address different approaches to machine learning, which are summarized in Fig. 9.4.

As shown in Fig. 9.4, the machine learning pipeline has been expressed several ways:

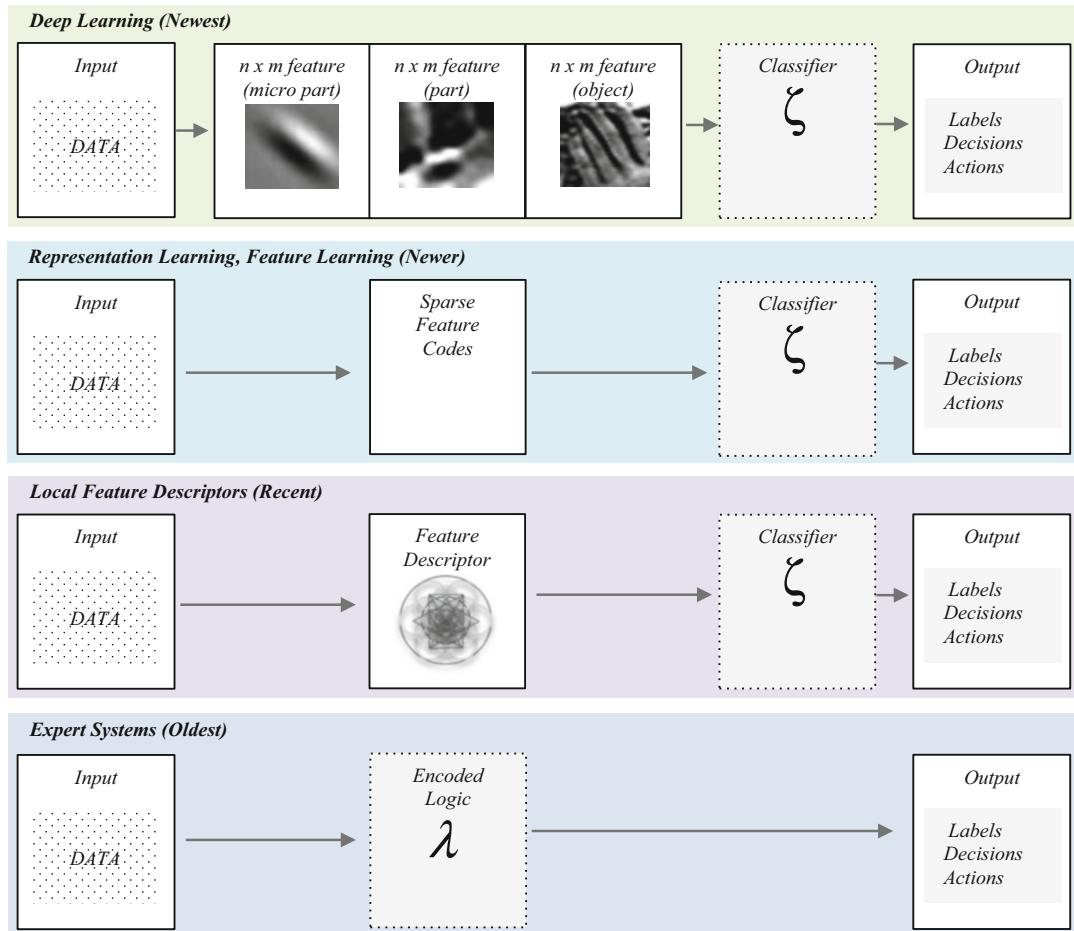


Figure 9.4 A simplified taxonomy of AI approaches, highlighting similarities and differences, after Bengio et al. [554]

1. Feature extraction, could be local feature descriptors or learned DNN features
2. Encoding features, perhaps keep all or only a sparse set of all features
3. Classifier design and training

The power of the DNN approach is that all three steps in the pipeline can be collapsed into a *single feature learning architecture*, applicable to a range of applications, which can be trained to generate (or learn) features from the training set. In essence, the goal is that the DNN network architecture is fixed; *the features become the program code, the learning parameters and training protocol are the programmer, and the DNN is the computer.*

See the resources in Appendix C, which lists some open source projects, commercial products, and links to follow several key researchers including Schmidhuber, Ng, LeCun, and others.

Expert Systems

At one time, *Expert Systems* were hot topics in AI research. Expert systems [556] are the equivalent of a system encoding the knowledge of an expert, and have been referred to as *Rule-Based Expert Systems*,

and *Decision Support Systems*. The rule-based method allowed for recursive process models [607], similar to the RNN. The main expert systems architecture components include (1) a *knowledge base* of rules and facts, and (2) an *inference engine* to make decisions from inputs using the knowledge base. To populate the knowledge base, experts are interviewed by programmers who code the expert's knowledge as logical decisions and rules. At one time, attempts were made to generalize and structure the concept of expert systems into software products, which were programmable and trainable for a range of applications. AI researchers developed several variants of commercial expert systems in the 1970s and 1980s, and then research interest tapered off. After some time, the ideas and key learnings from expert system have been integrated into business logic software and database software, and expert systems are hardly discussed in academic circles today. However, in many cases, expert systems have been embodied as ad hoc *systems*, often composed of expert-level logic hard-coded into software. Using a loose definition of expert systems as hard-coded expert logic, a great many software programs are, informally, expert systems. No attention is given in this chapter to *expert system methods*, but a few ad hoc expert system approaches are already covered in the Chap. 8 examples.

Statistical and Mathematical Analysis Methods

Perhaps the largest portion of AI is based on standard numerical analysis methods developed over the past few hundred years, using a huge range of regression, group distance, clustering, and error minimization methods to classify feature descriptors. There is no limit to the numerical methods applied to machine learning and computer vision tasks. We cover several statistical methods used for distance and clustering in Chap. 4, and some statistical methods as applied to deep learning and feature learning in this chapter, and briefly touch on SVMs in Chap. 10. See also Hastie et al. [698] for an overview of statistical learning.

Neural Science Inspired Methods

Inspired by the brain's neural networks as studied in neuroscience, *Neural Network Methods (NNs)* and *deep learning* represent intelligence using multilevel networks of primitive, artificial neurons. As shown in Fig. 4.23, the human brain composes quite complex neural networks, which apparently grow and change over time. Since the brain's neural networks are so complex and impossible to model in a real system, researchers have developed primitive models to mimic the brain's architecture, as illustrated in Fig. 9.10.

Neurobiology is at the forefront of artificial neural network research in the computer science community, and neurobiologists routinely modify ANN architectural models, such as CNNs, and also create new models such as HMAX and HMO. *Artificial neural networks are part of the future*: this author predicts that neural network research and the resulting architectures inspired by biological mechanisms will ultimately drive computer science to produce real products and commercial breakthroughs in the near future, based on a common *neural programming language and architecture* accelerated in silicon.

Table 9.1 provides a comparison of artificial neural methods to human neural biology.

Deep Learning

The term deep learning was popularized in the early 2000s by Hinton and others [539, 540] to describe hierarchical feed-forward neural networks with several layers. In DNN parlance, *deep* means

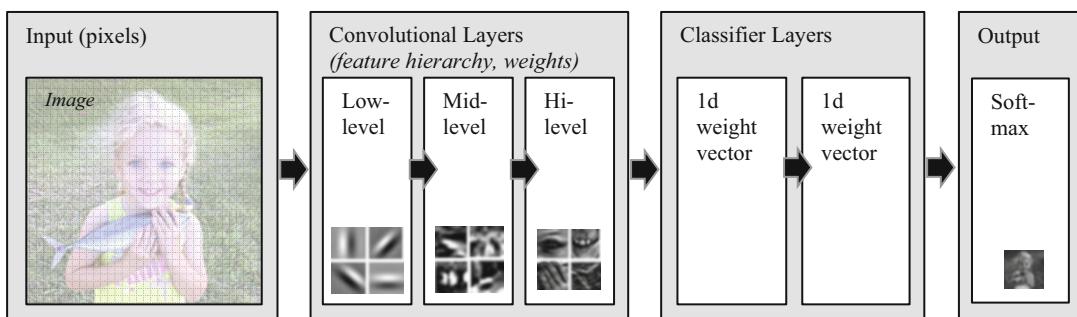
Table 9.1 A simple comparison of an artificial neural network to the human brain

Attribute	Artificial Neural Network	Human Brain
Types of neurons	Single type (usually)	Several types, number of types unknown
Neural activation, transfer function	Mathematical, logical function of inputs	Unknown non-linear functions based on electrical, chemical, high level reasoning, questions, excitement
Network Topology	Structured, known connections such as FNN or RNN	Unknown topological connection model, semi-unstructured, apparently grows and changes
Inputs	Known data such as pixels	Sensory inputs (touch, smell, sight, taste, hearing), and recursive inputs from unknown neural pathways
Number of neurons	<1,000,000 (typ.)	80,000,000,000 – est.
Number of connections	<1,000,000 (typ.)	100,000,000,000,000 – est.
Number of parameters	<1,000,000,000 (typ.)	?
Learning time	Days, weeks per item	Continuous over lifetime
Speed of recognition *round trip=one guess	<1 ms per round trip *getting faster...	~100 ms per round trip

a neural network with more than one *hidden layer* between the input and output. A hidden layer contains hidden units or *artificial neurons*, which represent learned features from the input data, see Fig. 9.2. In DNNs, layers may be replicated, and each layer learns and produces features. See Fig. 9.5. In CNN-DNN systems, the features are generated by the convolutional neurons at each layer as feature *weights* or correlation templates, and stored in a hierarchy of feature sets.

Earlier work starting in the late 1980s and early 1990s by LeCun and others [574, 576, 577], popularized as Convnets and CNNs at that time, can also be considered as deep learning. Deep learning was demonstrated to be effective by Schmidhuber in 1991 [582] using deep RNNs. Several feature learning concepts besides CNNs fall into the deep learning category. At this time, most DNN systems follow the work of LeCun and Hinton and are implemented as CNNs, or convolutional style networks, using correlation templates for the feature descriptor. Also, the HMAX model [812] was developed by Riesenhuber and Poggio in the late 1990s to model the entire visual cortex in a primitive manner, rather than just mimic parts of the neurobiology as CNNs do.

Deep Learning Neural Networks (DNNs) implemented as CNNs have become more common for solving computer vision problems, due mainly to advances in commodity compute power. DNNs have done well in computer vision challenge events and real-world applications since around 2010, resulting in many new computer vision researchers who have adopted DNN methods, as surveyed in Chap. 10.

**Figure 9.5** A typical deep learning architecture with input, Convolutional Layers, Classifier Layers, and the output as labeled objects. There are many variations covered in the surveys in Chap. 10

In a typical convolutional style DNN (CNN), each feature set in the hierarchy contains perhaps hundreds of individual *features* in each *hidden layer* in DNN parlance. Each feature is simply a *correlation template*. Each correlation template is represented as a matrix containing trained weights. Each feature captures a different type of detail, so that low-level features capture finer details, and higher-level features capture higher-level concepts like parts or objects. DNN features are a hierarchy of objects and parts of objects. Indeed, some have called deep learning a variant of the well-known method of *parts models* [549]. While this is conceptually true, the nuances of deep learning methods go farther than parts models since the architectures allow for a uniform training protocol, and the architecture of deep learning alone is a field in and of itself, since it relies heavily on neuroscience and machine learning, and is applicable to a wide range of problems besides computer vision.

Deep architectures, or hierarchical architectures, are not a new concept per se, but the nuances made possible by deep learning networks, such as feature learning, transfer learning (discussed later in this chapter and Chap. 10), and the common DNN architecture are novel and effective. The power of the DNN lies in the *quantity* of features in *each level* of the feature map hierarchy, and the *quality* of the features resulting from tuning each feature during training. In Chap. 10 we survey the limits and capabilities of selected DNNs, as well as fundamental innovations at various stages of the DNN architecture.

DNN Hacking and Misclassification

It should be pointed out that deep learning methods, particularly convolutional networks, are vulnerable to *adversarial misclassification*. For example when an image is modified with just minor changes to some of the pixel values, the DNN may misclassify. This is a serious problem for trusted applications of DNNs. In the future, alternative methods for training and classification will be needed to overcome such problems, yet even so we are likely to see hacking threats to computer vision systems, and perhaps hacking challenge events for computer vision systems.

Deep learning methods have been shown to fail to recognize features that are obviously correct, and succeed in recognizing features that are obviously incorrect [541]. See Goodfellow et al. [591] for details on DNN misclassification of adversarial data with high confidence, based on intentionally corrupted data (i.e., *DNN hacking*). See also Sutskever et al. 2014 regarding intriguing anomalies of DNNs.

History of Machine Learning (ML) and Feature Learning

The history of machine learning is fascinating, so we present this brief introduction which is well worth reading, especially for the novice. Perhaps the most detailed and complete historical information and references can be found in Schmidhuber [552], who is one of the pioneers in this field. In addition, the comprehensive survey by Anderson [555] provides references to many of the seminal research papers for machine learning and artificial intelligence. See also Haykin [647] for a comprehensive introduction to neural networks.

A review of history can never be comprehensive, since certain details may be left out. However, here we survey several historical developments in machine learning relevant to computer vision feature learning, specifically hierarchical or deep learning methods. We highlight the early foundations of ML, especially the inspiration from neuroscience. We also touch upon some related statistical and numerical methods. Machine learning has historically been discussed using several related terms and concepts, such as neurodynamics, cybernetics, autonomies, synnoetics, intelectronics, artificial intelligence, neurocomputing, pattern recognition, expert systems, analytics, and deep learning.

Historical Survey, 1940s–2010s

Much of the foundation for machine learning was laid in the 1940s, 1950s, and 1960s, *without the computing power and computing languages available today*. And as usual for technology fields, practitioners unfamiliar with historical research and concepts often reinvent the same concepts over again, giving them a new name and a new spin. Later in this chapter we will dig deeper into many of the key historical concepts, which have remained influential in feature learning.

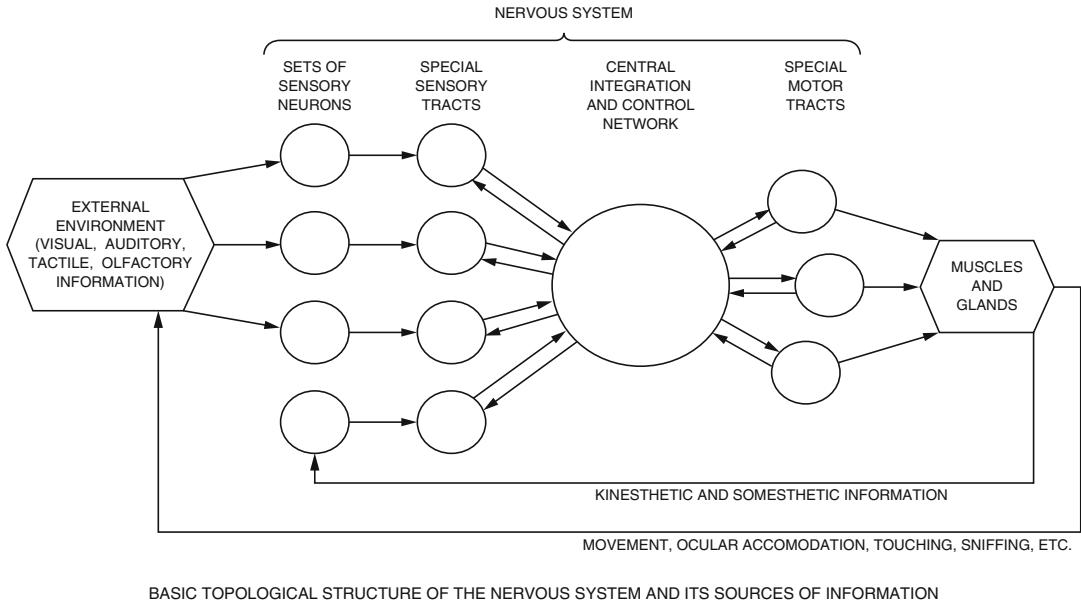


Figure 9.6 Prevailing theories of the nature of neuroscience around the early 1960s, image © Springer-Verlag, from Brain Theory, Palm, Gunther, Aertsen, Ad, Frank Roseblatt: Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms, Springer Berlin Heidelberg, 1986

1940s and 1950s

During the 1940s and 1950s, with the dawn of the first analog computers, we begin to see progress in the area of concrete, working artificial intelligence models. The first model of the brain was implemented as a Boolean circuit by McCulloch and Pitts in 1943 [561], using a hardwired neural model of their *Logical Calculus for Nervous Activity* in Boolean logic, using vacuum tubes. Rudimentary learning concepts soon followed, in the area of unsupervised learning by Hebb in 1949 [562] and supervised learning in the simple *Perceptron* model in 1958 by Rosenblatt [557, 563] and also in 1961 by Joseph [558]. The Perceptron is a major milestone, using adjustable weights for learning like CNNs, even though learning was limited to tasks with linearly separable training data. We will dig deeper into the Perceptron concepts later in Chap. 10. Wiesel and Hubel in 1959 [559] and 1962 [560] developed an influential *hierarchical* or *deep model* of the brain, where some neurons closer to sensory input detected *simple* (S) or low-level features, and other neurons detected *complex* (C) or higher level features, to enable high-level reasoning. This Hubel and Wiesel work on the concepts of *Simple Cells* (S) and *Complex Cells* (C) has inspired the development of most deep learning architectures to date, especially the notion that features exist in a hierarchy of detail, low, mid, and high detail. So we will dig deeper into the Hubel and Wiesel model later in this chapter (Fig. 9.11).

The Pandemonium model introduced in 1959 by Selfridge [815] is perhaps the earliest model for pattern recognition and *unsupervised feature learning* using a set of local feature detectors. The application was 1D signal Morse Code detection. The detectors are referred to as *computational sub-daemons*. The classifier is the *cognitive daemon*, which computes a *score* via a weighted sum of the detected features. The Pandemonium system also addresses the problem of finding the *highest score* using a *hill climber* method in the score space where gradient descent methods may find several solutions. In Pandemonium, each feature detector is scanned across the image in sliding windows to look for low level and mid level features. Then a decision demon is used to classify the signal based on the presence of a specific set of the strongest feature activations using tuned feature weights. Pandemonium defines two methods for feature learning, called *mutated fission* and *conjugation*, to determine if a new feature should be added to the feature set to increase the representational power, or removed from the feature set if little used and not needed.

1960s

In the 1960s, slightly larger computers and memory systems enabled more detailed neuroscience-inspired models to be implemented. Some of the first single-level feed-forward neural networks were implemented by Joseph [558] in 1961. Also, Rosenblatt developed multilayer Perceptron models with up to four layers, which are described using the terms “*forward coupling, cross-coupling, and back-coupling*” between layers (RNN style). Another one of the earliest multilayer or deep network is called the *Group Method for Data Handling* (GMDH), modeled as a multilayer Perceptron, demonstrated by Ivakhneko and Lapa starting in 1965 [564–566]. Note that Ivakhneko work is very much worth reading today, containing advanced concepts that are no longer in common use. Ivakhnenko’s work appeared frequently in *Avtomatika*, the journal of *Soviet Automatic Control*, and other Soviet publications, which were not available to western researchers. In addition during the 1960s, several researchers explored gradient descent and maximum descent methods applicable to back propagation algorithms, see Dreyfus in 1962 [572], Kelley in 1960 [578], and Bryson in 1961 [580].

In 1963, Vapnik and Lerner [594] introduced the *generalized portrait method* algorithm for statistical classification, which is the basis for the Support Vector Machine (SVM) still widely used in classification for machine learning. Later work on SVMs by Vapnik [597] and others [598, 599] has expanded the basic SVM model [585], and SVMs remain one of the most widely used statistical tools for machine learning classification.

Neural networks and AI in general went through a lull stage as the 1960s progressed, when many researchers became disillusioned due to limitations with the simple Perceptron model introduced in 1958 by Rosenblatt [557, 563] and also in 1961 by Joseph [558]. In 1964 Powell [600] developed a method for computing gradient descent without using derivatives.

1970s

In 1971, Ivakhneko [567] refined the GMDH model as the first deep NN-inspired model, which was fairly deep using eight layers, also known as a *Polynomial Neural Network* (PNN), which learned the number of layers, neural units per layer, and pruned units as needed. The methods in GMDH are quite advanced for their time, and some of the concepts have been rediscovered in recent years and renamed, and others still remain quite advanced and worth further research. An active GMDH research community continues today, see the GMDH survey in Chap. 10 for more details.

Also in the 1970s the first back propagation algorithms were implemented for tuning the weight factors in CNNs. Although back propagation and gradient descent were applied to NNs the 1960s, only a few people actually implemented the algorithms. For example, Dreyfus [573] developed back prop methods for error minimization using weight control parameters in 1972 and 1973. Also, in 1974

Werbos demonstrated a method for multilayer threshold adjustments in DNNs using backprop in his Ph.D. thesis.

1980s

The decade of the 1980s saw the introduction of the Neocognitron by Fukushima [570, 571] in 1979, based on his earlier work on the Cognitron [677], which was the first deep style of neural network, similar to the CNN/FNN architectures of today. The Neocognitron used convolutional-style features, where the features were modeled as rectangular correlation templates, biologically inspired by the 2D visual field spanning some small, local receptor distance. Each correlation template filter coefficient was given a weight factor, which was adjusted. The output of each filter stage was fed as input to the next convolutional layer of similar filters. Weight parameters were shared or replicated in each layer. The Neocognitron included subsampling units to look for the strongest filter activations in small local regions, to provide some translation invariance for feature detection. Instead of max pooling to find the best filter responses in the subsampling units, spatial averaging was used instead. The Neocognitron weights were tuned using unsupervised hard-coded learning rules, and Neocognitron did not use back propagation.

The 1980s also saw more back propagation innovations in deep learning. In 1981 Werbos [581] demonstrated a method for back propagation using gradient descent specifically for tuning weights in an NN, which is an early forerunner of current systems. Also, LeCun [574, 575] developed early success with back propagation. LeCun is one of the early pioneers in *deployed* neural network systems, and led the effort to create several, commercially successful NN systems [682], including handwritten postal code recognition, and bank check handwriting recognition, which were commercial successes. LeCun-style FNNs are often referred to as Convnets, and Convolutional Neural Networks (CNNs).

In 1989, LeCun et al. [576] introduced the basic convolutional neural network CNN architecture still used today often referred to as *LeNet*, including max-pooling, weight sharing via kernel-connected layers, and advances in back propagation [593]. LeCun's work also introduced the MNIST ground truth dataset for handwritten digits, one of the most widely known benchmarks in machine learning.

Also in the 1980s, early hierarchical, or deep networks of autoencoders were developed by Ballard and Hinton to learn feature hierarchies one layer at a time using unsupervised learning. Boltzman Machines [587, 588] were introduced by Ackley, Hinton, Sejnowski as a type of RNN, related to the Hopfield Network [589] RNN. Both Boltzman Machines and Hopfield Networks were difficult to train, but even so they were important theoretical advances. Smolensky introduced the Harmonium [590], a variant of the basic Boltzman Machine, which had restrictions on the connectivity making it a FNN instead of an RNN. The Harmonium was later reintroduced as a *Restricted Boltzman Machine* by Hinton and others after 2000, with a much-improved training protocol for faster learning.

1990s

According to Schmidhuber (*lecture in NYC on Deep Learning RNAissance*), Neural Network research went through a dark age from the early 1990s to the early 2000s, for about a decade, where nobody would fund it, and nobody outside the academic community was interested. According to LeCun (*comments made during CVPR private Intel meeting in Portland*), during this dark period, a lot of NN research applied to computer vision was ignored by the computer vision community. But major advances in compute technology, coupled with advances in key algorithms, profoundly advanced the science during this time, and propelled CNNs to the forefront of computer vision.

In the 1990s, several improvements were made to NNs. In 1991, Hochreiter's diploma thesis explored back propagation training using gradient descent, and identified the key problems:

(1) *vanishing gradients* where the errors become too small as the errors are propagated backwards to previous layers, (2) *exploding gradients*, and (3) *oscillating gradients* which prevented the algorithms from converging to tune the weights. For exploding and oscillating gradients, the gradient problems manifested themselves early in the training cycle vs. later, since the gradients are expected to get smaller and smaller as the error minimization functions converge at a solution. Hochreiter's insight, together with collaborations with Schmidhuber, led to several innovations in back propagation to overcome problems with gradients.

In 1992, Schmidhuber [582] proposed a method to overcome the gradient descent problems identified by Hochreiter using a hierarchical, or multilayer network, which could be pretrained layer by layer using unsupervised learning to yield a sparse, compressed set of reasonable and useful starting feature weights, that were subsequently fine-tuned using back propagation under supervised learning.

In 1992 the Cresceptron [616] was demonstrated by Wang et al., providing a hierarchical framework for learning multi-scale feature hierarchies over an image scale pyramid, similar to earlier work by LeCun [576, 577]. The Cresceptron defined a *neural plane* for recognizing low-level features anywhere in the image, and a *concept module* for classifying higher-level features. In addition, the Cresceptron could identify the coordinates of each feature found in the image by back-tracking the response paths through the network (good idea).

Then in 1997, the *LSTM* (Long Term Short Term Memory) RNN architecture was developed by Hochreiter and Schmidhuber [584] that includes memory units which can feed back prior information into the RNN at a later time to assist in learning time-related sequences. LSTM uses long-term and short-term memory cells as *carousels*, gating the introduction of backpropagated gradients in a controlled manner into a time-aware gradient descent algorithm. The original LSTM paper [584] also provides a very complete survey of gradient descent algorithm variants. LSTM is a major advancement in NN architecture, so we will study LSTM architecture in more detail later in Chap. 10. LSTM is not affected by gradient descent problems such as vanishing and exploding gradients, since the LSTM gradient descent algorithm is gated and controlled to condition the gradients. The LSTM makes the layer-wise training protocol using unsupervised pretraining followed by supervised training unnecessary, therefore dramatically speeding up the training process.

SVMs were enhanced to almost their current form in 1992 by Boser et al. [595] based on the original 1963 algorithm by Vapnik. Other SVM enhancement was made in 1993 by Cortes and Vapnik [596]. For introductory material see Ng [585].

A hierarchical model of the visual cortex dubbed HMAX was first introduced by Riesenhuber and Poggio in 1999 [812] taking inspiration from the known facts about the visual ventral pathway, largely based on experimental data from Logothetis et al. [813] who measured responses to shapes across the visual pathway in monkeys. Logothetis found that some groups of neurons along the hierarchy respond to specific shapes similar to Gabor-like basis functions at the low levels, and object-level concepts such as faces in higher levels. We survey HMAX in some detail in Chap. 10. This author contends that neuroscience is now the driving force in computer vision research, and will lead to synthetic vision systems with a biological interface.

2000s–2010s

Inexpensive computing power brought rapid advances in CNN-style networks. Personal computers containing relatively powerful GPUs, multi-core CPUs, large memories, and SIMD instruction sets, could be exploited via GPGPU, SIMD, and SIMT programming methods, providing massive parallelism ideal for backpropagation training using very large datasets, the key to effective CNN training. Many of the concepts developed prior to 2000 resurfaced as NN research accelerated. In some cases older concepts were refined further. The rapid innovation spawned several new architectural and

algorithmic variations to NNs and feature learning, which we survey throughout Chap. 10, such as pooling, subsampling, numeric conditioning, dropout, and mini-batch training to name a few.

The terms *Deep Learning* and *Deep Neural Networks (DNNs)* were popularized around 2006 when a think-tank was formed by Hinton, LeCun, Bengio, and others, who were funded by the Canadian Government's CIFAR program to promote research into neural networks; their work paid off. Key research breakthroughs and success followed. Beginning in the mid-late 2000s, DNNs began to excel in computer vision applications, and a new, larger group of academics and industrialists took notice. With the renewed attention to NN research and real-world applications, including thousands of new researchers and SW development engineers in industry, the science advanced rapidly.

In 2006 Hinton et al. [539, 540] demonstrated FNNs which were trained effectively using unsupervised pretraining, and promoted a style of NNs called Deep Belief Networks (DBNs) built as a deep hierarchy of Restricted Boltzmann Machine layers, trained layer by layer using contrastive divergence [601]. In the early 2010s, LeCun and Hinton led research groups who demonstrated success with FNNs, winning the top positions in several CVPR IMAGENET challenge events,¹ and influencing most of the other researchers to follow their architectures.

Research teams worldwide are continuing to create DNN innovations. For example in 2012, Ciresan et al. [602] demonstrated a DNN setting the record for the German traffic sign benchmark, beating human experts, and also winning several other international machine learning competitions, including the ICDAR 2009 handwriting recognition challenge to learn three languages. Schmidhuber's NN lab is one of the most diversified, has won several competitions, and extends into many areas besides computer vision.²

Based on CNNs styled after LeCun's earlier *LeNet*, plus the work of many others mentioned in this brief history, the new millennium has seen rapid architectural and algorithmic innovations to ANNs. In fact, recent work on local feature descriptor and interest point methods has slowed dramatically in the research community, as research has shifted attention to NNs, and eventually the research topics will settle down and become more balanced. While the newer DNN methods are effective, the best local feature descriptor methods, such as SIFT and FREAK, still provide superior invariance and robustness across a wider range of criteria. However, by introducing variations into the training set for the DNN, much of the desired invariance can be learned and represented in correlation templates, at a great cost of feature set size increase, and training time increase.

In summary, the groundwork was laid before 2000 for the progress in neural networks that we see today. Increased computing power and large sets of labeled training data images make real-world computer vision applications using NNs possible and very effective. In Chap. 10 we will dig further into the architecture and design innovations that have occurred in NNs, comparing and contrasting the various approaches.

Artificial Neural Network (ANN) Taxonomy Overview

As shown in Fig. 10.1, we break down ANN architectures into three types:

- **FNN**—Feed-Forward Neural Networks, where all the inputs move in the same direction towards the classifiers at the output stage. CNNs are equivalent to FNNs.

¹ <http://www.image-net.org> - see “Challenges” for 2013-2015 and onwards

² <http://people.idsia.ch/~juergen/> Schmidhuber's home page

- **RNN**—Recurrent Neural Networks, which allow for feedback paths in the network. In other words, an *arbitrarily connected network* (ACN).
- **BFN**—Basis Function Networks, which do not always follow neural network principles as defined in the CNN, FNN, and RNN models, and instead use alternate architecture configurations and basis functions such as Gabor features or SIFT features, rather than correlation templates as used in FNN-CNNs and most RNNs.

Note that BFNs are broken out here into a separate category, and the term BFN is introduced in this work for the sake of developing a taxonomy. However, several deep learning models, which we refer to here as BFNs, do not use correlation templates or filters, and instead use alternate basis functions such as Gabor, SIFT, and others, or else the BFNs deviate in some other fashion from a more neuroscience inspired network model as embodied in typical CNNs. So BFN is the catch-all category.

Further, since the general variants of the ANN architecture are applicable to a wide range of problems, we can expect to see more and more commercial and open source software resources to accelerate application development, increasing special purposes ANN accelerators, some standardization of ANN variants, and more classes offered in academia to train the engineering workforce. Selected resources are provided in Appendix C.

Feature Learning Overview

The idea of *feature learning* is to create features from a set of ground truth data, and also to tune the features to be optimal for all similar features in the data. For example, a feature learning problem may include learning the optimal feature sets to represent a wide range of human faces, as the faces are presented in the ground truth data. A typical feature learning goal is to collect only the *optimal features*, limiting the feature set size to only maintain the strongest and most common features, rather than an exhaustive set of all the features possible.

Learned Feature Descriptor Types

A range of feature description methods are used to learn features, and we take a broad view of feature learning to include:

- **Local feature descriptors**—Some are learned, for example SIFT, SURF, FREAK, ORB are trained to encode several dimensions of information to identify unique pixel patterns. Learned feature descriptor dimensions include gradient information, scale information, and pixel sampling pattern information.
- **Regional feature sets**—for example, Spatial Pyramid Matching by Lazebnik et al., [766] described in Chap. 6, trains a descriptor by dividing an image into regions, and each region is described separately to make up the final descriptor.
- **Basis feature sets**—may include selected sets of Gabor features, selected Fourier frequency components, or HMP-style [778] learned features composed by sparse coding alone. CNNs learn a hierarchical set of basis features.
- **CNN-style hierarchical feature sets**—composed of correlation templates, or filters, learned at each level of the network, represented as tunable weight matrices.

By definition, a neural network style feature learning approach does not mandate any particular style of feature, as long as the features can be learned and trained by the neural network. However, most neural networks for computer vision are using convolution-style features, or correlation templates, rather than other feature descriptor methods.

With local feature descriptors such as SIFT, a feature at an interest point in a local region is learned in a scale-invariant manner from an image pyramid by finding local maxima points present at adjacent scales in the image pyramid, and then local region *gradients* around the interest point are pooled and encoded into a *gradient orientation histogram*. A set of SIFT features are collected together for identifying an object. With SIFT style features, it is often possible to recognize objects with very small numbers of features, such as 10–20 features total, since so much local information is encoded in each descriptor. In some applications, hundreds of CNN features are equivalent to 10–20 SIFT feature descriptors.

Hierarchical Feature Learning

Typically, deep networks have 2, 3 or more layers of features. The feature set size is usually limited to a few hundred or so features per level, see Fig. 9.2. The first layer features in the neural network may be derived directly from pixels, and the subsequent layers of the neural network take the input from the pixel correlations at each layer in the form of a feature map with filtered output values (i.e. a processed image), so the values are no longer strictly pixels, but a *nonlinear mapping from pixels to intermediate values*. This nonlinearity is typically a feature of CNNs and viewed as desirable.

How Many Features to Learn?

In the extreme, if a complete memory image of all objects, at all angles, under all invariance criteria was stored, and the computer had enough memory and processing power to sample, match and classify all inputs to the stored features in a practical amount of time, then computer vision would be much simpler. If a method existed to automatically capture the ground truth data for subjects complete with all the desired invariance attributes, such as scale, geometric, and lighting variations, then training would be much simpler. In the extreme, as many features as possible could be useful.

However, practical problems, such as limited ground truth data with bad labels and lack of invariance attributes built-in, limits training efficiency. In addition, memory and compute practicalities limit the number of features that can be stored and searched. So the bulk of the time being spent in feature learning research seems to be towards *creating compromises*, or *optimizations*, devising more efficient and discriminating features and classifiers, working within the limits of practical feature compression into a sufficient number of features, and looking for optimal performance within the memory and compute limitations of the day. See Varma et al. [771] for a discussion of optimal descriptor characterization as a trade-off between invariance and discrimination.

The type of feature descriptor used does not seem to be nearly as critical as *the sheer number of features*. In this survey we will see a wide range of feature descriptors used successfully in various architectures. Simply using a large number of features, especially deep sets of features, seems to be a consistent ingredient for success, and the type of feature does not seem to matter. For example, large numbers of simple image pixel regions have been demonstrated by Gu et al. [706] to be very capable image descriptors for segmentation, detection and classification. Gu organizes the architecture as a robust bag of overlapped features, using Hough voting to identify hypothesis object locations, followed by a classifier, to achieve state of the art accuracy on several tests.

In general, more features are better, but not too many to manage, as discussed in the architecture survey later in this chapter.

The Power Of DNNs

By themselves, individual features learned by a neural network are not very useful. However, the power of deep neural-networks is a combination of several factors:

1. The *hierarchical nature* of the features to represent multilevel concepts. Spreading the features out across layers actually reduces the number of features a CNN would require, since eliminating layers results in the need to add more features to other layers to compensate.
2. The *quantity* of individual features in each layer. A CNN generates many features at each level, which actually increases the accuracy, since the features allow for an overcomplete set of features, rather than a sparse set.
3. Generic feature tuning. A CNN *tunes* each feature to *learn generic features*, which represent a group of similar features. Each feature is typically trained on thousands of similar images, and tuned via gradient minimization methods to best represent the average feature within a range of similarity. Essentially, the CNN style filters are well-tuned blob patterns and contour pattern detectors represent low-, mid-, and high-level concepts. The DNN features are ideally contrast invariant, which is accomplished by nonlinear transforms to the input, such as whitening, normalization, and local histeq. We know that the human visual system (as discussed in the SIFT survey in Chap. 6) is sensitive to gradients, and due to the local receptor pooling in the LGN, the gradients are allowed to slide around the retina and still be recognized. This provides limited deformation invariance for low level features, and extends to higher level features which pool local receptive fields.

The concepts used in deep learning and NN systems, such as hierarchical learning, algorithm pipelines, replicated compute stages (neurons), tuning features via training, and computing via graph methods, is nothing new or novel. Rather, the power of deep learning methods lies mostly in the synergy of the architecture and the sum of its parts.

Encoding Efficiency

It should be noted that the most effective local feature descriptors, such as ORB, SIFT SURF, or FREAK, individually encode much more information than any single CNN feature. In fact, many local feature descriptors are quite powerful, and entire images can be reconstructed fairly well from the local features alone, see Figs. 4.12, 4.13, and 4.14. However, single CNN features do not contain enough information to reconstruct an image very well. We could conclude that powerful local feature descriptors individually encode more information than individual CNN features.

Handcrafted Features vs. Handcrafted Deep Learning

There is some debate about the value of DNNs compared to other computer vision methods. Many DNN practitioners have developed a preference for deep learning methods, stating that learning features is better than designing features, thus taking a biased view against what they call *handcrafted*

features, such as SIFT, FREAK, and other local feature descriptors. However, DNN architectures and training methods are very handcrafted, and rely on several ad hoc design assumptions, such as the DNN architecture, the DNN training protocol, and the learning parameters. A DNN is much more difficult to develop and use compared to local feature descriptors. Local feature descriptors are handcrafted as much as DNNs are handcrafted, both involve empirical engineering processes, and trial and error are expected.

The CNN resembles a system composed of square puzzle pieces (i.e features), where the content of each puzzle piece is learned similar to the average value of several similar square pieces from the training images, and the final classification is a *best guess* based on the puzzle pieces presented. The puzzle pieces are not designed to fit together, since no spatial relationships or coherence are encoded in each piece. One could argue that this is not feature learning at all and more like serendipity.

While neural networks are inspired by a few *neuroscience* concepts, the best local feature descriptors, such as SIFT and FREAK, are inspired by the best *visual science*. So this author envisions a merger of the both the neurobiology inspired approaches, and the vision-science inspired approaches, combining the best feature representations with the best learning and training architectures into a common system.

This discussion and comparison of CNNs and local feature descriptors revolves around the conundrum of *top-down* vs. *bottom-up* design. One perspective is that local feature descriptors such as SIFT or FREAK are designed *top-down*, using high-level concepts based on the human visual system and intuition to guide the descriptor design, while CNN features are designed *bottom-up*, using large sets of very primitive simple correlation templates, simply relying on local receptive fields to guide the formation of the hierarchy of features.

Here are some of the comments this author has seen: “handcrafted features are fragile, inferior, over-specified, or incomplete,” “DNNs are the only method worth using today,” and “we must move beyond handcrafted features and simple machine learning.” Practitioners state that “they do not understand local features, like SIFT,” and surprisingly also state that “they do not really understand neural networks and back propagation . . . but they work.” And perhaps the most intriguing comment may be “handcrafted features are time consuming to create,” since training DNNs can be so time-intensive, data-intensive, and it is difficult to set up all the learning parameters, transfer functions, numeric conditioning, and other parts of the pipeline. Until recently when multi-core CPUs and GPUs with lots of memory were available, DNN training was often not practical or even possible except within academia for small problems. Local features are simple by comparison. Today, only very skilled practitioners, typically dedicated academic researchers, are presently capable of understanding enough to successfully design and deploy DNNs, although DNN toolkits and commercial products are making DNN applications simpler to develop by nonexperts. Soon DNNs will be commoditized, see the resources products in Appendix C.

No method is clearly better in all cases. Depending on the objective criteria chosen, any feature learning or feature descriptor method may be optimized to score better than another. Invariance criterion is often critical. If training speed is important, perhaps DNNs are not the right choice. If scale and geometric distortion invariance is important, perhaps some combination of local feature descriptors with a focused training protocol to present all the scale and geometric views is best.

Deep learning practitioners develop ad hoc methods to make DNNs work at all, and *to steer the feature learning in the right direction*. For this reason, practitioners often copy existing DNNs that work (i.e., *like cut and paste coding*), and make architectural adjustments looking for incremental improvements. The best performing DNNs are typically designed and maintained by teams of dedicated developers at major companies or universities, since the complexities are daunting for a single person to grasp. DNNs are handcrafted.

Invariance and Robustness Attributes for Feature Learning

Invariance and robustness attributes are a measure of quality for any feature descriptor method (See Figs. 5.1 and 5.2). DNN/CNN learned features are *individually* inferior with respect to invariance attributors compared to the better local feature descriptors, but collectively as a hierarchical group DNN/CNN features can be as good or better. The square kernel matrices used in typical CNNs are perhaps the least invariant feature descriptor type. However, by introducing variations into the training set for the DNN, including training samples with the desired variations such as rotations, deformations, contrast, scale, and other variations, the desired invariance can be learned and represented in CNN feature set. We survey training protocols and training sample variations in Chap. 10.

What Are the Best Features and Learning Architectures?

Researchers and practitioners are constantly trying to disentangle the components and architecture attributes of DNNs and other feature learning methods, to find the most critical variables and the best solutions. As the scope of the vision problems grow, the solutions are harder and harder to evaluate and compare. Many researchers are trying to design the *ultimate solution*: recreate human intelligence as artificial brains (See Table 9.2 on various initiatives).

Major areas for architecture optimization include:

1. **Training protocol**, ordering and dropping samples, sample batch sizing, and adding modified copies of each sample to the training set to add invariance, such as geometrically transformed images, and numerically conditioned images to affect contrast, noise, and other techniques surveyed later in this chapter
2. **Features**, the type of feature used (correlation templates vs. SIFT or basis features), the depth of the feature hierarchy or number of convolutional layers, number of color channels per feature, features per layer, and the size of each feature, such as 3×3 vs. 11×11 .
3. **Classifiers**, varying the number of convolutional classifier layers, or using other classifiers such as SVMs.
4. **Numeric Conditioning**, breaking apart the data via numeric conditioning to enhance the spectra, such as local equalizations, normalizations, and whitening methods, as well as the choice of activation functions.
5. **Pooling and subsampling** methods to reduce the feature size and add some invariance by varying the pool size, and pooling criteria, such as MAX pooling and CCCP pooling.

Various practitioners have offered research findings to point towards the best places for DNN optimizations, and findings vary.

Bergstra et al. [644] recommends random experiments to find the best architectures. One study by Russakovsky et al. [643] analyzed the types of errors found in the Imagenet challenge, where mostly CNN methods are performing best. However, only a small handful of comparative conclusions are provided. Many DNN architectures and other non-DNN approaches are used, and compete favorably. Usually, from year to year, the best methods from the prior year are taken as a starting point by many practitioners, and enhancements are made to compete for the next year. Currently, ensemble methods using several networks voting together are popular, as well as variations in the training protocol, covered later in the surveys in Chap. 10.

Coates and Ng argue [618] that trying to choose the best features, or basis functions, is not as critical as choosing the right architecture and encoding scheme. For example, Jarret et al. [619] found that using random features (untrained weights) for the feature set could perform quite well compared to templates with trained weights, which demonstrates that the power of the CNN architecture is *to leverage many features together*, rather than the choice of the *optimal basis features*. Using random weights for correlation templates is equivalent to a random set of edge detectors.

However, Parikh and Zitnick [636] found that *features are the most critical part* of the architecture, by comparing (1) feature descriptors, (2) learning algorithms, and (3) the training data. Parikh and Zitnick created a baseline by using human experts to take the place of algorithms, to compare what a human can do to increase performance in any area compared to various DNN algorithms.

Other research by Eigen et. al. [638] evaluated several parameters of deep CNNs to find the most critical elements. Parameters considered where number of layers, feature map dimensions, spatial kernel extents, number of parameters, pooling size, and pooling placements. Eigen's work focused on (1) the number of layers, (2) features per layer, and (3) the number of parameters, and their work demonstrates that, within the CNN architecture, increasing convolutional layers alone provides the most significant accuracy increase, compared to increasing the number of features in each feature layer in the hierarchy. Also, Eigen found that convolutional layers are fairly insensitive to the number of features, and more sensitive to the dimension (size) of each correlation template, i.e., the number of weights in each correlation template.

Other research by Fergus and his team at Microsoft Research involved removing layers from a CNN DNN, and found that deeper networks performed better. Fergus started by taking the Krizhevsky [640] CNN architecture (two classifier layers, five convolutional layers), and removed layers from the architecture, a few layers at a time, to check the effect on accuracy. In general, multiple convolutional layers and multiple classification layers all add to the effectiveness of the DNN. In addition, the Fergus slides [639] provide information on invariance attributes, confirming the general lack of robustness provided by static correlation templates to occlusion, scale and rotation, except for orthogonal rotations, which correspond to mirrorings. Of course, these results are expected because the correlation template features used are known to provide this level invariance, and nothing more.

Zeiler and Fergus [641, 642] explored methods to visualize the quality of the features in the hierarchy by using an instrumented CNN they call a *deconvolutional* network architecture, to enable visualization of strong feature activations in the input space. In other words, the image regions which strongly activate each feature can be displayed to visually check the quality of the features. However, no actual deconvolutions are performed. Rather, the method uses instrumentation built into the CNN to choose only the best or strongest activating functions in the forward pass by following gradient descent backwards, and setting to zero the lowest gradients to sparsify the features of interest. Also, during the forward pooling pass, a record is kept of the strongest features chosen, along with their Cartesian coordinates in the input space, allowing the region in the input space to be visualized as an image, which is useful to make sure the features are unique and discriminative.

In the view of several practitioners, DNNs are not as novel or different as claimed. See the *Deformable Part Models are Convolutional Neural Networks*, [549] Tech report, by Ross Girshick, Forrest Iandola, Trevor Darrell, Jitendra Malik. In this work, a deformable parts model is shown to be equivalent to a DNN, and a deformable parts model is refactored to use a basic DNN architecture.

Paradoxically, deep learning methods have been shown to fail to recognize features that are obviously correct, and succeed in recognizing features that are obviously incorrect [541, 591]. Just changing a small number of pixel values is all that is needed in some cases to fool the DNN. Of course, this is to be expected during the training of any computer vision system, and is correctable via

retraining and redesign. Perhaps we will see deep learning hacking contests [591] where coders try to spoof DNNs.

Finally, several novel architectures and findings from the latest research are covered in the architecture survey later in this chapter, to illustrate some of the better approaches.

Merger of Big Data, Analytics, and Computer Vision

A trend is becoming visible: neural network architectures are applicable to a wide range of analytic applications, such as speech recognition, computer vision, and general data analytics. The underlying neural network architectures for all the applications is remarkably similar, leading to the possibility of a common architecture for learning and artificial intelligence, such as a neural computer or NC. This means that a single NC architecture may become a common building block for analytics, similar to the idea of a common CPU architecture used for general computing problems. Neural computing will become a commodity item. Neural computing and similar hierarchical and multivariate methods will be used together as hybrids.

In the near future, we should commonly see mobile and hand-held devices with neural computers connected to a remote server used by business, commercial, governmental, military, law enforcement, and legal organizations to perform a complete audio, visual, historical, and textual evaluations of people for employment interviews, banking, commerce, law enforcement, or housing applications. The neural computers will evaluate facial expression, body language, and clothing style for emotions and intentions, as well as audio evaluation of the tone and rhythm of spoken words for latent intentions and assumptions, with complete NC textual analysis of the words they have written in email, texts and blogs, and other documents, including historical records from local governments, academic institutions, purchasing records, and other financial transactions, to develop a *composite character profile*—all using the same neural computing architecture. The result will be complete online, virtual personal profiles stored on the internet somewhere, which can be queried by voice commands or textual commands to learn about a person with or without their knowledge, and perform *what-if analysis and prediction of their future behavior within a set of circumstances, for example allowing a commercial enterprise to design situations or opportunities to suit their preferences and influence purchasing behavior, or by allowing governments to develop policies and propaganda to test the reactions of a population, their preferences, intentions, and personal beliefs.*

How much such information will be relied upon is another matter; however, the technology is in place today to be assembled and developed into a commodity appliance, *available for a fee*, depending upon the analysis desired, the databases you wish to access, and the turn-around time desired. NC-based personal behavior prediction services and NC-based personal profiling services are very near, and in the early stages now, with huge investments being made into early-stage analytics startups now. Computer vision will be a central component of the future of analytics. Imagine government policy and business plans being designed around the predictions generated by an NC to form *future programs*, and evaluation of each *program* by another NC to form *recommendations*, and the *recommendations* being made by another NC to the final *decision authority*—a human . . . or?

Neural network innovations have disrupted and refocused academic research and industry investments to apply neural network methods to more areas. Today, commercial NC products from major corporations enable rapid development of commercial neural network solutions. We are seeing *commoditization* and *standardization* of neural network methods, APIs and software libraries, see Appendix C. Fundamental research has laid the foundations to spawn the first-generation of commodity neural computing architectures in silicon, leading to widespread and pervasive solutions.

The race is on, and will play out similar to the space race and the weapons race, since artificial intelligence is a business and national security priority, garnering billions of dollars of government and industry investment today, and spawning commercial products and services. The stakes are huge, given the potential of creating machines that learn, reason, make decisions, and perform actions, and in some cases the machines will be preferable to humans. Major initiatives are being funded as shown below in Table 9.2. In the near future, robotics, automation, and analytics will become as pervasive as kitchen appliances and power tools, changing the nature of society worldwide, and changing the face of government, military, industry, and commerce.

Table 9.2 Major initiatives in neural computing, including research, products, and services

Investor	Initiative Description
USG / DARPA	Brain Machine Initiative [650] \$3 BIL USD over 10 years, similar to NASA level funding http://www.artificialbrains.com/darpa-synapse-program
European Union Human Brain Project, includes 24 countries	Basic research into neurobiology, genomics and neuroanatomy. [651] https://www.humanbrainproject.eu
Institute For Brain Science, (Paul Allen Foundation in Seattle)	Basic research into neurobiology, genomics and neuroanatomy. http://en.wikipedia.org/wiki/Allen_Brain_Atlas
NCAP Neural Computation & Adaptive Perception Canadian Institute For Advanced Research CIFAR	Canadian govt. funded basic research into neuroscience and artificial neural networks, [652] http://www.cifar.ca/neural-computation-and-adaptive-perception-research-progress
Baidu	Largest Asian search engine world-wide <ul style="list-style-type: none"> • Investing in large Deep Learning (DL) SW/Servers/HW world-wide • Institute of Deep Learning Silicon Valley 2013 • Investing heavily in large data centers • Marrying cloud + device DL together • Hiring key AI researchers • Licensing their DL technology to with mobile device companies http://technews.co/2014/08/01 MEDIATEK-baidu-developing-super-smartphones/
Apple	Largest computer company in the world, making major investments. <ul style="list-style-type: none"> • Apple currently offers the Siri voice recognition service using deep learning methods, • Working on visual recognition products as well. • Working on analytics products for their product lines to understand their customers better
Google Brain	Largest search engine company <ul style="list-style-type: none"> - Invests hundreds of millions of dollars annually in AI related research - Visual search, user analytics, and speech recognition, more - Related areas, robotic cars, Google Glass, more
Facebook	Major artificial intelligence laboratory, huge investment (\$ amount unknown)
Microsoft	Several projects including voice recognition, visual search, and various SW API's and libraries to enable developers use NC for general analytics, computer vision, speech recognition
Startups and VC funding	Several robotics, analytics and computer vision startups are being funded to apply deep learning to practical (and sometimes faddish) applications

Key Technology Enablers

Certainly, AI and machine learning methods have benefited and come into widespread use due to the technology available today, which has enabled a vast army of researchers to explore new concepts and improve methods. With the rise in compute horsepower, deep learning and neural network research has proliferated.

Some of the main technology enablers propelling advances in machine learning are:

- Compute power increase:** Commodity Graphics Processors (GPUs) available on laptops and desktops, as well as multi-core CPUs, provide huge SIMD and SIMT compute power that is well matched to large parts of the machine learning and vision pipeline. Note that in some cases, multi-core CPUs perform equivalently to a GPU, depending on the exact specifications of each compute unit. See Chap. 8.
- Programming language advances:** GPGPU programming languages such as CUDA and OpenCL have opened up direct methods for algorithm acceleration utilizing the SIMT methods and the SIMD instruction sets of CPUs and GPUs. See Chap. 8.
- Memory size increase:** with much larger main memory and cache memories, the large training data sets used in machine learning are, in many cases, easy to fit into fast memory.
- Training data size increase:** The Internet has made it possible to find and collect large labeled datasets on most any topic, such as pictures labeled as dogs or cats for example hosted on picture sharing sites. *Scraper scripts* can scour internet web sites to collect images with the desired labels, or by using search engines to find the images. Large labeled data sets are usually required for effective machine learning.

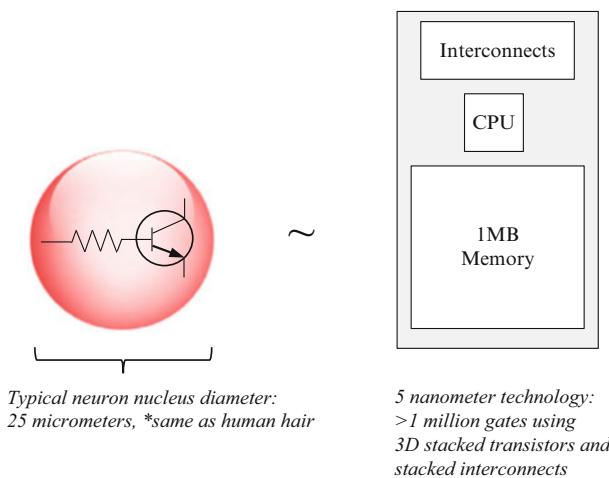


Figure 9.7 The figure illustrates how the 3D volumetric space inside a neuron nucleus can easily contain the equivalent of a small 25,000 transistor CPU and 1 MB memory using sub-5 nm silicon technology, stacked transistors, and stacked interconnects

Neuroscience Concepts

Since much of computer vision is inspired by neuroscience studies regarding learning mechanisms and the human visual system, we provide a high level overview here of selected neuroscience topics, with a focus on the visual pathway from the eye through the visual cortex. It is believed that each neuron performs pattern matching, internal processing, control of neural network connections, and memory storage management. The brain contains perhaps 100 billion neurons (*estimates vary*), each of which may vary in diameter from 4 to 100 μm (0.00015–0.004 in.), typically 25 μm for the nucleus alone (~0.001 in., or the diameter of a human hair). The volume of the neuron contains enough room for a substantial amount of processing, networking, and memory apparatus. By comparison with sub-5 nm silicon technology, a typical neuron contains enough room in the 3D volume for millions of gates to implement a small computer at least as powerful as an ARM/68000 class processor (~25,000 transistors) with perhaps 1 Mbyte of memory, and another ~25,000 transistors for interconnects, assuming 3D stacked transistors and interconnect methods, since a neuron is a 3D volumetric shape and several silicon stacks could fit inside as shown in Fig. 9.7.

Each neuron is connected to 10,000 other neurons on average, making over 100 trillion connections to control all memories and learned behaviors [858], compared to the estimated 200–400 million stars in the Milky Way galaxy. *By comparison, state-of-the-art artificial neural networks may contain over ten billion parameters, and require more than a week to train for a single dataset using thousands of CPUs and GPUs.* If we consider that each neuron may perform a mere 1 MOPS (ops per second) and has 1 MB storage, then the neurocortex contains perhaps 100 \times more MOPS per second and memory cells than the number of stars in the milky way.

Neurons represent both memories and behaviors. Biological neural networks are dynamic and adaptive: the neurons grow and shrink as they learn or forget, and new dendrites are formed to connect the neurons to each other in seemingly unlimited topologies, and the dendrite connections grow or shrink over time as well. The fragile nature of biological neural networks seems to corroborate the wisdom of Solomon “*lean not on your own understanding.*” We know that neural processing functions, memories, and connections grow like a plant, based on environment, circumstances, and experience, *so even DNA clones of a biological organism will develop biologically different neural topologies and traits if grown in a different environment.*

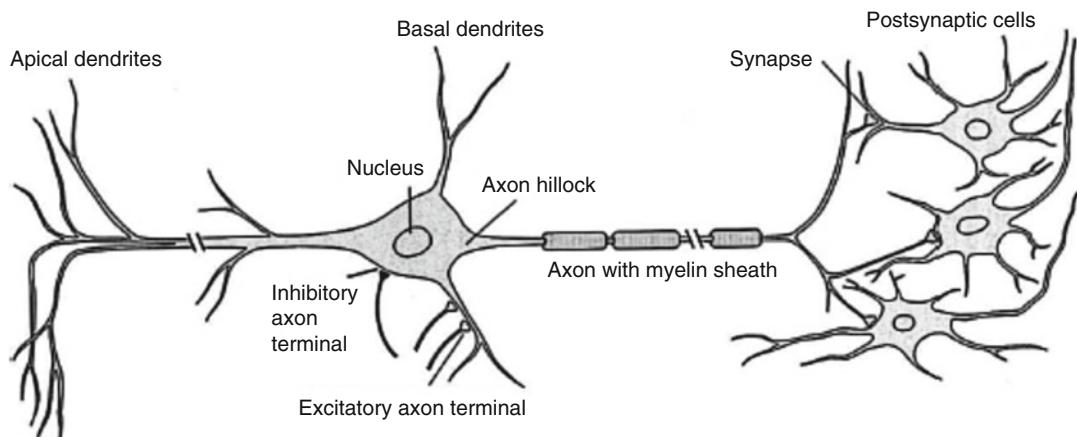


Figure 9.8 Neurons and dendrites. Image © Springer-Verlag, from Hierarchical Neural Networks for Image Interpretation, Sven Behnke [548], Draft submitted to Springer-Verlag Published as volume 2766 of Lecture Notes in Computer Science ISBN: 3-540-40722-7

Biology and Blueprint

There are basically two forces at work in the visual pathway of the human mind which inspire artificial neural networks: *biology* and *blueprint*.

The *biological structure* of neural networks is becoming more understood, being a complex electrochemical machine of connected neurons, the central processors of our nervous system. Researchers can measure the electrical activity in the brain to map out the various neural regions where processing occurs for sensory tasks, such as visual learning tasks, and where the optical nerves are connected. See the Human Connectome Project [686] for the latest images and simulations of neural pathways and Fig. 9.1. The Human Connectome Project uses various imaging modalities, similar to very fast MRIs, to image the neuron states and measure the electrical impulses when the neurons fire across dendrites to understand neural activity [631]. In some cases, neural imaging can reveal if a subject recognizes the object they are viewing, and whether or not they are telling the truth [852, 853].

From neuroscience, we know that neurons process inputs using a *trained* activation function which fires as needed, taking inputs as well as bias from electric-chemical stimulus. Apparently, the activation function for each neuron is developed over time via learning and experience. We know that neurons fire, or *activate*, in a binary or all-or-nothing fashion. We know that the neurons have some sort of memory for the input patterns they recognize, and memory for the concepts and perceptions formed in the higher level reasoning sections of the brain. But how does the memory work biologically? How much information can be stored? When is memory forgotten? What activates memories that have apparently been forgotten? How can neurobiological memory models guide the construction of artificial neural memories?

Nobody really knows the *blueprint* for the brain, except that it is encoded in DNA. As with the DNA programming code that genetic engineers are only beginning to understand, deliberate intelligence is evident in the design of the human visual system also. DNA is a programming language and that creates a learning machine—the brain. Two identical humans, twins or clones, can share the same DNA, yet their *learning experiences* determine how the biological neural network will grow in each one: the neural biology will turn out to be different, therefore each one develops in a different direction. There is some debate about how much neurology is learned vs. innate. Is a baby born with neurobiology pretrained to recognize its mother? Research suggests that DNA can be *imprinted* by our ancestors, genetically encoding predispositions towards disease, as well neurological predispositions—*perhaps visual memories and learnings are passed to us at birth*. Basic aspects of vision seem to be innate or else humans are predisposed to learn them, such as depth perception, size, texture, color, gradient detection, and shape.

Humans can discover the basic meaning of a scene in about 100 ms, and find specific targets in 150 ms as shown by Metin and Frost [626]. The visual system operates using very high dynamic range optimizations for color, gray scale, and various lighting conditions, taking input from rods and cones in the retina, able to evaluate a scene using a number of hypotheses to check assumptions and locate specific visual information under several robustness and invariance criteria (see Fig. 5.2).

And nobody really knows exactly how a neuron works, how neurons process inputs, why neurons fire, and how neurons learn. Nobody really knows how the connection topology between neurons is directed to form and grow as learning occurs, what causes neurons to grow and shrink, and what causes new dendrites to grow and connect to other neurons. And nobody really understands the center of the conscious spirit that directs the entire system.

Neuroscience has inspired researchers to develop *artificial neural networks* (ANNs), using simple models and simple assumptions, to mimic the biology of the human brain, while simultaneously guessing about the actual blueprint of how it really works. Thus, we say that the structure, or architecture, of ANNs is *biologically inspired*.

The Elusive Unified Learning Theory

According to neuroscience, the human brain is composed of several different interacting regions, where each region is dedicated to different tasks. For example, the five senses (visual, hearing, taste, touch, smell) are processed across separate and sometimes overlapping pathways across the brain, see Fig. 9.10. Emotional and rational thought are processed in separate regions of the brain as well, for example speech and vision have dedicated neural processing centers and unique neurobiology. However, since the underlying neuron biology looks generally the same in all regions, many researchers have speculated that the neurons are simply waiting to be trained according to a *universal learning mechanism*.

Metin and Frost [626] tested this universal learning hypothesis by rewiring the cortex of test animals, for example to swap the visual nerves into the audio cortex, and found that the rodents did in fact still learn to see using their audio cortex region instead of the visual cortex region, although the visual abilities were slightly impaired. Other similar experiments have corroborated such results. This universal learning hypothesis has inspired researchers to develop artificial learning models that can learn all types of information. As a result, ANNs are often applied, with little additional work, to different types of learning domains such as text, speech, and vision. Roe et al. [627] also rewired the visual receptive fields into the auditory pathway of ferrets and found that visual learning was accomplished.

In the 1960s, noted neuroscientist Bach-y-Rita postulated that *we see with our brain, not our eyes*, and his later research [628, 629] proved the concept by demonstrating a sensor for the tongue connected to a video camera, that allowed a blind person to be trained to see with their tongue, see Fig. 9.9. Subsequently, several commercial products have been developed along this line.

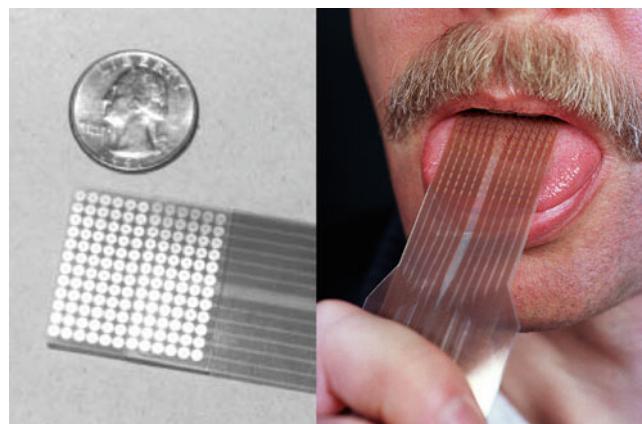


Figure 9.9 A set of electrodes representing pixels originating from a camera, which can be placed on the tongue, used to train blind people to see. Image © University of Wisconsin-Madison, used by permission

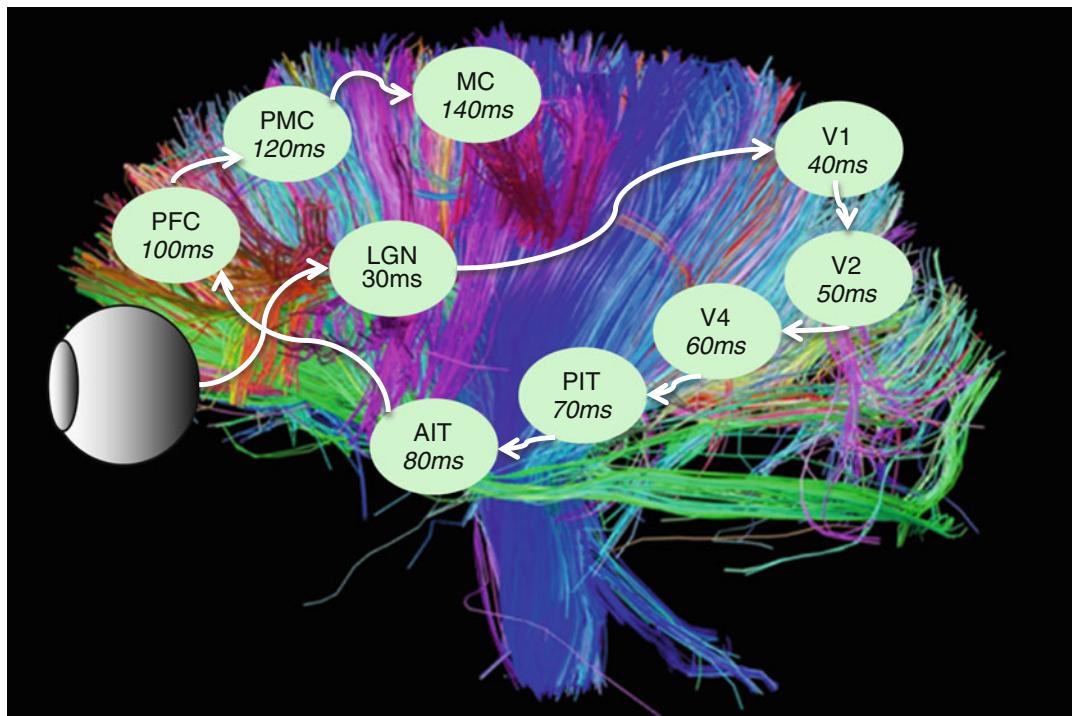


Figure 9.10 “Courtesy of the Laboratory of Neuro Imaging and Martinos Center for Biomedical Imaging, Consortium of the Human Connectome Project—www.humanconnectomeproject.org.” This Connectome Image has been overlaid with the visual pathway regions including approximate cumulative travel time between regions. Connectome images are maps or wiring diagrams of connectivity pathways, captured *in vivo* using multiple neuroimaging modalities

Human Visual System Architecture

It is useful to explore the human visual system from a *neurobiology* viewpoint, to enhance the discussion of artificial neural networks. So we provide a quick overview highlighting some of the concepts that inspire the computer vision related neural networks. ANNs *mimic* a few key of the biological structures and programmable behaviors of the brain that are observed by neuroscience. We will highlight key neurobiological structures and programmable behaviors of the brain in this brief overview. Additional background and references on the human visual system, particularly the spectral response of the eye from a pure illumination perspective, is provided in Chap. 1 on imaging. An excellent reference text for the human visual system is Kandel et al. [623].

A standard research model of the visual pathway has been developed by neuroscientists called HMAX, and many researchers have extended the basic HMAX model which we survey in Chap. 10. From the computer vision community point of view, local feature descriptors and more recently CNNs have been popular, and HMAX has been popular in the neuroscience community.

A simple model of the visual system is shown in Fig. 9.10, where the retina sends images to into the visual cortex. A good overview and reference on the history of visual cortex mapping is provided by Yeo et al. [631]. The knowledge of the neurobiological activity of the visual pathway is being filled in little by little, for example shift and size invariance in the visual pathway have been explored by Wiskott et al. [630]. Many more examples can be cited.

As shown in Fig. 9.10, the human visual pathway uses about six levels in the hierarchy (LGN, V1, V2, V4, PIY, AIT), and shares the higher level reasoning centers (PFC, PMC, MC). Various machine

learning practitioners have also found good results using close to six layers as well. According to Yann LeCun, there are between 5 and 10 separate layers in the visual cortex pathway, depending on what is counted. As a first approximation, the visual pathway resembles a feed-forward network, or FNN. However, the actual processing architecture, including feedback mechanisms and electrochemical stimulus algorithms, is unknown. There is some feedback in the visual cortex, and the processing is not all feed-forward, see Rao et al. [624].

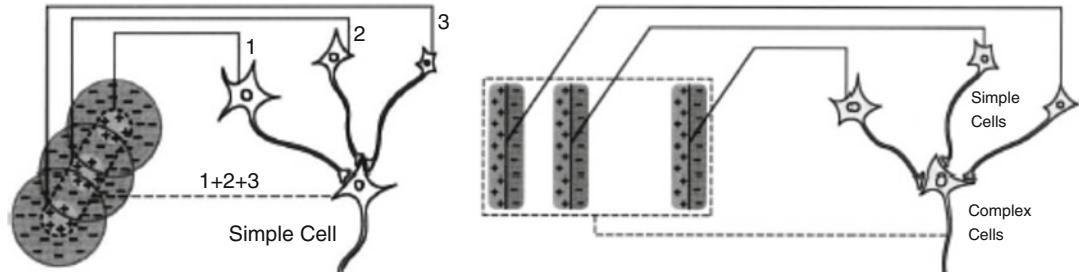


Figure 9.11 The Hubel and Wiesel model of simple cells and complex cells. Image © Springer-Verlag, from Hierarchical Neural Networks for Image Interpretation, Sven Behnke [548], Draft submitted to Springer-Verlag Published as volume 2766 of Lecture Notes in Computer Science ISBN: 3-540-40722-7

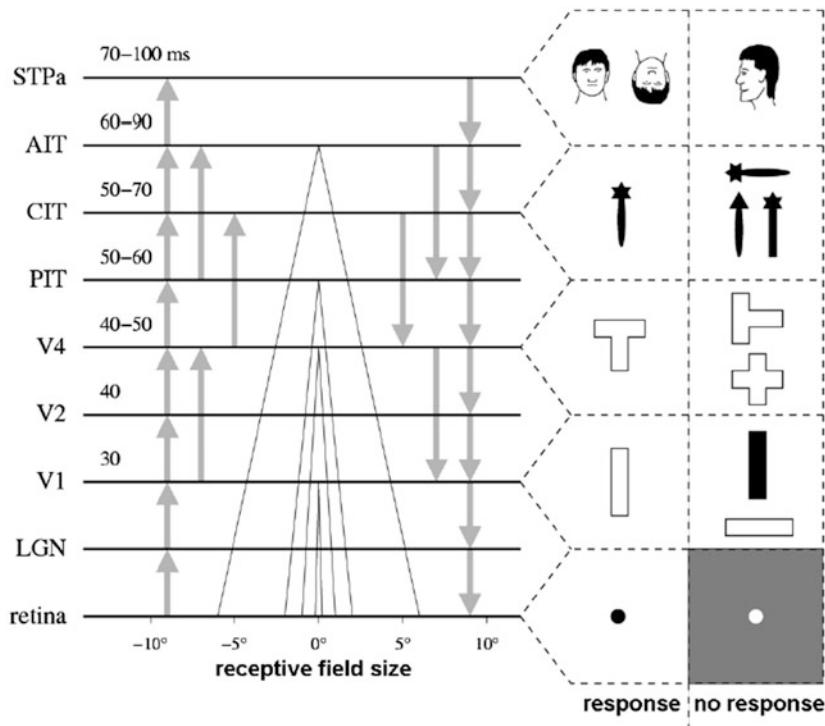


Figure 9.12 An illustration of the visual processing pathway (“standard model”), including the types of features represented in the visual neural region hierarchy. Image © Springer-Verlag, from Hierarchical Neural Networks for Image Interpretation, Sven Behnke [548], Draft submitted to Springer-Verlag Published as volume 2766 of Lecture Notes in Computer Science ISBN: 3-540-40722-7

As shown in Fig. 9.11, Hubel and Wiesel [559, 560] developed influential models for the lower levels of the visual pathway and introduced several concepts which have guided the development of *hierarchical artificial neural networks*. The basic concepts include (1) *simple cells*, which collect inputs from a local receptive region, and tune themselves to recognize oriented local features like gradient patterns or edges, and (2) *complex cells*, which pool and select the best activations from local receptive fields of simple cells.

As we summarize neurobiological architecture of the visual pathway below, please refer to Figs. 9.10 and 9.12 as we go along.

- **LGN, Lateral Geniculate Nucleus**, studied by Hubel and Wiesel [559, 560], who proposed that the LGN is a critical part of the visual cortex, and as shown in Fig. 9.12 composes the smallest aligned features from small, overlapping concentric regions in *local retinotopic fields*, and is the beginning of a *serial, hierarchical visual processing pathway*.
- **Localized Neural Field Interactions**, several neuroscience researchers [632–635] have observed that the local neural fields (neurons close together dealing with adjacent stimulus) interact with each other, similar to lateral controls between themselves. For example, the V1, V2, V2, and higher level components in the visual pathway combine local region inputs, and even scatter inhibitory and excitatory signals laterally across the local regions to change the perception parameters in the local region.
- **V1, S1**, receives inputs from the overlapping LGN local retinotopic cell regions, and forms *simple cells* (S1 cells) or *low-level features*, which have some amount of invariance to translation and rotational orientation. The outputs of the simple cells are therefore *aligned* with respect to rotational orientation. The V1 also receives backpropagated signals from the V2 area, apparently to provide guidance for constructing the local receptive fields into features. As postulated by Hubel and Wiesel, simple cells contain local features like edges and micro-textures, which are sent up the visual processing hierarchy as inputs to the *complex cells* or C1 cells.
- **V2, C1**, composes the low-level S1 features from the V1 into *mid-level features* or C1 complex cells. V2 also *pools* local features from a retinotopic region, as shown in Fig. 9.12. The C1 cells combine the outputs of aligned, overlapping retinotopic regions from V1 by local retinotopic region pooling, similar to overlapping feature kernels used in convolutional networks, where each feature map is independently sensitive to phase, translation and rotational orientation. The C1 complex cells combine the simple S1 cell features in a *phase invariant manner*, responding to edges and bars. The C1 receptive field is perhaps $2 \times$ larger than S1. Note that this concept of many simple cells, or low-level features such as edges, and local retinotopic pooling of features feeding into higher level cells such as mid-level features like motifs and object parts, is one of the key inspirations for CNNs.
- **V3**, some researchers have defined an additional low-level oriented edge feature layer as V3, which is similar to but higher level than V1 and V2.
- **V4**, here, mid-level concepts are assembled, such as motifs and object parts. The C1 cell outputs from the V2 are combined in V4 into higher level concepts, using the same types of feature map response pooling, as done in V1 and V2. So we see a common architecture among V1, V2, and V4 levels of the hierarchy, that is also another significant inspiration to CNN design, and is usually implemented as *multiple replicated hidden layers* composed of filtering, pooling, and a nonlinear neuron activation function. The nonlinearity of the data, and the nonlinearity of the neural transfer function of the ANNs is another method inspired from biology, since at each layer in the V1, V2, V4, and higher levels, we are moving farther and farther away from pixels, and more and more towards abstract concepts which are not pixels at all.
- **PIT, CIT**, contains mid-level concepts, such as directionality, motifs, hidden-layer architecture.

- **AIT, STPa**, contains higher-level concepts, such as object parts, hidden-layer architecture.
- **PFC, MFC**, contains classifier layers used to reach conclusions, make judgments, and make decisions. May also generate new hypothesis and corresponding neural programming and classifier programming, and direct multiple-hypothesis evaluations.
- **Hypothesis**, Each additional hypothesis, as directed by the higher-level consciousness of the brain in MFC and similar regions, requires a round-trip through the visual pathway to evaluate. The electrical signals in the visual pathway have been measured [626] to take about 100–150 ms per hypothesis.

Now that we have briefly surveyed the neurobiological architecture of the visual pathway, we will next summarize how ANN implementations typically translate neurobiology research into actual implementations.

- **Receptive Field Size**, the LGN sets the initial *visual receptive field size*, which is partially determined by the optic nerve architecture which arranges the impulses from the retina for transmission to the visual pathway. ANNs often represent the receptive field as the *correlation template window size*, or kernel size. It is not clear what the size for a receptive field actually is; however, CNNs typically use various sizes, ranging from 15×15 down to 3×3 , at different layers of the network, using empirical guidance based on the expertise of the practitioner. The shape of the receptive field is also unknown; however, the best local feature descriptors, such as FREAK and SIFT, use a notion of circularity to shape the receptive field, FREAK actually uses a circular region, and SIFT circularly weights the rectangular field.
- **Local Receptive Field Overlap**, the use of *local overlapping receptive fields* that are pooled together is inspired by the LGN cortex layer and the V1, V2 and V3 layers. The receptive field, or kernel, is scanned across the input field at some stride, such as at each pixel or striding every n pixels. Each hidden layer may have a different receptive field size.
- **Pooling, Subsampling**, the LGN, V1, V2, and V3 layers apparently pool features in a local receptive field to find the best feature for the current hypothesis under evaluation, to find the best one in the pooled local region, for example using *max pooling* or the strongest feature matched in the pool. Pooling can reduce noise effects. Feature subsampling resolution in ANNs, which involves striding the kernel window across the image, and the region size of the local pool, becomes lower resolution as the visual processing hierarchy moves upward to higher level concepts. As reported by several neuroscience researchers [632–635], biological neurons exhibit interesting programmable behaviors across local regions emulated in ANNs, such as *winner-take-all* or *max pooling, subsampling, automatic contrast gain control and other nonlinear numeric conditioning, hierarchical concepts, and noise suppression*.
- **Weight Sharing**, in CNNs this is simply sharing the same filter weights to use with each sliding window input kernel (i.e., kernel-connected layers) as an implementation convenience. Actual neurons apparently contain their own unique memory cells, so weight sharing is a purely computer vision concept. Compared to fully connected layers, kernel-connected layers are far more efficient in terms of compute and memory. This allows each artificial neuron to share the same parameters, which can be said to mimic the evident sharing of memories among neurons. Weight sharing also reduces the number of parameters in the network, enabling a simpler design using a replicated convolutional neuron design pattern.
- **Hidden Layers**, the visual pathway sections are implemented in CNNs using replicated functionality in V1, V2, V4 hidden layers, arranged in a feed-forward concept hierarchy, from low, through mid, and up to high-level features or concepts.

- **Feature maps**, are used as intermediate memories in the CNNs to record the response of each feature to the input. One feature map records the response to each filter.
- **Learning**, the mechanisms of neural *learning and training* are harder to understand, and less is known about how neurobiological learning really works. As discussed earlier, some amount of preprogrammed learning may be encoded genetically into the DNA (See Appendix F on Visual Genomes). CNNs typically use a variant of back propagation learning via gradient descent and similar methods, discussed in Chap. 10.

However, the DNN approaches are working well in terms of accuracy and performance in comparison to humans in limited circumstances. Many examples can be cited, such as the German traffic sign recognition benchmark competition won Ciresan et al. [602] that actually surpassed the accuracy of a human expert. In other work by Cadieu [617], as shown in Fig. 9.13 DNNs rival primates for visual recognition tasks, and the DNNs tested were actually faster and more accurate than primates as shown in Fig. 9.13. In Cadieu's work, humans and monkeys were tested for image recognition, and only given 100 ms to make a first guess and identify an image (no training or prior knowledge). The 100 ms time limit is in keeping with the visual pathway hypothesis turnaround time in humans [617] from LGN through AIT, see Fig. 9.12.

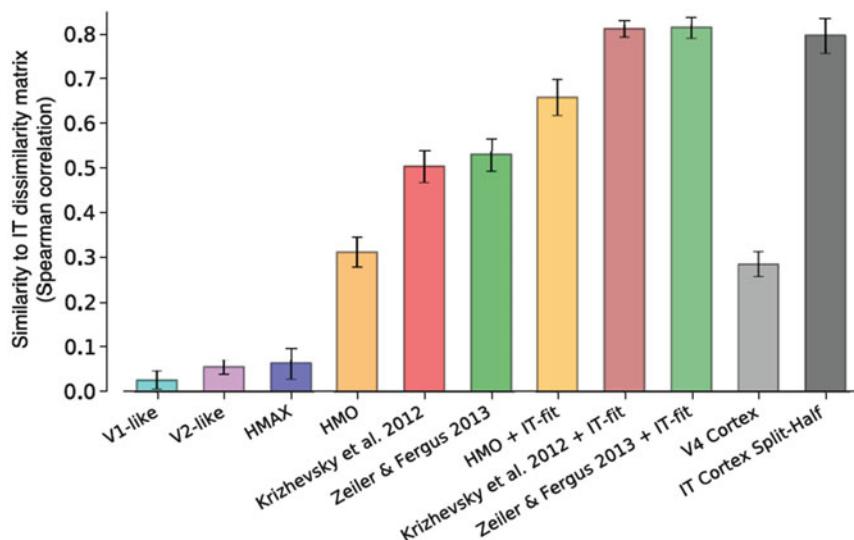


Figure 9.13 DNNs compared to actual primate visual response accuracy (V4 Cortex and IT Cortex Split-Half), from Cadieu et al. [617]. Note that DNNs rival primates according to the metrics used in the tests. The paper reports that DNNs rival the representational accuracy of primate IT cortex. Image used by permission, published under the Creative Commons Attribution (CC BY) license

In summary, neurobiology has revealed architectural concepts that have inspired machine learning, and it is remarkable that ANNs inspired by neurobiology seem to perform very well, even though the exact methods used seem to defy mathematical modeling and rational explanation in some cases. Indeed, much of the progress in ANN research seems to be due to perseverance of the practitioners, and tricks of the trade [654]. Few of the key elements of ANN design and architecture are mathematically modeled satisfactorily, and require a combination of trial and error, expertise, and good fortune to apply.

Taxonomy of Feature Learning Architectures

The taxonomy is intended to sketch out the overall architecture and components which have been used in DNNs, to provide a basis for comparison between DNNs. The taxonomy summarizes connections, layers, components, and algorithms. The taxonomy consists of:

- **Architecture Topologies**, connections and graph structure
- **Components**, algorithms and layers

For example, architecture topology defines the connections, data flow, depth of the network, and overall organization, while the components comprise the algorithms and methods used in each layer of the architecture. Table 9.3 contains the Taxonomy Summary, followed by a quick introduction to each the taxonomy component elements.

The taxonomy also shows that image processing, computer graphics, and media processing methods are being applied to ANN design; however, most DNN practitioners have little knowledge of image processing, computer graphics, media processing, and how GPUs implement and accelerate fundamental operations in silicon. As a result, DNN practitioners commonly reimplement common operations with new names or use slow algorithms, such as rank filtering which they call max pooling, and the implementation of average pooling in software instead of using image processing accelerators in DSPs connected the cameras, as well as silicon in the GPU for rescaling and anti-aliasing, other examples are pointed out as we go along.

Table 9.3 Summary feature learning architecture and component taxonomy

Feature Learning Architecture & Component Taxonomy Summary	
ANN type	Training protocols
FNN (no loops)	Randomizing training samples
RNN (loops)	Jittering/translating data
BFN (hybrids)	Warping samples
Memory Model	Reflection of samples
Simple Fixed Memory	Region proposals via segmentation
Spatio-Temporal Memory	Bagging
Associative Memory, CAM	Batch
	Mini-Batch
Input Sampling	Subcategory Mining & Fusion
Region non-overlapping, tiled	Adversarial perturbations
Region overlap, n-stride or each pixel	Layer totals
Region normalization (segment likely regions)	Total layers
Shape rectangle	Feature hierarchy layers ($n \times m$)
Shape circular	Classification layers (1×1)
Shape polygon	Other layers
Pattern every pixel dense	Features, Filters
Pattern trained sparse	Correlation Template/Convolutional Filter
Spectra float	MLP
Spectra int	CCCP feature map reduction
Dropout, reconfiguration, regularization	RCL feature
Input sample dropout	Basis Function or Local Feature
Bagging	Composite (inception)
Input weight to zero	Activation, Transfer Function
Drop connection (random, sparsification)	Binary Step Function
Drop output to zero	Linear Ramp

(continued)

Table 9.3 (continued)

Feature Learning Architecture & Component Taxonomy Summary	
Noise Injection	Saturating Linear Ramp
Pre-processing, numeric conditioning	Log-Sigmoid
Mean-zero normalization	Hyperbolic Tangent Sigmoid
Local EQ Normalization	Competitive
Global EQ normalization	Softmax
Whitening	Rectification (ReLU)
PCA	Parameterized Rectification (PrReLU)
Other	ABSVAL Rectification
Feature Set Dimensions	Radial Basis Functions
Feature patch size per layer	Maxout
Features count per layer	NiN. MLP
Feature initialization	Post processing, numeric conditioning
Transfer learning	Response Normalization (local, cross channel)
Unsupervised pre-training	Divisive normalization
Random feature initialization	Local EQ normalization
Fixed basis set	Other
Layer Connection Topology	Pooling, subsampling, upsampling
Kernel Connected	Tiled pooling
Fully Connected	Overlapped pooling
Sparse Connected	Stochastic Pooling
Gaussian Connected	LWTA pooling
Other	MAX pooling
	AVE pooling
	Overlapped pooling
	GPU pixel shaders for rescaling
	Upsampling
	Global Average Pooling
	Multi-way local pooling
	Spatial Pyramid Max Pooling (HMP)
	Affine pooling (SYMNETS)
	Multiscale MAX pooling (HMAX)

Note The architecture taxonomy and component taxonomy are introduced here first in summary format, followed by more details on each element. The terminology and explanatory information are useful to understand the architecture surveys in Chap. 10.

Architecture Topologies

The architecture topology deals with the *connection structure*, otherwise known as a network or graph, between the components or algorithm sections. The architecture topology defines the graph of connections between inputs, outputs, and components. Topology is the top-level characteristic in this architecture taxonomy, providing the structure to incorporate the additional details for each component. The basic ANN architecture topologies are shown in Fig. 9.14, and discussed below. See Bengio [507] for a good review of neural network connection topologies. A foundational text is provided by Rojas [787].

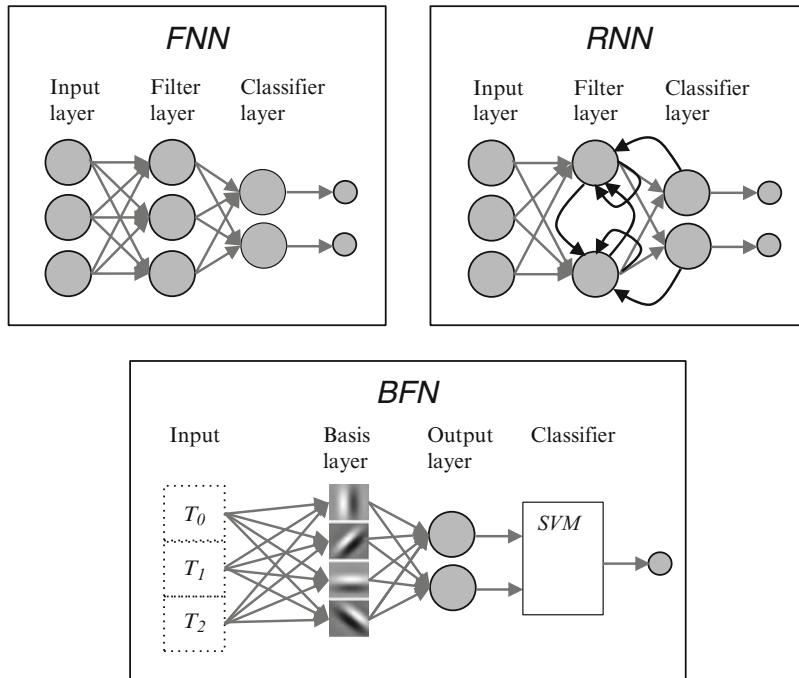


Figure 9.14 ANN architecture topologies, (top left) a Feed-Forward Neural Network or FNN, (top right) a Recurrent Neural Network or RNN, showing recursive, lateral, forward, and backward connection topology, and (bottom) a hypothetical Basis Function Network or BFN, using tiled input, basis functions such as Gabor functions, feeding into a fully connected 1D convolutional neural output layer, followed by an SVM classifier

ANNs (Artificial Neural Networks)

There is no clear pattern for ANNs to follow. Therefore, practitioners have defined a few common types of artificial neural networks that are practical to implement, being only a tiny subset of the unlimited variety of biologically plausible biological neural networks. Therefore, this taxonomy identifies three basic types of ANNs: (1) Feed-Forward Neural Networks (FNNs), (2) Recurrent Neural Networks (RNNs), and (3) Basis Function Networks (BFNs).

FNN (Feed Forward Neural Network)

Inspired by the visual pathway, FNNs provide an architecture analogous to a pipeline of replicated operations or stages. FNNs typically use a *simple memory model* which only stores the hierarchical feature set (weights) and some parameters. *For our discussion, FNNs are also a type of CNN, and use convolutional weight matrices as features and filters.*

RNN (Recurrent Neural Network)

Recurrent means loops exist in the network. In other words the network may be arbitrarily connected. However, many recurrent styles of architecture exist, such as FFNs with a few feedback loops, and Network in Network styles (NiN) with smaller ANNs inside the larger network. RNNs may also incorporate a *memory model* using temporal storage units for learning spatiotemporal sequences and related parameters. RNNs are related to CNNs and typically use the same type of *convolutional weight matrices as features and filters*.

BFN (Basis Function Network)

For this survey, BFNs are the *catch-all* category, incorporating all types of features and architectures. One of the most interesting architectures in the BFN category is the HMAX model, which is perhaps the most detailed model of the visual cortex, using basis function neuron models instead of the purely convolutional neuron feature model used in CNNs and RNNs. Often inspired by high-level reasoning approaches rather than the bottom-up neurobiology approaches, BFNs use a range of topologies and feature descriptors, a wide range of classifiers, and a wide range of architectures. For the BFN category, we consider *basis functions* to include *functional features* such as Gabor functions, Zernike Functions, Fourier features, plus all types local feature descriptors such as SIFT or FREAK.

Ensembles, Hybrids

Analogous to a room full of experts, ensemble methods combine several networks together, and perhaps use a master classification stage at the end to combine and vote on the results from each network. Hybrids may combine ANNs with other methods. We will survey a few examples of hybrids and ensemble networks in Chap. 10.

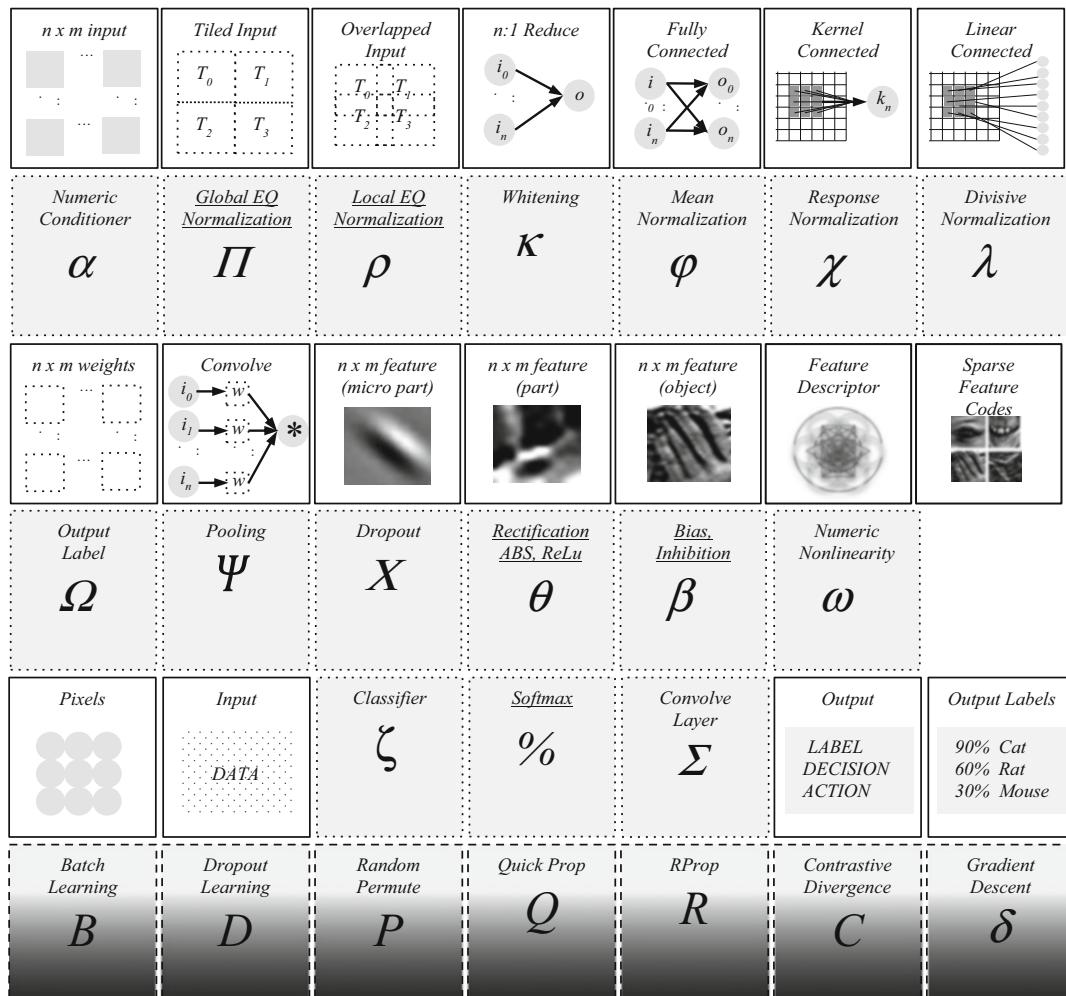


Figure 9.15 The figure illustrates selected components used in feature learning architectures; however, see Table 9.3 for the complete taxonomy

Architecture Components and Layers

The *components* are connected within the architecture as *nodes* in the graph or network, resembling a pipeline. Another way to describe a component is a *layer*. In some cases, a layer is a single function such as a convolutional filter layer or pooling layer, in other cases practitioners combined several components together into a layer. In DNNs patterns of layers are often replicated, such as convolution layers followed by pooling layers, see Fig. 9.16.

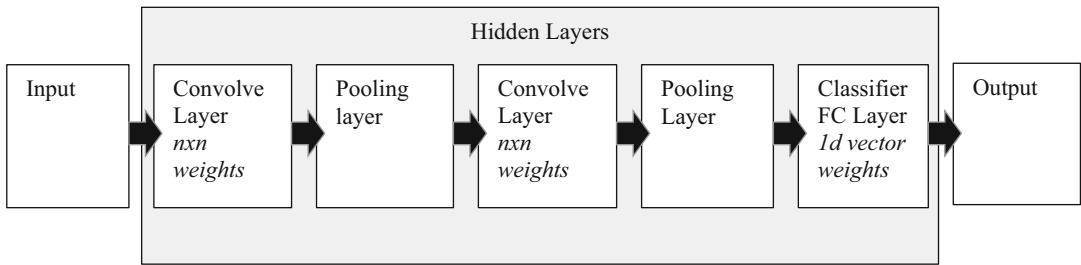


Figure 9.16 Typical CNN layers with replicated convolutional and pooling layers

Note that Fig. 9.16 shows the classifier and convolutional layers, which are referred to as *hidden layers* in DNN parlance (*note: the layers are not hidden, but any layer between the input and output is considered a hidden layer in DNN parlance*). The convolutional layers implement a hierarchy of low-level and high-level features.

In CNN parlance there is some ambiguity in terminology (*we point out terminology clarifications as we go along, apologies for repetition*). Here are some equivalences:

filters = weights = convolution_kernels = correlation_templates = features

feature_map = image = 2D array = output (filter \times input)

hidden layer = any layer between the input layer and output layer

layer = algorithm = set_of_algorithms = pipeline_stage

This ambiguity and equivalence of terms is unfortunate; however, we also follow this terminology in our discussions, since many terms are in wide use. For example, when discussing processing the input, we may refer to *features* as *convolution_kernels* acting as *filters* over the input to produce an output image or *feature_map*, and *correlation_templates* to measure the strength of the feature match for each feature over the input. When discussing back propagation to tune the *features*, we will refer to features as *weights*.

Next, we introduce and summarize the components in the taxonomy.

Layer Totals

Besides the input and output layers, the main types of layers are *kernel-connected* convolutional filtering layers, and *fully connected* classification layers.

CNNs use several replicated layers connected in pipeline fashion as shown in Fig. 9.16, where each layer contains a pipeline of processing elements, such as numeric conditioning followed by convolutions then MAX pooling. A regular architecture at each layer also supports sharing of parameters within each layer, such as feature weights, greatly simplifying implementation. Other ANN architectures are asymmetric, containing feedback loops or recurrence as in the RNN style, and are often more difficult to implement.

Layer Totals	
Total layers	The total number of layers in a feed-forward architecture, including the input layer, intermediate layers, and the output layer. Layers may include compound functionality with several steps grouped together in the layer, or else layers may be defined in a more fine-grained manner such as pre-processing layers, convolutional filtering layers, pooling layers, and post-processing layers.
Feature hierarchy layers	The low, mid, and high level features contained in different layers
Classification layers	The final layers which classify the top level features to identify higher level concepts such as objects and scenes. Classifiers may also use high, mid, and low level features together for classification, rather than just the high level features. FC-MLP ANN's may be used for classification (See Chapter 10 for FC-layers), and also statistical methods such as SVM's may be used (See Chapter 10).

Layer Connection Topology

Each layer in a typical CNN is connected to other layers in a potentially different input/output topology. For example, the input layers in convolutional networks are typically assembled into $n \times n$ kernel patterns containing *2D templates or patterns*, and each kernel is *kernel connected* to a (virtual) separate artificial neuron for convolutional processing. *Fully connected* layers are typically used in the last 1D classification layers, although in typical DNNs the full connectedness is always *forward* and not a fully connected mesh, for example connecting each feature into all neurons (See the FC layer discussion in Chap. 10). *Sparse connected* layers are found in RNNs supporting feedback, and also feature descriptors may use sparse statistical sampling patterns to group inputs, similar to the patterns used in ORB, FREAK, BRISK and other local binary descriptors discussed in Chap. 4. Sparse connections are also created using various dropout methods discussed in this taxonomy. Actually, from neurobiology we see that the entire neural network seems to be sparse connected with amazing variation.

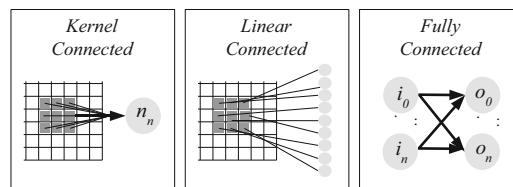


Figure 9.17 Fully connected vs. kernel connected vs. linear connected patterns. (*Left*) shows a 3×3 kernel with nine inputs connected to a single neuron, (*center*) showing a sparse linear connected 3×3 spreading pattern, and (*right*) a fully connected arrangement where each output is connected to each forward input

Layer Connection Topology	
Kernel Connected	Input regions are shaped like rectangular kernels
Fully Connected	All inputs are connected to all outputs
Sparse Connected	Interlayer connections are not dense

Memory Model

Several types of memory models are used in ANNs.

- **Simple, fixed memory:** typical CNNs use a simple model of fixed memory, storing a fixed number of features at each layer, and using dynamic parameters at run-time.
- **Spatiotemporal memory:** RNNs, such as the LSTM style networks [LSTM], have a *spatiotemporal* function to learn sequences of features or events, and therefore use a memory model which includes time-based memory management functions and *gating mechanisms* to control memory content, memory R/W access, and store a history of events.
- **Associative memory, CAM:** some ANNs, particularly RNNs, make use of dedicated memory units, known as content-addressable memories (CAM), or associative memories, so instead of using a memory address to access memory contents, the contents are accessed via a key which references the memory content.

Memory Model	
Simple Fixed Memory	Memory is allocated and used for the duration
Spatio-Temporal Memory	A type of local memory which is used for a period of time, may be implemented using RNN's
Associative Memory, CAM	Memory with a tagged or hashed memory address rather than a linear numeric address, may be implemented using RNN's

Training Protocols

Training protocols are concerned with methods of data preparation and presentation, such as creating rotated or cropped copies of the training data, and presentation of data in batches. The goal of the training protocol design is to ensure that sufficient samples and randomization of the training samples is provided to train the system, and many variations are in common use.

Training protocols	
Randomizing training samples	Grouping the training samples in a random order for each training run.
Jittering/translating data	Shifting the images
Warping samples	Warping the images
Reflection of samples	Reflecting or mirroring the images
Region proposals via segmentation	Segmenting the input images into regions, and sending each region for training as a separate image
Batch, randomization	All training images are sent through in a group, and then training parameters are changed to account for the entire group. The groups may be randomized in content or size.
Mini-Batch	Like batch, except that the training set is broken into smaller batches of images for training one batch at a time, and all weights are adjusted after the entire batch is run.
Subcategory Mining & Fusion	Intelligently grouping similar sets of images into training sets based on data mining analysis of the image content, rather than randomly creating training set batches, see Dong CVPR 2013
Adversarial perturbations	Deliberately designing adversarial images which intended to disturb and misclassify, see Fawzi CVPR 2015

Input Sampling Methods

Input to each neural layer is sampled using a wide range of methods:

- **Regions**, includes tiled nonoverlapping regions, or strided, overlapping regions, such as at each pixel or every n pixels.
- **Shape**, typically rectangular patches are sampled, such as 3×3 kernels, but could be rectangular or circular.
- **Pattern**, in CNNs the pattern is typically a dense pattern, using all values in a 3×3 kernel region for example. However, some local features, such as FREAK, use a dithering or saccadic style sampling pattern to provide more resolution and detail, modeled after the human retina as it examines a particular area closely.
- **Spectra**, for pixel input, the values are scalars, perhaps floating point, fixed point, or integers. For higher levels of input in a CNN style network, the values are no longer strictly pixels, since each value has been nonlinearly transformed by the filter, as well as the nonlinearity of the activation function, plus perhaps other numeric conditioning.

Input Sampling	
Region non-overlapping, tiled	Sampling the input image in non-overlapping tiles
Region overlap, n-stride	Sampling the input region in overlapping tiles, where the stride factor determines the tile overlap
Region overlap every pixel	Dense sampling at each pixel of the input image
Shape rectangle	The sampling shape is a rectangle
Shape circular	The sampling shape is circular
Shape polygon	The sampling shape is a polygon
Pattern every pixel dense	The feature descriptor densely samples each pixel within the descriptor shape region, such as correlation templates
Pattern trained sparse	The feature descriptor sampling pattern is sparse, such as the FREAK and ORB methods.
Spectra float	Descriptor is an array of floating point numbers, this is typical of CNN weights
Spectra int	Descriptor is an array of integers

Dropout, Reconfiguration, Regularization

Various methods, known collectively as *dropout*, are used for many reasons during training, and appear as layers in the architecture. Reasons for using drop include:

- **Ignoring data samples**, to prevent overfitting.
- **Dynamically reconfigure the network input/output topology**, to ignore samples for regularization, and randomize the connection topology to regularize training.
- **Dynamically setting weights to zero**, for model regularization.

It has been shown by several practitioners that dropout methods can improve training convergence speed, reduce overfitting, and perhaps training accuracy.

Dropout is used to prevent overfitting by ignoring (dropping), or setting inputs or outputs to zero. Dropout effectively alters the neural network model, randomly setting some of the data samples to zero for each forward pass through the network, which results in a set of semi-random network

variations which are averaged together during training. Dropout yields modest improvements when applied to most any neural network model [568, 660]. Dropout is most effective during early stages of training, where the tuning parameter step sizes are larger and still converging, rather than at later training stages where the tuning steps are smaller. Note that dropout appears to be mathematically motivated, rather than neurobiologically inspired.

Random sample drops have the effect of randomizing the architecture as well, creating sparsely connected layers at run-time with different connection topologies. The number of neural inputs dropped is chosen empirically, and many practitioners choose to drop $>50\%$ for only inputs, and other practitioners perhaps drop $<50\%$ for only outputs. Other methods to drop random neural inputs or outputs, or even individual filter weights, in convolutional layers have been tried as well [539, 660–662]. Also, $L2$ regularization and adding input noise have also been applied.

Dropout is intended to make DNN training possible by addressing overfitting and failure to converge. Overfitting during training can be caused by data samples that vary quite a bit, which causes the classifier to fit a model to bad values, causing anomalies like overshoot and undershoot. In addition, neural network models using back propagation weight tuning algorithms will easily overfit to the *data variations* rather than the *data trend*, manifested as oscillations and failure to converge, which can complicate and increase training time. Also, using too few data samples can also cause overfitting, since sampling artifacts due to undersampling are well known, such as Nyquist effects. Some practitioners also add noise, such as Gaussian noise, during training to regularize the data, but not during testing. The noise is mostly filtered out during training as all the samples are averaged together.

Part of the problem with any sampling operation is noise; the data samples by definition may not be a smooth, regular continuum of values, and instead may contain bad training samples and a non uniform distribution of values, with some strong outliers that will skew the results. Dropout is the inverse of *adding* noise to the data, and is analogous to *adding* Gaussian filtering to the data to remove noise. Dropout is intended to deal with noise or overfitting artifacts by *subtracting the noise out*. Of course, dropout is currently implemented using a random dropout mechanisms, so in fact the dropout effect is simply analogous to thinning out the data samples, which seems to work fine in practice.

In 1971, Ivakhneko's GMDH neural network model [564–566, 569], was the first deep NN-inspired model to use a parameterized data sample conditioning method to *ignore* certain data samples (an early dropout variant), which prevented model overfitting.

More recently around 2012, Hinton et al. [539, 660] popularized a drop-out method variant called *random drop-out*, to drop random training samples to prevent overfitting. One variant includes setting a *dropout mask* that is symmetrically applied to each round trip through the network: the same mask is used for both the forward pass from input to classifier, and the backpropagation tuning pass. The mask, of course, is random, and changes for each round trip. By using a value of zero for neuron output, back propagation correspondingly finds several zero-valued gradients during gradient descent, speeding up back propagation, and hopefully preventing overfitting.

Regularization is an attempt to force the models to train correctly by altering the training variables (making them more regular), sort of like Gaussian filtering or blurring, to try and eliminate artifacts manifested as “overfitting.” Data regularization methods may be considered as *nonlinear data reductions*, which either regularize data on the *input* side or the *output* side of each neural processor. Dropout regularization methods implement crude forms of *lateral inhibition* and *lateral communications* between neurons at the same layer of the hierarchy in the visual pathway of the brain [632–635].

Dropout, reconfiguration, regularization	
Input sample dropout	
Bagging	Another method used to selectively ignore data to prevent overfitting is the bagging approach, proposed by Breiman in 1994 [611, 662]. The basic idea is to divide the total training set into <i>bags</i> groups, and create corresponding models to train on each group. Training proceeds synchronously and in lock-step for all models and their corresponding bagged data sets, and the results of each model are averaged together, or else voting can be applied like a majority filter (see Chapter 2).
Input weight to zero	Setting randomly selected inputs to zero
Drop connection (random, sparsification)	A variant of dropout, DropConnect by Wan et al. [661] uses a random mask to zero out random filter weights during inputs to the neurons in the fully connected layers, instead of zeroing the outputs as per the dropout method, and Wan reports good results.
Drop output to zero	Setting randomly selected outputs to zero
Noise injection	Adding noise to samples

Preprocessing, Numeric Conditioning

Preprocessing of the data for purposes of numeric conditioning is common, for example to remove noise, squash or compress the data into a zero-centered range, or present only the most common data elements as in PCA. Several methods are applied from standard signal and image processing, see also Chaps. 2 and 3. However, it is worth noting that no numeric conditioning functions have been discovered from neuroscience research, so numeric conditioning, image preprocessing, and nonlinearities are used as ad hoc methods of compensating for other problems in the network.

Pre-processing, numeric conditioning	
Mean-zero normalization	Normalizing the data in a range about zero, such as [-1 … 1]
Local EQ Normalization	Normalizing the input data region using histogram equalization
Global EQ normalization	Normalizing the entire input image using histogram equalization
Whitening	A decorrelation transform that transforms a vector of variables into an uncorrelated vector of variables having a variance of 1, so the input then becomes like a white noise vector
PCA	Principal Component Analysis methods find the most common values in a distribution, similar to cluster centers, useful for boiling down the values to a smaller representative set
Other	Most any image processing operation

Features Set Dimensions

The feature set dimensions includes the number of feature sets in the hierarchy, and the number of features per set. Each architecture typically fixes the number of features per layer, perhaps increasing the number of features in higher layers.

Feature Set Dimensions	Feature Set Dimensions
Feature patch size per layer	In a CNN, each layer in the feature hierarchy may assign a different x,y dimension to each feature, such as 3x3 .. 31x31 or larger.
Features count per layer	In a CNN, each layer in the hierarchy may define a different number of features, such as 200. Other methods such as 1d vocabularies and FC layers may use thousands of features in a 1d single layer representation.

Feature Initialization

The method of initializing the features for a CNN will affect the final training results. Running the same training data over features initialized slightly differently will lead to a different feature set, likely due to the different local minima in each set. Many practitioners have used random initializations, and in some cases it is beneficial to use transfer learning and start learning from a set of preexisting trained features. In other cases, an unsupervised pretraining session is run to build up a feature base, and then supervised training is used to refine the features.

Feature initialization	
Transfer learning	Use pre-existing features that have already been trained, perhaps use only the lower layers.
Unsupervised pre-training	Train the network using unlabeled data, and then start a training session using labeled data to refine the base features.
Random feature initialization	Initialize feature weights to random values, note that random value weight matrices are equivalent to a set of edge detectors.
Fixed basis set	Use fixed basis features for the lower layers, such as in the HMAX model surveyed in Chapter 10.

Features, Filters

Typical DNNs follow the CNN model and use correlation templates such as $n \times n$ weight kernels as the *features*. The weight kernels also double as *convolutional filters*, used to transform the input in a nonlinear fashion producing a subsampled reduction known as a *feature map* in CNN parlance, for input to the next layer.

However, some ANN methods (BFNs) may use basis features, or local feature descriptors such as SIFT, instead of simple convolutional filters.

Features, Filters	
Correlation Template/Convolutional Filter	A weight matrix define a feature as a template
MLP	Using a multilayer perceptron to define a feature, as used in the NiN method (see Chapter 10)
CCCP feature map reduction	Filtering a set of feature maps from a large set into a smaller set, as used in NiN and Inception. This effectively changes the feature results.
RCL feature	A recurrent convolutional layer which substitutes an RNN for the convolutional features to consider the spatial relationships in local regions via the RNN
Local Feature Descriptor	A feature such as SIFT, ORB, and other local features.
Basis Function Set	Basis functions computed over a space such as Fourier, Gabor, Zernike. The basis functions are not learned or designed, but rather generated from parametric functions.

Activation, Transfer functions

The basic artificial neural processor model used in most DNNs consists of two parts: (1) a convolution function: $s = f(inputs \times weights)$, and (2) a nonlinear activation function: $a = n(s)$ to spread or squash the data, which produces a scalar value. Several types of activations functions are used; see Hagan et al. [668] for details and Fig. 9.18.

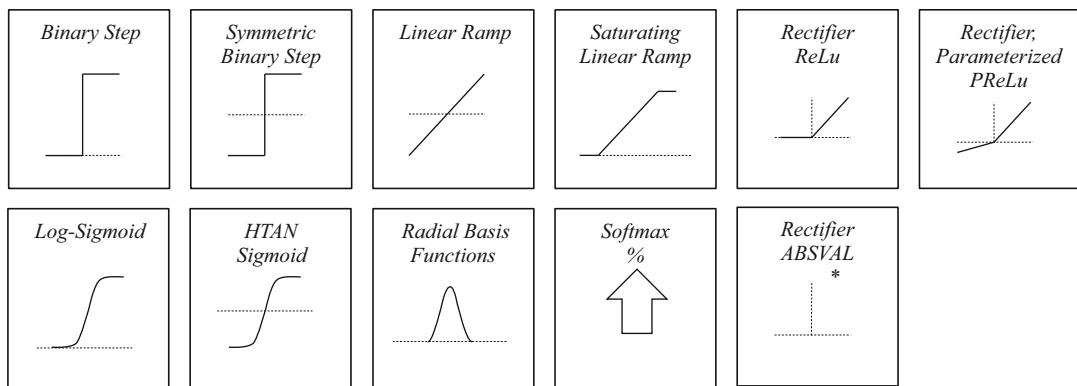


Figure 9.18 Various activation functions, which act as threshold functions, numeric range compressors, companders, or remappers

Activation functions, or transfer functions, are used to (1) introduce nonlinearity into the neural function, (2) prevent saturation of values, and (3) ensure that the neural output is differentiable to support back propagation methods using gradient descent. One key goal of nonlinear activation functions is to project the purely linear convolution operation into a nonlinear solution space, which is believed by many to improve results. In addition, the nonlinearity may result in faster convergence during backpropagation training to move the gradient more quickly out of flat spots towards the local minima.

However, the actual nonlinearity function for neurobiological activations is still an unknown function, if it exists at all. Artificial neural networks, such as CNNs, do not fire in a binary all or nothing manner as real neurons, but rather produce an *analog output* which is then numerically conditioned by the activation function and other functions such as pooling. The simple Perceptron model according to McCulloch and Pitts [561] does not use activation functions, instead using only weights on the inputs.

Activation, Transfer Function	
Binary Step Function	Uses a threshold to determine the neural activation output as a binary all or nothing value
Linear Ramp	Assigns the activation within a linear range, which may be a compressed range ramp or a full range map, see remapping in Chapter 2.
Saturating Linear Ramp	Assigns the activation within a linear range, with a saturation threshold value to cap the linear range
Log-Sigmoid	Assigns the activation using a log sigmoid style function
Hyperbolic Tangent Sigmoid	Assigns the activation using a hyperbolic tangent sigmoid function

Activation, Transfer Function	
Competitive	An activation method which couples several neural outputs together, and inhibits the losing neural outputs.
Softmax	A probabilistic activation function, which produces a percentage or strength activation rather than an all-or-nothing binary activation
Rectification (ReLU)	A rectification of output values to eliminate polarity, useful for identifying the same feature under polarity changes
Parameterized Rectification (PrReLU)	A rectified linear unit with parameterization to control the threshold of where negative polarity is determined
ABSVAL Rectification	Using the absolute value of an activation to eliminate negative polarity
Radial Basis Functions	Controlling activation by matching with RBF's
Maxout	Pooling over a range of feature map channels, see details below.
MLP (NiN)	Using a multilayer perceptron as the activation function, rather than a simpler convolutional template of weights, as used in NiN

Maxout networks, or deep maxout networks (DMNs), as proposed by Goodfellow et al. [610], are an interesting variation on CNNs using a new type of *activation function*, called maxout, which pools across spatial regions and across feature map channels from prior layers, able to approximate any convex function. The basic idea is to pool a small group of nonoverlapping neural outputs, and select only the MAX activation from the group to pass forward to the next layer, and zero out the others as in dropout. This is similar also to MAX pooling, but replaces the activation function and can take input from prior layer feature maps. The maxout function can implement various activation functions, such as absolute value rectification, and quadratic functions. The end result is a reduction of parameters for the model, amenable to smaller computers.

Post-processing, Numeric Conditioning

Some practitioners further post-process the results from the activation function or the pooling function, for example using normalization, compression, and expansion. Post-processing adds further nonlinearity, and also may correct numeric range problems. There seems to be no limit to the *other* post-processing methods in use.

Post processing	
Response Normalization (local, cross channel)	A non-linear spreading function to normalize a region of pixels or inputs, which may use local regional values, or values from other channels such as color, intensity, or gradients.
Divisive Normalization	Scales the value of a local region using a divisive factor based on the activity of a large number of nearby neurons, see D. J. Heeger 1992.
Local EQ normalization	Normalization of a local region of pixels or values using a histogram equalization function
Other	Several methods are used including whitening, rescaling, and contrast windowing.

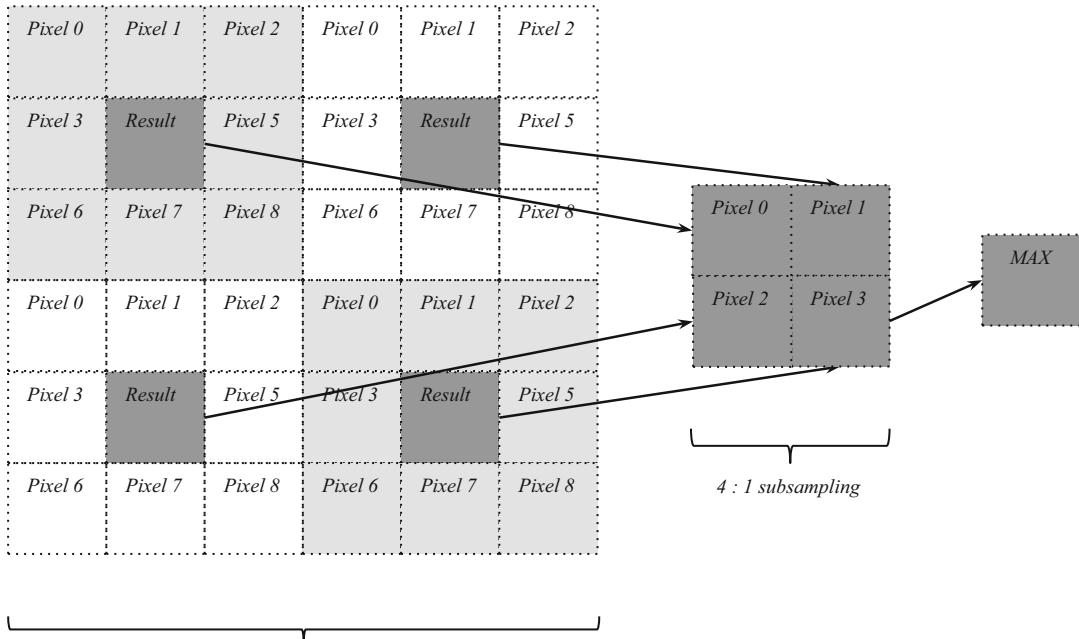


Figure 9.19 Pooling from 3×3 convolutions taken across nonoverlapping tiles, yielding a 4×1 pooling set from which the MAX value is selected. Total subsampling for the MAX pooling is 36:1

Pooling, Subsampling, Downsampling, Upsampling

Pooling is a method of combining several neural outputs consisting of the scalars from the activation or transfer function into a pool, or group, and creating a new output from the pool. Several methods are used to select the value from the pool. Actually, computer graphics methods for scaling and anti-aliasing using pixel shaders may be a preferred approach, and GPUs even provide hardware acceleration for this fundamental operation, but the author is not aware of any practitioners using GPU anti-aliasing. Upsampling is a method to regularize features for some classification schemes, see Fig. 9.19.

Pooling, subsampling, upsampling	
Tiled pooling	Pooling results over non-overlapping regions
Overlapped pooling	Pooling results over regions which overlap, adjusted using a stride factor
Stochastic Pooling	Used to eliminate sample noise to prevent overfitting by pooling over regions using semi-randomly determined activations besides the max or average value, using a multinomial probabilistic function for a region, see Zeiler [663]
LWTA pooling and dynamic connections (see Figure 9.21 below)	LWTA pooling turns off selected forward propagation of neuron values from a layer, except for the winning neuron's value. See details below.
MAX pooling	Uses the MAX activation from a group of neurons as a representation of the group.
AVE pooling	Uses the AVE activation from a group of neurons as a representation of the group.

Pooling, subsampling, upsampling	
GPU pixel shaders for rescaling	An embodiment of subsampling and pooling using pixel shaders and silicon accelerators in the GPU to implement the computations.
Up-Sampling	Interpolation of a small group of pixels into a zoomed-in larger region.
GAP Global Average Pooling	Replace the fully connected layers in a CNN for classification, by taking the spatial average of all the features in the last layer for scoring, as used in the NiN method.
GMP Global Max Pooling	Combines the strongest activations from feature maps at each layer, which feeds into a softmax classifier, as used in the RCL-RCNN method.
Multi-way local pooling	Pooling features that are similar in feature space, or spatially nearby
Spatial Pyramid MAX pooling (HMP)	Pooling the MAX value of features across a spatial pyramid resulting in a combination of features sizes, as used in the HMP method.
Affine pooling (SYMNETS)	Pooling the strongest activations over a set of similar feature activations under affine transformations as used in SYMNETS
Multiscale MAX pooling (HMAX)	Pooling over local regions of feature activations using similar scales of the same feature as used in HMAX.

As demonstrated in the Compete to Compute (CTC) approach taken by Srivastava et al. [613] the LWTA pooling and dynamic connections method is a form of *Local Winner Take All* dropout, which generates *dynamic neural connections* based on the competition between local neurons, essentially turning neural outputs temporarily off when they lose the local competition. LWTA is a form of local activation inhibition. Note that LWTA does not downsample the output like other pooling methods, but rather changes the network topology or *sparsifies* the topology temporarily. As shown in Fig. 9.20, LWTA methods first groups neurons into blocks at each layer. The blocks within each layer compete. Each block contains logic to *turn off* the output connections from the losing neurons, and turn on the output connection output from the winning neuron. LWTA is also shown to add a form of memory to the network, preventing what the authors call *catastrophic forgetting*. Neurobiology shows [228] that there is local pooling, or competition, among neurons, as leveraged in the SIFT approach which pools local gradient magnitudes in a histogram feature.

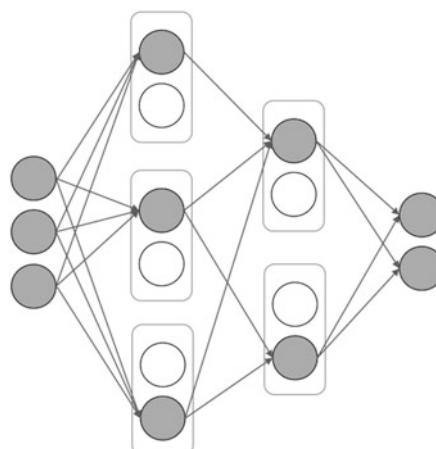


Figure 9.20 LWTA pooling and dynamic connections [613], where local groups of neurons, groups of 2 in this case, compete within their group to forward their output up to the next layer. Winning neurons (dark) forward output to the next layer, losing neurons (white) do not. Winning neurons form a dynamic network topology, which changes with input data

Classifiers

While classification methods are not the focus of this work, we include some notation in this taxonomy for classifiers, and provide some background discussions in Chaps. 4 and 10. Classification is typically the last stage of the computer vision system, where the presence and absence of features is used to make decisions (*NOTE: the Google Inception Architecture uses multiple classifiers at various layers, surveyed in Chap. 10*). The classifier matches detected patterns against learned patterns to identify the class of the input and make a score to show confidence.

An ensemble of classifiers may also be used, working in tandem. First, *a hypothesis is determined*, such as *am I looking at a dog or a cat?* The hypothesis determines which parameters and features are used to set up the classifier, using the same architecture which has been trained for several recognition tasks using several classifiers.

According to Cadieu et al. [617], the human brain *sequentially* processes multiple hypothesis. The brain may take about 100 MS to make a forward pass through its neural network for a first evaluation of a hypothesis against a new scene. After that, other passes may be made to compare other hypothesis against each other for the new scene, generating additional hypothesis in response to a question, uncertainty, or need for further analysis, and so on. Thus, the brain acts as an *ensemble classifier*, checking a hypothesis against the scene, and then perhaps changing the hypothesis and making further evaluations, and then choosing the best hypothesis.

Classifier	
FC-layer classifiers	In convolutional style networks, the classifier may be made up of 1d fully connected layers containing convolutional style neurons, which are amenable to back propagation style training, instead of traditional statistical methods such as regression models, SVM's, and kernel machines. A discussion on FC layers is found in Chapter 10.
Softmax	A softmax layer is a probabilistic representation of correspondence, similar to a percentage score, weighting the contribution of each feature to compose the final object.
SVM	Support Vector Machines (SVM's) provide a framework for evaluating multiple representations or encodings of features with various distance functions in a higher dimensional space, where features become separable to allow for distance measurements. Some background is provided in Chapter 10.
Statistical, other	Other: A wide range of regression models and statistical methods are used, see the references scattered in Chapters 4 and 10 (outside the scope of this work)

Summary

This chapter lays the groundwork for the feature learning architecture survey in Chap. 10, and should be read prior to reading Chap. 10. The neuroscience inspiration behind deep learning networks and the visual pathway is explored here, including key ideas from the history of neuroscience and artificial neural networks. The idea of synthetic vision was introduced, where complete systems are being designed to mimic the entire human visual system along the lines of other prosthetic sciences such as robotics. A brief history of the various approaches to machine learning are discussed which includes expert systems, local feature description, representational learning, and deep learning. The notion of *deep learning* systems is introduced where multiple levels of features or concepts are learned at

different scales in a hierarchy, similar to the hierarchy in the visual pathway. Key terminology is summarized, and a taxonomy of deep learning architectures is developed. The taxonomy includes three architecture families: Feed Forward Neural Networks (FNNs), Recurrent Neural Networks (RNNs), and Basis Function Networks (BFNs) to capture all other feature learning methods which do not necessarily use ANN methods to learn all the features. The taxonomy also includes a breakdown of the various design elements and algorithms used in the feature learning network layers.

Chapter 9: Learning Assignments

1. Describe several goals of feature learning.
2. Describe an artificial neuron model typically used in deep learning networks for computer vision, and describe each model component.
3. Describe, at a high level, hierarchical learning (deep learning).
4. Compare hierarchical learning to dictionary learning.
5. Describe a visual vocabulary, and discuss the types of features that can be used.
6. Describe several goals of sparse coding.
7. Describe basis features and how they are used in deep learning for computer vision, and give at least two examples of basis features.
8. Describe an expert system and how it is designed.
9. Describe the design of a local feature descriptor of your choice.
10. Compare and contrast CNN feature learning against local descriptor feature training using the ORB descriptor.
11. Compare a Feed-Forward Neural Network (FFN) with a Recurrent Neural Network (RNN) and a Basis Function Network (BFN).
12. Provide an estimate of the number of neurons in the human brain, the number of neural connections in the human brain, and the recognition speed of the human brain.
13. Describe the architecture of a convolutional neural network (CNN), including a description of typical layers.
14. Describe what is known about the architecture of a biological neuron, including the component parts of the neuron.
15. Describe the layers in the visual pathway including all layers between V1 . . . AIT.
16. Describe the higher level classification layers of the visual pathway PFC, MFC.
17. Provide a theory about how the human brain develops conceptual visual understanding and high-level visual reasoning.
18. Provide a theory about the neurological mechanism for introducing a hypothesis to the visual pathway for evaluating the visual field.
19. Provide a hypothetical model of a neuron that includes local neural memory, shared neural memory with other neurons, and enumerate the neural processing operations and parameters.
20. Discuss the local receptive field in visual cortex, and the local receptive field as implemented in CNNs.
21. Describe at least two goals for an activation function (i.e., transfer function), and provide algorithm descriptions of at least two types of activation functions.
22. Describe at least two goals of pooling and subsampling in CNNs, and compare at least two pooling methods.
23. Describe how a fully connected neural layer is organized, and how it can be used in a CNN.
24. Discuss the Hubel and Weiss model of S-cells and C-cells.
25. Discuss the concept of a hierarchy of features (deep learning), and how deep learning is inspired by neurobiology.
26. Discuss CNN training protocols including batch and mini-batch training.
27. Discuss training considerations such as number of training samples, and modifications to the training samples.
28. Describe hidden units and hidden layers.

29. Describe how a convolutional neuron model works, including the inputs, feature weights, and *at least four (4)* other possible functions in the model including the activation function and the pooling function.
30. Discuss the goals of a classifier.
31. Name at least two types of classifiers, and discuss the design and operation of each classifier.

“...the code is more like ... guidelines.”

—Captain Barbosa, Pirates of the Caribbean

In this chapter we survey a wide range of *feature learning architectures and deep learning architectures*, which incorporate a range of *feature models and classification models*. This chapter digs deeper into the background concepts of feature learning and artificial neural networks summarized in the taxonomy of Chap. 9, and complements the local and regional feature descriptor surveys in Chaps. 3–6. The architectures in the survey represent significant variations across neural-network approaches, local feature descriptor and classification based approaches, and ensemble approaches. The architecture taken together as *the sum of its parts* is apparently more important than individual parts or *components* of the design, such as the choice of feature descriptor, number of levels in the feature hierarchy, number of features per layer, or the choice of classifier. Good results are being reported across a wide range of architectures.

Feature learning architectures in the survey cover two broad categories:

1. **Statistical learning methods** using a wide range of feature descriptors, learning methods, sparse coding, and statistical classifiers.
2. **Neural network methods** inspired by a few simple neurobiology concepts, such as concentration of local receptive fields into artificial neurons, and wide and deep feature hierarchies.

This chapter surveys both historical and recent examples of deep learning architectures, especially the area of *feature learning*, where the features composing objects and the relationships among features are *learned in common architectures*. We examine the components used in each architecture, and discuss some of the motivation and intuition behind the designs.

Several detailed background sections are provided to explain key concepts for each type of architecture, such as artificial neuron models, backpropagation methods such as gradient descent, sparse coding, and visual vocabularies.

Deep learning architectures are used to *generate features* from training data under the control of a skilled practitioner—there is a learning curve. And the sheer amount of training data available and its preparation is usually the most important factor for successful feature learning. Here are some insights from skilled practitioners.

“It’s not who has the best algorithm that wins, it’s who has the most data”
Bank and Brill

“The weights are the program code”
“It’s all about compression”
Juergen Schmidhuber

Architecture Survey

The survey includes selected *representative architectures* following the taxonomy from Chap. 9. The goal is to explore the boundaries of innovation across the architecture families. Most of the architectures surveyed can be considered deep learning methods, but we also survey a few hybrids and exceptional cases. The survey focuses on architecture variations, rather than which architecture is the leader in a single benchmark. Unfortunately, we cannot survey every significant development in this field. However, we refer the reader to Schmidhuber’s excellent historical survey [552]. A good introduction to basic neural network designs, including descriptions of several influential FNN architectures, is provided by Hagan et al. [668]. In addition, we provide a list of key journals and conferences in Appendix C for the interested reader to follow the latest research.

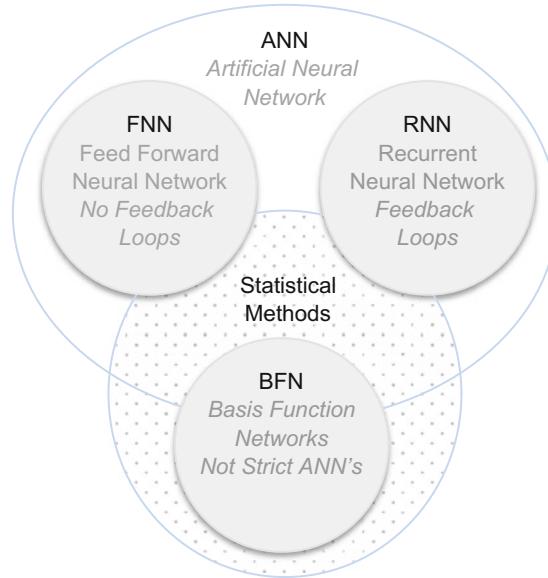


Figure 10.1 This figure shows a simple taxonomy of feature learning architecture topologies used for the survey. Note that neural network methods and statistical methods overlap

As shown in Fig. 10.1, the survey is taxonomized into three architecture families:

- FNNs feed-forward convolutional style networks
- RNNs recurrent networks, primarily convolutional style
- BFNs typically basis functions are used as the features, such as Gabor or Fourier

Artificial neural networks (ANNs, NNs) are closely related to statistics methods, and in fact can solve many of the same problems. Note that we show FNNs and RNNs as subsets of the ANN category, and BFNs as hybrids, since some BFNs incorporate a few neural network concepts, as well as statistical and heuristic methods.

Neural networks are *function approximation engines*: they learn features in a compressed, sparse manner to reconstruct their input functions, which are images or image sequences in computer vision applications. The architecture of the NN determines the ease of training and the effectiveness of the function approximation.

ANN design may be inspired by neurobiology, but the architecture and design of real systems is an art form. Actually, the leading practitioners advocate experimenting with different architectures, components, and training protocols, and comparing results to tune the architecture incrementally. Often, researchers start with an existing architecture, and then make small changes, measure the results, and publish.

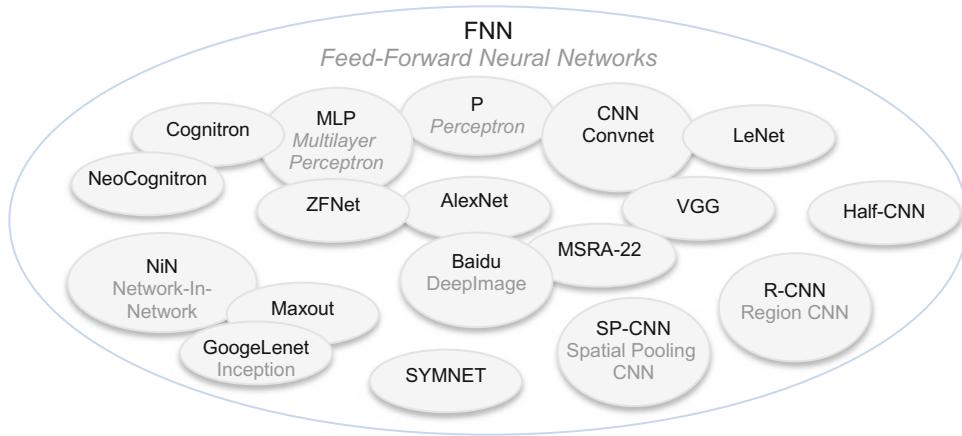


Figure 10.2 This figure shows the FNN architectures in the survey

FNN Architecture Survey

We start the architecture survey with feed forward neural networks or FNNs as shown in Fig. 10.2, since many of the first successful neural network applications used feed-forward models. In particular, we spend considerable time on a survey of *Convnets*, or convolutional neural networks (CNNs), which are implemented as FNNs. Convnets have been influential, forming the basis for substantial research, as pioneered by LeCun and others. The Perceptron, surveyed later, is often cited as the basis of convolutional models, where weight factors acting as the features are applied to the inputs to measure correspondence. FNNs have been demonstrated to work in various application domains, providing a fairly regular architecture that has been extended and refined by several practitioners.

P—Perceptron

The Perceptron model developed by Rosenblatt in 1958 [557, 563] was part of a classified artificial intelligence project, studying what Rosenblatt termed *Neurodynamics* for the US Navy, that took place during the 1950s, later declassified [604] in 1961. Reaction of the public to news about *artificial*

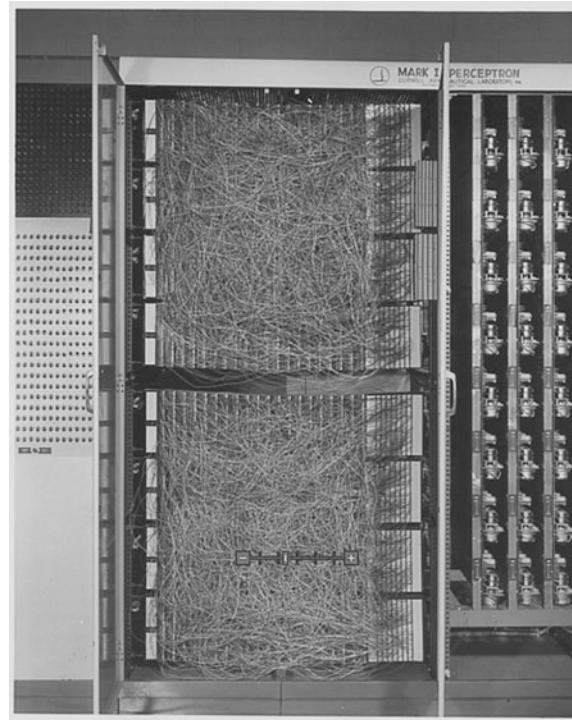


Figure 10.3 This figure shows the original Perceptron machine [557, 563], implementing a primitive artificial neural network model. Note all the individual wires used to connect phototransistor pixel inputs to potentiometers storing the filter weights, and define the filter shape. Image © Cornell University, used by permission, *Cornell University News Service records, #4-3-15. Division of Rare and Manuscript Collections, Cornell University Library*

intelligence may best be summarized as wild expectations, as stated by The New Yorker December 19, 1959 [604] “*That’s a simplification. Perception is standing on the sidewalk, watching all the girls go by.*” The original Perceptron was implemented in hardware, and used 400 photosensors to compose a raster image, where each photosensor was connected to an electrical potentiometer which could be adjusted to control the weight factors, implementing a primitive single-layer neural model. There was no hierarchy of features, and no concept of a fixed-sized local receptive field, since all the 400 photoreceptors represented the receptive field, and the 400 weights were tuned together as a single feature. The Perceptron was not purely feed-forward, but allowed for some feedback laterally and backwards to provide positive and negative reinforcement. However, the Perceptron is the basis for most feed-forward neural networks, therefore included here in the FNN architecture survey. As shown in Fig. 10.3 the Perceptron was housed in a cabinet, with a cable patch bay for manually connecting photosensors to weight potentiometers to form receptive fields. Weight tuning could be done using motors connected to each potentiometer under software control, or manually via knob adjustments.

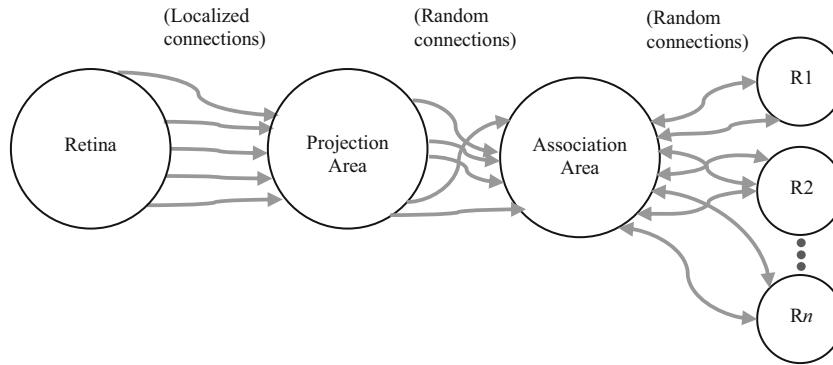


Figure 10.4 This figure shows the basic Perceptron architecture [557, 563] composed of Retinal stimulus (pixels), a Projection area to form localized receptive fields, an Association area to multiply pixels by weights, and a Response area (R_x) for multiple classifiers. Note the semi-random connection patterns and the feedback loops, following RNN models

The Perceptron is the basis for many artificial neural network concepts; in particular convnets are based on the Perceptron neural model of weight adjustment to learn features. The single-layer Perceptron architecture limited learning and accuracy to simple problems, and initially led many researchers to abandon artificial neural network research for many years. However, many of the limitations of the single hidden-layer Perceptron were overcome by the Multilayer Perceptron (MLP) model developed later and refined in the 1980s, using backpropagation and learning methods. We survey various MLP architectures in the next section. Some researchers have developed parallel perceptron models, and corresponding training algorithms, see for example Auer et al. [673].

Here is a summary of interesting features of the Perceptron, including the architecture, weight tuning, and learning.

Perceptron Architecture

- The Perceptron architecture is three layers: pixel stimulus (S-units), Associated pixels (A-units), and responses (R-units), as shown in Fig. 10.6.
- S-points, or input stimuli, impinged upon a retina, or image.
- The S-points are clustered as a local receptive field about a point, but not densely, rather semi-randomly.
- The density or number of S-points in the cluster decreases exponentially with distance from the central point, which is based on biological evidence of the radially decreasing retinal nerve distribution revealed in more modern research [671], and seems to support contour detection. See a similar local response field distribution model used by FREAK in Chap. 4. In other words, high-density S-points are desired concentrated in the center of the receptive field, and sparser S-points are desired as distance from the center increases.
- A-units, or associated cells, receive groups of S-points. A-unit connections define a receptive field. The associated cells are referred to as the *projection area*.
- The pattern of associated cells in the local receptive field is assumed to be random, not structured as an $n \times n$ or circular kernel, but shaped similar to a blob to allow for contour detection. See Fig. 10.5.
- The S-points may be excitatory, or inhibitive, as represented by their weights. The weights either excite (positive values) or inhibit (negative values).
- The algebraic sum of the chosen S-points for an A-unit causes the A-unit to fire all-or-nothing (however, Perceptrons allow an analog value or scalar firing event).

- The Responses, or R-units $R_1 \dots R_n$ (or labeling mechanisms, classifiers) receive input from a set of A-units with assumed semi-random input. The best output is a *binary* all-or-nothing firing event, inspired by neurobiology. Using several R-unit responses combined together to determine object recognition is better than fewer R-units. However, the *binary* nature of firing events means that the Perceptron can model the AND function well, but not the XOR function, which was viewed as a severe limitation to the capability for classification, which discouraged many researchers from Perceptron-related research (see Minsky and Papert, 1969).
- Responses are *mutually exclusive*, and only one response to input is expected, and any strong responses tend to inhibit other responses.
- The Perceptron memory was described as associative and distributed, allowing for some memory units to be removed via the weight settings with only slight decreases in overall accuracy.

As shown in Fig. 10.4 and 10.5, the Perceptron allowed for a variable connection topology, including *localized connections* between the retina stimulus (S-points, photoreceptors, pixels), and *random connections* between the neuron (A-units) and R-units (classifiers).

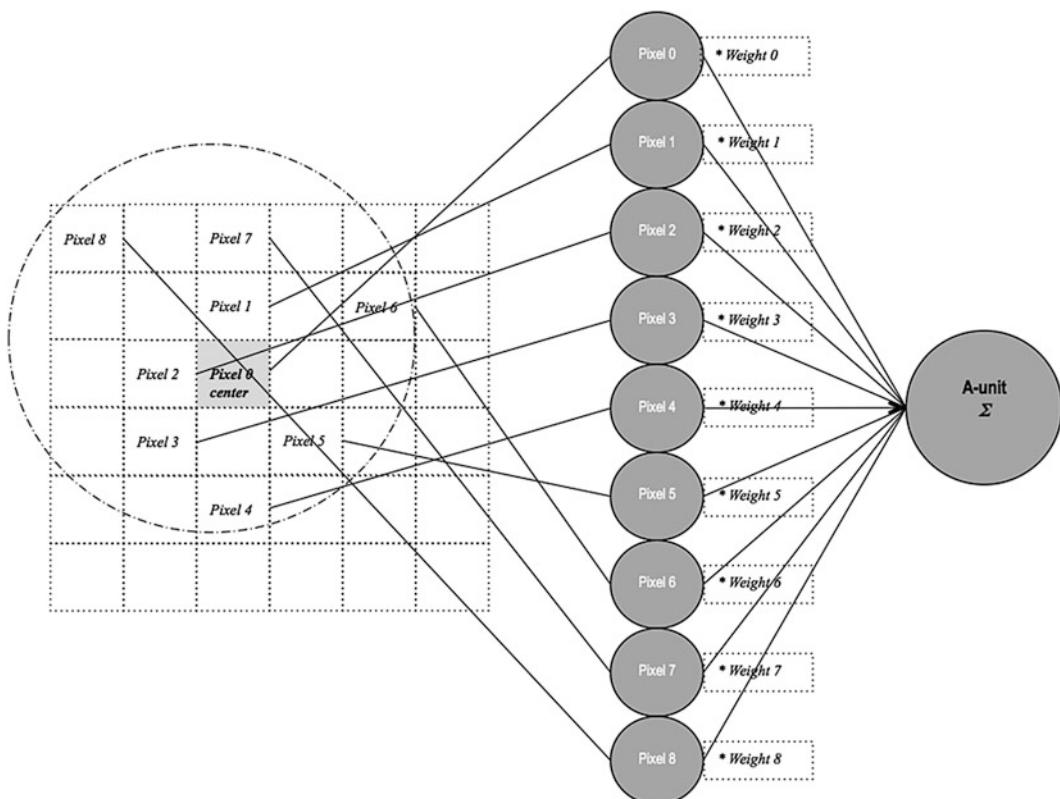


Figure 10.5 This figure shows a hypothetical, nonsymmetric, sparse feature sampling pattern allowed in the Perceptron architecture, which would be manually created by connecting cables in a patch bay between photoreceptors for the raster image to weight potentiometers. Note that the patterns allowed in the Perceptron are similar to modern local binary descriptors such as FREAK, BRISK, and ORB as discussed in Chap. 4. Modern DNNs typically use fixed $n \times n$ matrix patterns

Perceptron Weight Tuning

Learning rules used in the Perceptron are novel, and included several techniques. Feedback in the system, laterally and backwards, was implemented similar to the RNN concept, to adjust or *bias* the weights collectively in different layers, in the form of inhibitory signals from R-units (classifiers) to A-units (receptive field concentrator neurons), and even signals between A-units, to effectively increase the strength of the best responding R-unit, decreasing the strength of weaker responding R-units, implementing a form of reinforcement learning. See Fig. 10.6.

One of the novel learning concepts explored by Rosenblatt include *bivalent weight reinforcement* to implement reinforcement learning, which is worth reconsidering today for incorporation into DNNs. Bivalent adjustments allow for inhibitory (negative) and excitatory (positive) weight tuning, as well as recursive, lateral, and backwards *bias adjustments* between A-units and R-units in an RNN style.

The Perceptron was very sensitive to initial values for the weight setting, and different initial values would lead to different solutions. Best results were achieved by setting the initial weights close to zero.

The overall Perceptron weight tuning approach, and learning in general, was termed *trial and error learning* by Rosenblatt. (*In the opinion of this author, DNNs today have not overcome the trial and error paradigm.*) Several types of feedback systems and weight training protocols were investigated in the Perceptron research:

- **Time-unit gain**, where the active A-units were amplified or their weight increased for one unit of time when activated.
- **Permanent gain** of weight values which when activated, causing weights to keep growing unreasonably.
- **Bivalent reinforcement**, decrease weights for inactive A-units, increase weights for active units.
- **Magnitude-proportional** weight decay, so that inactive A-units receive weight decay proportional to the magnitude of the weight, rather than a more localized distribution of weight adjustments in a receptive field.

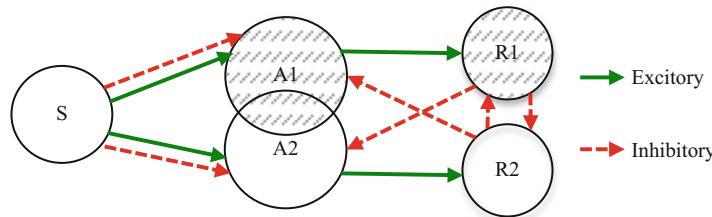


Figure 10.6 This figure shows how the Perceptron used RNN-style feedback laterally and backwards for reinforcement learning adjustments to the weight parameters, similar to a reinforcement bias, see Rosenblatt [557]

Perceptron Learning, Training, Classification

The Perceptron is considered a type of linear classifier. Perceptron learning and training presents several problems such as (1) separable data can lead to several solutions depending on the initial weight values, (2) the training iterations required to converge can be very large, similar to modern Convnets using the same style of incremental adjustments analogous to averaging the weights for the best fit over all training examples, and (3) if the data are not separable, convergence does not occur, and oscillations develop, which can be long duration cycles, difficult to detect.

Rosenblatt investigated several training and learning protocols [672]. Since the local receptive fields were intended to be semi-random sparse patterns surrounding the center position, and the connection patterns were allowed to decrease in density with distance from the center, designing good feature extractors was truly, as Rosenblatt remarked, “*... a trial and error learning system*” and required considerable expertise to get anything to work—*encore vu* in DNNs today. Since a cable patch bay was used to design the sparse feature sampling patterns by manually connecting photoreceptors to potentiometers, Perceptron feature extractors were truly *handcrafted*.

For a given Perceptron, six parameters define the system in terms of training, learning, discrimination, and generalization:

- x : the number of excitatory inputs to each A-unit,
- y : the number of inhibitory inputs per each A-unit,
- z : the threshold value of the A-unit,
- w : the ratio of A-units connected to R-unit,
- T_a : total number of A-units,
- T_r : the total number of R units in the system.

Rosenblatt reported that increasing the number of A-units, or receptive fields, generally increased accuracy to a point, and increasing the number of R-units (classification categories) generally decreased accuracy. Also as the size of the retinal area increased, the number of S-points needed ceases to be as significant.

The training method involved setting all 400 potentiometers independently to best match the target raster pattern. During training, the potentiometers could be set with electric motors. The Perceptron is a simple *linear classifier*, and requires linearly separable data in order to be trained. Although the Perceptron was criticized for limited function representations which hampered learning nonlinear problems, the basic concept of convolution introduced in the Perceptron, *inputs * weights*, is still the predominate basis for convolutional style neural networks today. However, the modern Convnets use enhancements such as local receptive fields, several convolutional layers with sets of hierarchical features instead of just one, and other techniques such as pooling.

The Perceptron learning rule for linear classification has been highly influential, and inspired the Support Vector Machine and Kernel Machine classifiers, as discussed elsewhere in this work. The basic Perceptron learning rule algorithm can be expressed as follows:

```

Define feature sampling pattern in cable patch bay
Define target_pattern rasters, for example digits 0..9.
Initialize weights to random values, such as small random values close to 0
Compute difference test_input : target_pattern for each weight
If target_pattern weights != test_input weights, adjust weights:

```

$$w_j(t+1) = w_j(t) + n(d - y)x$$

where:

```

y = (target_pattern - test_input)
d = desired output
t = iteration number (0 ... 400)
n = the learning rate constant, 0 ... 1
wj = weights
Else leave weights alone, do not adjust
Repeat until error minimum reached, or iteration count limit reached

```

As shown in the algorithm above, if the weight settings = the test inputs, no weight adjustments are made. Thus, the Perceptron implements a sort of *repulsive learning rule*, only changing weights when the weights are incorrect. The goal of the weight adjustments was to classify unique patterns and separate the patterns in weight space. Note also that the inputs are not normalized, and neither is the pattern normalized, which made parameter adjustments hard to manage, so subsequent researchers addressed this weakness by adding various forms of normalization and numeric conditioning to the network, surveyed later.

See Hagan [668] for a detailed introduction to Perceptron design and learning, with worked out examples, mathematical proofs, sample algorithms, and coding guidelines.

MLP, Multilayer Perceptron, Cognitron, Neocognitron

The Multilayer Perceptron, or MLP, is a deeper model based on the Perceptron model, where the MLP contains more hidden layers and other refinements. Several types of MLP architectures have been developed, which are essentially the forerunners of the Convnet-style architectures, such as LeCun's *LeNet* architecture surveyed later, which provides the basis for most of the current generation of DNNs.

As shown by Hornik et al. [890], an MLP can be devised as a general function approximator, taking scalar inputs, convolving them with weights, and adding bias factors to provide a scalar output. The MLP function approximation idea is analogous to the Fourier Series concept, where frequency inputs can be combined to produce arbitrary functions in the frequency domain. However, the backpropagation training used in MLPs is not in any way analogous to the inverse Fourier transform.

The Cognitron and Neocognitron are early and influential examples of the MLP architecture, which we survey next.

Cognitron

Fukushima's Cognitron [677] was demonstrated in 1975 as one the first *multilayer Perceptrons*, enhancing the basic Perceptron model into a deep architecture. The Cognitron is an FNN. However, the Cognitron is primitive given the engineering capabilities of the time, and did not support much invariance for low level features such as translation or scale invariance over smaller receptive fields in low-level features, but for higher-level features more invariance was demonstrated, in part due to the fact that higher level features cover a larger receptive field due in part to convolutional subsampling. The Cognitron is inspired by the neurobiology of the time, and Fukushima provides several interesting observations about neurobiology, for example that the hierarchical Hubel and Wiesel model does not hold for all types of visual reasoning, but may represent a major part of the neurological visual pathway. Fukushima also notes that the Hubel and Wiesel model does not specify higher level cells above complex or hypercomplex cells, yet higher-level cells are known to exist which respond very well to larger features under all sorts of invariance, such as scale and deformation, referred to as *Grandmother cells*.

The Cognitron, as shown in Fig. 10.7, is the first generation model proposed by Fukushima, followed a few years later by improvements to add translational invariance and some distortion invariance known as the *Neocognitron*, which we discuss after the Cognitron.

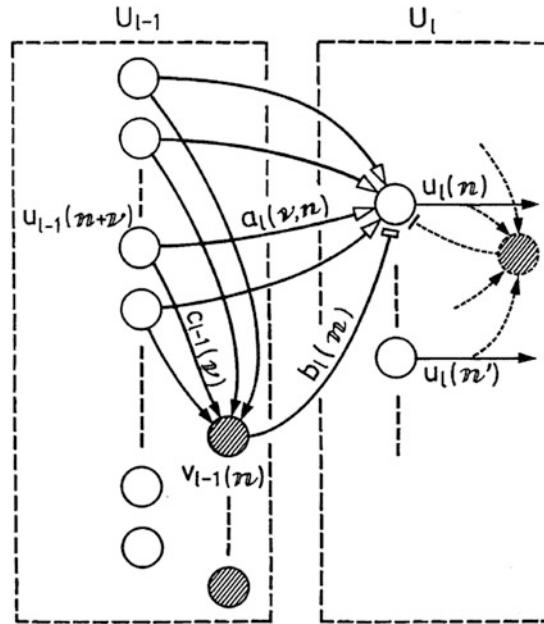


Figure 10.7 This figure shows the basic Cognitron concept of excitatory weights U_n and inhibitory weights V_n . Image © Springer-Verlag, used by permission, from “Cognitron: A self-organizing multilayered neural network”, K. Fukushima, used by permission, Biol. Cybernetics 20, 121–136 (1975) Springer-Verlag

The Cognitron learning rule embodied a concept Fukushima called *Dynamic Equilibrium*, which increased weights for the strongest feature matches to make the matches stronger, and decreased other weights for inhibition to balance the excitatory and inhibitory weights. However, the Cognitron weight settings were difficult to manage to balance excitatory and inhibitory factors. The *excitatory and inhibitory weight adjustment rules* allowed the excitatory weights to keep growing unchecked, so Fukushima introduced a *weight limiter function*. If the weights were set correctly then better recognition of overlapped patterns was possible than the Perceptron. Also, better discrimination between binary and gray-scale patterns was also possible with the Cognitron, provided the weights were set correctly.

The Cognitron weight adjustment rules are summarized in Fig. 10.8.

*Excitatory weight adjustment rule**

$$e = \frac{P * v}{n}$$

where

e = excitatory increment

P = proportionality constant

v = line value (input value)

n = # inputs in pattern subcircuit (i.e. kernel)

*Inhibitory weight adjustment rule**

$$i = \frac{G * s}{t}$$

where

i = inhibitory increment

G = generality constant

s = sum of input values in subcircuit (i.e. kernel)

t = sum of weight values in pattern kernel

*Excitatory weight limiter function

$$l = \frac{e - i}{I + i}$$

Figure 10.8 This figure shows the Cognitron weight adjustment rules

Neocognitron

The Neocognitron [571, 679] is an extension of Fukushima's Cognitron [677]. One major enhancement in the Neocognitron over the Cognitron is translational invariance, enabled by OR'ing a local region of subcircuit comparator outputs together (i.e., convolutional filters pooled together) at the output of each layer, so that the same feature could be detected in multiple locations, limited only by feature overlap range.

The Neocognitron is the forerunner of the Convnet style DNNs. The Neocognitron was first demonstrated by Fukushima in 1980 [570, 571] capable of *self-organization* in the words of Fukushima, referred today as *unsupervised learning*. The learning method produces features that capture the *Gestalt* or geometric gist of each feature, providing for pattern matching based on geometric similarity with translational invariance. The Neocognitron is an FNN, composed of the input layer, followed by regular two-part layers mimicking the Hubel and Wiesel model [559, 560] as simple cells containing input patterns, followed by complex cells containing higher level concepts derived from the simple cells. Each input (or synapse as Fukushima notes) is afferent, plastic, and modifiable (i.e., via the weights). After training, the last layer of C-cells become a set of trained classifiers, responsive to only a single type of high level concept or feature, with some deformation and translation invariance.

Fukushima notes that the division between excitatory cells and inhibitory cells is a form of *lateral inhibition*, a neurobiologically inspired concept, since the inhibitory weight factors will tend to allow the feature pattern to shift in position, yet still be recognized from location to location since the excitatory weights will yield a strong response at various positions. The Neocognitron uses *Simple Cells* to act as convolutional features (S-cells), *Complex Cells* to pool S-cells (C-cells), and V-cells to sum C-cell activity, which is used to provide a gain control for S-cell convolutions.

Fig. 10.9 (top) shows the basic types of cells represented in the Neocognitron based on current neuroscience, starting with the retina on the left, proceeding through simple and complex cells grouped together into receptive fields as per the Hubel and Wiesel model, finally leading to *hyper-complex cells* representing high level concepts, and *Grandmother cells* which represent the highest level concepts. Fig. 10.9 (middle) shows the cell features actually cover more pixels as they are represented in higher levels of the network, illustrated by the large letter A in the input image at the

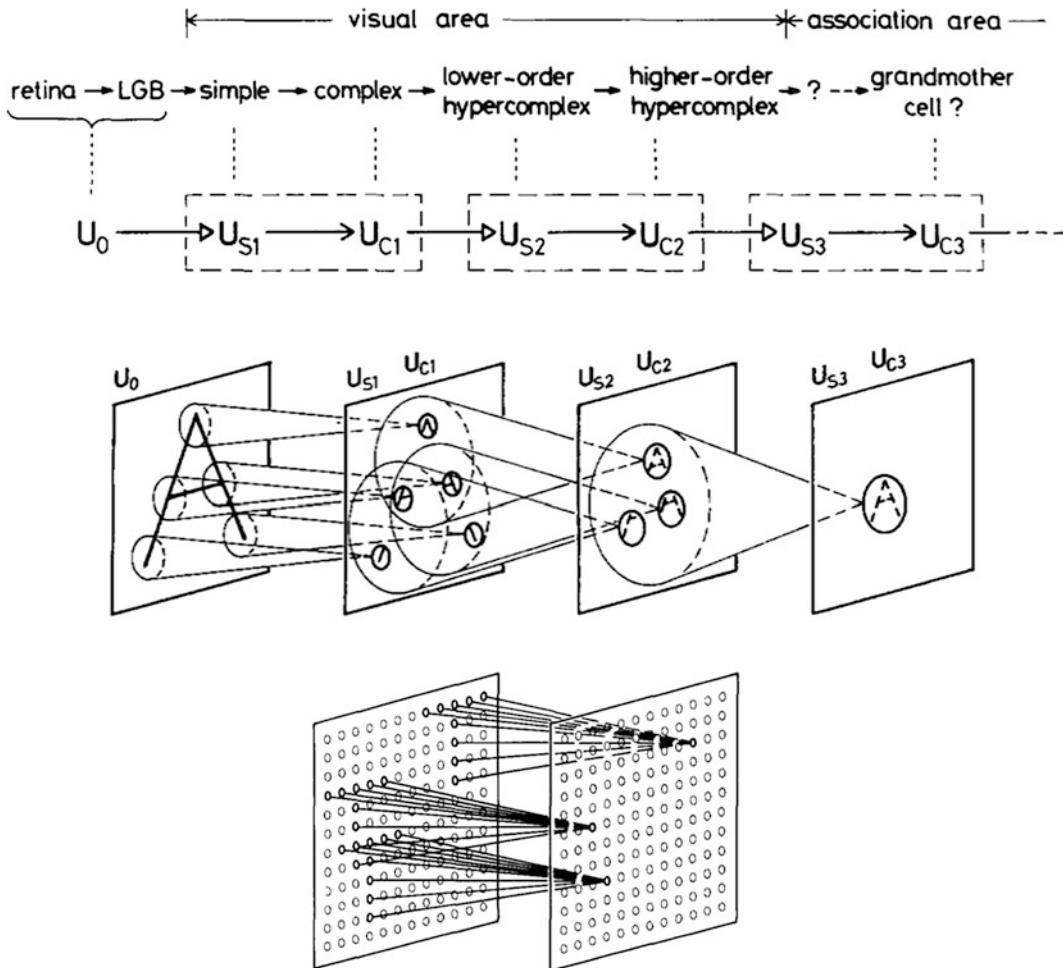


Figure 10.9 This figure illustrates Neocognitron architecture concepts. (Top) The correspondence between Hubel and Wiesel simple and complex cells and the Neocognitron layers. (Middle) Illustration of the hierarchical, cascaded feed-forward nature of the layers and corresponding features. (Bottom) An illustration of a regular hierarchy of Convnet style filtering, note the pattern “T” in the inputs. Note the excitatory weights U_n and the inhibitory weights V_n . Image © Springer-Verlag, used by permission, from “Cognitron: A self-organizing multilayered neural network”, K. Fukushima, used by permission, Biol. Cybernetics 20, 121–136 (1975) Springer-Verlag

left, which is eventually represented as a condensed, subsampled letter A feature at the top of the feature hierarchy on the right. The condensation of the feature is the result of convolutions and subsampling, resulting in a representation of a set of unique top-level features. Also note that each feature is represented as a hierarchy of feature parts, or deep representation. The bottom image in Fig. 10.9 shows excitatory cell outputs as dark lines between layers that best match the input pixels representing the letter T; however, the inhibitory cells are also present in the system, but not shown with synaptic connection lines.

The Neocognitron inputs were not normalized around zero, but were floats mostly in the range 0–1 (sometimes >1), also the neuron outputs or C-cells were not normalized either. Note that in subsequent generations of CNNs, we see greater attention given to data normalization to pair the data with various activation functions that operate in a mean-zero normalized range.

S-columns represented the *Hyper column* concept introduced by Hubel and Wiesel in 1977, illustrated in the Neocognitron layers as shown in Fig. 10.10, and note that a single cell may overlap and contribute to several S-columns. The S-column concept describes a focused formation of the features up through the hierarchy, and tends to suppress feature overlap.

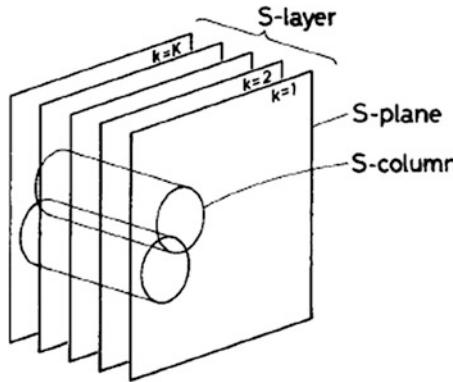


Figure 10.10 This figure shows S-columns which focus, combine and condense features in the S-layer hierarchy. Image © Springer-Verlag, used by permission, from “Cognitron: A self-organizing multilayered neural network”, K. Fukushima, used by permission, Biol. Cybernetics 20, 121–136 (1975) Springer-Verlag

Fukushima had several more experiments in mind to test and refine the Neocognitron, but only went as far as possible given the computing power available. The entire Neocognitron was simulated on early *digital computers* typically containing less than 64,000 bytes of memory, probably hand-wound core memory. The major innovations of the Neocognitron architecture that inspired the later Convnet-style developed by LeCun and other DNNs are summarized here:

- Regular, feed-forward layers composed of replicated functions.
- Input windowing using striding to assemble $n \times n$ input kernels for filters.
- Convolutional layers, composed of filters (features) in hierarchical sets.
- Weight replication and sharing in each layer, for parallel convolutions, which reduced the number of parameters.
- Subsampling layers fed by convolutional layers, to provide translation invariance. Subsampling used local spatial feature averaging of entire features.
- Learning via interesting weight tuning methods, although backpropagation methods were not used, the inspiration and foundation for tuning algorithms was laid, see [678, 679].
- LWTA weight tuning and training. Instead of backprop, weights were tuned either using (1) local winner-take all (LWTA) unsupervised learning, or (2) by pre-wiring the electrical circuit (transfer learning).

Concepts for CNNs, Convnets, Deep MLPs

Convolutional Neural Networks, abbreviated as *Convnets* or CNNs, are a good starting point for learning about DNNs in general. A CNN is a type of Multilayer Perceptron (MLP). A summary of CNN motivations is provided by LeCun [655], who is considered one of the pioneers in this field. In this section, we will survey the basic architecture and components of a CNN, including features, types of layers, convolutional neuron models, and backpropagation and training parameters, to lay the foundation for surveying several innovations on the basic CNN architecture.

Convolutional neural networks leverage the historical concepts developed in the Perceptron, Cognitron, Neocognitron, and other systems described in the history section earlier. See Table 10.1 for a summary comparison of the progression in CNN developments.

Table 10.1 Summarizing Convnets compared to historical P and MLP models

CNN Architecture Feature	Perceptron	Cognitron	Neocognitron	LeNet
Raster pixel input grid	y	y	y	y
Convolutional filters/features/weights	y	y	y	y
Hierarchical deep network	-	y	y	y
Sliding window $n \times m$ input kernels	-	y	y	y
Weight sharing in layers	-	y	y	y
Non-rectangular feature shapes	y	-	-	-
Pre-processing, global normalization	-	-	-	y
Non-linear activation functions	-	-	-	y
Pooling & subsampling	-	-	y	y
Post-processing, local normalization	-	-	-	y
1d classification layers	-	-	-	y
LWTA or hard-wired training	-	y	y	-
Backpropagation training	-	-	-	y

Convolution is the basic operation used to model an artificial neuron to both learn and detect the features, using a *weight matrix* convolved against an input window of pixels. Convolutions are used like a correlation template or feature detector. The output of each convolutional filter is assembled into an output image referred to as a *feature map*, which is sent along as input to the next layer. One output image is created for each filter, and there are usually hundreds of filters per layer. Each convolutional filter acts as *both* a feature detector and a filter. The convolutional filters are composed together into feature sets at each level of the hierarchy, and each filter is tuned during training as discussed in the backpropagation section later.

As shown in Fig. 10.11, Convnet feature layers can be conceptually represented as a *volume* or stack of separate input images and output images. For example the input volume may contain a stack

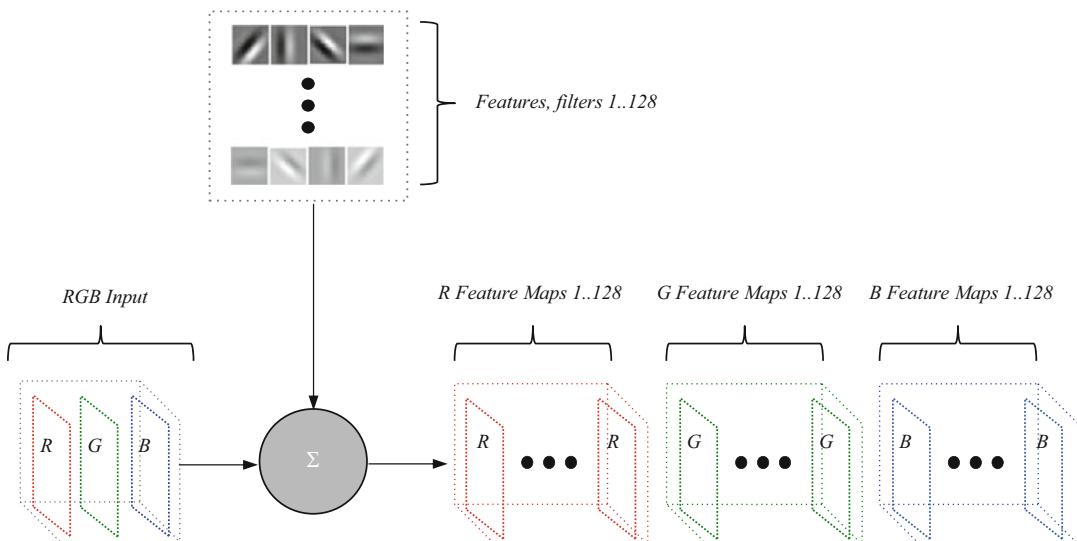


Figure 10.11 This figure illustrates convolutional layers taking an RGB input volume (stack of 2D images) and producing a volume or stack of 2D output images or feature maps, one per feature per input image

of three images, one for each RGB color channel, and if the convolutional layer contains 128 filters (features), then the output of the layer is a volume or stack of 128 feature maps. So for an RGB input containing three channels, the output is 3×128 feature maps.

Note that convolution is only one way to model the neuron in DNNs. We survey a range of models in this chapter. For example, in the NiN model by Li [636], convolution as a generalized linear model is replaced by an MLP model, acting as a fully connected *micro-classifier* inside each neuron to achieve equivalent, of not better, results than convolution. Polynomials are used to model the neurons in the PNN model surveyed later. Also, using basis features in place of convolutions to model the neuron is demonstrated in the HMAX model [812] surveyed later. Raw memory impressions are used to model the neuron in the Visual Genomes model [534], see [Appendix E](#).

CNNs are used for many applications including (1) *classification* of families of objects, (2) *recognition* of specific objects within a class such as the specific face of a known person, (3) *localization* of objects to find coordinates, (4) *segmentation* of region of pixels into classes such as water, grass, or roadway, and (5) general *regression analysis*. We will survey the architectures and design considerations used for these tasks in this chapter.

The foundational architecture used by many CNNs today, LeNet-5, is described by LeCun [682], and surveyed later in this section. LeCun notes [655] that Convnets are a fundamental type of multi-stage Hubel and Wiesel model, using a design pattern of convolutional filter banks as simple cells, and pooling layers as complex cells.

The Convnet is a *feed-forward* network, with data moving from input through processing to output classification. However, the training phase for feature learning operates *backwards*, computing the error between expected results and computed results, and distributing the errors back to their source to correct the filter weights (features) in a method referred to as *backpropagation*, discuss in some detail later.

While current generation Convnets are all based on discreet convolution of scalar data in square $n \times n$ kernels, future research is pointing towards alternative basis spaces for the data, such as Fourier space [680], allowing other dimensions to be mapped into the Convnet framework.

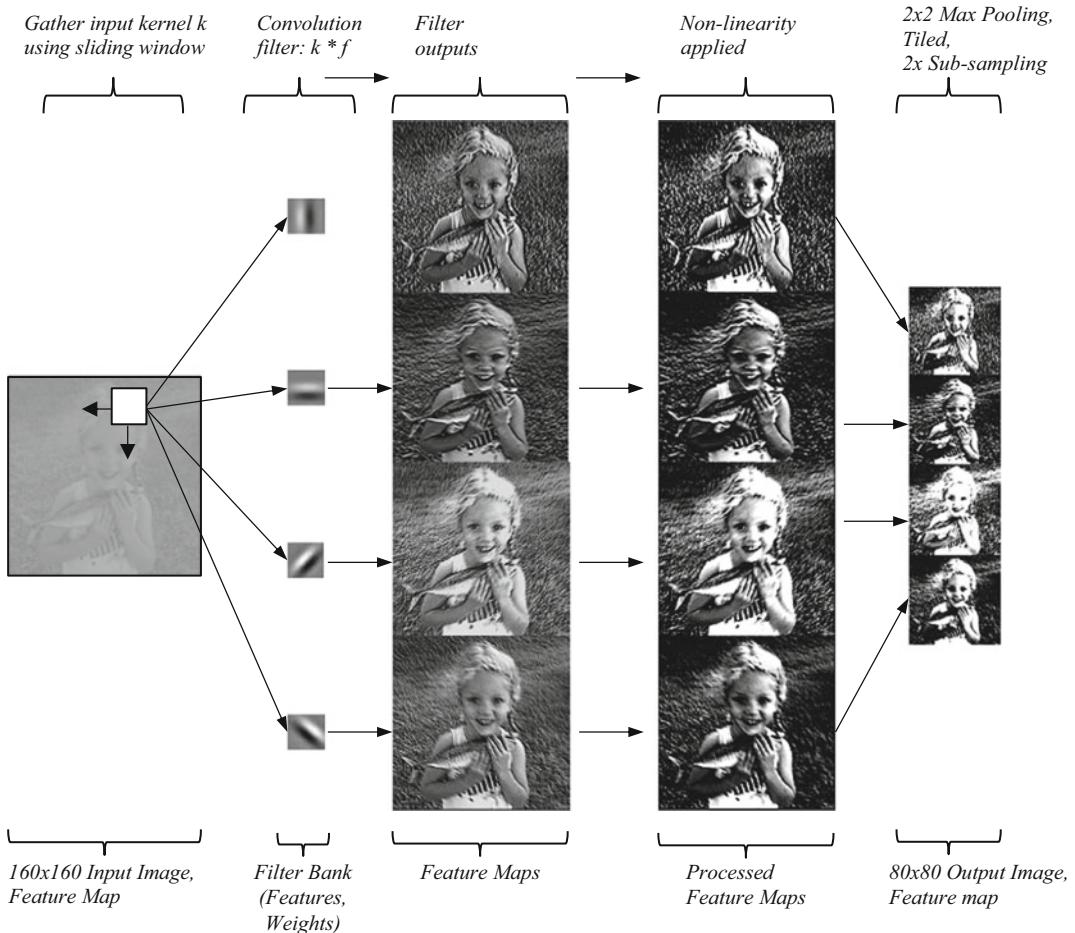


Figure 10.12 This figure illustrates the basic operations that occur in a convolutional layer. Note that the output image is reduced in resolution (we assume that a 1-pixel padding is used around all images in this example, so no border pixel resolution is lost to convolutional window border effects), based on the reduction from the 3×3 kernel (3×3 region \rightarrow 1 pixel) and the final pooling and subsampling layer (2×2 region \rightarrow 1 pixel)

Forward and Backward Pass Through the CNN

Next, we describe the basic CNN operations in both directions as the starting point for the survey of the myriad CNN variations used in practice.

Forward pass overview (feature detection pass, training pass covered later):

- Each training sample, one at a time, is moved forward through the network to compute the error between the sample and the closest matching target feature.
- Inputs, pixels in the case of the first layer and feature maps for subsequent layers, are *fed forward* into convolutional layers in local pixel groups of say 3×3 taken from a sliding window over the image.
- Some numerical conditioning may be applied to the input data, for example mean-zero normalization to condition the data for a mean-zero activation function.
- Correlation/Convolution: The data is then convolved with an $n \times m$ filter (i.e., weights) to compute the filter correlation or result value.

- Stored results are kept in the output feature map (*the filtered image*) from the convolution, and also a derivative is stored by taking the difference between the current convolution results f' and the previous results f'' . The derivative is computed and stored for each training sample for each feature. Initially, each stored feature weight matrix may be substantially different than the closest matching training sample, until corrected by backpropagation tuning to reduce the difference.
- Some post-processing is typically applied to the filter result using an *activation function*, such as a zero-centered sigmoid, which introduces nonlinearity into the result. One goal of nonlinearity is to project the purely linear convolution operation into a nonlinear solution space, which is believed to improve results. In addition, the nonlinearity may result in faster convergence during backpropagation training to move the gradient more quickly out of flat spots towards the local minima. Also, the nonlinearity is used to ensure that the value is differentiable for backpropagation.
- Pooling and subsampling is typically applied to the output feature map in a pooling layer, where local regions are combined into a single value, such as 2×2 regions, via averaging values together or choosing the max value. The result is a smaller output image or feature map, see Fig. 9.19.
- At the end of the CNN, the FC layers reduce and classify the features by performing 1D vector convolutions, see Fig. 9.5. The last layer of the FC classifier produces a best-guess label or identification of the input data.
- The output classification guess is measured as an *error term* (expected vs. computed result) to be used for backpropagation tuning, discussed next.

Backward pass overview (weight tuning, or learning):

- The *total error term* computed during the forward pass at the classifier output is the basis for backpropagation. See the section on backpropagation below for details. The output value is a classification result, which has an error term (computed vs. expected result). The total error term at the classifier output *is a combination of errors from all the previous convolutional layers*, including contributions from the feature weights, processing steps such as pooling and normalization, and learning parameters.
- Backpropagation traces the error backwards through each feature layer to the contributing weight sources, and decomposes the error term into smaller and smaller parts (partial derivatives or gradients) at each layer, one feature weight at a time, in order to adjust each weight to reduce the error. Backpropagation is discussed in detail later in the section ‘Backpropagation, Feature Learning, Feature Tuning’, see also Figs. 10.22 and 10.23.
- Each feature weight is tuned independently by scaling the gradient against the derivative error $f' - f''$ stored at each neuron. The derivative corresponds to the response change between current weights and previous weights, useful for scaling weight adjustments with the gradient to adjust each weight, see Fig. 10.23.

Fully Connected (FC) Layers, Flatten, Reduction, Reshape

Feed-forward networks typically make use of special network connection arrangements for particular goals such as (1) fully connecting feature weights to a 1D vector for linear classification, and (2) reduction of large fully connected layers into smaller layers to reduce the parameter count, see Fig. 10.13. In some architectures in this survey, layer connections may be designed to reshape, split, or to go between 1D and 2D shapes. A fully connected network is sometimes referred to as a Perceptron (P), and serially connecting several Perceptrons together is known as a Multilayer Perceptron (MLP). An FC layer can be considered an array of neurons, and is trainable via backpropagation.

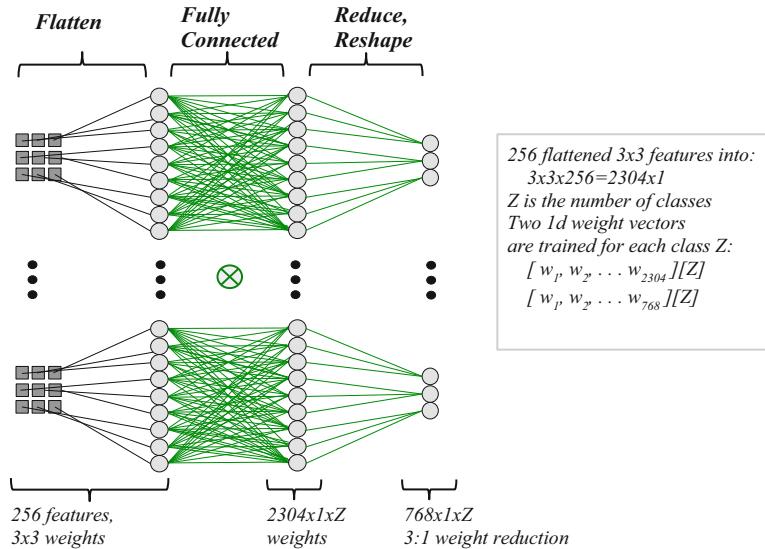


Figure 10.13 This figure illustrates flattening of 256 3×3 features into a 2304×1 vector, fully connected to a 2304 FC layer of neurons with 2304 weights per neuron, finally reduced 3:1 into an FN neuron layer of 768

In DNNs, the FC layers act as a *bridge* between the filtering layers and a classifier. FC models are seamlessly trainable in CNN architectures, since the same convolutional neural weight model is used. Since fully connected layers can be prone to overfitting for high parameter counts, various mitigation strategies are used during training to reduce and regularize the parameters, such as randomly dropping data (see *dropout*, Chap. 9).

The first FC layer usually contains the largest number of parameters in the entire system, specifically the most connections and weights. Subsequent FC layers are typically reduced in size for practical reasons. For example, the 2304 element FC layer in Fig. 10.13 must compute a 1×1 convolution over each of the 2304 connections for each FC neuron ($2304 \times 2304 = 5,308,416$ MADD instructions + some sort of activation function). Also, each FC layer maintains a set of 1D weight vectors and bias vectors, one for each class, to be trained via backpropagation. For example, the FC feature vector for a 2304-wide FC layer may be an array of [5308416][number_of_classes] 1D feature weight vectors and bias vectors, compared to a kernel-connected layer with only [256][3×3] feature weights and a bias for each feature. A kernel-connected layer preserves local spatial relationships in the receptive field, while a 1D vector in an FC classification layer does not necessarily do so.

An FC layer is a typical building block in a DNN, typically used in the last layers for classification. Alternative methods of employing or ignoring the FC layers are surveyed later, including purely kernel-connected convolutional filter layers as used in the Half-CNN, and the Inception architectures with multiple FC classifiers at different layers in the network. An FC network can be described in several different ways depending on the intended use:

- As a simple Perceptron P, or as an MLP when several FCs are stacked
- As a *dimensionality reduction layer* to reduce the input features to the desired number of output classes, like a final pooling layer
- As a *normalization layer* to feed into a softmax classifier
- As a *linear regression network* over the FC inputs

- As a *logistic regression network* for binary classification
- As a general *function approximator*

A fully connected layer is a *general function approximator*, capable of implementing a *linear classifier*, or a *logistic regression function* for binary classification, perhaps using Hamming distance for matching. By weighting each input to each neuron in the FC layer, the combination of inputs and weights forms a linear classifier. Usually one or two FC layers stacked together are sufficient to approximate the desired function. Even a single-layer or shallow FC network has been shown to be able to approximate any function to arbitrary precision as shown by Hornik et al. [890]. The final FC layer vector length will typically be *reduced* in a reduction layer to provide the correct number of class outputs for an application. The FC layer uses one 1D weight vector for each label or class, so an array of 1024 1D weight vectors is needed to classify 1024 objects, and each of the weight vectors must be trained and evaluated to find the best match. Sometimes, FCs are replaced after training with an SVM or other classifier to increase accuracy.

Sparse Gaussian-connected layers, instead of fully connected layers, have also been used to reduce the connection count, and other sparse topologies are possible. A circular matrix projection connection topology proposed by Cheng et al. [695], and other methods discussed in the components taxonomy in Chap. 9. For example, a *Global Average Pooling* (GAP) layer as introduced in the Nin [546] architecture has been used to replace the FC layers and vastly reduce parameter counts.

When the author implemented his first FC neural network to recognize audio guitar chords from a microphone on a PC in 1999 using Fourier spectrum features, it was difficult to believe that this simple FC neural model was little more than hype, since it seemed so simple from a programming perspective, but it worked.

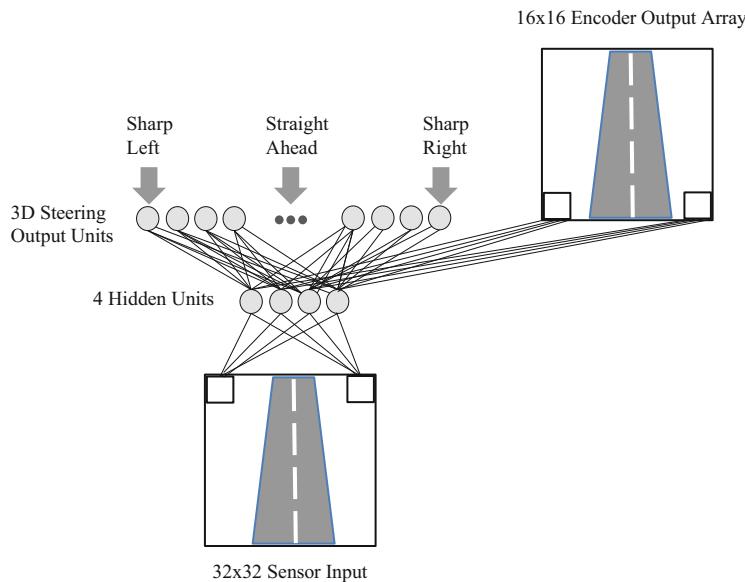


Figure 10.14 Images describing the ALVIN driving system developed by Jochem and Pomerleau in 1996 [696], which used fully connected layers to learn to steer a car on a roadway

The FC model provides a *framework to think about classification*, forcing the system to be designed around vectors of features to support the FC layers. A small network of FCs can regularize the input to the correct dimension and label the results, for example reducing a larger number of

features into a classification vector of the desired number of classes. An FC can be designed to filter out inputs, similar to dropout, by setting the 1D vector to contain zero-valued mask weights for pixel inputs not of interest, and non-zero values to include those of interest. For normalization and other processing, the FC weights may be set to *excite* ($i > 0$) certain inputs, *inhibit* ($i < 0$) certain inputs, and *ignore* ($i = 0$) certain inputs. For binary classification and logistic regressions, the weight values of 1 and 0 can be sufficient.

A fully connected mesh topology may be used for example to connect each pixel in the image together for a segmentation application (a huge number of connections!). To illustrate the capabilities of a fully connected mesh, Fig. 10.15 provides a few feature weight visualizations created by Andrej Karpathy¹ using the Caffe open source neural network library. The FC mesh was trained on Imagenet data to create *very primitive features*. The architecture is just a single FC mesh connection all the pixels in the image, simply intended for visualizing and studying linear classifier failure modes, and is not intended to be competitive or accurate, since this architecture is reported to be $<3\%$ accurate at best. The training protocol involves resizing each image in the 1.2 million Imagenet image training set to 64×64 pixels to reduce training time, then train using backprop. All input pixels in each image are *fully connected* into a linear classifier of size $64 \times 64 = 4096$. The resulting features are visualized as a blob-like color feature showing the dominant color distribution across all the training set images for each of the 1000 images per class.



Figure 10.15 This figure shows a single-layer FC classification and visualization of all the pixels from a few of the 1000 classes in the Imagenet data. Image © Andrej Karpathy, used by permission

We can illustrate a very simple FC as shown in Fig. 10.16, using three ASCII characters as input, with an objective to classify a set of three-letter character strings into three classes: CAR, MOT and PED. The FC layer contains three artificial neurons, each with a weight and bias. As the input values are propagated through the FC layer, each input letter is multiplied by the corresponding weight factor and summed in the neuron, and the weights are designed so that matches will sum to 3 (based on ASCII character values). In this simple example, the bias is a scaling factor, not an

¹ See <http://karpathy.github.io/2015/03/30/breaking-Convnets/> for the training parameters and goals.

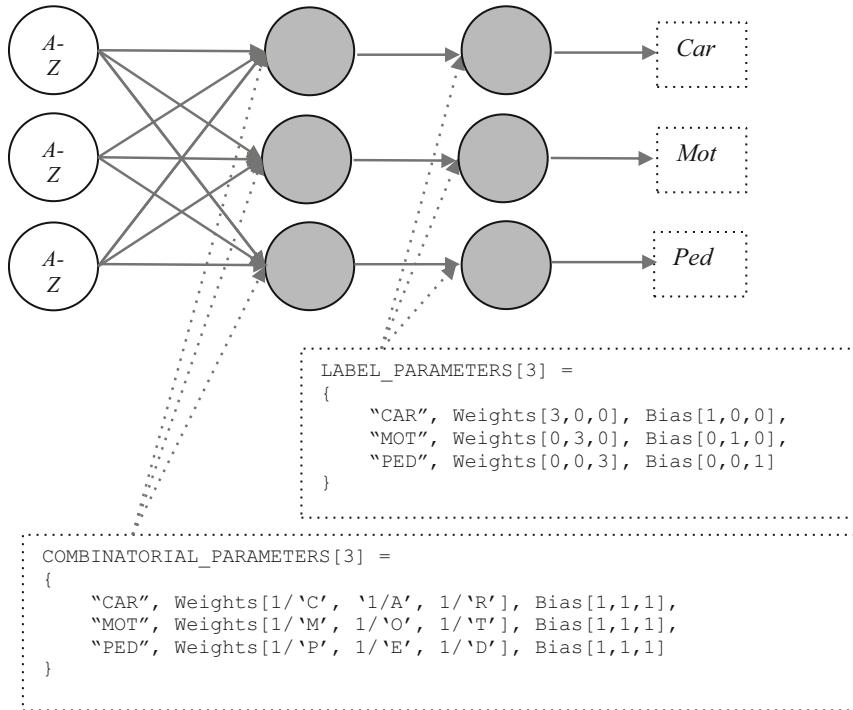


Figure 10.16 This figure illustrates a simple FC network used to classify three-letter character strings. Note: in this example, we use the bias as a multiplicative scaling factor for the convolution result, not as an additive bias. Typically, CNNs ignore the bias and set it to 1

additive bias. The FC output is passed directly to the final single-connected classifier layer, and a correct match for each class will yield a value of 3 when all three characters match the expected values. A nonlinear activation function is not needed for this simple example, since we are seeking a linear classification.

Layers and Depth

CNN architectures are typically described a layer at a time. For simplicity, we distinguish between the following major layers:

1. Input Layer
2. Convolutional Feature Layers (*a hidden layer in DNN parlance*)
3. Convolutional Classification Layers (*a hidden layer in DNN parlance*)
4. Output Layer

Note We taxonomize *convolutional layers* to be an aggregate layer, containing multiple operations such as numeric conditioning, convolution, activation functions, and pooling. We do not call out separate layers for all of the various operations, since there are too many possibilities. See Fig. 10.19 and the taxonomy in Chap. 9.

One key CNN concept is the use of *replicated* convolutional layers in the architecture. As shown in Fig. 10.17, a few basic types of layers are used. The *input* layer is the simplest, consisting of pixels for a 2D imaging application, feeding into convolutional layers. The output of convolutional layers consists of: (1) Feature, or filters as $n \times n$ weight kernels, and (2) filtered images (one image per feature), referred to as *feature maps* in CNN parlance, which are fed as input to the next convolutional layers.

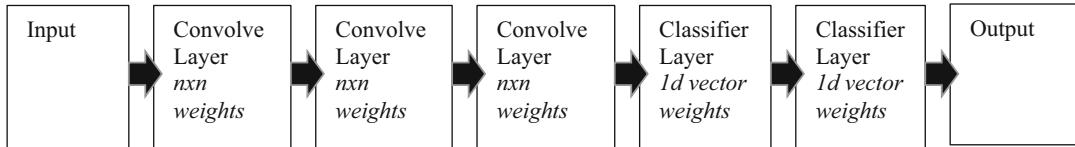


Figure 10.17 This figure shows the layered architecture of Convnets. Note that there are five layers in this network, three convolutional filter layers, and two convolutional classification layers

Typically the convolutional layers are replicated, or else changed slightly from layer to layer. For example, a convolutional layer may optionally include numeric conditioning of the input data, convolutional filtering, followed by a nonlinear transform of the convolutional result, pooling and subsampling, and local region post-processing of the data. See Fig. 10.18 and Table 10.2.

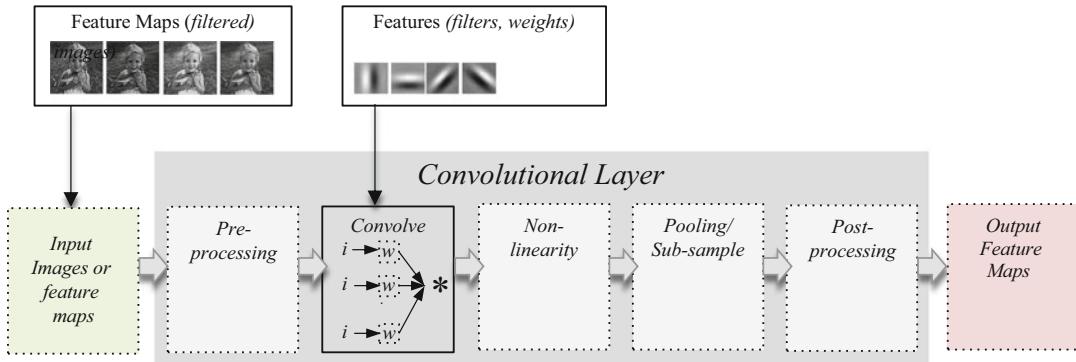


Figure 10.18 This figure shows the basic components in a convolutional layer, described in Table 10.2

Table 10.2 Summary description of typical components in a CNN

Convolution Layer Component	Description and rationale
Input	Pixel images for input layer, processed images for subsequent layers
Pre-processing	Numeric conditioning, dropout
Filter bank, features, weights	Correlation, projection on overcomplete basis, dimension expansion
Nonlinearity, activation	Lateral inhibition, sparsification, squashing, spreading
Pooling and subsampling	Subsampling over tiles, translational invariance
Post-processing	Numeric conditioning, dropout
Output	A feature map, or Image, to feed to the next convolutional layer

Classification layers in a CNN may also be implemented using statistical methods such as an SVM, rather than as convolutional layers (see *FC layers discussed above*). FC classification layers are similar to pooling layers, but used for regularization. The classifier layers flatten out the 2D weight kernels from the feature layer(s) into a 1D vector to allow for (1) a 1D linear classifier to be modeled and tuned, and (2) to support training via backpropagation.

The depth of the entire network is an architecture variable, and typically the final depth is arrived at after trial and error. In Fig. 10.17 we see a network with a depth of five convolutional layers total: three filtering convolutional layers, and two 1D convolves for classification. In practice up to 20 or more convolutional layers are used, as discussed in the architecture survey later in this chapter and Table 10.3.

Several practitioners have demonstrated that depth of the feature hierarchy is more important than the features and the classifier, including Coates, Lee, and Ng [603], Simonyan et al. [656], Ren Wu (comments at the Embedded Vision Summit Oct. 2014), and Szegedy in the Inception architecture [547, 608]. Using more and deeper features is an intuitive advantage.

Convnets have been implemented using a *strided window* method to gather pixels from spaced tiles in the input image for faster compute, or to introduce subsampling to the input to condition the data, or to introduce translational invariance. However, more recent Convnets set the stride to 1 to gather input over each possible window and generally see better results.

Modeling an Artificial Neuron

As shown in Fig. 10.19, an artificial neuron is modeled using a pipeline of *operations*. We will discuss methods to optimize the pipeline later in this section, for example, see Mamalet et al. [664]. Here is a discussion of the components in typical convolutional neuron models.

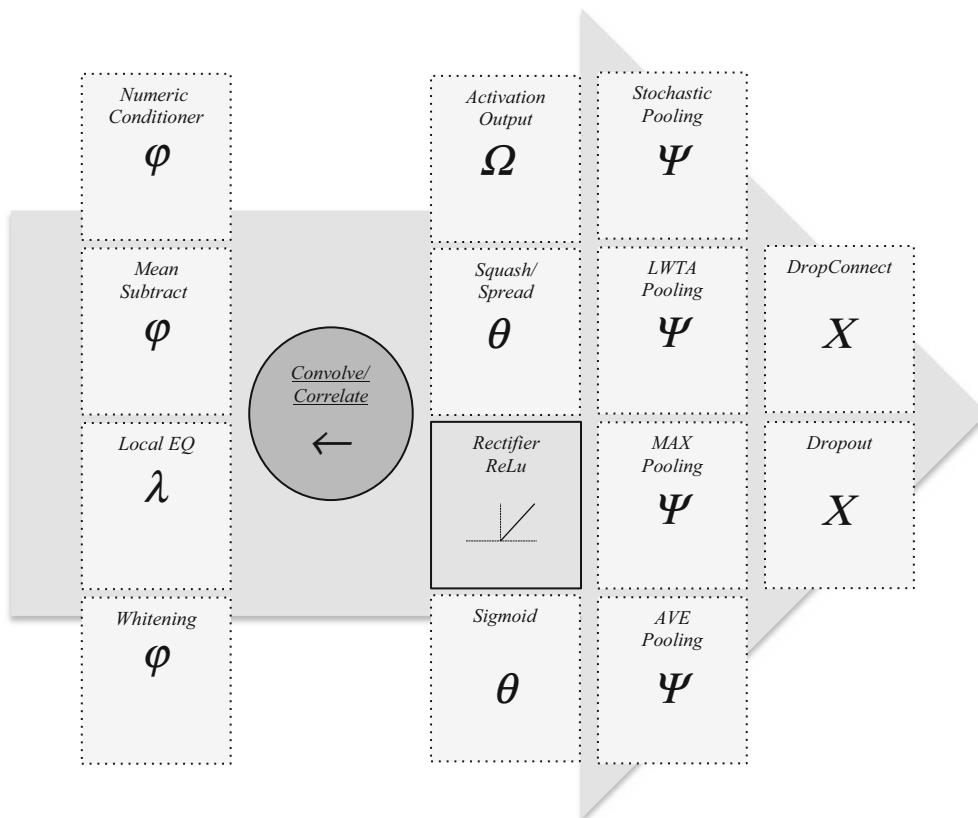


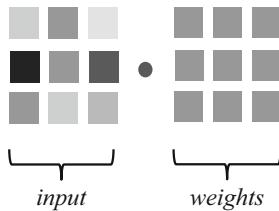
Figure 10.19 This figure shows common which may be combined in various fashions combined to model an artificial neuron in a convolutional feature layer

Convolutional Features, Filters

Convolutional feature layers represent the neural function as a convolution, or a template match via correlation, and *the features are weight matrices*, which are tuned to the training set using various training protocols and backpropagation methods. There is some variation in terminology among practitioners, and subtle nuances in intent, which we cover in this section. For 2D images, the weights are modeled as 2D kernel arrays or *weights*. For *fully connected classification* layers (also discussed in this chapter), the weights are flattened into a 1D vector to feed a 1D array of neurons.

The main operation of each artificial neuron is convolution of inputs and weights. The term convolution is used very imprecisely in CNN discussions, and what is meant is the *dot product* of the input vector I and the weight vector W :

$$I \cdot W = \sum_{i=0}^n I_i W_i = I_0 W_0 + I_1 W_1 + \dots + I_n W_n$$



Actually, the mathematical definition of convolution assumes that one of the input vectors has been reversed and shifted prior to the dot product (i.e., the filter has been flipped horizontally and vertically); however, correlation by definition takes straight data. In typical image processing packages, there is usually a scaling factor s applied to the convolution result, and possibly an added bias b :

$$\text{convolve}(I, W) = s \left(\sum_{i=0}^n I_i W_i \right) + b$$

Convolution and correlation are mathematically equivalent given some setup assumptions. *Template matching* is another term commonly used to describe correlation. Also, filtering is a term typically used in CNN discussions for convolution, which produces a filtered *feature map* (*output image*), since the dot product results can be interpreted as a filter. See the section on *Convolution vs. Correlation* later in this chapter for more details on convolution and correlation.

Besides the more complex neural model pipelines for 2D feature filtering layers, the FC layers contain a simpler neuron pipeline usually consisting of only convolution and an activation function for the 1D feature vectors, see the discussion on FC layers.

For the feature layers, a set of unique filters are kept at each layer in the hierarchy, and the number of features is typically limited to a few hundred features per layer. Convolutional filters are symmetric, rectangular, such as 3×3 , 7×7 , 11×11 . A hierarchy of feature concepts are kept in higher layers of abstraction, consisting of edge and texture filters at low layers and higher level concepts at higher layers, such as motifs, object parts, complete objects, and scenes. Each filter is run against the input feature map exactly like any other spatial filter, typically as a sliding window across the image at each pixel or strided, producing a new feature map. So Convnet filters are dual purpose: (1) the filters are features (correlation templates), and (2) the filters are in fact filters. For the FC layers, a set of 1D vectors are kept, one vector for each output class, for example if the output classes are *dog*, *cat* and *bird*, the FC layer keep a separate 1D vector to train for each class.

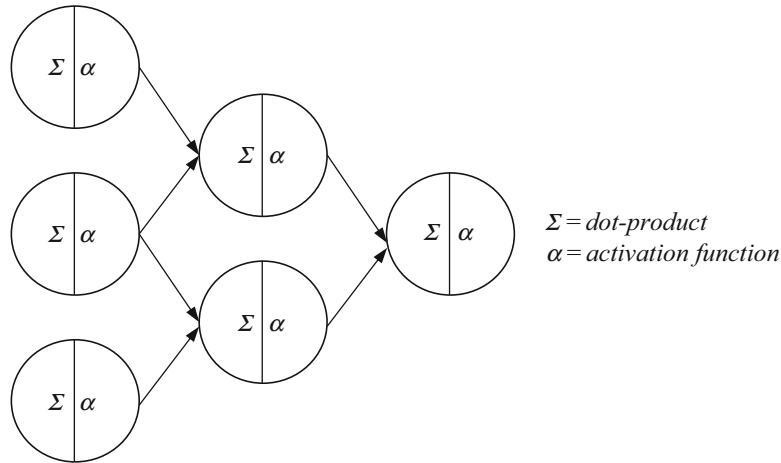


Figure 10.20 This figure illustrates the basic convolutional artificial neuron model composed of a dot-product filter followed by an activation function, or transfer function, to provide nonlinearity to the response

As shown in Fig. 10.20, the convolution (i.e. dot product) is typically followed by an activation function, or transfer function, to provide nonlinearity to the response, discussed in the next section. The nonlinearity ensures that the value is differentiable for backpropagation using gradient descent and similar methods to tune the feature weights, as discussed in the backpropagation section.

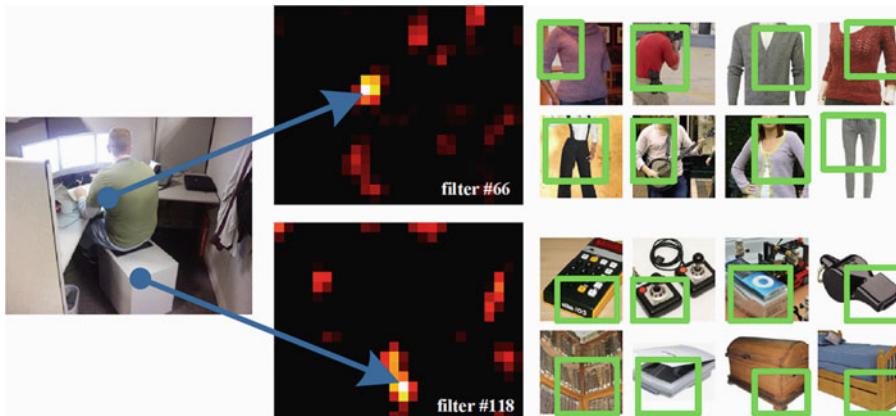


Figure 10.21 This figure illustrates convolutional filter response, (left) input image, (center) results of applying two filters highlighted by strength of filter response, filter #66 and #118, and (right) examples of filter response, top right showing V-like features which responded to filter #66, and bottom right showing a corner-like features which respond to filter #118. Image © Springer, from He et al. [542], used by permission

Transfer Function (Activation Function)

Activation functions may be linear, such as simple thresholds or ramp functions, or nonlinear, such as sigmoids, see the discussion on activation function in Chap. 9, and see also Fig. 9.18. In a CNN, the convolution result is sent to the activation function to perform a nonlinear thresholding mechanism. One goal of nonlinearity is to project the purely linear convolution operation into a nonlinear solution space, which is believed to improve results. In addition, the nonlinearity may result in faster convergence during backpropagation training to move out of flat spots towards the local minima.

Also, the nonlinearity is used to ensure that the value is differentiable for backpropagation. A *Threshold Bias* can be provided to the activation function; however, it is rarely used in most CNN models except for mathematical convenience in matrix operations and set to 1, see Fig. 4.23.

Nonlinearity is believed to help solve problems of data saturation, perhaps caused by numeric overflow perhaps due to poor lighting or very strong lighting. For example, if the correlation output is 255 in range 0–255, a well designed nonlinearity function will redistribute the limit value 255 somewhere within the range, say to 180, to overcome saturation and avoid the limit. The nonlinear distribution produced by the activation function is affected by any numeric conditioning functions used on the input data, such as normalization or whitening, also referred to as local response normalization or LRN. Some researchers report that local response normalization does not work well enough to justify the increased compute time.

LeCun² has stated that a fundamental goal of the activation function is *to break apart* the data and *project to other spaces* in a nonlinear fashion, since each space may provide a better way to represent and match features. Scaling and image pyramids do not meet the *break and project* criteria. A bias factor is used for the nonlinear projection, either by scaling or in an additive manner. The bias can also be used as a mask to influence which features are important.

Feature Weights and Initialization

The *feature weight matrices are the features*, typically 2D matrices or kernels for computer vision applications. Each layer of the network contains a set of feature weights, perhaps a different number of features for each layer. Typically, floating point values are used for the feature weights, and the size of the features ranges from 3×3 up to perhaps 29×29 or larger, as we point out in the surveys of real CNNs in this chapter.

The method of feature weight initialization determines the final outcome of the feature learning. For example, starting from biased feature weights will lead to biased feature learning. Random weight initialization has been used in many systems. The feature weight initializations are critical, since the initial values lead in the direction the weight tuning will follow. Transfer learning is a method to use pretrained features as starting points. Layer-wise pretraining is another method, starting at the bottom layer and working upward, training the features at each layer until they seem to converge. For more information, see the discussion on Feature Initialization Chap. 9.

Local Receptive Field

As shown in Fig. 10.26, each convolutional neuron is fed by a local receptive field from the input image or feature map from a small window. The concept of a local receptive field is influenced by work in neurobiology. Local competition among the local regions processed by neurons at each layer is observed in neurobiology [657, 658], and there is speculation that the neural firing activation function includes a nonlinearity similar to a local histogram equalization among spatially adjacent pixels, which is often implemented in artificial neural models to mimic competition. In addition, competition is mimicked by local receptive field overlap at the input side, or pooling at the output side.

The input is modeled as a receptive field of view, foveal region, or attention field. *This attention field moves across the image as directed*, for example scanning across the image or directed to stare at a location, or perhaps using saccadic movements to dither for greater resolution. Convnets implement the neurological concept of local receptive fields, using $n \times m$ sliding windows across the 2D image to mimic the local receptive fields in the human visual system. Convolutional windows may overlap

²Private presentation to Intel during CVPR 2013.

adjacent windows according to a stride factor of 1 or more, although convnets may use nonoverlapping tiled windows.

Note that the local receptive fields are typically processed and stored as independent, spatially disconnected, and unordered, with no association with other windows or features. The Convnets simply learn and record the features, and the classifier discriminates based on the strength, presence and absence of features. Rosenfeld [625] provides some early work on preserving spatial relationships between features used in scene labeling applications.

Receptive Field Compression via Input Striding or Output Pooling

Subsampling or compression is achieved equivalently at each neuron by either (1) striding the input window, or (2) pooling windows in the output. To gain some invariance to local feature deformations, Convnets may use *strided input windows* to reduce input resolution, or *pooled output windows* to reduce output resolution. Strided input windows are run against the filters to effectively *down-sample*, or reduce the resolution up front, for example sliding the filter window across the image at a stride of 2 pixels. The pooled output window consists of grouping the filter results in small local regions such as 2×2 regions, and choosing the max or average value in the local region, and using this pooled value as the filter result for the pooled region, effectively downsampling or reducing the resolution at the output. However, Schmidhuber [552] notes that the Creseptron [616] added blurring layers to add a measure of translational invariance, yielding similar results to pooling.

Trainable Bias

Each feature may have a trainable bias, which is applied to the results of the convolution operation, implemented as a scale factor or additive bias, before the nonlinearity and subsequent processing. Bias factors are biologically inspired, although nobody really knows how they operate, or how many bias factors biological neurons may have.

Memory for Current Neuron State

For purposes of backpropagation and learning, each convolutional layer maintains several state variables in local memory, such as the *current output* and *previous output* of the neuron after the activation function, and the derivative of the neural state (*current output* – *previous output*). See the next section on backpropagation.

Backpropagation, Feature Learning, Feature Tuning

The feature learning in a CNN takes place during *backpropagation*, where features are tuned by distributing the classification errors back through the network, layer by layer, to adjust the feature weights to minimize the errors.

We provide *only a brief introduction* to backpropagation fundamentals here, since this is an area of active research, suitable to an entire book all by itself. Likely several months time will be required to master the concepts and gain practical experience.

For an introduction to backpropagation and a survey of methods see Hagan [668], and other good discussions are found in Werberos [581]. See Rojas [787] for a readable step-by-step explanation. For a history of backpropagation with details and references on key innovations see Schmidhuber [552]. Some of the earliest work to establish a backpropagation algorithm using gradient descent was developed in 1986 by Plaut et al. [855], which is still the basis for many methods today. Perhaps the best place to start to learn how backpropagation really works is to use open source code packages and DNN software libraries such as Caffe, some of which are listed in the [Appendix C](#) resources.

Backpropagation is *multidirectional* and follows many separate *gradient descent paths* back down the network through all contributing feature weights. Gradients are found using (1) *numeric methods* such as Newton's iterative method which are simpler to implement, provide close approximations, but slower convergence, and (2) closed-form *analytic methods* which can be more accurate, faster to compute, but usually harder to parameterize. In practice, both numerical and analytic methods can be used together as parallel baselines to cross-check each other.

Some fundamental problems for a backpropagation algorithm to solve include:

- Adjusting the backpropagation learning parameters
- Setting convergence criteria and stopping criteria
- Choosing the error minimization algorithm
- Avoiding over fitting
- Training time reductions
- Folding error term corrections back into the weights

The basic order of events for backpropagation includes the following:

1. **Forward pass** through the network
 - (a) Compute responses $f()$ at each neuron for each feature to feed forward
 - (b) Store response derivatives $f'()$ at each neuron for each feature for backprop
 - (c) Compute classification scores and errors
2. **Backward pass** through the network
 - (a) Distribute errors backwards to each contributing neuron
 - (b) Adjust weights using errors and stored response derivatives
 - (c) Compute residuals for current layer, distribute backwards, repeat
3. *Continue until stopping criteria reached (threshold, iterations, elapsed time)*

Ideally, *convex data* is desired with well-defined local minima for use in gradient descent and other backpropagation methods, so input data conditioning can be used to reduce noise, reduce outliers, and hopefully eliminate spurious basins of attraction. Backpropagation methods using gradient descent rely on the neural *transfer function* to provide a nonlinearity to their output response to ensure that the value is *differentiable* for backpropagation. In addition, the nonlinearity may result in faster convergence during backpropagation training to move the gradient more quickly out of flat spots towards the local minima.

Several backpropagation approaches are used (see Schmidhuber [552, 584]):

- Gradient descent variations—uses the chain rule from Calculus
- Conjugate Gradient—approximates the gradients for speed improvement
- Levenberg–Marquardt—a damped least-squares method for pattern matching
- Cascade Correlation—adds unique features instead of training old ones
- Rprop Algorithm—optimized method using partial derivative errors
- LSTM—optimization to gate the introduction of gradients over time
- Quickprop Algorithm—iterative loss function following Newton's method
- SuperSAB Algorithm—adaptive backpropagation for faster convergence

Next we will conceptually describe backpropagation using gradient descent to develop some intuition; however, *see the references given above for actual algorithms* since there is no need here to duplicate the algorithms found in the references. The concepts discussed below do not follow a specific backpropagation algorithm and are for purposes of illustration only.

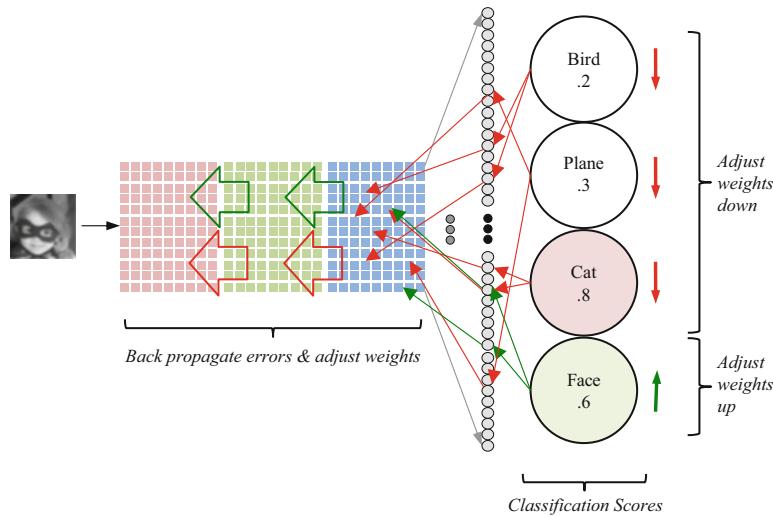


Figure 10.22 An over-simplified example illustrating the final classification error and weight tuning. *Face* feature score .6 is too low, compared to the winning *Cat* score .8 (classification error). Adjust *Face* feature weights up, but adjust *Cat* and other feature weights down

As shown in Figs. 10.22 and 10.23 example, we have four classes to classify: Face, Cat, Plane, and Bird. The class scores are a weighted sum of the trained features * the input. In our example, a Face image candidate is presented to a hypothetical CNN, and the resulting Face score is .6 which is smaller than the Cat score of .8 so there is a classification error. For each image candidate x , the actual output score o for each class is measured against each desired target class score t . For example, we expect a match to be close to 1.0, and a miss to be close to 0. The total error E is the sum of the *difference* between the actual score and desired score for all classes. For example, SSD may be used to sum the total error E :

$$E = \frac{1}{2} \sum_i [o_i - t_i]^2$$

The total network error term E is broken apart and distributed proportionally to all the contributing feature weights in a process called *backpropagation*. Each gradient portion is passed backwards through the network through each contributing neuron as a *signed gradient* to allow for the weights to be adjusted higher or lower. For example, from Fig. 10.23 we can visualize the class gradient as a signed value for each feature class as follows:

$$\nabla_{\text{Face}} \propto + (1.0 - .6) \propto + \left(\frac{\delta E}{\delta \text{Face}} \right)$$

$$\nabla_{\text{Cat}} \propto - (0 + .8) \propto - \left(\frac{\delta E}{\delta \text{Cat}} \right)$$

During backpropagation, the class error gradients are fed backwards to their contributing neural inputs, and then the class error gradient is scaled in proportion to the error contribution of each neuron. Here are the key points to follow the backward pass:

1. Determine the derivative change at each neuron *output* (forward pass)
2. Determine the error contribution at each neuron *output* (backward pass)
3. Proportionally scale and distribute the error to contributing neurons (backward pass)
4. Adjust the weights for this neuron (backward pass)

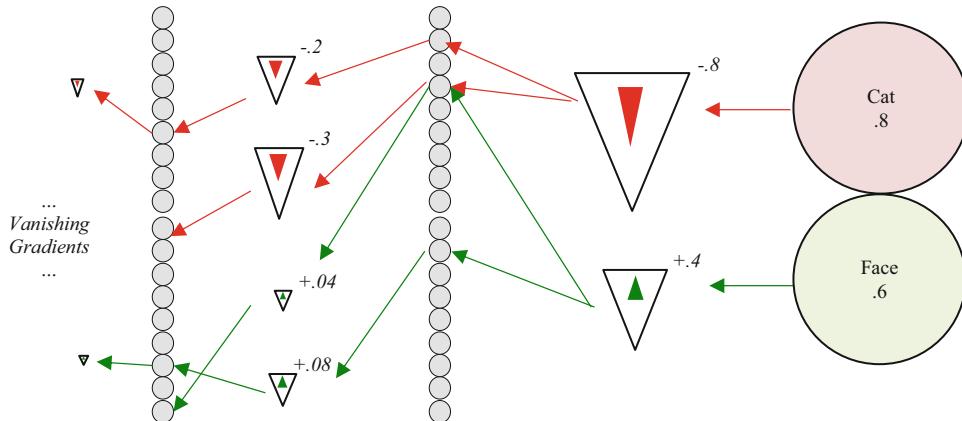


Figure 10.23 This figure illustrates backpropagation of gradients

The network output N is the *functional composition* of all n neural responses, composed of a set of feature weights against which candidate images are classified. To find the source of the classification errors, a *functional decomposition* of the entire network N is made using the chain rule from Calculus to find the *derivative function contributions* N' for each neuron output with respect to each input x :

$$N'_{1\dots n}(x) = \prod_{k=0}^n N'_k(N_{(k+1\dots n)}(x))$$

The weight adjustments are made proportional to the derivative of each neuron response N' . In other words, whatever changes were made to the weights during the last forward pass through the network have contributed to the current error term E , so each neuron's derivative response N' is used as the baseline increment for weight adjustments. As shown in Fig. 10.24, during the forward pass, partial derivatives at each neuron are *stored* to be used during backpropagation. The convolutional neural response N is the *input * weight dot product* followed by the activation function.

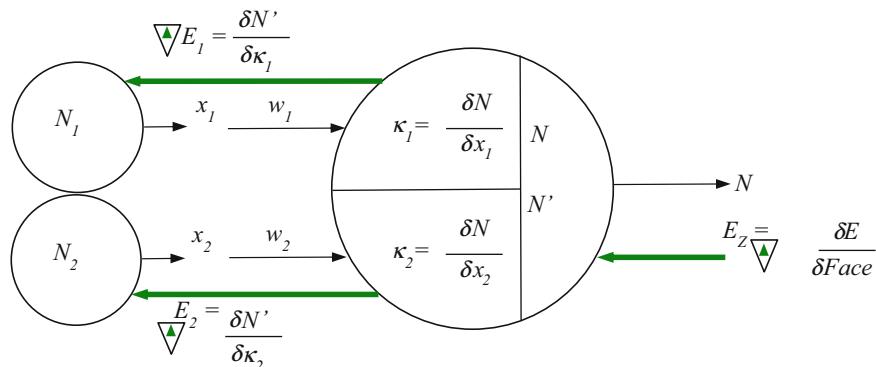


Figure 10.24 The partial derivatives for each neural input s_n are computed and stored during the forward pass, and the neuron output N and its derivative N' are stored as well for use in backpropagation

Fig. 10.24 illustrates how the backpropagated gradient E is fed into neuron N and proportionally scaled into E_Z , and then *the proportional gradient contribution for the neuron N inputs is computed* to backpropagate to the contributing neurons N_1 , and N_Z , and the backpropagated gradient is computed as E_1 , and E_Z . One key to understanding gradient backpropagation is *visualizing the proportional scaling of the gradient as it is backpropagated as an incoming value*. We will work through a simple example below to illustrate the concept and show how the weights are updated.

To begin with, on the forward pass the derivative N' of each neuron response N is stored. We can *visualize N' as $(N_{\text{previous}} - N_{\text{current}})$* which is *the difference between the current response and the previous response*. Likewise, the partial derivatives κ_1 and κ_2 are the *contributions* to neural response N taken as a partial derivative with respect the neural inputs x_1 and x_2 and stored for use during backpropagation.

$$N' \propto (N_{\text{current}} - N_{\text{previous}})$$

$$\kappa_1 = \frac{\delta N}{\delta x_1}$$

$$\kappa_2 = \frac{\delta N}{\delta x_2}$$

We will show how s_1 and s_2 and N' are used to scale the incoming gradient contribution and compute the outgoing backpropagated gradient contributions next.

Intuitively, the stored derivatives N' and κ_1 and κ_2 represent the change in network response from the last round of weight changes to the neuron compared to the current response, providing the basis for weight adjustment relative to the last weight change. Usually, the first hundred or so weight adjustments are larger and hopefully result in faster convergence towards the local minima, and subsequently smaller weight adjustments are made as the partial derivatives become smaller and smaller, and convergence may be slower. The vanishing gradient problem occurs especially in deeper networks as the gradient error becomes infinitesimally small.

To determine the gradient error contribution E_N of the neuron relative to the response derivative N' , the derivative of the backpropagated gradient error E_z is taken with respect to the neuron's response derivative N' :

$$E_N = \left(\frac{\delta E_z}{\delta N'} \right)$$

Note that the feed-forward step computes the response of each weight w with each input x as wx , and since w is the derivative of wx then the weight w is a derivative of the input function, and is therefore used for both the forward and backward directions to proportionally adjust both the input x and the backpropagated gradient error term E_N .

The error contribution for the neuron E_N is combined with each weight to scale the weights to compensate for the neuron N gradient error E_N :

$$\Delta w_1 = \left(\frac{\delta E_N}{\delta w_1} \right)$$

$$\Delta w_2 = \left(\frac{\delta E_N}{\delta w_2} \right)$$

The stored input partials s_1 and s_2 are combined with the gradient contribution of the current neuron N' to propagate backwards to the contributing neuron for each input x_1 and x_2 to produce gradient error terms E_1 and E_2 feed into the input neurons N_1 and N_2 :

$$N_1 \leftarrow E_1 = \left(\frac{\delta N'}{\delta s_1} \right)$$

$$N_2 \leftarrow E_2 = \left(\frac{\delta N'}{\delta s_2} \right)$$

Conceptually, a simple method for adjusting each weight is to change each weight proportionally with respect to the accumulated error, in other words using the partial derivative of E with respect to each weight:

$$\Delta w_i = -\beta \frac{\delta E_N}{\delta w_i} \text{ for } i = 1, \dots, n$$

where β is a learning constant to scale the weight update to limit oscillations.

However, there are many methods used in practice to proportionally distribute and combine the error term with each weight. Typically, a weight tuning function is defined for weight adjustments, incorporating various *learning parameters*. Typically, the gradient value is scaled smaller using a *learning rate* parameter β to prevent too much oscillation due to noise and numerical artifacts. Also, A *momentum* function can be devised to control the weight adjustment using a history buffer of recent error activity, acting as a 1D convolution function across the history buffer to *smooth out the trajectory of the error curve* to reduce noise, overshoot, and undershoot.

Note that in a typical CNN, the feature weight contributions are intertwined, since a single low-level feature weight may contribute to several higher-level features. This makes backpropagation very difficult to describe analytically, and typically very time consuming to compute since the weight adjustments may alternate over the course of training between better and worse for various classes and training examples. Backpropagation is analogous to a *tedious averaging procedure* resulting in features that are tuned generically over all training samples but not specifically for any.

If the weights are immediately adjusted for each forward pass, the gradient descent may not follow a direct path to the local gradient maxima and may oscillate significantly and prevent convergence, or in the best case the oscillations may be beneficial to help avoid shallow local minima. If the weights are updated in batches of training images, there is added storage needed in the CNN to save all the intermediate values. Besides the uncertainty regarding the weight adjustments regarding oscillations and false local minima, the error term shrinks in size and vanishes as it is propagated to each lower layer. Hochreiter et al. [721] is credited with identifying and quantifying vanishing gradient problems. When the gradients are too small, further training is useless. See “Neural Networks Tricks Of The Trade” [654] and especially the chapter on Stochastic Gradient Descent Tricks.

Backpropagation can take days and weeks even on the fastest computers. In fact, backpropagation does not work at all if the learning parameters are not set up correctly, and the difficulties of training with backpropagation, especially for deep networks, are well known in the machine learning community. LaRochelle [694] names *convergence at local minima* and *basins of attraction* in the feature space as two major challenges. It is well known that gradient descent can get stuck in certain local minima before reaching the lowest minima, and that random weight initialization contributes to this phenomenon. In fact, each layer has a different set of local minima, so reaching all the lowest minima becomes cumulatively more difficult with deeper networks. In fact, weight initialization values usually determine where the basins of attraction may be located.

Zeiler et al. [641, 642] developed methods for visualizing the convolutional feature weights to help with devising better backpropagation methods, since actually viewing the weight matrices as images can lead to better feature tuning, see Fig. 9.2. Zeiler found that if the visualized features look about the same or indistinct, perhaps the learning parameters are wrong, and if the visualized features look different and distinct, then perhaps the learning parameters are better.

Backpropagation learning is not inspired by neurobiology, and in fact the dendrites leaving neurons are one-way firing mechanisms generally feeding forward into the visual pathway. Neurobiology does not validate the neural model used by backpropagation learning. In fact, humans learn far faster than backpropagation. It seems more likely that the *view-based models* of the visual pathway are more realistic, adding new views of objects with corresponding the hierarchy of features as needed to build up better models, see Tarr [814] and the discussion of HMAX later in this chapter, and also Appendix F on Visual Genomes [534]. There are no neurobiological mechanisms or connection paths to support backpropagation learning. Note that actual physical neurons use a binary step function for activation and fire *all-or-nothing* across dendrite connections, while the CNN and related ANN models fire through an activation function yielding a firing range or *strength*.

Alternatives to Backpropagation

Note that backpropagation is a *blurry* operation, since it is not entirely clear if all the feature weights should be adjusted indiscriminately in the direction of the gradient, as backpropagation and gradient descent methods typical operate, ignoring the more complex and difficult questions regarding individual feature weight balance and weight independence. Individual weights in each feature could be treated more independently and adjusted up, down, or proportionally. This is a complex problem. Note that we survey other learning methods later, including the NAP architecture which incorporates spatial relationships in the learning process, and adjusts each weight separately using Hebbian learning principles. Also, the dasNet architecture uses novel attentional learning and boosting methods to tune weak and misclassified features, one at a time. See Schmidhuber [552] for more on learning methods.

Note that early CNNs did not use backpropagation, see the description of the Perceptron in the previous section.

Features per Layer

A sufficient number of features at each layer is required to create a robust hierarchical feature model. Too many features and too many parameters are counter-productive, too few features may lead to over fitting and difficulty in training. The feature counts per layer are ad hoc numbers. In some architectures, the CNN feature count increases with higher levels of the network. Later, we survey specific CNN examples to explore feature count as a component of the overall architecture. In Table 10.3 we summarize the number of layers and feature sizes is a variety of CNNs.

Table 10.3 This table showing a comparison of convolutional features is several CNNs

CNN Name ^a <i>typical configuration shown</i>	Year	Total layers	Filtering layers <i>Convolve n × n</i>	Filter sizes <i>Used in various layers</i>	Pooling layers	Classification layers <i>Convolve 1D vector</i>	^a Other layers (not counted in totals)
LeNet5	1998	7	2	(3 × 3), (5 × 5)	2	3	–
Ale × Net	2012	10	5	(11 × 11), (5 × 5), (3 × 3)	2	3	–
NiN ^b	2014	6	4 MLP's	(11 × 11), (5 × 5), (3 × 3), (3 × 3)	0	1 Global Ave. Pool 1000	–
Inception ^a	2014	41	22	(1 × 1), (3 × 3), (5 × 5)	14	5	11
VGG-19	2015	24	16	(3 × 3)	10	3	19
MSRA-22	2015	29	19	(3 × 3)	6	3	–
HMA×	1999	5	2	(3 × 3), ... (29 × 29)	2	1	–
DRL ^c [872]	2015	100–1000	Variable	Variable	Variable	Variable	Skip connections

^a*Inception Note:* Inception's architecture is not a straightforward CNN, therefore difficult to compare, since at several layers there are parallel convolutions, pooling operations and classification operations. So, the author is not sure if the Inception layer counts above are comparable to other CNN's

^b*NiN Note:* Based on the ILSVRC slides for NiN classification [592]

^c*DRL note:* we discuss DRL in more detail later in this chapter in discussed in more detail in the Deep Neural Network Futures section at the end of this chapter.

Table 10.4 This table itemizing convolution compute costs

Convolve size nxn	Operations Per Convolve ($R_p + R_k + I + W$)	R_p Input reads	R_k Weight reads	M Multiply/Add instructions	R_c Convolve Result Write
3x3	28	9	9	9	1
5x5	76	25	25	25	1
7x7	148	49	49	49	1
9x9	244	81	81	81	1
11x11	364	121	121	121	1
13x13	508	169	169	169	1
15x15	675	225	225	225	1

Compute Cost of Convolutional Features and Layers

Each feature carries a compute cost, and each layer carries a compute cost. For features, the larger the kernel size, the larger the compute cost. The current state of the art training methods can take days and weeks given the propensity for deeper networks.

Usually, the minimum feature considered for convolutional layers is 3×3 , since this allows for a center pixel and a 1-pixel neighborhood to be represented, including nine orientations of rotation ($0^\circ, 45^\circ, 90^\circ, 135^\circ, 180^\circ, 225^\circ, 270^\circ, 315^\circ$). In fact, one of the CNNs in this survey, VGG [656], exclusively uses stacked 3×3 convolutions, see Fig. 10.33. However, many CNN architectures in the survey below use a range of different filter sizes together in the same network, ranging from perhaps 1×1 up to 11×11 . See Table 10.4 and Fig. 10.25 for a summary of the hidden compute

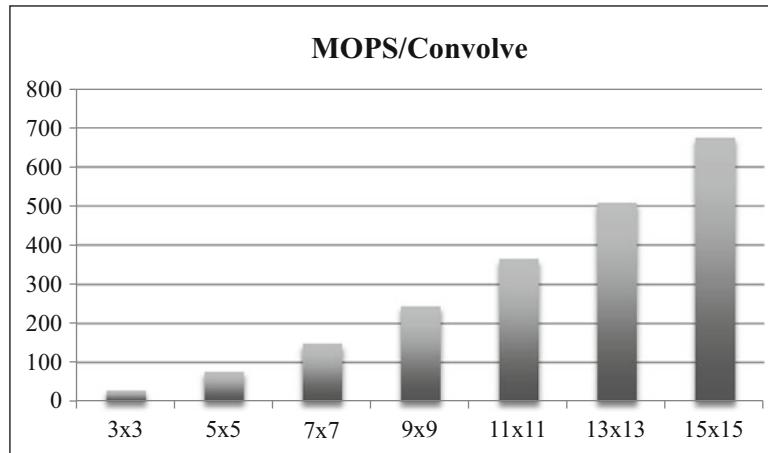


Figure 10.25 This figure shows the compute cost of various sizes of convolution kernels using the data from Table 10.4 across a 1024×1024 image, with a simplistic breakdown of *unoptimized* micro-operations per convolution kernel size, including memory read/write and machine instructions

costs of an unoptimized convolution, assuming all data loaded in registers for single clock-cycle instruction operation, prefetched with no cache misses. Convolutions can be heavily optimized using SIMD instructions, SIMT, memory tiling and pipelining, and dedicated silicon. See Chap. 8 for details on optimization strategies. Convolutions are also optimized using separable kernels, discussed later.

See also the section on Parameters and Hyperparameters below for more information on compute cost. For more discussion on convolutional acceleration, see the discussion on *Boxlets and Convolution Acceleration* in Chap. 8.

Filter Shape and Size

Convolutional networks typically use square kernel shapes for the filters, such as 3×3 or 5×5 . However, square features are the least invariant shape with respect to rotational invariance, see Chap. 4. To compensate for the rotational invariance of the single rectangular features, the training data can be augmented to include rotated copies of the data, and additional features can be trained, so in the end many more features are generated, along with the associated compute and memory cost.

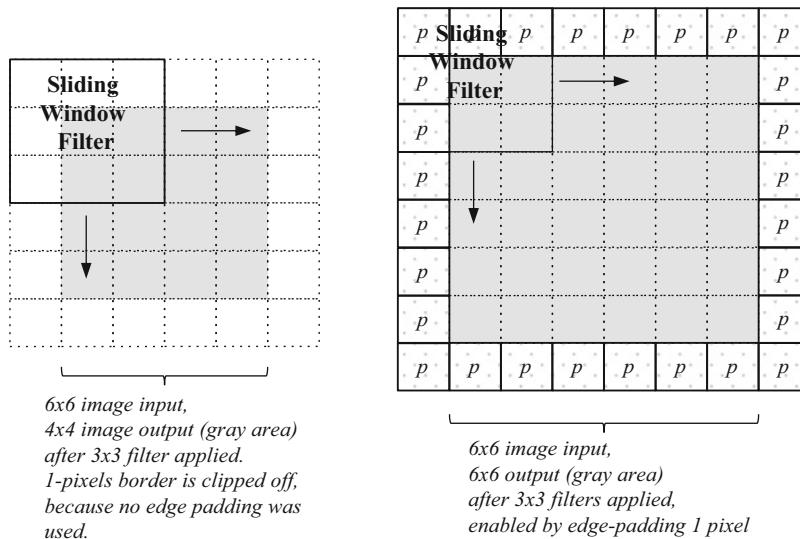


Figure 10.26 This figure illustrates image padding to preserve the input image size

As shown in Fig. 10.26, the filter shape also contributes to image boundary effects. To mitigate the boundary effects of square filters, the input image can be made larger by *padding* one or more pixels around the edges to preserve the true image boundary, which would otherwise be clipped off due to the convolution. Padding can be implemented using a replicated copy of the boundary pixels, the mean value of the entire image, and other methods.

In practice, the filter size at each layer is empirically determined to find the optimal size and set size granularity for a given data set. According to LeCun [682], for the MNIST data set composed of rather small 28×28 pixel images, each containing handwritten characters and numbers, a 5×5 convolution kernel for the first layer proved to be beneficial. For other datasets using larger image sizes such as 256×256 and up, larger filter shapes have been used at the first layer and higher layers, such as 11×11 or 15×15 for the first layer. However, *stacked convolutions* using a pipeline of smaller kernels such as size 3×3 as shown in the VGG architecture surveyed later, can be a good alternative to large kernels, with the added benefit of reducing compute cost. Stacked kernel are discussed next.

Stacked Convolutions

It is possible to stack several smaller convolutions together to approximate a larger convolutional kernel. As shown in Fig. 10.27, a stack of three 3×3 kernels can be used to approximate a 7×7 kernel, since the first convolution reduces the 7×7 image to 5×5 , and the second convolution reduces the image to 3×3 , and the final convolution produces 1 pixel output. Simonyan and Zisserman [656] demonstrate a very deep convolutional network called VGG, using up to 19 layers of exclusively 3×3 convolutional features, noting that stacked 3×3 convolutions are a viable alternative to larger kernels, and offer equivalent accuracy. For example, by looking at the total receptive field, it is apparent that two 3×3 kernels stacked together is equivalent to a 5×5 kernel. The key advantage of smaller kernels is computational efficiency.

Several practitioners have attempted to reduce the computational burden of *larger convolutional kernels* by using a striding factor to skip pixels during input kernel assembly. For example,

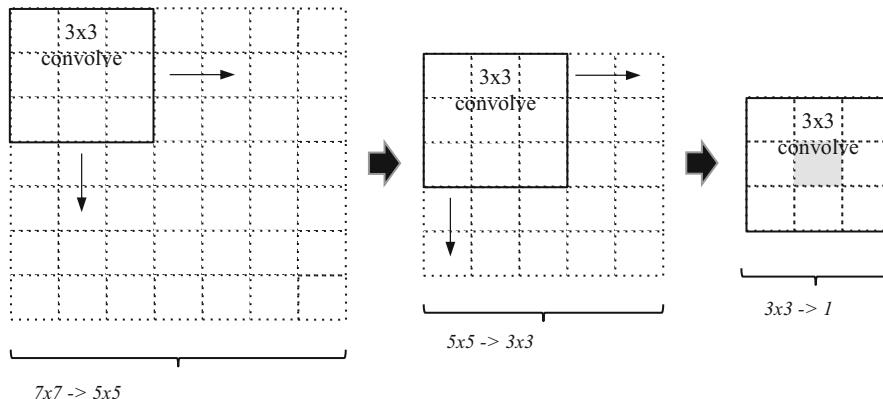


Figure 10.27 This figure illustrates stack convolutional equivalence, showing three stacked 3×3 convolutions applied to the 7×7 input image yield a 5×5 , followed by a 3×3 convolution to the 5×5 image yields a 3×3 image, followed by a 3×3 convolution on the 3×3 yielding a single pixel output, yielding the same effective receptive field coverage as a single 7×7 kernel

Khrishevsky et al. [640] developed an architecture called *AlexNet*, as shown in Fig. 10.30, using several kernel sizes, including 11×11 , 5×5 , and 3×3 , and to mitigate the performance for the 11×11 kernels, a stride of 2 is used. Stacked convolutions have limited usefulness, since a larger kernel such as 11×11 may be the right choice depending upon the input data and the application, since larger kernels capture more local region information to describe higher-level object features.

Stacking convolutional kernels is also claimed [656] to add some *regularization* by adding rectification or nonlinearity after each 3 kernel, forcing a decomposition through the 3×3 filters in the chain, claimed to add increased discrimination to the final result. And of course, the number of parameters is greatly reduced for smaller convolutions, perhaps increasing performance depending on the overall architecture. For example, assuming the inputs and outputs to a stack of convolutions are the same C , a 3-channel RGB input to the stack of 3×3 convolutions uses $3(3^2C^2) = 27C^2$ weight parameters (excluding the bias parameters) to cover a 7×7 region, while a single 7×7 convolution uses $(7^2C^2) = 49C^2$ weight parameters (excluding the bias parameters).

In summary, stacked smaller convolutions rather than single larger kernels are a viable alternative in some applications to reduce the number of weight parameters.

Separable and Fused Convolutions

Another approach for reducing convolution overhead is to use *separable* filters. Many 2D filters, such as convolutions, are separable filters, which can be broken down into row and column operations. For example, the Discrete Fourier Transform is a separable filter and can be broken down into overlapping 1D FFTs across a 2D or 3D field. Convolutions can also be implemented as separable filters. This means that rows and columns can be processed independently as 1D vectors, and vectors can be processed using fast SIMD instructions to accelerate processing.

For example, a 2D Gaussian blur kernel can be *separated* into two 1D kernels as follows:

$$\frac{1}{4} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Another approach is to *fuse* several convolutional filters together, collapsing into a single filter. Since convolution is associative and implemented as multiplication in the frequency domain and

multiplication in the spatial (pixel) domain, a set of convolutional filters can be convolved together, or pre-convolved, yielding a *single filter* representing a set of filters. For example, Mamalet et al. [664] developed a method to apply *fused convolutions* to combine the filter, activation function, and pooling function and approximate the same results within <1 % of the same accuracy. Although the fusion is not perfectly equivalent to the separate steps, the performance is increased by 2–6×, depending on the configuration, and whether the network is in the training phase or the operational matching stage. In addition, Mamalet developed backpropagation methods for fused and separable filters.

Convolution vs. Correlation

The terms *convolution* and *correlation* are often used interchangeably, although the actual mathematics and intended use is often different. We will discuss some of the possible points of confusion here.

The term convolution is used very imprecisely in CNN discussions, and what is meant is the *dot product* of the input vector \mathbf{I} and the feature weight vector \mathbf{W} :

$$\mathbf{I} \cdot \mathbf{W} = \sum_i I_i W_i = I_0 W_0 + I_1 W_1 + \cdots + I_n W_n$$

In fact, mathematically convolution and correlation are closely related, since convolution is equivalent to correlation by simply rotating the convolution template by 180°, or in other words reflecting the rows and columns of the matrix. A variant of correlation, *normalized correlation*, is computed to allow for invariance to scale using a scale factor x :

$$\frac{\sum_i (I(i)W(x+i))}{\sqrt{\sum_i I(x+i)^2} \sqrt{\sum_i (W(i))^2}}$$

In fact, the term convolutional neural network is somewhat confusing, since both correlation and convolution are implied. And coming from the image processing perspective, some confusion arises also, since the term *convolutional filter* is an image processing operator, and the term *correlation* is either a statistical metric or a template matching feature detector. However, both methods are mathematically about the same. Convolution is typically used to change the image, and correlation is used as a hit-or-miss pattern matching operation, which can alternatively be implemented with morphology operations, see Chap. 2. Typically correlation kernels represent patterns, shapes and structure used for template matching. Correlation matching occurs pixel-by-pixel within the image region, and correlation strength is measured using a similarity metric (several methods are used such as SSD, see Chap. 4 for a discussion on distance metrics) to find the difference between corresponding pixels in the template, and sum a final score. See [683] for a comparison between correlation and convolution.

In image filtering operations, convolution kernels most often are designed using a *scaling factor* to condition the results as needed. The convolutional scaling factor is analogous to the *bias factor* parameter in CNNs, which is tuned along with each feature during CNN training. So each convolutional feature consists of a kernel with individual weight parameters and a bias factor parameter, for example a 3 × 3 feature kernel consists of 9 + 1 tunable parameters:

$$\text{bias} \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} \quad \text{or alternatively bias} + \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix}$$

The bias factor may also be implemented as an additive scaling factor, instead of a multiplicative scaling factor, with the convolution kernel.

Convnets use convolutional filters for two purposes: as feature weight matrices analogous to correlation templates, and as filters to change the input image to produce output for the next convolutional layer. In CNN parlance, filters produce *feature maps* containing latent features.

For symmetric features built around a central origin, such as Gaussian blur kernels, convolution and correlation are essentially the same. However, the learned features in Convnets are not constrained to be symmetric, and rarely are.

For example, a nonsymmetric edge detector as used in a convolution:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

is equivalent to correlation using the same kernel rotated 180°:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

For more information on convolution and correlation, see Chap. 2. For more information on correlation used as a template matching feature detector, see Chap. 6.

Pooling, Subsampling

Pooling is another name for subsampling in CNN parlance (i.e., image rescaling in image processing parlance), and several methods are discussed in Chap. 9. Pooling is typically used as the last step, or one of the last steps, in each convolutional layer. The Neocognitron [571, 679] was the first CNN to use pooling in 1980. The stated goal for pooling is typically to add some translational invariance, or to simply subsample the image smaller to reduce compute and parameters. However, the subsampling methods used in ANNs are different than standard computer graphics sampling methods such as linear interpolation and other anti-aliasing methods. We discuss several variants here.

For CNNs, simple approaches such as taking the average value of the pool have been used. However, the most popular method seems to be the RANK filter method (see Chap. 2) to selecting the MAX value of the local region. Choosing the MAX value seems intuitive and obvious, since the MAX activation in a local region is analogous to the strongest feature, while using the average value implies uncertainty.

Is the pooling layer needed at all? Dosovitskiy et al. [684] argue that replacing the max pooling layer by a convolutional layer with a stride of at least 2 is equivalent in terms of accuracy. Graham proposes to create variable sized max pooling regions [685]. In practice, many of the CNNs we survey do not always use a pooling layer. However, pooling operations to subsample or upsample the images are often required to size the outputs to match the inputs for a fully connected classification layer.

Convolutional neural layers *also subsample* the image, and larger kernels, such as 11×11 , may be used to produce more subsampling than smaller kernels. In addition, a stride factor for the sliding window larger than 1 also subsamples the image. So the combined subsampling effect of the convolution kernel, stride, and the pooling region size is an important consideration for understanding the level of detail represented by the features.

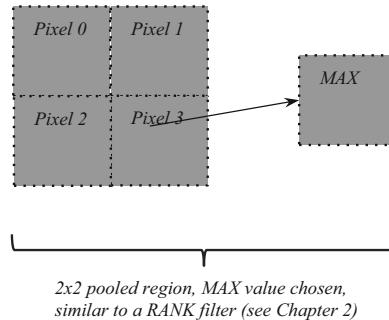


Figure 10.28 This figure illustrates MAX pooling, where a RANK filter is run over a local region and the MAX value is selected to assemble the final output image

Parameters and Hyperparameters

Several practitioners separately summarize (1) the *architectures parameters*, and (2) the *learning hyperparameters* for a CNN. Architecture parameters are useful to summarize the moving parts in the CNN, which corresponds to training difficulty and overall performance. CNN parameter counts typically ranging from millions to billions of parameters for the entire network. The hyperparameters used for learning, such as momentum and learning rate, are discussed in the *Backpropagation* discussion in this chapter.

CNNs contain a dizzying number of parameters comparable in magnitude to the neurons in the human brain, which contains billions of neurons and connections, see Table 9.1. Fortunately CNNs have a much more regular architecture, which simplifies the architecture. We encourage the reader to develop a mental visualization of CNN architecture parameters in order to better understanding the fundamentals.

Architecture Parameters

Parameters are the program code for the CNN. The architecture is put in place to program itself via tuning the learning parameters and the training data, resulting in feature learning. The parameters are analogous to the *neural DNA code* in the brain. When analyzing architectural complexity, we may be interested in analyzing parameters to compare design alternatives. For example, we may want to explore the cost of using larger images as input, more image channels, larger features, or deeper networks. Since each parameter implies corresponding memory and compute operations, parameter analysis is useful for estimating training time and run-time performance.

Key architecture parameters types include:

- *General Parameters:* typically computed, for a convolutional layer, as the total of $(\text{input_feature_maps} \times \text{output_feature_maps} * \text{weights_per_kernel}) + \text{bias factors}$ for each kernel at current layer. The parameters are the *neural DNA code*.
- *Neurons:* the number of *artificial neurons simulated* in the network. The actual number of neurons may not be simultaneously in operation, since artificial neurons are typically shared *serially* to process for one feature at a time, rather than running simultaneously in parallel for all features. However, parallel implementations are used as well to increase performance at the cost of system size.
- *Connections:* the total number of connections between artificial neurons. This is a simple measure of *artificial dendrites and axons simulated* in the network.

Here we provide a few methods to compute architecture parameters, which may be different than the methods used by other practitioners to compute parameters, operations and connection complexity, where:

Subscripts denote:

t = totals

l = layer

$l - 1$ = previous layer

n = single neuron

Parameters:

n_t = total neurons for network

c_t = total connections for network

p_t = total parameters for network

s_l = total parameters in a single convolutional layer

i_l = inputs to convolutional layer, from prior layer,

either $n \times n$ sliding windows from each feature map for kernel filtering

or fully connected weights from each feature map for 1D convolve

d_l = depth, number of channels for this feature map

f_l = number of features for this layer

m_l = number of feature maps output for this layer ($m_l = f_l$)

f_{l-1} = number of features for previous layer

m_{l-1} = number of feature maps output from previous layer ($m_{l-1} = f_{l-1}$)

k_l = number of feature weights in each $n \times n$ kernel, for example $3 \times 3 = 9$

b_l = bias factor, 1 per feature (=total features per layer)

x_n = operations per neuron, such as normalization, max pooling, etc.

(1) Neurons: For 2D convolutional filtering layers, each neuron receives a single $n \times n$ kernel input window (kernel-connected), counted as 1 input, from each feature map in the prior layer. For each 1D convolutional 1D classifier layer, each neuron receives fully connected inputs from each weight in each feature from the prior layer.

$$(\text{total neurons for network}) n_t = \sum_{l=1}^{\text{total layers}} i_{l-1} m_{l-1} f_l$$

(2) Parameters: Simple parameter total for convolutional neurons, including the weights and biases.

$$(\text{network}) p_t = \sum_{l=1}^{\text{total layers}} m_{l-1} k_l m_l + b_l$$

(3) Operations: The activation function and convolution are implied as default parts of the neuron, and are not counted as separate operations here. Instead, operations are extra functions in the neural pipeline, including preprocessing (*dropout*, *numeric conditioning*, ...), and post-processing (*response normalization*, *pooling*, ...). We consider operations here as a serial part of the artificial neuron pipeline (part of the hidden unit), rather than a separate layer. Operations as a metric are a simple measure of neuron complexity, though some operations are more complex than others. Operation functions can independently be assigned an *error term* during backpropagation, which some practitioners refer to as *loss weights*, to allow each operation function to be tuned to reduce their error contribution, for example by adjusting the parameters of the operation.

$$(\text{network}) \quad x_t = \sum_{l=1}^{\text{total layers}} x_n n_l p_t$$

(4) Connections: we count *connections* as inputs from each feature map at the previous layer to each weight in each filter at the current layer. An input may be either a kernel-connected window, or fully connected weights as per the 1D convolve layers.

$$(\text{total connections for network}) \quad c_t = \sum_{l=1}^{\text{total layers}} i_{l-1} m_{l-1} k_l$$

Note that the first fully connected 1D classification layer may have more parameters than other layers, since it is the first *fully connected* layer, taking input as each feature weight from the last convolutional layer fully connected to each neuron in the classification layer.

Table 10.5 This table illustrating hypothetical network parameters

Convolutional Layers	Parameters	Neurons	Connections
Layer 0 - INPUT 640x480 image monochrome, 1 chan. <i>fully padded output</i>	640x480x1 * <i>*Fully padded</i> <i>i.e. 642x472 for 7x7 kernels</i> P = 307,200		
Layer - 1 FILTER 256 bias factors 256 7x7 features (op*1) mean-zero norm (op*2) ReLu activation <i>fully padded output</i>	(1 * 7x7 * 256) + 256 P = 12,800	640x480 * 256 N = 78,643,200	640x480 * 256 * 7x7 c = 3,853,516,800
Layer 2 - FILTER 512 bias factors 512 5x5 features (op*1) ReLu activation <i>fully padded output</i>	(1)(256 * 5x5 * 512) + 512 P = 3,277,312	640x480 * 512 N = 15,7286,400	640x480 * 512 * 5x5 c = 3,932,160,000
Layer 3 - FILTER 1024 bias factors 1024 3x3 features (op*1) ReLu activation <i>fully padded output</i>	(1)(512 * 3x3 * 1024) + 1024 P = 4,719,616	640x480 * 1024 N = 314,572,800	640x480 * 1024 * 3x3 c = 2,831,155,200
Layer 4 - CLASSIFY 8192 1d vector weights	1024 * 3x3 * 8192 P = 75,497,472	1024 * 8192 N = 8,388,608	1024 * 8192 * 3x3 N = 75,497,472
Layer 5 - CLASSIFY 4096 1d vector weights	8192 x 4096 P = 33,554,432	8192 x 4096 P = 33,554,432	8192 x 4096 * 1 P = 33,554,432
Layer 6 - CLASSIFY 1024 1d vector weights	4096 x 1024 P = 4,194,304	4096 x 1024 P = 4,194,304	4096 x 1024 * 1 P = 4,194,304
Layer 7 - OUTPUT Labeling	1024 Softmax	-	-
TOTALS	121,563,136	596,639,745	10,730,385,408

Learning Hyperparameters

Learning Hyperparameters include a range of variables and constants used in the DNN training process, such as initial feature weights, bias, momentum, learning rate, and several other parameters. We do not delve deeply into learning parameters here, and instead refer the interested reader to better

references provided in the backpropagation section and elsewhere in this work. Hyperparameters are known to be difficult to choose, understand, and control, see for example LeCun et al. [654] “*Neural Networks, Tricks of the Trade*”.

From statistical analysis, the term hyperparameter is determined from a *prior*, and therefore must be initialized intelligently, and optimized from there. However, many DNN practitioners argue that there are no known methods to initialize DNN training hyperparameters, and advocate empiricism [644]. In this respect, we may say that DNN practitioners often use a *best guess*, rather than statistically generated hyper-parameters. However, for feature weight initialization, *transfer learning* is effective, and follows the statistical analysis pattern of using priors, since existing trained feature weights are reused to initialize weights, usually at the lower levels of the CNN. Besides transfer learning, there seems to be little formal guidance to follow to initialize and tune hyperparameters. See the discussions in Chap. 9, and LeCun [687].

Next, we will move past the convnet fundamentals discussed above, and begin a survey of several illustrative examples of CNN architectures.

LeNet

LeNet is the canonical CNN architecture developed by Yann LeCun [574, 575] in the late 1980s, and is a good starting point for understanding CNNs. Probably Yann is most associated with the research, development, and successful deployment of *practical* Convnets to solve real-world problems [682], such as commercially deployed hand-writing recognition systems for zip code recognition for postal sorting, and handwriting recognition systems for bank check processing. Given that fundamental Convnet research was completed in the mid-1980s, and sufficient compute power was available, LeCun was able to synthesize and improve the Convnet architecture in the LeNet architecture [574, 575] which has progressed up to and beyond the LeNet5 version, forming the basis for most of the DNNs used in academic research and industry today.

Here we survey an early LeNet architecture [576], which was deployed by the US Postal Service for mail sorting. The LeNet architecture was one of the first to successfully apply backpropagation to a real world problem [577].

The training data consisted of 9298 individually digitized handwritten numerals or digits, taken from hand-written zip codes from actual mail, written in many sizes and styles. The training data also contained a sampling of unrecognizable, ambiguous and misclassified digits. For training data preparation, each digitized numeral was rendered into a 16×16 pixel template using linear interpolation to preserve the aspect ratio of each digit. Random, erroneous markings surrounding each digit were manually removed via pixel editing.

Each 16×16 input image was processed as a gray-scale image, preprocessing using mean-zero normalization in the range $-1 \dots +1$. The input images were fully padded around the edges using a value of -1 . So including padding, the input features are treated as 24×24 pixel images to allow for complete 5×5 input window sampling of each pixel in the 16×16 template by padding 4 pixels on each side ($16 + 4 + 4 = 24$). Subsequent layers included padding for the input feature map images as well. Weights were initialized with random values, which is a form of primitive edge detector.

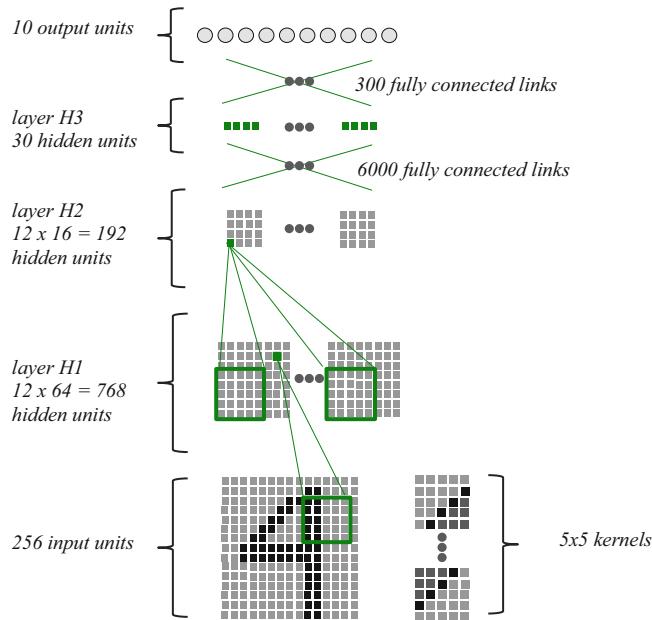


Figure 10.29 The basic LeNet architecture, after LeCun from 1986 [576], used for digit recognition

One goal for LeNet was to reduce computational overhead, so careful attention was paid to reducing parameters. Weight sharing was used to allow for one feature at a time to be searched for. The filter weights and bias were shared for all neurons simultaneously in the same layer, so *virtual neurons* are implemented to share weights instead exhaustively implementing all neurons with their own copy of the weights—*shared weights and virtual neurons are an innovation successfully demonstrated In LeNet*. While this seems natural, LeCun et al. was among the first to employ this technique in the context of a regular convolutional network. Note that LeCun used the terminology *feature map* to represent the result of convolving weights with the input, which is often confusing, since the result of convolution is just another *image* and not a feature at all, since the *weights* are features. However, the term *latent features* is used in CNN parlance to describe the features lurking in the feature maps, waiting to be discovered at the next layer.

The architecture is illustrated in Fig. 10.29. There are three hidden layers (convolutional layers), labeled H1, H2, and H3. Note that all images at each layer are fully padded. The input (bottom) is a 16×16 mean zero normalized gray scale template. The input kernel window size is 5×5 strided at each pixel. The layer H1 output feature map size is 8×8 , a $2 \times$ reduction from the 16×16 input, due to the 2-pixel stride resulting in an *undersampled* input image. The rationale for the reduction was to provide some translational invariance. Likewise, the 2D input undersampling was used for layer H3 resulting in a reduction from 8×8 to 4×4 feature maps. No pooling layers were used in the first version, but subsequent LeNet architectures added pooling and subsampling layers [682] equivalent to undersampling the input at a stride of 2 or more.

Today, some practitioners refer to a *hidden layer* as a complete convolutional layer, and a *hidden unit* as a single convolutional neural processor in a hidden layer. However, note that LeCun used the term *hidden unit* for a single pixel in a feature map, which is equivalently the output of a neural processor. Each LeNet hidden unit is computed using convolution weights, whether they be kernels in filter layers, or 1D vectors in fully connected layers. H1 contained 12 feature maps of size 8×8 , for a total of $12 \times 8 \times 8 = 768$ hidden units. H2 contained $12 \times 4 \times 4 = 192$ hidden units. H3 is a *fully*

connected layer with the largest hidden unit count at $30 \times 192 = 5750$ hidden units + 30 biases = 5790. The input contained 16×16 pixels, or 256 hidden units. The output was a decimal classifier with ten outputs, one per digit. In summary, layer H1 contained $12 \times 5 \times 5$ features, H2 contained $12 \times 5 \times 5$ features, and H3 contained a 1D vector with 30 weights, fully connected to the 192 weights from H3.

LeCun is also a pioneer in hardware accelerated CNNs. For example, the initial LeNet was implemented using a DSP for acceleration, which is a departure from most academic research which is usually only concerned with exploring new concepts, and leaving the optimizations to applied researchers and engineers. In addition, Farabet, LeCun, and others [689, 690] developed a hardware accelerated CNN using an FPGA and later an ASIC, which was the basis for a start-up company.

Name	LeNet
ANN type	CNN
Memory Model	Simple, fixed
Input Sampling	Sliding window stride=2
Dropout, Reconfiguration	-
Pre-Processing, Numeric Conditioning	Mean zero input normalization
Feature Set Dimensions	$5 \times 5 \times 12, 5 \times 5 \times 12, 30 \times 192, 1 \times 30$
Feature Initialization	Random
Layer Totals	2 filter, 1 classify
Features, Filters	Convolutional
Activation, Transfer Function	Sigmoid
Post processing, Numeric Conditioning	-
Pooling, Subsampling	Later versions used $2 \times 2 \times 2$ ave. pooling

AlexNet, ZFNet

The LeNet architecture is the basis for several modern CNNs including the AlexNet CNN variation developed by Krizhevsky et al. [640], which was the first CNN to realize the potential of Convnets on large natural image data sets.³ AlexNet has likewise become the basis for subsequent CNNs used for image recognition.⁴ In fact, Zeiler and Fergus [641, 642] improved the AlexNet architecture, referred to as ZFNet, which was subsequently commercialized in a start-up called Clarifai. Part of the success of ZFNet is due to Zeiler's method to visualize learned features corresponding to the pixel regions they match in the input image, referred to as *deconvolutional networks*, to develop intuition about how to enhance the learning hyperparameters to improve weak features. In addition, ZFNet reduced the number of hyperparameters, and expanded the convolutional filtering layer depth, improving the accuracy by several percentage points.

The basic AlexNet architecture is shown in Fig. 10.30. Several major innovations were introduced to provide optimizations for engineering efficiency (unusual for academic work), which we will survey here:

- Optimized to run across two GPUs in parallel
- Overfitting mitigation techniques
- Training time algorithm optimizations

³ Imagenet competition 2012, “Supervision”, <http://image-net.org/challenges/LSVRC/2012/results.html>.

⁴ A version of AlexNet is implemented in the Caffe open source package, and accelerated by NVIDIA, see <http://caffe.berkeleyvision.org>.

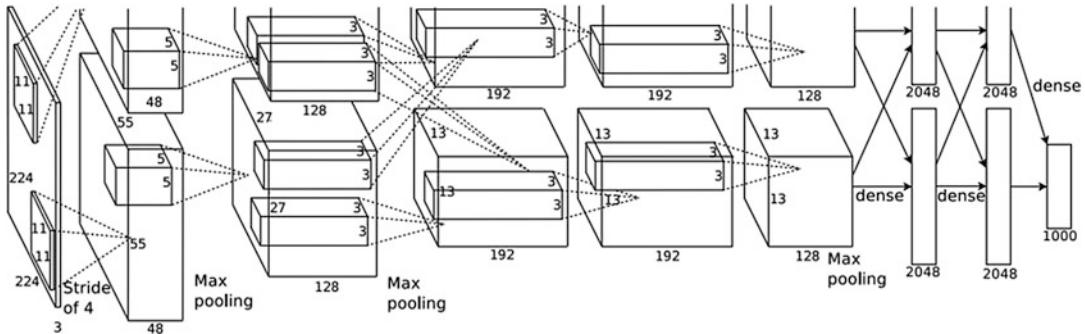


Figure 10.30 This figure shows the AlexNet CNN architecture, with two parallel paths for implementation on two GPUs. Image used by permission, © Alex Krizhevsky from [640]

As shown in Fig. 10.30, AlexNet contains five convolutional filtering layers, and three 1D classification layers. Fewer layers were tried, but accuracy decreased. As discussed in Chap. 9, Rectified Linear Units (ReLU) were used for the activation function, a departure from sigmoid shaped functions commonly used at the time. Training time is reduced using ReLU since it uses a simpler, ramp style function. Also, ReLU's can operate on unnormalized data without saturation, eliminating the compute cost of normalization during preprocessing. The GPU parallelization was designed to run half of the feature kernels on each GPU, with minimal communication between GPUs. In fact, the memory limitations of the GPUs were one of the primary reasons that the parallelization was followed.

One interesting side effect of the dual-GPU partitioning in AlexNet is that each GPU focused on learning an independent feature set. As shown in Fig. 10.31, one GPU generated the top features, which are mostly gradient-style monochrome features, while the other GPU generated the bottom features which are mostly color blob-style features. Fig. 10.31 also serves to illustrate the point that CNNs will generate different features based on the convergence of the gradient descent algorithm for the given data, the method in which each weight is actually updated and tuned, and on the initial values of the weights, all of which affect convergence. Gradient descent and backpropagation are



Figure 10.31 This figure from Krizhevsky et al. [640] showing the partitioning of features learned on separate GPUs, (top) gradients learned on GPU A, and (bottom) color blobs learned on GPU B. Note: same training data split between GPU A and B. Image used by permission, © Alex Krizhevsky

difficult to visualize, predict or control. The parallel GPU results for AlexNet are similar to cloned humans with identical DNA: both individuals are formed by their conditions and experiences, leading to different outcomes.

AlexNet uses *response normalization* over a local region in the convolutional output feature map using a fairly compute intensive function:

$$x_t = a_{x,y}^j / \left(k + \alpha \sum_{j=\max(0, i - \frac{n}{2})}^{\min(N-1, i + \frac{n}{2})} (a_{x,y}^j)^2 \right)^\beta$$

The local response normalization includes contributions from overlapping local regions in the output feature map. Hyperparameters are determined from the training set, including k , a , α , and β . The normalization objective was a brightness adjustment, similar to other methods like local histogram equalization or a LUT ramp, see Chap. 2. Note that some recent practitioners do not use response normalization and find no benefit to justify the compute cost [656]; however, Krizhevsky reported that response normalization was advantageous to decrease the error rates.

AlexNet used a method of pooling via overlapped regions, rather than nonoverlapping regions used in other CNNs of the day, and reported <1 % accuracy improvements compared to nonoverlapped pooling.

Memory and training time are the key bottlenecks that limit the architecture, so Krizhevsky's architecture can be adjusted to fit within the available memory and training time budget. The training data set (ILSVRC) included over 1.2 million images with 256×256 RGB resolution, mean-normalized. New training samples are introduced using mirrored and translated version of each sample, to incorporate multiple views into the scoring. Also, the RGB channels for each image are preprocessed using a PCA-based algorithm to select dominant components and add variance to the intensity and color of the images, which reduces the error rate <1 %. During training, dropout is used to randomly set the output of half of the neurons to zero on fully connected layers. During test time, the output of each neuron is multiplied by 0.5 to approximate the geometric mean of the cumulative dropout effects. Weights are initialized from a zero-mean Gaussian distribution, and biases are initialized to 1.

In summary, the AlexNet architecture was the first to demonstrate the potential of CNNs for natural image datasets, enabled by GPUs for compute acceleration and large labeled datasets.

Name	AlexNet
ANN type	CNN
Memory Model	Simple, fixed
Input Sampling	Sliding window stride=4 1 st layer, 1 otherwise
Dropout, Reconfiguration	Dropout 50% on fully-connected layers
Pre-Processing, Numeric Conditioning	Mean-zero normalization
Feature Set Dimensions	11x11x96RGB, 3x3x256RGB, 3x3x384RGB, 3x3x384RGB, 3x3x256RGB, 1x4096RGB, 1x4096RGB
Feature Initialization	Gaussian distribution
Layer Totals	5 filter, 2 classify
Features, Filters	Convolutional
Activation, Transfer Function	ReLU
Post processing, Numeric Conditioning	Local Brightness EQ
Pooling, Subsampling	Max pooling 2x2x2

VGGNet and Variants MSRA-22, Baidu Deep Image, Deep Residual Learning

The VGGnet architecture developed by Simonyan and Zisserman [656] has been highly influential, spawning variants from Microsoft MSRA [670] and Baidu [700], so we briefly discuss VGGNet and variants here, with a more detailed survey on each variant later in this section. At the time of this writing, the VGGNet variants from MSRA and Baidu, along with Google Inception, are achieving almost identical state-of-the-art results *within a few tenths of a percentage point difference*, which won several Imagenet competitions. Based in part on VGGNet, the current leader in most Imagenet competition categories as of 2015 is the Deep Residual Learning (DRL) method of He et al. [872], discussed in more detail in the Deep Neural Network Futures section at the end of this chapter, which supports very deep networks—over 1000 layers have been tried—the deepest DNNs to date.

Simonyan and Zisserman [656] developed the first versions of VGGNet using up to 19 layers of 3×3 convolutional kernels rather than an assortment of larger kernels, with minimal other operations, namely pooling and ReLu, with excellent results. The central concept was to use *stacked convolutions* to reduce the parameter count and increase performance. As explained in the *Stacked Convolutions* section above (*worth reviewing prior to reading this section*), a stack of three 3×3 convolutions covers the same receptive field and approximates a 7×7 convolution; however, using stacked convolutions uses far fewer parameters, resulting in significant performance advantage at training time and run time. However, it may be argued that larger features such as 11×11 or 15×15 are invaluable for some applications requiring more detail to describe features, where 3×3 stacked reductions may not work due to the limited receptive field.

VGGNet has inspired some notable CNNs. Note that Microsoft MSRA [670], surveyed later, also used a modified VGGNet architecture with 22 convolutional layers, with other modifications such as spatial pyramid pooling [542], parametric ReLu (PreLu) [670], improvements to weight initialization methods, some larger convolution kernels, and more aggressive downsampling. Baidu Deep image [700], surveyed later, also used a 19 layer VGGNet with a huge investment in the training protocol

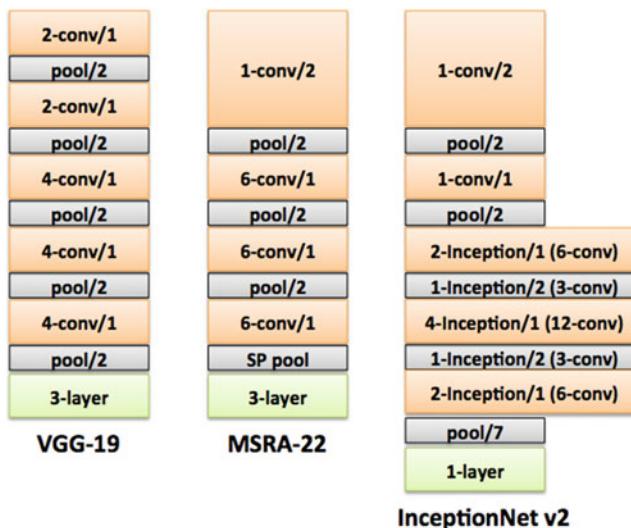


Figure 10.32 This figure shows (left) the VGGNet-19 variant used by Baidu, (center) the Microsoft MSRAA 22-layer VGGNet variant, compared with (right) the 33-layer Google InceptionNet V2 architecture. Together, these architectures currently (2015) represent the state-of-the-art in accuracy. Image © Karen Simonyan, used by permission, see also [701]

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 10.33 This figure shows six different deep architectures using exclusively 3×3 convolutions, image from Simonyan and Zisserman [656] in CVPR, © Springer-Verlag used by permission

incorporating vastly more training samples than anyone else to date, assisted by a custom Baidu-designed DNN supercomputer and other customer hardware achieving 1.9PFlops.

For the first generation VGGNet, the input images are 224×224 RGB, which are preprocessed by subtracting the RGB mean from each pixel. The training process involves random color shifting, cropping, horizontal mirroring, and *scale jittering* by randomly rescaling the training samples to add scale invariance in the range $[256 \times 256 \dots 512 \times 512]$. Scale jittering increased accuracy by $\sim 1\%$, however, at a compute cost. In addition, horizontal mirroring of test images is applied, and at the softmax layer the average score of the original and mirrored version is used for scoring. Random RGB color shifting is also used on training samples; however, this seems to be more of a guess than a rifle shot, yielding less than 1 % improvement, since shifting color in RGB space is quite ambiguous and unnatural, see Chap. 1 for more information on color spaces. Multiple crops of each image were evaluated but proved to be insignificant, equivalent to fully padded dense evaluation using overlapping kernels.

The basic VGGNet architecture is based on AlexNet.⁵ Several network depths were tried as shown in Fig. 10.33, including various sizes of convolution stacks followed by MAX pooling at stride 2. To preserve spatial resolution, the stride is fixed at 1 to incorporate each pixel, and full edge padding is used. Each convolution uses the ReLu nonlinearity. The authors note that local response normalization at the output was tried, and proved to be costly and did not improve performance. The best results

⁵ VGGNet and AlexNet can be fully implemented in the open source Caffe neural net package, and the best performing VGGNet models are available as Caffe configuration files, see http://www.robots.ox.ac.uk/~vgg/research/very_deep/.

were obtained by exclusively using deeper stacks of 3×3 convolutions. Three fully connected classification layers are followed by a soft-max classifier.

Weight initialization was done in two steps. First, random weights were zero-mean normalized and trained on network A shown in Fig. 10.33. Then the weights were transferred to the other layers and training continued. The authors also recommend the training procedure proposed by Glorot and Bengio [697] for better random weight initialization. Training follows a mini-batch protocol.

For classification testing, all input images were scaled to a uniform minimum size for a test scale, which could be different than the training scale. In addition, the first FC layer was converted to a 7×7 convolutional layer, and the last two FC layers used as 1×1 convolutional layers for feature map reductions, see the NiN survey below regarding 1×1 convolutions. The network is applied densely over the entire image, so no cropping or region proposals were used. The test set is also sent as a *mirrored image pair* through the network, so the final soft-max classifier averages the image pair to obtain the final score. The authors evaluated using multiple crops from each test image as input, but did not find that the cost justified the results (Fig. 10.34).

VGGNet is a preferred choice for many research tasks, especially for creating initial features for transfer learning, and several working models are available in the Caffe open source neural network library.

ConvNet config. (Table I)	smallest image side		top-1 val. error (%)	top-5 val. error (%)
	train (S)	test (Q)		
A	256	256	29.6	10.4
A-LRN	256	256	29.7	10.5
B	256	256	28.7	9.9
C	256	256	28.1	9.4
	384	384	28.1	9.3
	[256;512]	384	27.3	8.8
D	256	256	27.0	8.8
	384	384	26.8	8.7
	[256;512]	384	25.6	8.1
E	256	256	27.3	9.0
	384	384	26.9	8.7
	[256;512]	384	25.5	8.0

Figure 10.34 This figure shows (top) the architecture parameters for VGGNet, and (bottom) the test results for a single scale set of test images, from [656] in CVPR, © Springer-Verlag used by permission

Name	VGG
ANN type	CNN
Memory Model	Simple, fixed
Input Sampling	Sliding window stride=1
Dropout, Reconfiguration	Dropout 50% on 1 st two fully-connected layers
Pre-Processing, Numeric Conditioning	RGB Mean-zero normalization
Feature Set Dimensions	3x3RGB, 1x4096RGB, 1x4096RGB, 1x1024RGB
Feature Initialization	Random mean-zero distribution
Layer Totals	16 filter, 3 classify, 5 maxpool, 1 softmax
Features, Filters	Convolutional
Activation, Transfer Function	ReLU
Post processing, Numeric Conditioning	Local Brightness EQ
Pooling, Subsampling	Max pooling 2x2x2

Half-CNN

The Half-CNN [693] was proposed by Yuan et al. as a whole image regression model or *locally correlated classifier*. The half-CNN is a viable alternative to an SVM or other regression model. The Half-CNN method entirely removes fully connected layers, which eliminates the need for classifier design. Instead, the output is a feature map showing the correlation between the input and the features learned in the trained network, which has applications in detection and segmentation. As shown in Fig. 10.35, note the use of the novel *upsampling layer* which follows the pooling layer to normalize feature sizes to support a linear combination of detected features, followed by a sigmoid activation function, into a uniform sized image. The goal of the system is local correlation of input to features, leveraging the local nature of $n \times n$ convolutions over sliding windows. Note that the MSRA-22 [670] system surveyed in this section also makes use of a strategy similar to upscaling to normalize feature sizes, with a different intent, called SPP [542] discussed later.

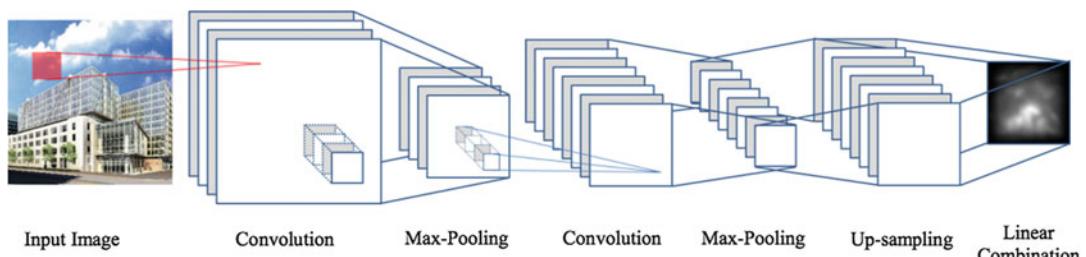


Figure 10.35 This figure shows the Half-CNN architecture. Note the novel up-sampling layer to support variable sized input images. Images from [693] in CVPR, © Springer-Verlag, used by permission

Since the Half-CNN is a *full-image* generic regression classifier, it does not depend on region proposals, and is designed to support unequally sized input images. The ground truth data consists of *prepared images* using a Gaussian weight mask generated inside of a marked region (detector region), and the Gaussian weight mask is centered in the marked region, for example the center of the facial landmarks as shown in Fig. 10.36. Each image is padded to 256×256 . This allows for the input image to be segmented in a smooth, Gaussian manner. The output in Fig. 10.36 is a linear combination of the convolutional features learned from the input images, which can be used as a mask to merge with the input image for segmentation output.

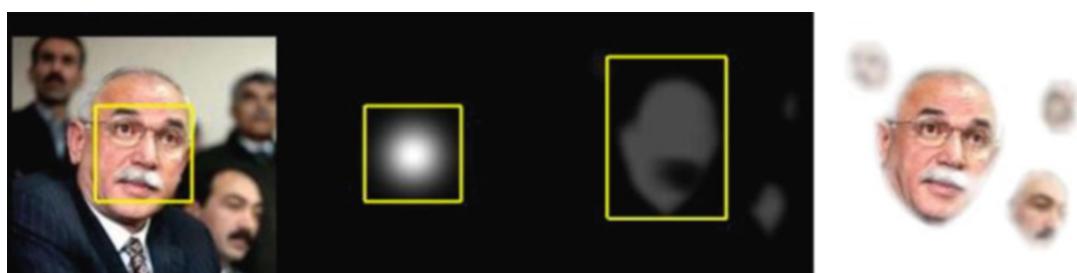


Figure 10.36 This figure shows the output of the Half-CNN, showing simultaneous detection and segmentation. The *left image* is the input image, *center left* is the ground truth feature map Gaussian mask fitted to the facial landmarks, *center right* is the output shown as a linear combination of the learned features, and the *right image* is a merge of the input image with the output segmentation region mask. Images from [693] in CVPR, © Springer-Verlag, used by permission

The elements of the architecture are composed of strictly kernel-connected convolutional kernel filtering layers, ReLu activation, and max pooling, along with a final filtering layer using a novel upsampling function to force all output images to the same size, eliminating the need for input images to be the same dimensions. No fully connected layers are used. The filter dimensions for each layer are $11 \times 11 \times 5$, $7 \times 7 \times 5$, and $5 \times 5 \times 5$. The last layer features are upsampled to allow for a linear combination of features into a final feature map, which can be applied as an overlay mask to the input image to visualize correspondence.

As shown in Fig. 10.35, instead of regularizing the feature sizes in a first fully connected layer, the upsampling layer takes care of regularizing the dimensions by rescaling all the features to the same size. However, due to the amount of downsampling in the maxpooling layers, there is a limit to the amount of upsampling that is possible, particularly for smaller images. Yuan reports that larger training sets (i.e., Imagenet sized) are required for good results.

Name	Half-CNN
ANN type	CNN
Memory Model	Simple, fixed
Input Sampling	Sliding window stride=1
Dropout, Reconfiguration	-
Pre-Processing, Numeric Conditioning	RGB Mean-zero normalization
Feature Set Dimensions	$11 \times 11 \text{RGB}$, $7 \times 7 \text{RGB}$, $5 \times 5 \text{RGB+upsampling}$
Feature Initialization	-
Layer Totals	3 filter, 3maxpooling, 1 upsample, 1 linear combine
Features, Filters	Convolutional
Activation, Transfer Function	ReLU
Post processing, Numeric Conditioning	Local Normalization
Pooling, Subsampling	Max pooling 2x2x2

NiN, Maxout

The Network in Network (NiN) model developed by Lin et al. [546] is perhaps the most novel and significant architecture in the DNN survey, and has introduced fundamental changes to the approach taken to create an artificial neural model. Besides the original paper [546], the NiN slides [592] and the poster from ILSVRC 2014 provide a good overview. The NiN authors were inspired to improve upon the Maxout network [610] developed by Goodfellow et al. by adding the MLP (multilayer perceptron) instead of simple convolution kernels as the activation function. Maxout networks use stacks of convolutional layers followed by Maxout layers, followed by a classification layer. Both NiN and Maxout use multilayer, or cross-channel methods, to create input from columns of pixels across all input feature maps, which we refer to as *Z-columns*. So we survey both NiN and Maxout methods together here, focusing on NiN and briefly discussing the relevant features from Maxout networks.

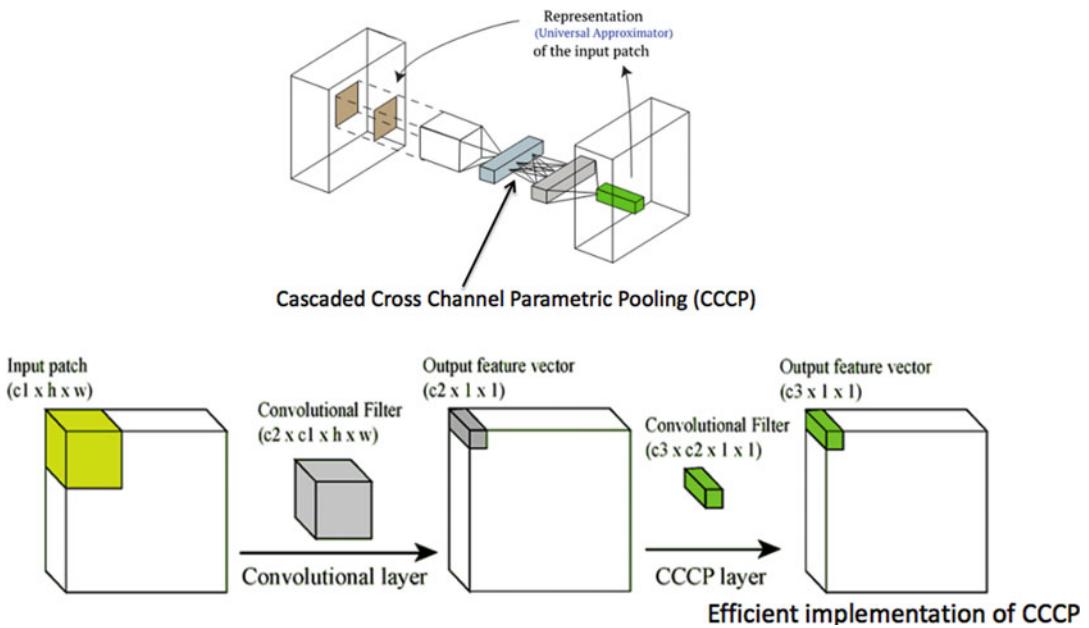


Figure 10.37 This figure shows (top) illustration a convolution pair $5 \times 5 \rightarrow 1 \times 1$ as used in the MLP layer, by pairing the 1×1 MLP convolution immediately after the 5×5 convolution layer, and (bottom) showing another view of the CCCP layer following the convolutional layer. Images from [592] in CVPR, © Springer-Verlag, used by permission

NiN innovations include:

1. **MLP feature model**, using a MLP (*Multilayer Perceptron*) supporting a nonlinear learning model instead of the general linear model of convolutional features. The MLP is implemented as a three-layer MLP micro-network in place after the convolution operation, which gives rise to the name *network-in-network* (NiN). See Fig. 10.37.
2. **Z-columns for 1×1 convolutions** is used as a method of reducing the number of feature maps, as shown in Fig. 10.38. Z-columns are taken from all input feature maps at once, which the authors refer to as *cascaded cross channel parametric pooling* (CCCP) to perform $1 \times 1 \times n$ convolutions ($n = \text{number of input feature maps}$) to reduce the volume of input feature maps. The $1 \times 1 \times n$ vectors are fed into the MLP. The Maxout [610] authors refer to the Z-column concept as cross-channel pooling (CCP), used for taking the max across n feature maps. Z-column convolutions and Z-max-pooling have huge implications, raising questions about the precise reasons why feature maps are needed, what functions should be used to create feature maps, and how feature maps contribute to accuracy, since the Z-columns and 1×1 convolutions recombine the feature maps generated from the carefully crafted and tuned feature weights.
3. **Convolution Pairs $n \times n \rightarrow 1 \times 1$** , the NiN MLP layers use a pairing of a normal 2D convolution immediately followed by a 1×1 convolution of a Z-column, to add richer representational power to the convolutional features. This is different from stacked convolutions, discussed earlier, since the goal is different.
4. **Use of Global Average Pooling (GAP)** to replace fully connected layers for classification, by taking the spatial average of features in the last layer for scoring. This reduces the training load and bypasses overfitting issues as well, see Fig. 10.39. GAP is in some ways analogous to the Maxout Z-pooling.

Maxout innovations include:

1. **Z-columns, Z-pooling**, a method of reducing the number of feature maps by taking the max value of several feature maps and combining into a single map, reducing the feature map count, which the Maxout authors refer to as cross-channel pooling (CCP), combining single-pixel columns from input feature maps, along with spatial x,y max pooling at current layer, which adds a more robust convex function approximation. Note that Z pooling reduces the number of feature maps.
2. **Dropout optimized**, Maxout is designed to be complementary to dropout, and uses the same dropout mask for cross-channel pooling, and incorporates dropout results into the maxout function.

Terminology Note: the terms cross-channel pooling (CCP) and cross channel parametric pooling (CCCP) are used in the NiN and Maxout literature to describe collecting z columns of pixels from a set of input feature maps, rather than collecting x,y kernels as tiles from one feature 2D map at a time. We use the term *Z-columns* here instead, to describe the multilayer input pattern, since the term *channel* may be confused with RGB or other channels.

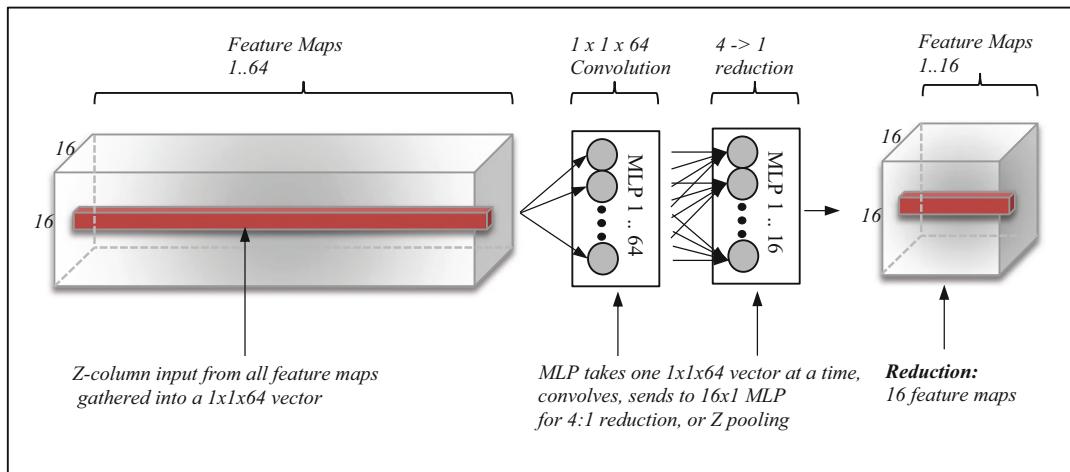


Figure 10.38 This figure shows an example method for *reducing* the number of feature maps (input = 64, output = 16), implementing *Z-column pixel input* for *Z-pooling* or 1×1 convolution, combining pixels from the complete set of input feature maps (i.e., the feature map volume) into a 1D input vectors, which the NiN method uses for input to the MLP, and the Maxout network uses in the max pooling operation, and InceptionNet users for dimensionality reductions. The total number of feature maps is reduced

As shown in Fig. 10.38, the Z-columns directly support the concept of 1×1 convolutions, which is also referred to as CCCP and CCP. As shown in Fig. 10.37, the MLP operation immediately follows a 2D convolution operation as a convolutional pipeline. The 1×1 convolutions are performed as part of the MLP feature detector in a sliding window across all the input feature maps along with the 2D convolutions. 1×1 convolutions have several interesting properties compared to 2D x,y input patterns. First, 1×1 convolutions can be used for *dimensionality reduction* in the z direction, to reduce the number of output feature maps. Z dimensionality reduction is invaluable for reducing system parameters. For example, if there are 256 input feature maps of dimension $640 \times 480 \times 256$ (x,y,z), the 256 feature maps can be reduced to 64 feature maps (z reduction) by performing a set of 64 $1 \times 1 \times 256$ convolutions on the input set, yielding a smaller or reduced set of feature maps as desired. In addition, 1×1 convolutions are novel and produce rich features, and have proven to yield

excellent results in Maxout networks, NiN, and GoogLeNet (Inception), complementary to x,y input patterns, and may be used together.

The implications of the success of using 1×1 convolutions across Z-columns for feature map space reduction are profound, touching the basic assumptions of CNN design. The same can be said for global average pooling, see Fig. 10.39. Consider that CNN practitioners often, without questioning why, use each convolutional feature at each layer as a filter to transform the input images into output feature maps, assuming that the filters are all needed and effective. CCCP reduces the feature maps into a smaller set which reduces the specific contribution of each filter, raising questions about the intrinsic value of each feature and feature map, and suggesting serendipity. The effectiveness of CNNs seems to result from the sheer number of features learned in the hierarchy (*averaged and generated features*), rather than the methodology of the CNN itself. CNN-style 2D filtering involves an empirically selected pipeline of numeric conditioning, convolution, a nonlinear activation function, and pooling to create each 2D feature map. In the end, the resulting feature map is declared effective if it works good enough or better than another approach. However, since the 1×1 convolutions reduce the entire feature map space into a new space, whatever supposed benefit is gained from the choice of filters and the processing pipeline is moot, since the 1×1 convolutions produce an entirely different view of the feature maps from the originals. Therefore, representing neurons as convolutional filters to produce feature maps is apparently a serendipitous design choice. The NiN and Maxout results also call into question the very notion of generating the convolutional filter weights via backpropagation in the first place, even though it seems to work. Perhaps preexisting features, such as basis functions or *visual genomes* ([534] see Appendix F) are more logical starting points. Future research into the area of feature map Z transformations is certainly a fruitful area.

Maxout uses a close variant of ReLu, but Maxout does not produce a zero value. NiN MLP uses ReLu as the activation function. Convolutions are linear functions, and often a nonlinear activation function is applied to add nonlinearity. Note that since the MLP is a nonlinear function itself, there is no strict need to apply a nonlinear activation function to the MLP result.

Maxout adds generality to the activation function, providing for the definition of arbitrary convex functions such as ReLu, ABS and quadratic functions, and is motivated by and intended to operate well with dropout. However, compared to the convex functions of Maxout, MLP as used in NiN is not limited to convex functions, which motivated the NiN authors to provide a better MLP-based feature to limit feature explosion and complexity in the higher layers of the network.

The NiN approach is guided by the observation that higher-level features are composed from lower level features, therefore better abstractions via the MLP model for lower level features will contribute to better and fewer higher-level features. The NiN approach addresses *the fundamental limitation of convolutional features as linear functions*, or a simple linear sum of *scalar values * weights*.

To maximize the effectiveness of convolutional features in a CNN, an overcomplete set of features is needed in each layer of the DNN feature hierarchy to separately capture nonlinear variations among very similar features, which adds more parameters to the system. The MLP is capable of approximating richer features with fewer parameters, compared to convolutional features. By using the MLP instead of convolution, the feature count is reduced from over-complete to sufficient. Since an MLP has weights, MLP is compatible with the general CNN architecture and supports backpropagation training.

NiN systems using the MLP model and *global average pooling* (GAP) shown in Fig. 10.39 greatly reduce the parameter count, mostly due to the reduced parameter count of GAP compared to a set of FC layers (see the discussion on fully connected layers in this chapter). Note that an NiN system with four MLP layers and a final global average pooling layer was demonstrated by the authors using only 7.5 million parameters and 29 MB of memory space, compared to the Khrishevsky architecture using 60 million parameters and 230 MB, with NiN achieving equivalent performance and half the training time [592].

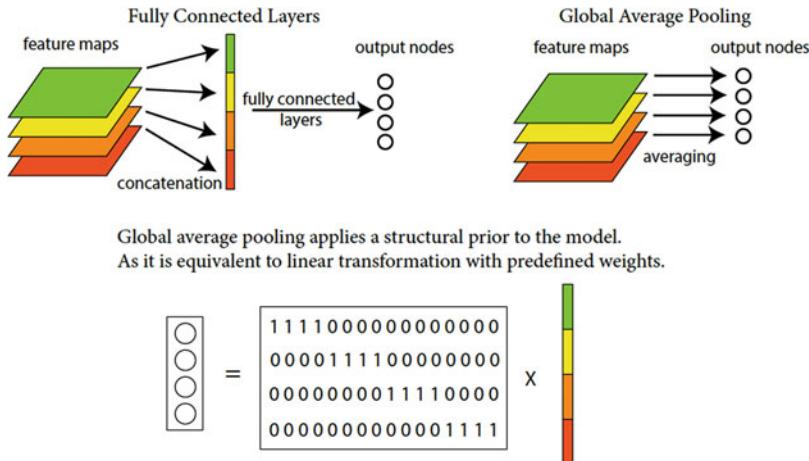


Figure 10.39 This figure illustrates the global average pooling classification concept compared to the FC layer classification concept. Images generated by Lin et al. [546] from CVPR poster talk, © Springer-Verlag and used by permission

In Maxout pooling, Z-column input (which the Maxout authors refer to as cross-channel pooling) and x,y pooling regions from the current layer are combined into a novel three dimensional x,y,z pool from which the max value is selected, prior to an activation function being applied. Typical max pooling alone only uses x,y region features from the current layer, pooled after applying an activation function to add nonlinearity. Maxout pooling apparently works well without rectification, and should be used separately for best results.

As shown in Fig. 10.39, Global Average Pooling (GAP) is a novel operation, which reduces the number of parameters and eliminates the FC layers often used for classification in CNNs. FC layers are typically the most parameter and connection intensive layers, and GAP provides a much *lower-cost* approach to achieve similar results. The main idea of GAP is to generate the average value from each last layer feature map as the confidence factor for scoring, feeding directly into the softmax layer. Global average pooling makes sense, since stronger features in the last layer are expected to have a higher average value, see Fig. 10.40. So GAP can simply be used as a proxy for the classification score. As the NiN authors state, the feature maps under GAP are interpreted as *confidence maps*, and force correspondence between the feature maps and the categories. GAP may be a particularly effective if the last layer features are at a sufficient abstraction for direct classification; however, GAP alone is not enough if multilevel features should be combined into groups like parts models, which is best performed by adding a simple FC layer or other classifier after the GAP.

Also with GAP, training becomes simpler with fewer parameters, and overfitting is not a problem as it would be in FC layers, since the parameters are so greatly reduced. Again, the NiN authors expect that the MLP generated features are higher quality to begin with, and can therefore be relied upon individually for supporting the global average pooling method. *This author expects other trainable classification methods to emerge along the lines of global average pooling, rather than FC layers.*

Global average pooling is an intuitive and sensible alternative to FC layers. FC layers are not optimal solutions, since FC layers increase the number of connections and weight parameters quite a bit, and are prone to overfitting, requiring regularization methods such as dropout to prevent overfitting.

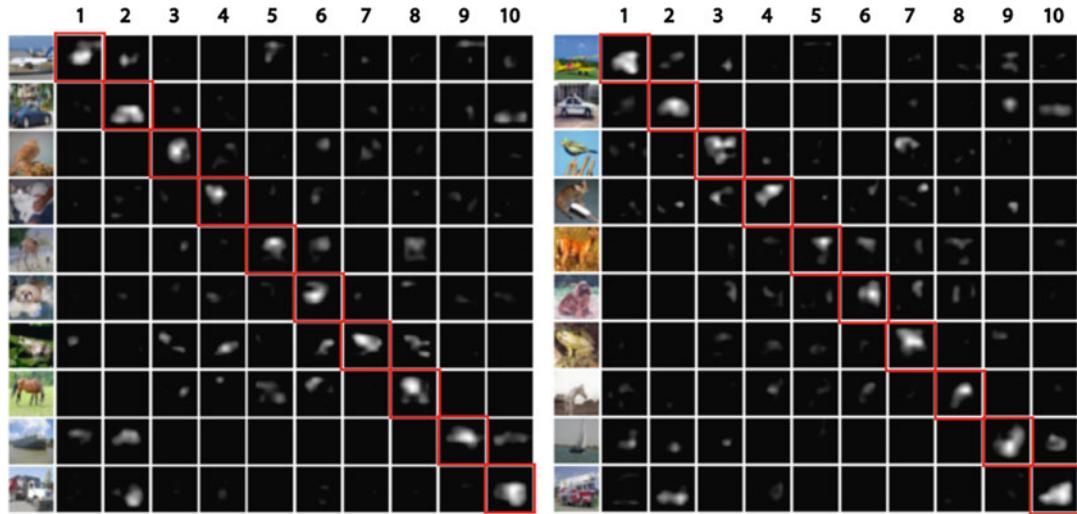


Figure 10.40 This figure shows the average strength of features in the diagonal as used in the global average pooling method. *Left and right images* are different classification sets. © Min Lin and used by permission, image taken from Lin et al. [546]

In summary, NiN is a profound architecture, and should significantly influence the path of CNN research going forward.

Next we will survey the GoogLeNet Inception architectures, which are influenced by NiN.

Name	NiN
ANN type	CNN
Memory Model	Simple, fixed
Input Sampling	Sliding window stride=1
Dropout, Reconfiguration	Using 70% dropout rate
Pre-Processing, Numeric Conditioning	RGB Mean Zero Norm.
Feature Set Dimensions	-
Feature Initialization	-
Layer Totals	22 convolutional, 5 pooling
Features, Filters	MLP
Activation, Transfer Function	ReLU
Post processing, Numeric Conditioning	-
Pooling, Subsampling	4 Max pooling, 1 Ave. pooling 5x5-stride=3

GoogLeNet, InceptionNet

The InceptionNet architecture introduced by Szegedy et al. [547, 608], otherwise known as GoogLeNet, demonstrates several novel architecture concepts for creating complex CNNs, leveraging a few concepts from the NiN architecture [546] and Maxout architecture [610] surveyed previously. Perhaps the major innovation in GoogLeNet is the *Inception module* as shown in Fig. 10.41, which defines an *aggregation* of different sized convolution filters at the same layer, providing (1) multi-scale convolutions, and (2) feature space dimension reduction using the 1×1 convolution model introduced by NiN. The InceptionNet architecture provides a $\sim 10\times$ parameter count reduction over the Krizhevsky architecture, and $\sim 25\%$ less compute. InceptionNet is currently one of the three top CNNs for various Imagenet benchmarks. Since the system is proprietary to Google, some details are not known.

InceptionNet is perhaps the most *baroque* CNN developed to date. An excellent overview of InceptionNet V2 is provided by Simonyan [701], and see also some overview slides by John Schlens [702]. Details on recent InceptionNet variants is provided by Ioffe et al. [703], and further developments incorporating Resnets and DRL concepts covered in Szegedy [893].

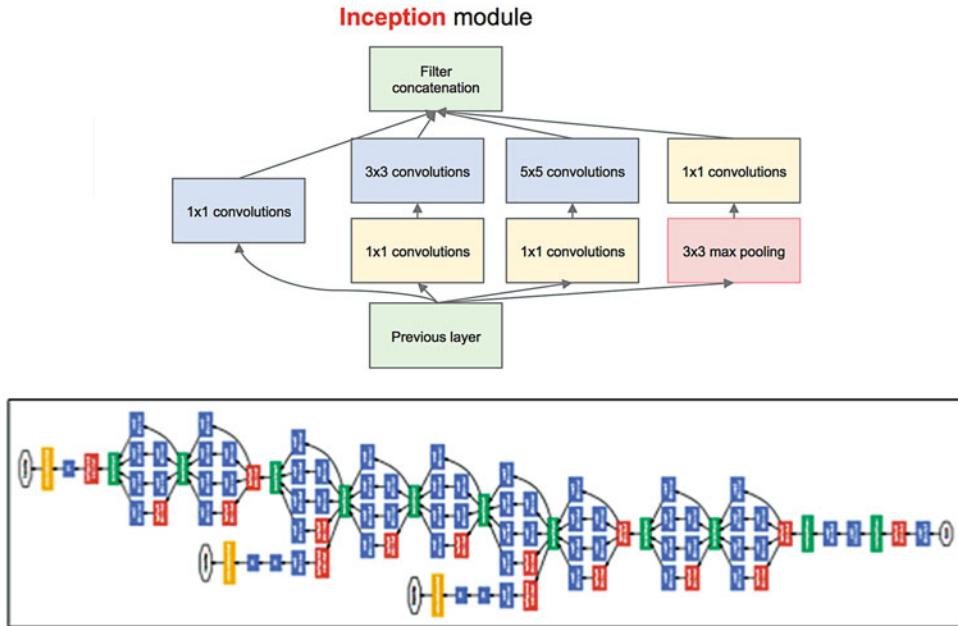


Figure 10.41 This figure shows (top) a single Inception module, combining 1×1 , 3×3 , and 5×5 convolutions into convolution pairs, and (bottom) the InceptionNet V1 architecture, other more recent variants exist. Note the inception modules, and the three branches for separate classifiers (yellow boxes). © Springer-Verlag, used by permission, taken from CVPR [547]

As shown in Fig. 10.41, the basic Inception module is similar to the NiN module with respect to the use of $n \times n \rightarrow 1 \times 1$ convolution pairs, i.e., 1×1 convolutions and $n \times n$ convolutions together. However, Inception uses several pairs of $1 \times 1 \rightarrow n \times n$ convolutions, using several parallel 2D spatial convolutions to increase the representation of *multi-size* features together, such as 3×3 , 5×5 , and also uses maxpooling. So the output of Inception modules is multi-scale due to the convolutional reductions, even if the input is not.

Here we highlight a few novel features of the InceptionNet architecture. Note that this summary covers information from several variants of the basic architecture, since exact details are difficult to obtain.

- **CNN In early layers:** A typical CNN architecture is used in the layers closer to the input, preferring medium sized features, such as 5×5 and 7×7 for more aggressive downsampling to reduce parameters and reduce the compute load.
- **Z-column 1D MLP convolutions:** Like the NiN architecture, 1×1 convolutions are used for both (1) adding representational power and (2) allowing for aggressive reduction of feature map count, which would otherwise explode exponentially in higher layers. See Fig. 10.38 in the NiN survey above.

- **Inception Modules:** Each module aggregates a series of convolutions of differing sizes, and may include a pooling layer and a branch off at intermediate levels for a softmax classifier, supporting multi-scale classification. One stated Inception module goal is to support parallel multi-scale feature extraction and classification at multiple layers.
- **Multi-sized feature aggregation and concatenation:** Multiple sizes of convolutions, such as 1×1 , 3×3 , and 5×5 are performed in each Inception module, and concatenated together as a single feature vector for output to the next layer, which reduces the parameter count. Inception features provide a type of multi-scale feature representation at each level, since the convolutional output is scale reduced at each layer. The feature vector contains 1×1 , 3×3 , 5×5 , and 3×3 maxpooling $\rightarrow 1$ features. Feature aggregation adds representational power inspired by the NiN method, surveyed earlier. The authors state that the major reason for the 1×1 convolutions is to reduce dimensions of the input feature maps prior to the 3×3 and 5×5 convolutions.
- **Max, Ave, striding and convolutional pooling:** Max pooling is used during feature downsampling between layers, and average pooling is used prior to softmax classification. Use of stride > 1 and convolutional downsampling is also used instead of strict pooling for some layers.
- **Branched Classifiers, Reduced FC layers:** InceptionNet allows for branching off from the lower levels of the network to classify features, so the network includes (1) lower level classifiers using FC layers and a softmax layer, and (2) a final softmax classifier with a single FC layer. InceptionNet’s classification is multi-scale. Splitting the classification into several parts also reduces the total FC network parameters in the system.
- **Preference for smaller convolutions:** Similar to the VGGNet concept using 3×3 stacked convolutions, smaller kernels are preferred to reduce parameters and compute overhead.
- **Aggressive spatial downsampling in lower network layers** to reduce parameters.

The InceptionNet training protocol involved independently training seven slightly different InceptionNet architectures to arrive at an *ensemble score*. The training for each of the seven architectures only differed in the training protocol selection of images and the order of presentation. The final softmax score is averaged over multiple images from each scale and crop. A final FC layer using average pooling is used to arrive at the softmax score, and the authors note that average pooling worked best among several alternatives evaluated.

Image augmentation during training included an aggressive set of operations, which the authors note keeps changing over time, and is hard to summarize here, such as four scales of images (256×256 , 288×288 , 320×320 , 352×352) with crops taken from each scale image, and mirrored versions of each crop. Other scaling of patch candidate images was from 8 % to 100 %. Aspect ratio changes were chosen randomly in the range $\frac{3}{4}$ to $4/3$. Also, some geometric distortions were applied for some images, and also different methods for computing scale changes and other geometric distortions were tried including bicubic interpolation and other standard computer graphics methods. However, due to the huge number of training image variations and changes to the hyperparameters during training, no conclusions were made regarding the best training protocols.

InceptionNet is quite complex, and defies simple mathematical analysis. In fact, the authors are very cautious about making claims for the architecture, since they are not sure if the success is attributable to the guiding principles, or just informed serendipity and hard work. But it works. See Table 10.3 for a *parameter comparison* of the top performing DNNs including InceptionNet, which is available in the Caffe open source library, and more pretrained network models may be released in the future.

Name	InceptionNet V_n
ANN type	CNN
Memory Model	Simple, fixed
Input Sampling	Sliding window stride=1 or 2 for pooling layers
Dropout, Reconfiguration	40% dropout
Pre-Processing, Numeric Conditioning	-
Feature Set Dimensions	7x7 at first layer otherwise 3x3 kernels, features/layer include 64, 192, 256, 320, 576, 1024
Feature Initialization	-
Layer Totals	33 (not comparable to non-inception architectures)
Features, Filters	2d convolutional, 1x1 MLP
Activation, Transfer Function	ReLU
Post processing, Numeric Conditioning	-
Pooling, Subsampling	Max, Ave, striding, convolutional reductions

*Note: parameters are guesses based on published info of different versions

MSRA-22, SPP-Net, R-CNN, MSSNN, Fast-R-CNN

The MSRA-22 architecture developed at Microsoft by He et al. [670] is based on the VGGNet architecture, with enhancements in the area of weight initialization optimized for the Parametric Rectified Linear Unit (PReLU) introduced by He et al. [670], and also using Spatial Pyramid Pooling (SPP) introduced by He et al. [542] to reduce the fixed-size input image limitation of the CNN to speed up training and improve scale invariance. We will summarize the major MSRA-22 innovations over VGGnet here. Since the system is proprietary to Microsoft, some details are not known.

The SPP approach reduces the need for fixed-sized region input, similar to other region proposal selection methods, so we provide a brief survey and discussion here of similar methods. The Fast-R-CNN system (Region-CNN) developed by Girshick [707] is a simplified and optimized version of the SPP approach. *Note: the SPP method was proposed to speed up the R-CNN approach, then Fast-R-CNN was proposed to speed up the SPP approach.* R-CNN is based on Girshick et al.'s earlier work [704] on segmented region proposals as input to the CNN to separately learn features and classify each region. A related method using a saliency score to group similar region proposals is found in [47] by Erhan et al. The Fast-R-CNN method applies a training protocol using mini-batches of images, and a sparse set of region proposals. Girchick provides interesting research on the major bottleneck of the method: how to determine the optimal number of region proposals. Apparently, comparing the results of dense vs. sparse region proposals demonstrates that somewhere over 1000 sparse region proposals are optimum. Sparse regions proposals seem to reduce false positives. Another method of creating region proposal candidates is via segmentation using super-pixels as proposed by Farabet et al. [850]. Super-pixels can be among the best segmentation techniques, see Chap. 2 for more on super-pixel segmentation and related methods. Uijlings et al. [667] present another method of generating region proposals using segmentation based on image partitions.

One problem addressed by MSRA-22 is scale invariance, which is often addressed by training multiple networks with different scales, and averaging the results. Another problem addressed by MSRA-22 is weight initialization and the activation function, which are correlated together and affect training and convergence towards best features.

SPP is also inspired by the HMP method of spatial pooling [109, 132] surveyed later in this chapter, and the Spatial Pyramid Matching (SPM) method [516], surveyed in Chap. 6, which divides the image into nested regions, and computes features for each region. The SPP authors leverage the

spatial pooling concept, and simply assemble all the extracted CNN features into a spatial pool from the last feature maps. The feature maps act like a scale pyramid of feature maps. Spatial pooling regularizes the input to a single size, since the number and proportion of the spatial pooling regions are the same, regardless of the image or feature map size. Note that the Half-CNN system [693], surveyed in this section, also makes use of a strategy similar to SPP, called upsampling, to normalize feature sizes.

Spatial Pyramid Pooling [542] is used to reduce input size restrictions on the images, since most CNNs are designed to support a fixed-size image. CNN training protocols often incorporate cropping, rescaling and warping regions into fixed-sized windows, which can distort and cut off image information. The authors note that SPP addresses the fixed-size limitations of CNNs that come from the FC classification layers which must be sized to the correct number of classification slots, while the convolutional feature extraction layers can accept any sized image. This observation inspired the SPP authors to move the scale normalization after the feature extraction, using a *spatial pooling pyramid*, just before the classification stage, to ensure class alignment, as shown in Fig. 10.42. However, the authors note that image scale is critical no matter what, and SPP can partially overcome scale issues. If pixel resolution is critical to capture local texture, then missing local texture information (i.e., pixels and scale) cannot be reconstructed.

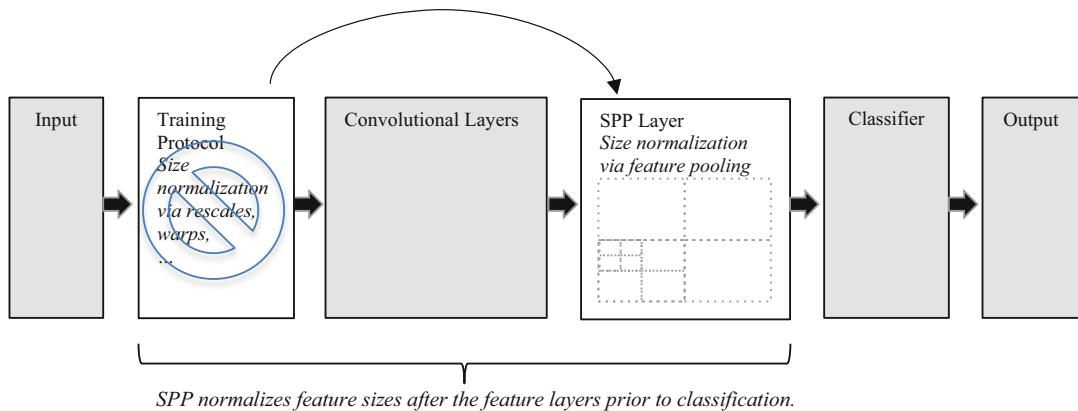


Figure 10.42 This figure illustrates how the SPP layer replaces the cropping and geometric transformations in the Training Protocol layer at the input by using the SPP layer prior to classification. The training protocol layer cropping and geometric transformations are not needed under SPP

First as shown in Fig. 10.42, the SPP layer shifts the scale invariance steps into the SPP pooling layer following the last convolutional feature layer, prior to classification. SPP eliminates the repeated convolutional feature extractions required using different scales of images, so only one pass over the training image is needed, rather than multiple crops and geometric transforms. The SPP layer pools and scale-normalizes the features prior to feeding into the classification layers, in this case an FC layer. SPP simplifies and accelerates the training protocol significantly. Performance is claimed to be significantly faster than other methods, between $24\times$ and $170\times$, depending on the comparison. Speedup can be partially attributed to the feature pooling used by SPP, since SPP pools the final features from a relatively small set of feature maps, rather than from large sets of cropped or geometrically transformed region proposals from the input images, so the SPP pooling space is far smaller.

Second, the feature weight initialization is optimized to work with the PReLU activation function introduced by He et al. [670]. The PReLU method is motivated, in part, to eliminate zero gradients,

which stall backpropagation training. PReLU is an activation function with a learnable parameter that can be tuned during training to define the zero threshold. PReLU is inspired by the method of Glorot and Bengio [705], which chooses a scaled uniform weight initialization to work well with a linear activation function. So the PReLU authors created a weight initialization method designed to work well with the PReLU activation function, which turns out to be based on a mean-zero Gaussian distribution with standard deviation $\sqrt{2/n_i}$, which works well with rectifier nonlinearities. The authors claim that the PReLU weight initialization method allows deeper networks to converge better, while the Glorot method does not. More discussion on activation functions is found in Chap. 9.

The MSRA-22 training protocol includes mean-zero pixel normalization, random cropping, scale jittering, horizontal mirroring, random color shifting, and mini-batches. Separate scores are developed for mirrored and unmirrored images. In addition to training on a single network, the MSRA-22 team tried a modified training protocol using two networks trained in parallel on separate sized images, sharing all weights between the networks. One network accepted 180×180 images, and the other accepted 224×224 images. This choice was motivated by the Imagenet competition, which provides 224×224 images. The model scores were averaged together.

input size	VGG-19 [25]	model A	model B	model C
224	$3 \times 3, 64$ $3 \times 3, 64$ $2 \times 2 \text{ maxpool}, /2$	$7 \times 7, 96, /2$	$7 \times 7, 96, /2$	$7 \times 7, 96, /2$
112	$3 \times 3, 128$ $3 \times 3, 128$ $2 \times 2 \text{ maxpool}, /2$	$2 \times 2 \text{ maxpool}, /2$	$2 \times 2 \text{ maxpool}, /2$	$2 \times 2 \text{ maxpool}, /2$
56	$3 \times 3, 256$ $3 \times 3, 256$ $3 \times 3, 256$ $3 \times 3, 256$ $2 \times 2 \text{ maxpool}, /2$	$3 \times 3, 256$ $3 \times 3, 256$ $3 \times 3, 256$ $3 \times 3, 256$ $2 \times 2 \text{ maxpool}, /2$	$3 \times 3, 256$ $3 \times 3, 256$ $3 \times 3, 256$ $3 \times 3, 256$ $2 \times 2 \text{ maxpool}, /2$	$3 \times 3, 384$ $3 \times 3, 384$ $3 \times 3, 384$ $3 \times 3, 384$ $2 \times 2 \text{ maxpool}, /2$
28	$3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $2 \times 2 \text{ maxpool}, /2$	$3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $2 \times 2 \text{ maxpool}, /2$	$3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $2 \times 2 \text{ maxpool}, /2$	$3 \times 3, 768$ $3 \times 3, 768$ $3 \times 3, 768$ $3 \times 3, 768$ $2 \times 2 \text{ maxpool}, /2$
14	$3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $2 \times 2 \text{ maxpool}, /2$	$3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $spp, \{7, 3, 2, 1\}$	$3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $3 \times 3, 512$ $spp, \{7, 3, 2, 1\}$	$3 \times 3, 896$ $3 \times 3, 896$ $3 \times 3, 896$ $3 \times 3, 896$ $spp, \{7, 3, 2, 1\}$
fc ₁			4096	
fc ₂			4096	
fc ₃			1000	
depth (conv+fc)	19	19	22	22
complexity (ops., $\times 10^{10}$)	1.96	1.90	2.32	5.30

Figure 10.43 This figure shows various MSRA-22 architecture variants, based on VGGnet, image © Springer-Verlag used by permission, taken from CVPR [670]

As shown in Fig. 10.43, MSRA-22 architectures use up to 22 convolutional layers, including a few examples using some layers with larger convolution kernels and larger strides for more aggressive downsampling. Note that InceptionNet, along with VGG-net variants MSRA-22, Baidu Deep Image, are the first DNNs to surpass human accuracy on some Imagenet benchmarks.

Name	MSRA-22
ANN type	CNN
Memory Model	Simple, fixed
Input Sampling	Sliding window stride=1
Dropout, Reconfiguration	-
Pre-Processing, Numeric Conditioning	RGB Mean-zero normalization
Feature Set Dimensions	7x7 at first layer otherwise 3x3
Feature Initialization	-
Layer Totals	19 filter, 3 max pool, 1 SPP
Features, Filters	Convolutional
Activation, Transfer Function	PReLU
Post processing, Numeric Conditioning	-
Pooling, Subsampling	Max pooling 3x3, 2x2, 1x1

Baidu, Deep Image, MINWA

Funded by search engine giant Baidu, the *Deep Image* system [700] introduced by Wu et al. employs a custom supercomputer hardware architecture called MINWA, and the most complex DNN training software architecture to date, achieving results that virtually equal the best systems in the world from Microsoft and Google. Deep Image is based on the VGGnet style model surveyed earlier in this chapter, and uses a 19-layer model with 3×3 convolutions at all levels, further underscoring the value of smaller convolutions and deeply stacked convolutions.

Deep Image and MINWA are in a class by themselves, and employ far more compute power, huge groups of labeled training samples, and more extensive training sample augmentations than any system known to the author. The compute resources employed are staggering.

The MINWA hardware is used to accelerate the *Deep Image* DNN, providing 6.9 TB of host memory, 1.7 TB local GPU device memory, and .6PFlops of total single-precision performance. The total compute power employed by Baidu in MINWA is in a class by itself compared to the other systems in this survey. MINWA is fundamentally a GPU cluster with high-speed Infiniband interconnects for direct GPU-to-GPU RDMA memory access, huge local memory for each GPU, and very large global system memory, similar to Cray supercomputer architectures from the 1990s, but at a fraction of the cost due to commodity GPUs.

In addition to the huge compute power, the software is extensively optimized for parallel operation to fully take advantage of MINWA, far beyond any other DNN optimization reported to date (*NSA has not reported in yet ...*), obviously intended for heavy commercial use. One technique mentioned regarding the optimizations is called *model-data parallelism*, where fully connected layers are split up to run in segments across several GPUs. FC-layers are the major bottleneck in most DNNs, so model-data parallelism is apparently a key to optimized performance. Convolutional layers express a kernel-connected workload, and can be accelerated reasonably well in GPUs *as-is* using the SIMD and SIMT capabilities of GPUs, and perhaps silicon accelerators in the GPUs for convolution, if available.

Deep Image implements the most extensive training data augmentation regime of any known DNN, made possible by MINWA. For training data augmentation, the authors reference the phrase “*the more you see, the more you know*” [700] to guide the preparation of training data—lots of data.

The goals for MINWA are far higher than any academic DNN published in research journals to date. The authors claim to have 10,000 times more training data than other systems. Various batch sizes are used for training different classes, tuned to work best across MINWA. The training samples are broken down 75 % for training, and 25 % for testing. Multiple resolutions of data are used, including higher resolutions than smaller systems can handle, up to 512×512 . Scale normalization combines multiple image scale results at the softmax layer. Note that use of higher resolution images is made practical via MINWA. Use of higher resolution images preserves detail, especially in image crops. Other augmentations include color castings to alter the RGB components independently in abnormal ways, vignetting, lens distortion, rotation, flipping, and cropping. In summary, no other DNN to date has been able to support so many augmentations and large datasets due to the huge compute requirements.

The training protocol uses several known labeled image databases for pretraining to generate a base set of features for various classes of images. To date, no other system has apparently been trained on so many labeled databases. Apparently, the pretrained features are used at all layers to start additional training sessions, and all layers are fine-tuned during ongoing training.

In summary, Deep Image and MINWA are in a class by themselves, leveraging massive data sets and a huge compute infrastructure to accelerate learning and analysis, *pointing towards a future with massive, dedicated deep learning systems, employed by various state and commercial enterprises.*

Name	Deep Image
ANN type	CNN
Memory Model	Simple, fixed
Input Sampling	Sliding window stride=1
Dropout, Reconfiguration	Dropout 50% on 1 st two fully-connected layers (like VGG)
Pre-Processing, Numeric Conditioning	-
Feature Set Dimensions	3x3RGB, 1x4096RGB, 1x4096RGB, 1x1024RGB
Feature Initialization	Massive pre-training
Layer Totals	16 filter, 3 classify, 5 maxpooling, 1 softmax
Features, Filters	Convolutional
Activation, Transfer Function	ReLU
Post processing, Numeric Conditioning	-
Pooling, Subsampling	Max pooling 2x2x2

SYMNETS—Deep Symmetry Networks

Gens and Domingos introduced Deep Symmetry Networks or *Symnets* [665] to address the fundamental invariance limitations of static weight templates normally used as features in CNNs. SYMNETS provide invariance to the feature space, which is missing from convnets, by projecting input patches into a six-dimensional affine feature space yielding *affine-invariant features*. A SYMNET is like a CNN, except that the feature generation layer operates in a six-dimensional affine space, using a set of affine transforms applied to each feature weight matrix prior to filtering. While the affine transform space is chosen to demonstrate the SYMNET concept, other symmetry spaces could be implemented. The features learned in SYMNETS are powerful, and more similar to deformable parts models [549] rather than typical convolutional style sparse features. SYMNETS incorporate several novel features not found in other CNNs. Note that a related concept for using invariant features is used in HMAX for lower-level features for composition into higher-level features, surveyed later in the BFN section.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \quad (\text{S1})$$

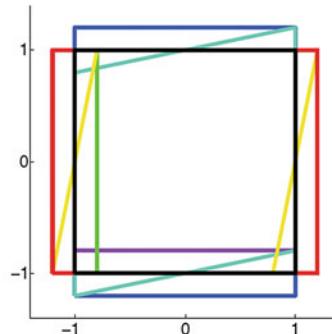


Figure 10.44 This figure shows (top) the affine transform S1, and (bottom) an exaggerated rendering of the six-element generating set of affine transforms (six transforms including identity, rotation, scaling, shear, reflection and translation) applied to a square, representing a square feature weight matrix. Image from [665], © Robert Gens, used by permission

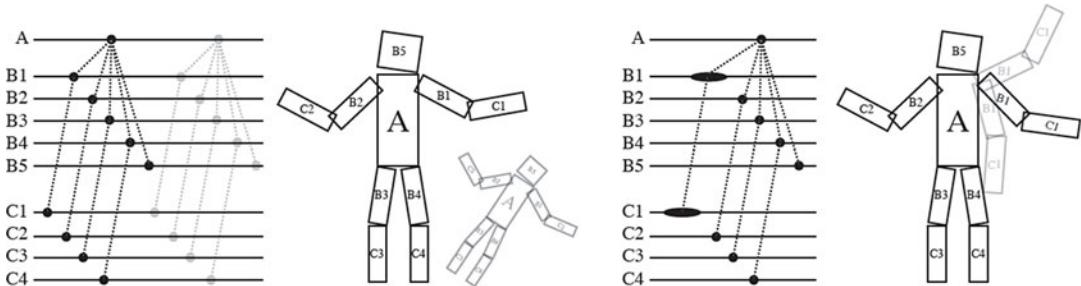


Figure 10.45 This figure shows how SYMNET tracks features under affine transforms. Each part A, B, C corresponds to a different part of the cartoon object in a feature hierarchy, and horizontal lines represent the feature position within affine feature space. *Left* shows unpooled affine feature positions in affine space for the single dark cartoon, *light gray* image represents the cartoon at another pose in affine space, and *right* shows how affine pooling kernels track the same feature under affine transformation, notice how B1 and C1 are affine transformed in *light gray*, and their position is located by affine pooling, represented by the wide ovals for B1 and C1. Image from [665], © Robert Gens, used by permission

Gens and Domingos take inspiration from *symmetry group theory* to develop an affine class of invariant features. A symmetry transform preserves the *class* or *identity* of the feature, and is invertible. SYMNETS support the affine symmetry group including rotation, scaling, shear, reflection and translation, as illustrated in Fig. 10.44. So there is one set of identity feature weight matrices learned in the CNN for each level, and the affine feature variations are derived from the identity feature for filtering with the input as shown in Fig. 10.45.

SYMNETS offer a feature representation that is consistent across pose variations and object part deformations in affine space. CNN training protocols often augment the training data via geometric and intensity transforms to add variation to the training set, which adds a degree of invariance to the feature set, at the cost of increasing the size of the feature set and training time. Instead, SYMNETS uses affine invariant features, which reduces training protocol augmentation requirements, and is demonstrated to speed up training convergence.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \quad (\text{S1})$$

SYMNETS mutilevel feature representation allows for composition of higher-level affine-invariant features from combinations of the low-level affine-invariant features. As shown in Fig. 10.45, the identity of the high-level features is preserved as the low-level features move in affine space, since the identity of the low-level features is also preserved as they move in affine space. This is a powerful concept not found in other CNNs. Invariance is accomplished using a set of 100 affine variations of each identity feature, allowing each feature to be tracked in affine space. Each affine variation is used as a filter across the input image space, and all filter results are pooled (we discuss more details on the pooling method later). Since the higher level features are composed of lower level identity features, the lower-level identity features may be detected under affine transforms, preserving the higher-level feature identity.

SYMNETS maintains an *identity set* of 20×20 pixel feature weights at each level in the hierarchy, used as in other CNNs for gradient descent tuning during backpropagation. However, for filtering, a set of novel *sparse affine features*, composed of 100 affine variations of each identity feature is computed at semi-random uniformly spaced *control points* within the 6D affine space. A control point in affine space is computed using Eq. (S1) in Fig. 10.44, by selecting appropriate affine transform coefficients $[a, b, c, d, e, f]$, and then rendering the identity feature using the affine transform into a 20×20 pixel feature weight matrix. To prepare for filtering, a forward-compositional (FC) warp extension to the Lucas-Kanade method is used to *align* the affine-transformed feature patch matrix at *local maxima* at fractional resolution near grid points at a chosen stride, such as 5, and the exact grid point position is adjusted during LK alignment with the feature. Using the local maxima surrounding grid locations is similar to using an interest point, and is a novel approach for CNNs.

Next, the *dot product* of each 20×20 feature is taken aligned at each adjusted grid point. By aligning the feature with the input window maxima, the dot product is maximized. Next, a sigmoid nonlinearity is applied, and the output for each of the 100 features is stored in a vector, analogous to a feature map per each of the 100 affine transforms, to allow for pooling of all the affine activations to find the strongest activation among all the 100 affine transforms to reveal the location of the identity feature in affine space.

The result of all the 100 affine filters is reduced to a summary feature map containing the strongest activation of all 100 affine filters, using novel *affine pooling kernels* to reveal the position of the strongest activation in the 6D affine space, or to locate a feature under a specific combination of affine transformations. The summary feature map can be considered like an *accumulator* of all 100 affine transforms, similar to the accumulator used in the Hough transform (see Chap. 3). If we visualize for a moment that each of the 100 filters produces a separate feature map in a volume, then the affine pooling kernels cross the Z dimension of the feature map volume (similar to Z-columns use in NiN and Inception for 1×1 convolutions, surveyed earlier), so all affine feature activations at the current position are part of the affine feature activation pool. SYMNETS allows for affine pooling kernels to be designed to pool over combinations of transforms in the affine feature space to represent expected affine transformations of real objects. For example, a pooling kernel can be devised to favor pooling over a small range of *scales* and a wide range of *rotations* to represent realistic movement for specific objects such as faces, or arm movements, as the features are transformed across the 6D affine space. See Fig. 10.46.

The combined results for all the filters acts as a *symmetry pool* in the local region, supporting either average pooling or max pooling, max pooling being preferred by the authors. As shown in Fig. 10.46, symmetric kernel group pools can detect the same features under affine transforms.

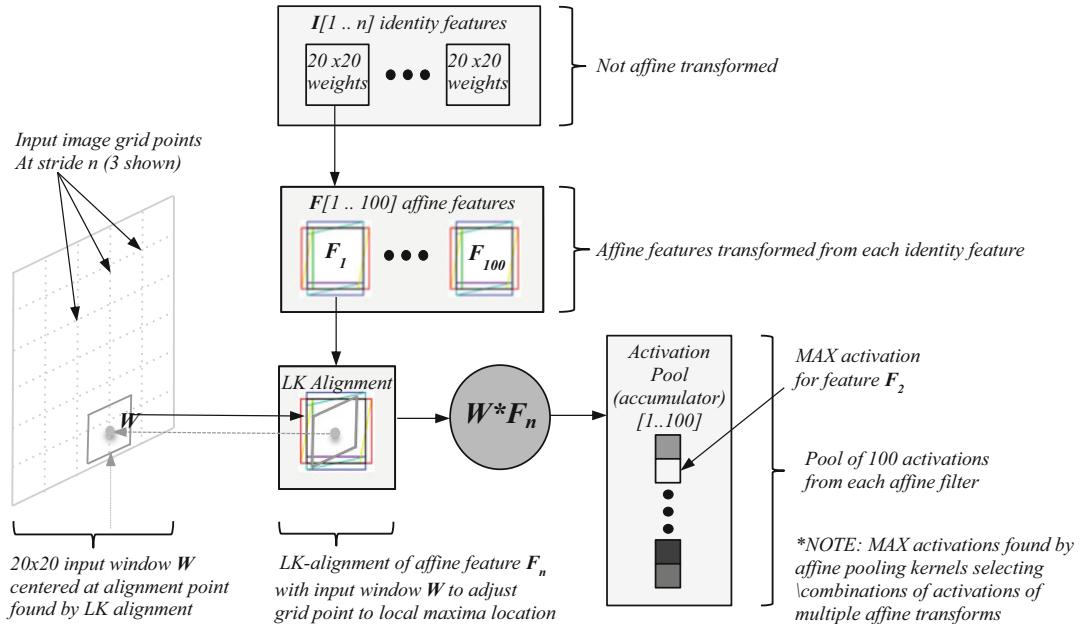


Figure 10.46 This figure illustrates the process of affine feature generation, affine filtering, and affine kernel pooling

Due to the affine nature of the features, the training protocol does not require extensive additions to the training set to include affine transformed test images. SYMNETS are trained as Convnets in mini-batches, using gradient descent and backpropagation. The authors demonstrate that a SYMNET can achieve faster training convergence with fewer training samples than a typical Convnet.

SYMNETS is compute intensive with respect to the affine transformations, and the control point alignments. However, *CPU SIMD instructions, GPGPU kernels, image processing libraries and GPU hardware accelerators could be employed, but are not mentioned*. So without optimization to compute affine features at control points, the affine transforms would be intractable for larger images.

Both one layer and two layer networks are evaluated, the two-layer network performs best.

The resulting features are fed into a fully connected layer with 500 connections, then a softmax layer. Larger features perform best in SYMNETS, such as 20×20 feature patches, compared to the trend in CNNs to use smaller features such as 3×3 or 5×5 , thus SYMNET feature convolutions are very compute intensive, and in fact a single 20×20 convolution is equivalent to a stack of 9 sequential 3×3 convolutions (see Stacked Convolution section in the Convnets discussion).

In summary, SYMNETS provides an elegant path forward towards adding invariant features into the basic Convnet architecture. Related work regarding adding invariant features to Convnets includes the work by Bruna et al. [780] using *cascaded wavelets* composed into feature descriptors similar to SIFT. A scattering-based wavelet transform adds invariance to the basic wavelets by using a nonlinearity to produce variant wavelets. The resulting nonlinear wavelet transforms are used in place of convolutional features in a CNN, which Bruna refers to as a *Scattering Convolutional Network*, which includes a novel affine space classification model.

Name	SYMNETS
ANN type	CNN
Memory Model	Simple, fixed
Input Sampling	Sliding window at feature-aligned grid points, stride=1
Dropout, Reconfiguration	-
Pre-Processing, Numeric Conditioning	-
Feature Set Dimensions	<100 features per layer, 20x20 size
Feature Initialization	100 affine transform permutations per feature
Layer Totals	1-2 feature layers, 1 FC layer, 1 softmax
Features, Filters	Convolutional, 100 affine permutations
Activation, Transfer Function	Sigmoid
Post processing, Numeric Conditioning	-
Pooling, Sub-Sampling	Max pooling over 100 affine activations per feature, no subsampling

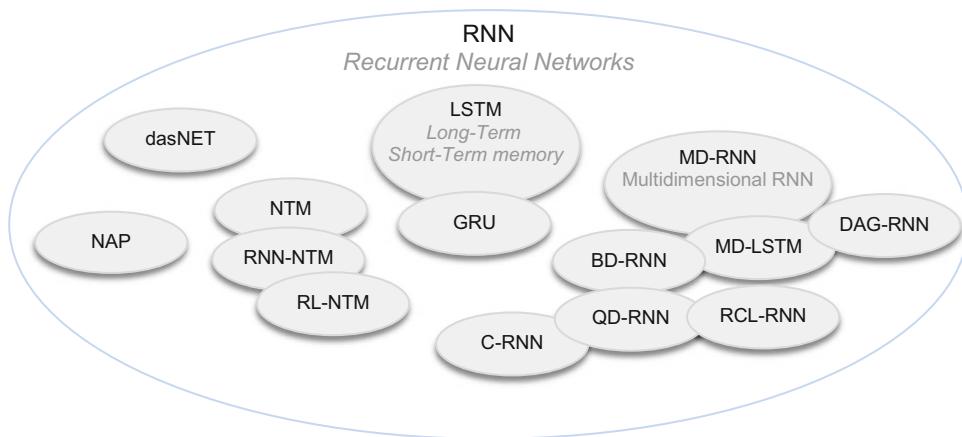


Figure 10.47 This figure illustrates the RNN variations covered in the survey

RNN Architecture Survey

RNNs are the most complex type of ANN to discuss, since there is no clear architecture pattern. Since this survey focuses on feature metrics and computer vision, we only cover the basic concepts of RNNs. The RNN architectures surveyed here are selected to span a range of novel applications to computer vision, which is an emerging area of research. We consider Spiking Neural Networks (SNNs) to be related to RNNs, since SNNs use a complex neuron model allowing ad-hoc feedback paths between neuron groups to influence neuron group firing. However, we do not cover SNN's and other complex neuron models in this brief survey. For more on complex neuron models, consult a standard text such as "Theoretical Neuroscience", Dayan and Abbot, MIT Press.

Perhaps the most popular application for RNNs is *sequence learning*, such as text processing, since RNNs provide primitive memory cells to store previous states and predicted states. However, today RNNs are being combined with CNNs for computer vision applications, for example where the CNN performs feature learning and classification, and the RNN is used for video frame sequence learning and video captioning, and the CNN classification of the images can be used to influence the RNN sequence learning via bias weights.

To dig deeper specifically into RNNs, the best resources include the work of Schmidhuber, Graves, Hochreiter, and Bengio, whose publications are cited as we go along. However, we focus here on selected research applicable to computer vision. To dig deeper into the field of RNNs, see Bengio et al. [554], Schmidhuber [552], and the upcoming book on RNNs by Schmidhuber. To get how-to information about designing and using RNNs, see the open-source resources in [Appendix C](#), and follow the source code. We deliberately steer clear of how-to materials and detailed math here, which is already presented very well in the references provided.

Early success with recurrent networks for 1D sequence learning was demonstrated in 1986 by Rumelhard and McClelland [739], and subsequently further developed in 1990 by Elman [740–742] and many others. LSTM-RNNs, a more versatile variety, are being applied in computer vision to spatiotemporal video sequences for applications such as activity recognition and video captioning, often combining both a CNN and an RNN in the same system. For example, video captioning work by Vinyals et al. [709] uses a combination of a deep CNN for analyzing the images, and an LSTM-RNN [584] for generating the text captioning. A video question-and-answer system was demonstrated by Ren et al. [710] utilizing a VGGnet style CNN for image feature learning, combined with an LSTM-RNN and softmax classifier to generate answers, using the DAQUAR labeled video question and answer dataset for ground truth. Venugopalan et al. [711] demonstrate matching image sequences to a series of words which are formed into sentences, using a combined CNN and LSTM-RNN network. Graves [712] demonstrates a novel method of applying an LSTM-RNN to the task of analyzing handwriting styles, and generating plausible handwriting in a selected style. Socher et al. [729] combine CNNs and RNNs together for RGB-D image classification, using a tree of RNNs for hierarchical feature pooling. Gregor et al. [728] developed a novel RNN for image generation.

At this time, only a small percentage of computer vision literature is devoted to RNNs. RNNs were originally designed to deal with sequence learning, which has been primarily researched as a *one-dimensional* problem, therefore computer vision applications using 2D data are rare. However, *Multidimensional RNNs* (MDRNNs) are emerging designed for 2D image data, which we survey later in this section. In the future, this author envisions RNNs applications proliferating in computer vision, for example applied to aggregating feature descriptors together for image classification, using RNNs to define pattern adjacency associations among features in local regions, where ordered sets or sequences of features in a region are fed into an RNN for feature adjacency signature encoding, prediction, and matching. The classifier may then be realized using RNN signature encodings for various classes and objects.

Notable methods which we do not survey include the *Recurrent Attention Model* (RAM) developed by Mnih et al. [754], which uses an RNN to select regions of the image to track and process at high resolution, incorporating a *Glimpse Sensor* and a *Glimpse Network*, which is trained using reinforcement learning rather than BPTT (*Backpropagation Through Time, used to train RNNs*). Another method using RNNs to process images selectively at high resolutions is proposed by Mnih et al. [708]. Volodymyr et al. [708] proposed an RNN which can selectively processing images only at high resolution in regions of interest.

Concepts for Recurrent Neural Networks

We will look into some basic concepts of RNNs here to set the stage for the survey, such as different RNN architecture concepts, how to unfold an RNN into an FNN to understand the forward pass and backward tuning pass, RNN weight sharing, and the types of memory implemented in RNNs.

A recurrent neural network is a class of dynamic, nonlinear systems for mapping *sequences to sequences using a concept of virtual time*. The RNN uses an *internal state space* composed from a trace of the inputs seen so far, see Boden [746]. RNNs also implement a form of *memory* via the recurrent inputs, which is useful for modeling sequences composed of current and past states or

events. Compared to other finite state models such as HMMs, the RNN is trainable, and much more efficient and compact for sequence representation and prediction, distributing the memory states across the network in uniform memory cells, rather than forcing each state of the model to store all possible state transitions. The RNN stores the state transitions in learned weights, like other artificial neural models. The RNN is also trainable via backpropagation. See Pascanu [648] regarding the difficulty of training recurrent neural networks. An RNN is like a finite state machine. Also, an RNN can emulate a finite state machine. See Tino et al. [736] and Arai et al. [737] for a discussion on the differences between finite state machines and RNNs. See also Pascanu [681] for some fundamental considerations on RNN design.

The basic ideas embodied in RNNs include the following capabilities:

- Memory Cells: recurrent inputs are a form of *memory, inputs persist*
- Serial Sequence Storage: sequences are learned and stored in RNN memory
- Time-based and state-based shifting of input through network
- Lateral Inhibitors/Excitators: recurrent inputs can be *lateral inhibitors and excitators*
- Backwards Feedback and Controls: feedback and controls to *same or other neurons*
- Weight replication over time: weights may be used and updated at each time step
- Correlation between input data separated by long and short time intervals
- Complex input pattern dependency representations to form sequences
- Reuse of neurons in hidden layers to accumulate sequence state

Depending on the goals of the RNN and the exact architecture, recursive inputs can be used along with recurrent inputs for specific purposes including:

- Control signals from other parts of the network
- Self-feedback for short-term memory to store state and sequence data
- Resilience to noise
- Excitatory signals (i.e., Hebbian learning) from associated cells
- Inhibitory signals

RNN Contrasted with CNN

A simple comparison between FNNs and RNNs is as follows:

FNNs can approximate arbitrary functions.

RNNs can approximate arbitrary programs⁶ and sequences.

Like the CNN, an RNN incorporates the basic convolutional artificial neural model, using weights and bias factors multiplied against the inputs. However, RNNs contain recurrent connections, combined with some feed-forward connections. The difference between a *recursive* network and a *recurrent* network is that the recurrent network is organized with chained connection structures to support sequences, while a recursive or arbitrarily connected network is not restricted by time or sequences. Recursive NNs have been applied to natural language by Socher et al. [752, 753]; however, we do not survey recursive NNs here. A recurrent network can be unfolded into virtual time into an FNN, but a recursive network likely cannot be unfolded into virtual time. See Figs. 10.48 and 10.50.

⁶In fact, Zaremba and Sutskever [738] design and train an RNN to evaluate short python programs, acting as a Python language interpreter.

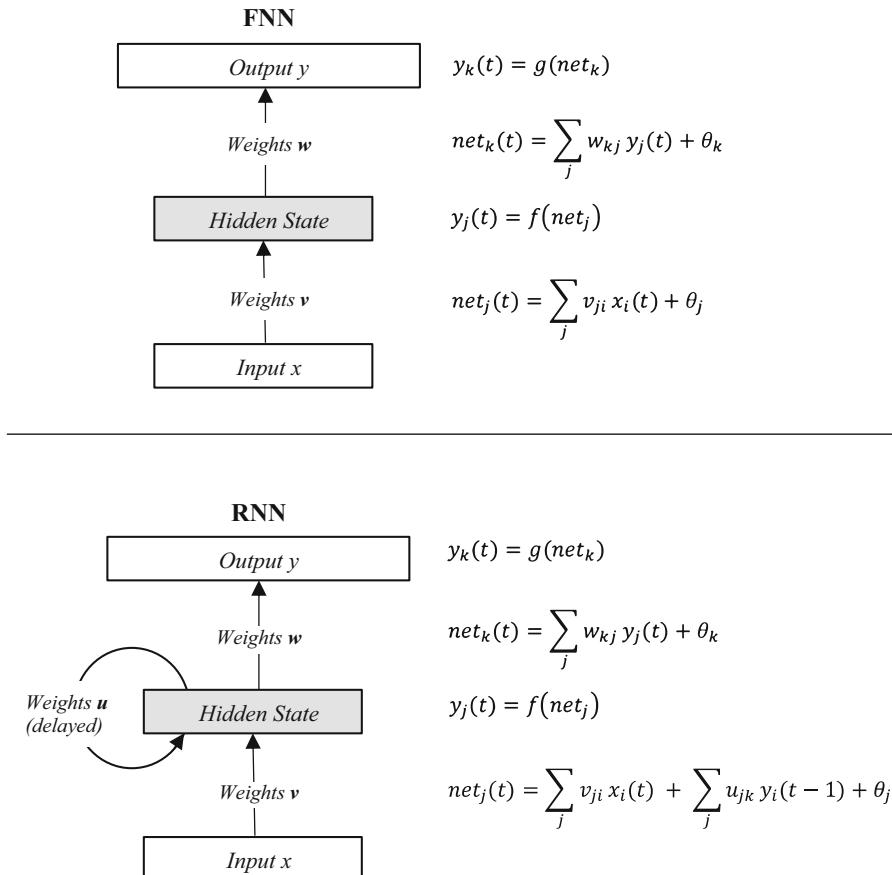


Figure 10.48 This figure illustrates basic differences between an FNN and an RNN, after Boden [746]

Unlike the FNN, the RNN's time-based or state-based sequence analysis and prediction structure allows the data to be repeatedly, sequentially and serially shifted into the network as if the RNN were a giant shift register or ring buffer. This fundamental notion of time and sequence underlies the RNN concept, providing both advantages and limitations. In contrast, the CNN takes 2D input window inputs from the current image frame, feeds the feature results forwards, then discards the entire frame and starts on a new frame.

*It should be noted that FNNs are also bidirectional and recurrent during backpropagation training. For example, gradient descent is feed-backwards, and tuning parameters such as momentum may use recurrent feedback for self-adjustment. Also, an RNN can be converted into an FNN by unfolding over time, as discussed later in this section.

Next we examine the method of transforming an RNN into an FNN via unfolding, which allows for visualizing the network as it acts on the input sequence, and is also useful for backpropagation training.

Unfolding an RNN into an FNN

An RNN can be remapped as a flow graph over a sequence of inputs, and then the flow graph can be unfolded into a feed-forward network (FNN), or chain of events. Unfolding allows the forward pass and the backward pass through the RNN to be visualized, and also enables backpropagation through time (BPTT [746]). See Figs. 10.49 and 10.50, which show equivalency between an FNN using a

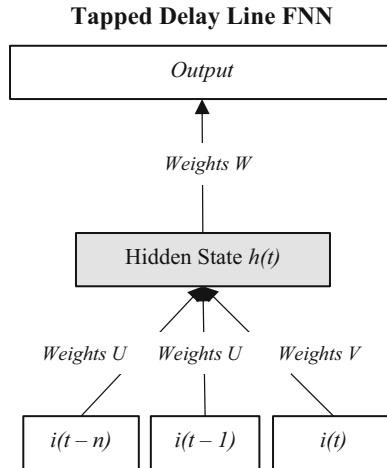


Figure 10.49 This figure illustrates a set of delay line inputs implemented with an FNN. Note the weight sharing

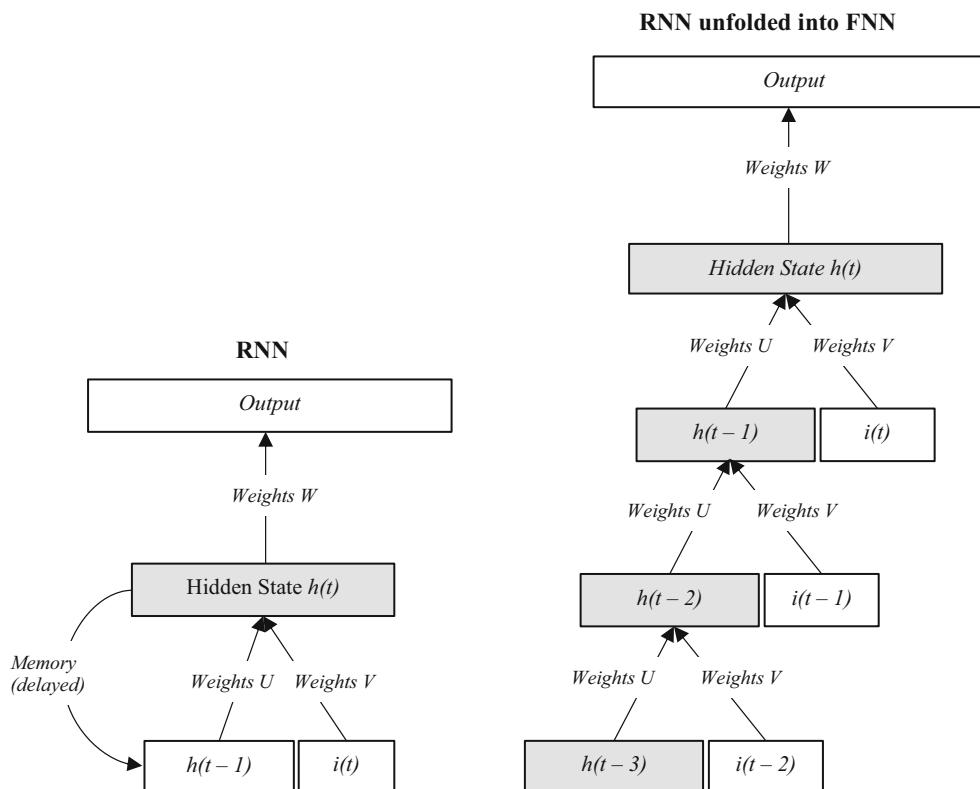


Figure 10.50 This figure illustrates (left) a simple RNN, and (right) the simple RNN unfolded into an FNN. Note the weight sharing. Unfolding an RNN into an FNN is useful for visualizing the input data sequence over time, and also useful for backpropagation training over time (BPTT)

tapped delay line, an RNN, and an unfolded RNN. A simple summary of unfolding in the context of backpropagation is described by Boden [746], see also Bengio [554].

The size of the sequence determines the size of the unfolded graph, or FNN. An RNN is designed with the sequence size in mind. Or, perhaps the RNN is designed with multiple sequence networks operating in parallel, each of a different length. The *state* or output of the RNN is composed of all prior inputs in the sequence:

$$o_t = h_t(i_t, i_{t-1}, i_{t-2}, \dots, i_2, i_1)$$

Note that the RNN state is not precise for every sequence that fits within the possible sequence size, but suffers some loss as the contributions are summed together, since depending on the training protocol, some contributions are weighted more precisely than others. The larger sequence sizes especially allow for generalization to sequences not found in the training data. The best precision for sequences will be found by designing the RNN with several parallel graphs containing different sized sequences, supporting sequence lengths expected in the training data and test data. However, with larger networks, of course more parameters and connections are introduced, increasing complexity.

As shown in Fig. 10.50 a simple RNN can be *unfolded* along its input sequence as time steps, where the forward pass is used to compute the current state of each cell by combining the new input with the previous state. Note that the weights in an RNN are shared across the cells at each layer, and the current state is stored in the RNN cell, which we discuss later. The unfolded network appears as a CNN, so during the backward pass, *Backpropagation Through Time* (BPTT) is used to tune the weights; the error is computed as a *partial derivative* at the output of the network, and for propagating the error backwards, the partial derivative is partitioned into *contributions* for distribution to each of the contributing RNN cells at the nearest layer, and the process repeats backwards through the network to the input. We refer the reader to Graves [725] regarding backpropagation methods used in RNNs, including BPTT as described by Williams et al. [743] and Werbos [745], and Real Time Recurrent Learning (RTRL) as described by Robinson et al. [744]. See Also Bengio et al. [554].

RNN Weight Sharing and Probabilistic Matching

Weight sharing is an artifact of RNN unfolding into an FNN, as previously described. As shown in Fig. 10.50, the same weights are shared at different time steps in the graph for the unfolded FNN. This reduces parameters, but also reduces precision to a point. Weight sharing is apparent in the unfolded graph, but in the recurrent graph the weights are implicitly shared at each time step, so the single RNN cell does not share weights, but rather *reuses* the same weights for each time step.

In RNNs, the idea of *weight sharing across time* allows for the same weights to apply to a range of sequences and subsequences such as “abcd” and “abcdefg,” and anomalously to different sequences of the same length such as “abcd” and “abcz.” The idea of sharing weights allows for *generalization to new sequences* similar to the learned sequences. But in this respect, weight sharing provides for a *statistical modeling capability* allowing for generalization and approximation, rather than an exhaustive, logical exact-match modeling capability requiring a larger memory system containing all known sequences to match against (which may be preferable if obtaining the complete training set is possible, and exact matches are required). Generalization and weight sharing for sequences and subsequences implies variable precision.

The final effect of weight sharing is that the sequence matching is not precise, but rather approximated, so an appropriate distance function must be used to predict the match probability. Therefore the final classification and matching is similar to polling each RNN cell in the sequence, and then combining the strength of the activations of each cell into the final match probability for a given sequence.

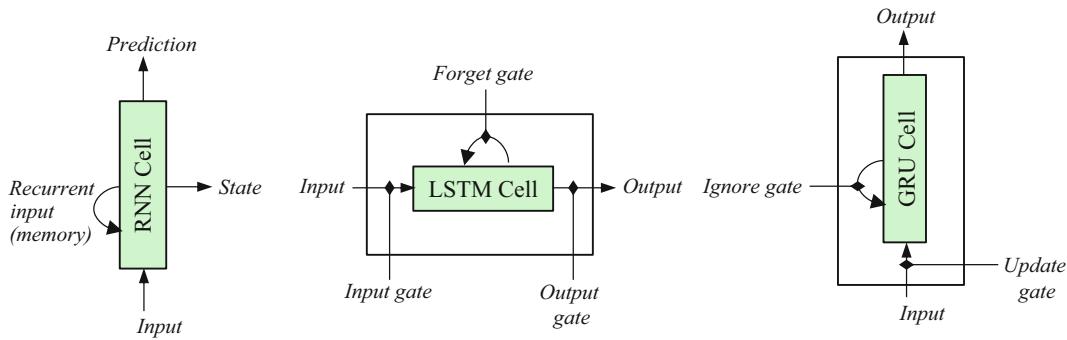


Figure 10.51 This figure illustrates a few possible RNN cell concepts. The RNN cell is a hidden unit, or artificial neuron. The recurrent feedback is a form of memory. (*Left*) Example RNN cell, (*center*) example LSTM cell with input trainable combinations of input, current state, and output, and (*right*) GRU cell combining weight combinations of input and current state. Cell types discussed later in this section

RNN Cell and Network Taxonomy

For the sake of this basic overview to illustrate the RNN concept, we taxonomize RNNs with the following examples of *cells* and *network topologies*, knowing that this list is far from complete (see Schmidhuber [548] to dig deeper and find detailed references).

RNN Cell Type, as illustrated in Fig. 10.51, which implement a form of memory, utilizing recurrent inputs, input from other RNN cells, and programmable gate functions for variable combinations of values. RNN cells have been developed and improved to provide more control over the memory, and in particular the *Long Term Short-Term Memory* model (LSTM), introduced by Schmidhuber [584] in 1991, is particularly attractive, which we survey later.

For the RNN cell taxonomy, we recognize:

- **RNN Simple Cells**, using a simple recurrent input, each cell is an artificial neuron with inputs, weights, bias, and output.
- **LSTM Cells, GRU Cells**, like the simple cells, with the addition of gating functions to allow the memory to persist and be more controlled.

RNN Network Topology, composed of RNN cells and other artificial neuron cells, using various connection topologies. RNN architecture topology may include recurrent portions, FNN portions, and arbitrarily connected portions. There is no common RNN architecture.

Various RNN network topologies include:

- **DAG-RNN**, Directed Acyclic Graph RNN, which may use recursive connections in a lattice, tree or other graph structure topology. DAGs provide a basic topology used in multidimensional RNNs, see [727].
- **ACRNN**—Arbitrarily Connected Recurrent Neural Network, which is the model neurobiology reveals; however, it is difficult to model.
- **DTRNN**—Discrete-Time RNNs, which are *synchronous*, and operate like a state machine driven by a clock.
- **CTRNN**—Continuous-Time RNNs which are *asynchronous*, operate similar to a state machine, and react dynamically.
- **BRNN, BLSTM**—Bidirectional RNN or LSTM, composed of two RNNs working in opposite directions. The BRNN provides context in both directions by incorporating inputs into the RNN

cell from both the forward (next) and backwards (prior) direction, combined into a single output, also known as a 1D DAG-RNN.

- **RRNN**—Reverse Direction RNN, where the input sequence is read in reverse, rather than forwards, see Sutskever et al. [749].
- **MDRNN, MDLSTM**—Multidimensional RNN, capable of supporting 2D imaging and computer vision problems, using an extension of the BRNN into two or more dimensions.

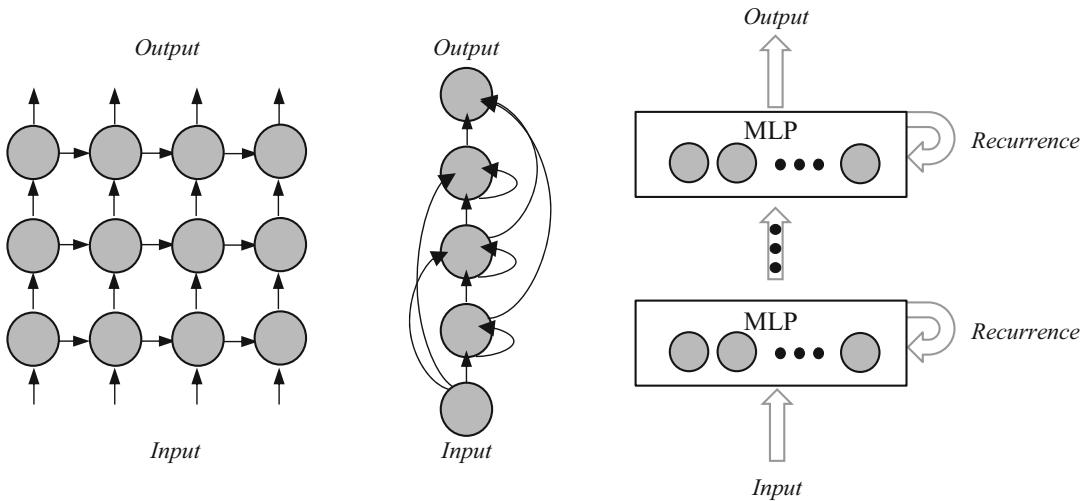


Figure 10.52 This figure illustrates a variety of RNN architecture connection topologies

RNN topologies shown in Fig. 10.52 include (right) an MLP modified to accept recurrent inputs at each layer, (center) skip connections used to distribute the input to all cells in the network, to reduce the path for computing the gradients so the gradients do not vanish, and (left) a deeper RNN for modeling longer time sequences, where for example the network is divided into an encoder section to store sequence tokens, and a predictor section that interprets the tokens.

RNN Sequencing and State

RNNs implement a neural model that stores sequences in *distributed memory*. The memory may persist over a variable length of time, and is distributed over several memory cells depending on the depth of the RNN architecture. RNNs may be designed to operate more like a finite state machine, rather than an FNN. The concept of time is central to RNN sequence learning, and may be synchronous or asynchronous. For example, data flows through the RNN one input per time step, and past input is stored in the network as desired. While FNNs are typically organized after the concept of layers, RNNs are not easily decomposed into layers, and more often resemble a network of *cells* or *cell groups*.

A lucid overview of neural methods for sequence processing is provided by Cho et al. [734], summarizing key research and concept development, and providing good intuition and working knowledge. Cho also introduces the gated recurrent unit (GRU), surveyed later along with the LSTM. As shown in Fig. 10.53, a sequence is first *encoded* from the input and stored in the RNN memory cells until the sequence is determined to be complete, and subsequently the sequence can be translated into another sequence or *decoded*, and then the sequence memory is released to process the next input sequence. For example, an English sentence with ten words may first be encoded into a vector representation, and subsequently decoded into a sequence of 14 French words, where each word, word pair, or phrase may be a recoded as needed to a correct length sequence.

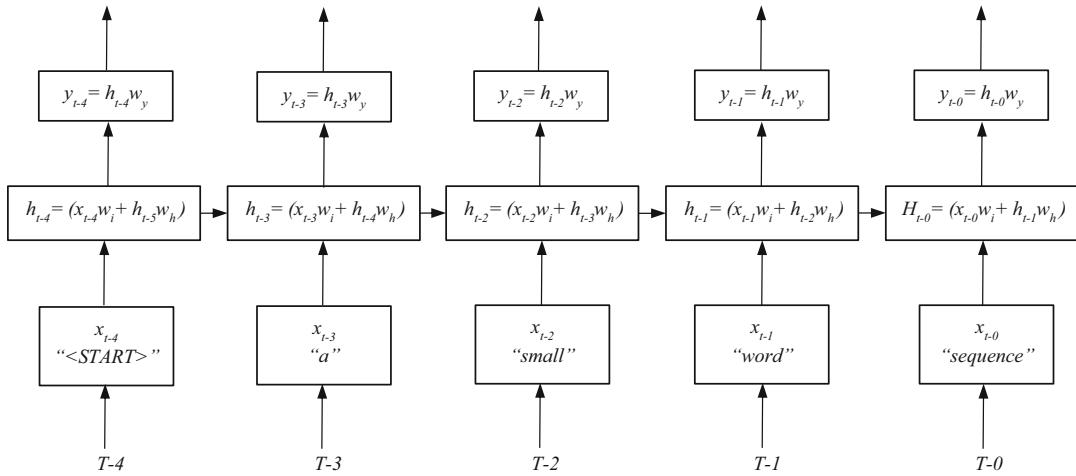


Figure 10.53 This figure illustrates the processing of a simple sequence in an RNN. Note that for this illustration, each input token “string” x is internally encoded as an integer acting as an index into a table of strings

The RNN cell state depends on the past state combined with the present inputs. RNNs are often used to *predict future states*, since a series of past states which have been learned and stored in the network are the basis for future sequential states added to the known past states. By combining RNN cells into deep networks, deep sequences can be learned. In fact, RNNs are often very deep compared to CNNs. We will survey a few examples later in this section.

Complex spatiotemporal patterns or rhythms can be learned and stored in RNNs. The RNN takes input, records state in memory, and produces outputs. RNN state memory can change based upon input and other gating factors, and produce state-related outputs. Other methods used to learn sequences such as finite state machines, Hidden Markov Models (HMMs), and Gaussian Mixture Models (GMMs), are not trainable using differentiation and backpropagation.

While an MLP or CNN is a general *function approximator*, and RNN is a general *sequence approximator*, and have been used to created program code sequences [738]. However, the sequence state is *probabilistic* due to the representation as a combination of weights and bias, since the RNN does not contain a full dictionary of every possible word and phrase combination; instead, the RNN goal is to *generalize* and learn via phrase encoding. In the RNN, the probabilistic notions are encoded in the individual weights for each cell and combined into a summary probability, rather than comparing complete phrase candidates to known phrases via probability. RNNs take advantage of temporal state information in the RNN cells for dynamic learning and prediction of spatiotemporal sequences. While FNNs learn classes in batches, the RNN can dynamically learn over time. RNNs are more versatile than state-less models, such as HMMs and SVMs which are applicable to classification. However, an RNN can be applied to both classification and sequences. An RNN can perform parallel and sequential computing. In general, recurrent networks are designed with far fewer parameters than an FNN, resulting in faster learning. In addition, RNNs can be designed with good generalization compared to strict CNN features.

RNN Memory Models

A central concept in the RNNs is *memory models*. Most RNN variations center around the memory model used, summarize below. The RNN memory models are designed to be intelligent, smart building blocks to implement spatiotemporal machine learning algorithms. So far, most of the work applying smart memory systems to computer vision is very primitive, and is done within the RNN architecture rather than the CNN architecture. However, we highlight the work of Weston

et al. [714] as one recent example of applying a more advanced memory model to machine learning for text recognition. We expect that smart memories alone could provide impressive tools for developing entirely new approaches to machine intelligence and computer vision, since biological intelligence denotes associative memory categorization and recall. Smart memory is a trend to watch for the future. See Appendix F for a discussion of Visual Genomes [534], which are a form of computer vision feature memory impressions.

For example, it is possible to represent a CNN by *an associative memory matrix* (AMM) as demonstrated by LaRue [715] under contract with DARPA. LaRue converts a CNN into a bidirectional memory matrix of relatively small dimensions to replace an entire CNN, compressing all hidden layers in a single association matrix, which is faster to train and faster to execute—a full order of magnitude faster is claimed. Smart memories for computer vision and machine learning are a fruitful area for future research, but outside the scope of this brief survey.

Here we briefly summarize a few types of smart memory, which have been incorporated into neural networks, with some of the methods implemented using RNNs. References are provided to dig deeper.

- **CEC LSTM, GRU**—Constant Error Carousel (CEC) and Long Short-Term Memory (LSTM)— Developed by Schmidhuber et al. [584] CEC and LSTM allow memory to persist, and errors to remain constant, rather than vanishing or exploding. See Fig. 10.54, LSTM is especially effective over long time lags. LSTM is a trainable artificial neural model using weights and bias factors, where the memory cell is surrounded with a few *gating functions*, to control memory persistence, updates, and reset. The *constant error carousel* is the mechanism of preserving the memory unmodified at the current state. We survey LSTM along with the variant GRU later in this section.
- **Semi-infinite Tape Model (NTM)**—The Neural Turing Machine RNN, developed by Graves et al. [583], uses a semi-infinite tape memory model containing LSTM cells; the memory persists as long as needed. NTM is novel, but difficult to optimize for random-access workloads. Instead of hard-coding the number of LSTM cells, the semi-infinite tape of LSTM memory cells can grow and shrink. NTM also allows forward-backward traversal of time (Virtual Time).
- **CAM, AMM, Hopfield Networks, SOM**—Content Addressable Memory was first implemented as an RNN by Hopfield [719] in 1982, also known as a *Hopfield Network*. In a CAM, the memory address corresponds by association to the contents of the cell, similar to a hashed-key of the cell contents. CAM memories have been applied to a variety of problems in computing for several decades, but are not popular in machine learning for computer vision. An Associative Memory Matrix (AMM) is similar, but can be derived from a CNN, see LaRue [715], which can also be used to implement a BAM (Bidirectional AMM) using SVD and PCA on the vectors to mitigate spurious basins of attraction. See also Kohonen [718] for a discussion on a similar concept, the *Self-Organizing Map* (SOM).
- **BAM**—Bidirectional Associative Memory in an RNN WAS first demonstrate by Kosko [717] in 1988, similar to CAM memory, except that each memory cell contents is a key to other related memory cells. The concept of associations is a vital notion within intelligence, so BAM memory systems can be used similar to distance functions (see Chap. 4) to identify feature candidates for classification and feature matching. LaRue [715] has developed a novel CNN using a BAM running concurrently with a CNN, which apparently is much faster to train than a CNN alone. Much earlier work has also been done in the area of BAM memories and variants, see for example Zhou and Quek [716] for a discussion on DBAM and DCBAM. See also LaRue [715].

LSTM, GRU

The LSTM is a type of RNN designed to implement a memory cell providing *short-term memory (STM)*, which can bridge *long (L)* time lags ($L + STM$). LSTM was introduced by Schmidhuber

[584] in 1991 as an improvement on the RNN model to overcome gradient descent problems, and to efficiently *compress* learned representations in a deep architecture using groups of interconnected RNN-LSTMs. The LSTM model provides for *constant error flow back in time*, preserving the RNN memory cell contents when needed, and preserving backpropagated gradient information over long time lags to avoid vanishing gradient problems. RNNs have been limited by backpropagation training difficulties, and the lack of any standard RNN architecture. However, the LSTM-RNN innovations have made RNNs much more trainable and have enabled deeper networks. See Also Ders [747] for a detailed analysis of LSTM strengths and weaknesses.

An LSTM is a *linear integrator*, like other artificial neural models, using weights and bias to integrate over inputs. The LSTM contains training mechanisms (gates) to determine *when* and *how much* the integrator listens to inputs to influence the current state. However, as shown in Fig. 10.54, the LSTM is a more complex artificial neural model of a memory cell than a simple RNN, incorporating a combination of *gates* as follows:

- Input gate (*logistic unit, weights = [1:0]*)
- Forget Gate, controls the *current state* (*linear unit, weights = [0...1]*)
- Output gate (*logistic unit, weights = [1:0]*)

The input gate and the forget gate together determine how much input is stored, and how much of the current state is forgotten. This arrangement allows for long-term dependencies between sequence elements to be represented and learned. For example, with the forget gate weight set to 1, the LSTM will retain its *current state* value over time, and when set to 0 the current state is forgotten. Intermediate weight values are possible also for *partial forgetting*. The input and output gates with the value of 1 or 0 produce simple derivatives during backpropagation, so since the derivative of a constant is zero, no error is propagated.

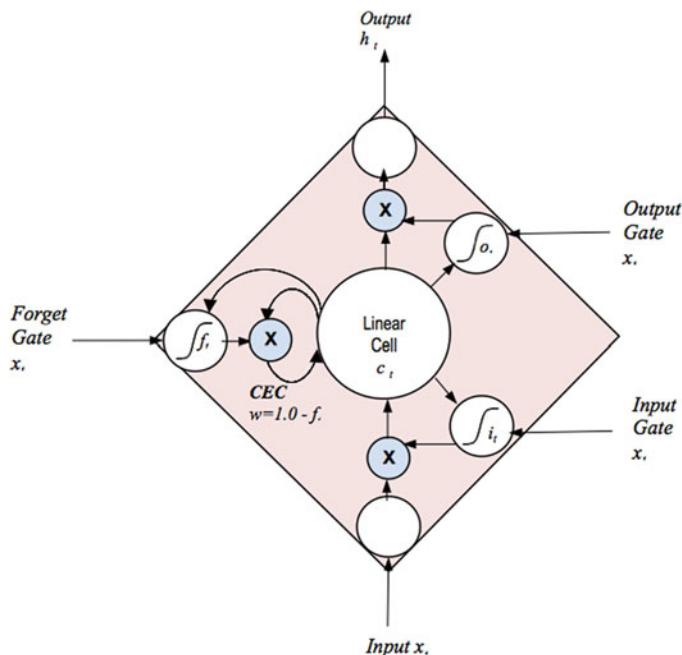


Figure 10.54 This figure illustrates one embodiment of an LSTM memory cell, others are possible. Note that the *Forget Gate* controls the error carousel (CEC), and the CEC value stays at 1 to preserve the memory until the Forget gate changes. The *Input Gate* and *Output Gate* control their corresponding gate units. See Schmidhuber [720] and Graves [712].

The LSTM cell state in Fig. 10.54 can be computed as follows:

$$\begin{aligned} i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \\ f &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \\ c_t &= f_t c_{t-1} + i_t \tanh(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\ o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_{t-1} + b_o) \\ h_t &= o_t \tanh(c_t) \end{aligned}$$

*where σ is the sigmoid function.

A recent variation of the LSTM is the Gated Recurrent Unit (GRU) developed by Cho et al. [734, 735, 750], which uses only two control gates: the *input gate* and the *dynamic gate*, providing a simpler model for backpropagation tuning. If we consider the current cell value and the new input value as two inputs to the GRU neural function, then the GRU dynamic gate allows for weight combinations of current value and input value. As shown in Fig. 10.54, when the input gate is closed, the contents of the memory are overwritten with new input. When the dynamic gate is closed, past information from previous cells is incorporated in the weighting computations at the desired strength proportional to the dynamic gate value, which can be used to reduce vanishing gradient problems. Also, Cho developed a *gated recursive convolutional neural network* (grConv) based on the GRU for text translation, see [735]. Note that the LSTM concept was originally implemented with fewer gates also, like the GRU.

A weakness of basic RNNs is that there is no way to control updates to each memory cell, which is simply a recurrent feedback or input with no controls. The LSTM allows data in each cell to be protected, and retain state as long as desired, using a combination of gates. The gates are analogous to memory read/write, and reset operations. However, the gates may use a nonlinear sigmoid function range 0–1. If the output gate = 0, the LSTM cell cannot be read. If the forget gate is zero, the data in the cell is zero, or reset.

One fundamental problem solved by the LSTM is elimination of vanishing gradients, which are the thorn in the side of gradient-based backpropagation methods. The solution: the *Constant Error Carousel* (CEC), which provides a *constant* backpropagation error flow to one or more neural units, since LSTM cells can be integrated together to share the CEC. The *forget gate* is used to control the CEC on one or more cells. (NOTE: there is no fixed LSTM design, since the CEC can be shared among LSTM cells). The constant value of 1 is used multiplicatively as the CEC weight to preserve the current state, rather than feeding an *infinitesimal gradient* in, which would scale the cell value towards zero. When the forget gate is changed to a value other than 1, the neuron cell value will change. The CEC also mitigates the problem of oscillating gradients. The work of Lyu et al. [748] on the gating functions shows that steeper gating functions, such as a steeper sigmoid that forces values to 1 faster, are better for accelerating learning and forcing convergence.

Within a connected network, the LSTM can learn when to forget memory, and when to update memory. LSTMs typically use the basic convolutional neural learning model to support backpropagation by gradient descent. However, other training approaches besides backpropagation via gradient descent are possible using LSTMs, see Schmidhuber et al. [713], and there is no clear best method for training RNNs since the topology varies widely. The LSTM concept can be implemented in several ways, for example with or without the forget gates. One method is illustrated in Fig. 10.55.

LSTM is a time-aware learning method. Sequences such as algorithm steps, or larger processes can be learned. LSTM can learn to reproduce sequences, organize and recall data associatively. LSTM can handle sequences, and disconnected sequence delimiters analogous to quotes or brackets. For

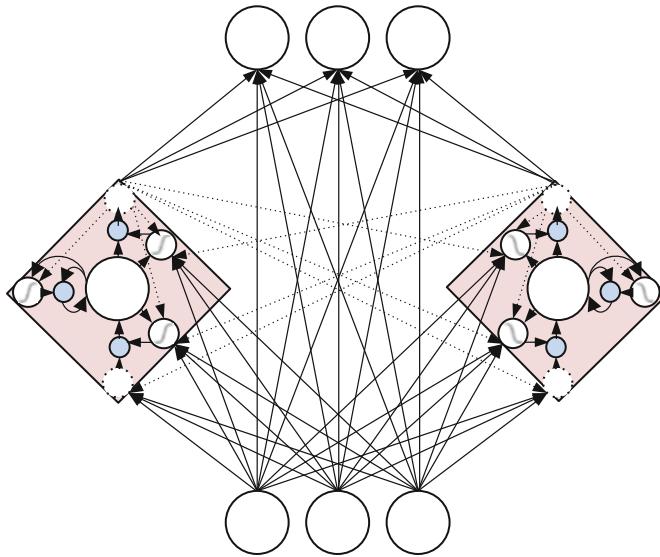


Figure 10.55 This figure illustrates LSTM cells interconnected with other cells, after Schmidhuber [720]

example, an LSTM cell could be set to recognize an open bracket ‘[’, and trained to reset itself on detection of close bracket ‘]’. LSTMs operate well when the training data, or input stream, contains sufficient redundancy and predictability. Purely random data would not do well in an LSTM architecture, since the purpose is sequence learning.

Another key concept of LSTMs is *compression*. For example, an LSTM cell may be set to monitor a sequence of characters looking for delimiter sequences, such as “;”. By doing this, the intermediate values are not stored, only the delimiters are stored. This is *sequence compression*. The sequence compression idea can make the LSTM network more resilient to errors, since sequences which are much larger than expected are required to break the paradigm. LSTMs are often organized into layers to represent time steps, or sequence steps. The lower layers would learn initial sequences, and the higher layers would learn the unknown sequences which lower layers did not learn, in more and more compact form. In addition, LSTMs can be organized into larger memory blocks containing entire sequences per block, with a common CEC to control the block.

Future directions for RNN-LSTM style approaches includes meta-learning *or learning to learn*, and memory networks, see Schmidhuber [722]. A good introduction to meta-learning using LSTMs is found in Hochreiter et al. [721], which an ideal application for LSTMs since an LSTM can change its own weights to improve its own algorithm. We discuss 2D LSTMs applied to computer vision in the multidimensional RNN (MDRNN) section below.

NTM, RNN-NTM, RL-NTM

The Neural Turing Machine (NTM, or RNN-NTM) is an RNN developed by Graves et al. [583] to implement an RNN as a trainable, probabilistic *memory controller* for a physical memory unit such as a DRAM. The NTM is designed to automatically learn algorithms, particularly basic memory access patterns (sequences) and the memory values (CAM) written to memory. The NTM can also generalize the learned algorithms, which is much more demanding than learning simple sequences such as a word phrases or sentences typically performed via RNNs. Currently the author knows of no computer vision applications for the NTM; however, computer vision applications are expected as research continues.

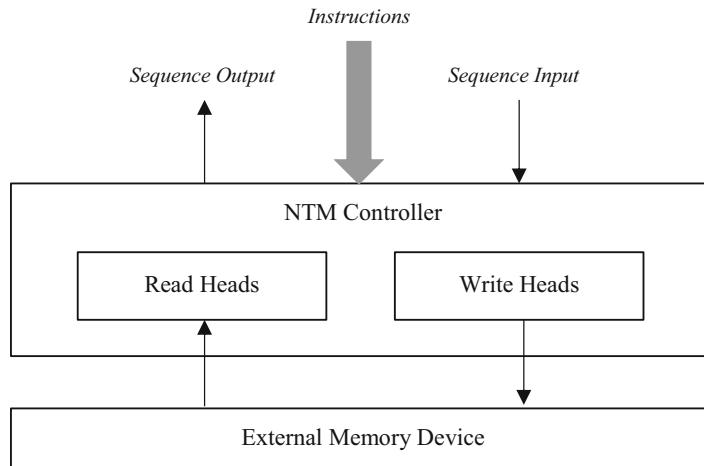


Figure 10.56 This figure illustrates the basic NTM concept. Note that the NTM is a controller for an external memory device, with an instruction set controlled by weights, trainable like other RNNs

The NTM memory model is demonstrated as an enhancement to the LSTM model [584]. However, as the NTM model's memory size and sequence size increase, compute complexity exponentially increases also, since RNNs access each memory cell at each time step. The physical memory address is actually represented by NTM as a *probability distribution* over the possible memory addresses, rather than as an absolute memory address. Therefore, as the memory space size increases, the probability computation increases exponentially, limiting NTMs to sequence sizes practical for computation on the chosen platform. To address limitations of the NTM, the RL-NTM (Reinforcement Learning NTM) was developed by Zaremba and Sutskever [757] to improve the NTM model, substituting reinforcement learning rather than an RNN to learn memory access patterns. We only briefly survey the NTM and RL-NTM here as extensions of the LSTM concept, illustrative of the capabilities of RNN style memory, and refer the interested reader to the original papers cited.

Graves et al. [583] provide the analogy that the NTM-RNN is like a Turing Machine acting on an external memory device, so following the Turing Machine model which uses a semi-infinite tape as memory, the NTM model incorporates the archaic concept of read and write heads (*a read/write head is used on magnetic tape drives*). Like the Turing Machine, the NTM incorporates a notion of instruction sequences such as sequence position, read, erase, and add, as well as RNN constraint concepts including time and clocking.

As shown in Fig. 10.57, the NTM implements a *probabilistic* read and write address method which provides an *attentional focus window* on variable sized groups or windows of *concepts*, which can be one of:

- A CAM: a value or range of values spread across memory locations
- A Sequence: an address (time step) or range of addresses

The attentional focus window can be narrow or wide, depending on a *blurriness* weighting factor and other weighting factors, which make the learning process differentiable and amenable to gradient descent learning like other RNNs, but therefore probabilistic rather than absolute. Blurriness allows memory concepts to be selectively ignored while others are in focus, as shown in Fig. 10.57. It is not entirely clear how the blur parameter works from the NTM paper [583], since the NTM system is proprietary.

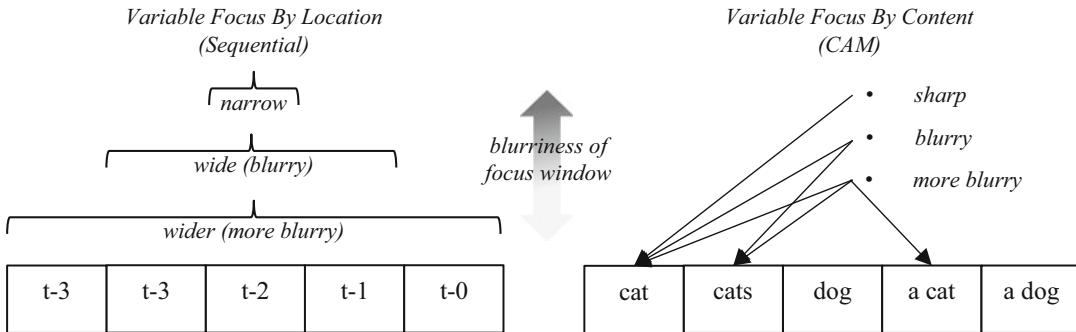


Figure 10.57 This figure illustrates the variable-focus memory window concept in the NTM model, where the level of focus can be increased to be more specific, or decreased to be more probabilistic. (*Left*) A variable-width of a time-based sequence window, and (*right*) a variable-specificity content-based window

The NTM CAM addressing mechanism is similar to content addressing in a Hopfield network [589], using a blurry, *approximate* address, or sparse address, which uses *part* of the concept’s data, which is then compared into the other concepts (chunks) to find the chunk containing the closest match. Another way to describe the NTM memory is similar to a classifier, which finds the closest memory concept given a sparse descriptor or key, and weights.

The NTM authors cite a range of neurobiological research as inspiration. For example, Baddeley [605] and others have shown that human learning and reasoning process typically keeps several concepts at attention simultaneously at the request of the *central executive*, which is directing the reasoning task at hand. The central executive concept assumes that inputs may come in at different times, thus several concepts need to be at attention at a given time for the best learning to take place. Perhaps up to seven concepts can be held at attention by the human brain at once, thus Bell Labs initially create phone numbers using seven digits. Selected concepts are kept at attention in a working memory or *short-term memory* (i.e., *attention memory*, or *concept-memory*), as opposed to a *long-term memory* from the past that is not relevant to the current task. As shown by Goldman-Rakic [606] for the human brain, the attention-memory or concepts may be accessed at different rates, for example checked constantly, or not at all, during delay periods while the central executive is pursuing the task at hand and accessing other parts of memory. The short-term memory will respond to various cues, or according to some logic or rules, and loosely resembles the familiar associative memory or content-addressable memory (CAM) used for caching in some CPUs.

Graves et al. demonstrate that the NTM can be trained and optimized for sequential access copy operations, chunked access, and sorting, further illustrating how the NTM can learn longer sequences than an LSTM model. However, since the LSTM can be arranged in a variety of configurations, it is not clear how the comparisons are made. In addition, the exact NTM architecture is not provided in the paper [583], neither is the LSTM architecture provided which is used for the comparison.

The applications of the NTM seem well matched to searching and sorting operations performed by search engines. If developed further, the NTM concepts could be made into a standardized type of silicon device with wide applications to machine intelligence (NTM-RAM or SMART-RAM), storing and organizing learned sequences and tuples such as n-grams, which could be a viable alternative to most classification methods, and also support other general purpose computing applications for data analysis.

Multidimensional RNNs, MDRNN

We introduce the Multidimensional RNN (MDRNN) and variants here as *a group of related architectures*, to point out their similarities and applications to 2D images. The MDRNN is a special case of the DAG-RNN model introduced by Baldi and Pollastri in 2003 [727], sometimes implemented as a sequence model for 1D via the BRNN (Bidirectional RNN), which uses a directed acyclic graph model and recursion within the RNNs to handle sequences. DAG-RNNs can also be implemented in multiple dimensions such as for 2D images, for example supporting a 2D network with a separate RNN for the forward and backward passes. The DAG-RNN concept is described by Baldi and Pollastri as analogous to two wheels, containing weights, moving in opposite directions across the data sequence, combining the current input and bidirectional wheel outputs to compute the prediction.

2D RNNs and 2D LSTMs for Computer Vision

A few applications of RNNs and LSTMs to computer vision are appearing in the literature, we briefly highlight a few here. Graves developed an MDRNN [725] for image segmentation, including the related MD-LSTM variation [732]. Of all the RNN architectures, the MDRNN is most applicable to 2D imaging and computer vision, since 2D and higher dimensions are directly supported. Donahue et al. [723] developed an LSTM-based image captioning and activity recognition system using LSTMs and CNNs together, see Fig. 10.58. Byeon et al. [781] developed a novel Quad-LSTM arrangement to capture sequences in the four compass directions (n, s, e, w) process an image frame using four parallel LSTMs, feeding the four sequences into a CNN to sum and squash the sequences using a nonlinearity, and feed the result into a softmax layer, the goal is to model local and global pixel dependency sequences for scene labeling. Yong et al. [782] apply RNNs to action recognition, using the RNN to learn motion sequences.

In many cases, 2D RNNs and CNNs are combined so each can accomplish different complementary tasks, usually relying on CNNs to learn features, and RNNs to learn sequences of features from frame to frame, or spatial relationship sequences between local features. For example, CNNs for visual recognition and LSTMs for sequence prediction are found in Venugopal et al. [711], Ren [724], and Donahue et al. [723] who applies LSTMs for activity recognition and video captioning.

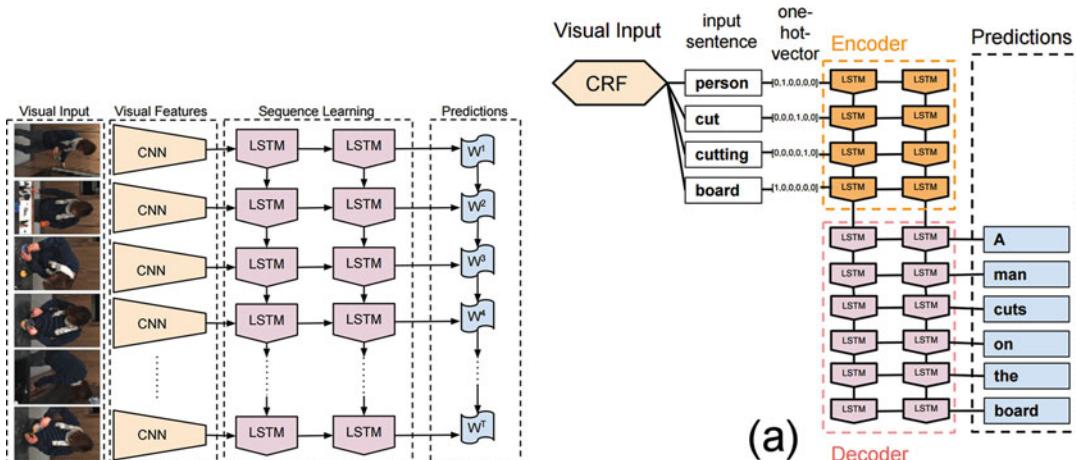


Figure 10.58 This figure illustrates (left) a hybrid CNN and LSTM architecture for sequence learning in activity recognition and video captioning applications, (right) the concept of encoding and decoding sequences using LSTMs, images from Donahue et al. [723] CVPR 2015, © Springer-Verlag, used by permission

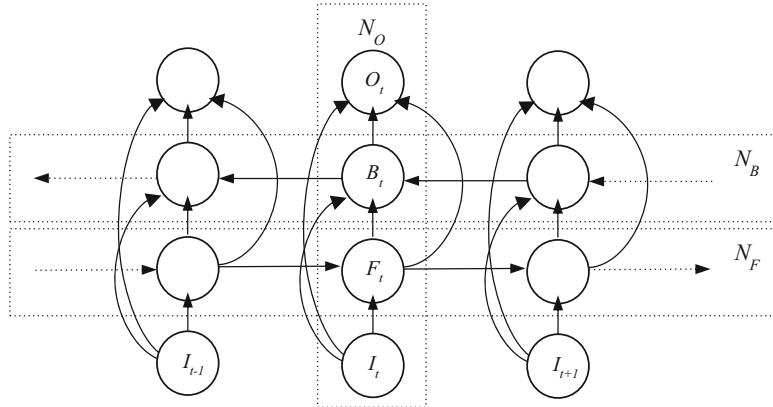


Figure 10.59 This figure illustrates a simplified BDRNN, or 1D case of the MDRNN. Note that the forward layer F and the backward layer B are composed of a separate path of hidden units

Later, we also survey the C-RNN [731] in more detail as an example of an MDRNN application to computer vision.

MDRNN, MDLSTM, DAG-RNN, BDRNN, RRNN

We will survey variations of the basic MD-RNN in this section. Variants include the multidimensional LSTM (MDLSTM), and the directed acyclic graph RNN (DAG-RNN). Also, we discuss the related 1D RNNs on which the 2D variants are based including the bidirectional RNN (BDRNN), and the reverse RNN (RRNN). To understand the concept of the MDRNN for 2D images, we first look at the 1D case, the BDRNN or bidirectional RNN (sometimes abbreviated BRNN) as shown in Fig. 10.59, which is composed of three RNN networks:

1. F_i —forward network to predict the upstream sequence
2. B_i —backward network to predict the downstream sequence
3. N_O —summary network to combine the output from (1) and (2).

*Note that F_i and B_i run in opposite directions on the data sequence.

For each position in the sequence, the forward and backward networks are combined by N_O into a final prediction, and errors are propagated in both directions along F_i and B_i . Imagine *two overlapping sequence predictions running in opposite directions*, distributed about the current position: the BDRNN output prediction at the current location is a combination of both directions.

According to Fig. 10.59, the 1D BRNN output O_i is composed from the N_O , N_B , and N_F RNNs and may be computed as follows:

$$O_i = N_O(I_i, F_i, B_i)$$

$$F_i = N_F(I_i, F_{i-1})$$

$$B_i = N_B(I_i, B_{i+1})$$

Note that the BRNN reads sequences in both directions: forward and reverse. However, Sutskever et al. [749] developed an RNN for *text-to-text* translation which *reads the input sequence in reverse*, which we refer to here as an RRNN (Reverse RNN). Sutskever found that by reversing the input strings during training, the RNN-LSTMs used were able to deal with much larger sentences, and

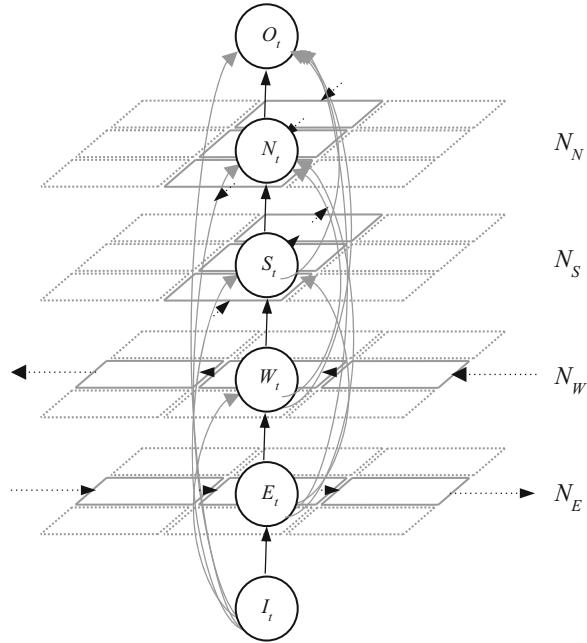


Figure 10.60 This figure illustrates the 2D MDRNN concept illustrating a 3×3 pixel 2D image as input to four RNN sequences (N , S , E , W) combined into a fifth output RNN. Note: compare this illustration with the 1D BRNN example in Fig. 10.59

performed measurably better. Sutskever believes that training similar RNN networks with reversed input sequences will enable them to perform better as well.

To extend the BRNN into the 2D case, we introduce a 2D network using four overlapping networks following the compass directions NSEW as N_N , N_S , N_E , and N_W within the image raster, centered about the pixel at the current time step, illustrated in Fig. 10.60.

According to Fig. 10.60, the 2D MDRNN output O_i can be composed from the N_O , N_N , N_S , N_E , and N_W RNNs as follows:

$$\begin{aligned} O_{i,j} &= N_O(I_{i,j}, E_{i,j}, W_{i,j}, S_{i,j}, N_{i,j}) \\ E_{i,j} &= N_E(I_{i,j}, E_{i-1,j}, E_{i+1,j}) \\ W_{i,j} &= N_W(I_{i,j}, W_{i+1,j}, W_{i-1,j}) \\ S_{i,j} &= N_S(I_{i,j}, S_{i,j-1}, S_{i,j+1}) \\ N_{i,j} &= N_N(I_{i,j}, N_{i,j+1}, N_{i,j-1}) \end{aligned}$$

As described by Graves et al. [725, 726, 732], the basic idea for the MDRNN or MDLSTM is to extend the single recurrent input with as many recurrent inputs as there are dimensions. As illustrated in Fig. 10.60 for the 2D case, the MDRNN is modeled as *an ordered sequence of pixels* on a series of scan lines in Cartesian coordinate space. Note that the pixels from the image are scanned into the RNN as a *sequence*, one line at a time, from left to right, from top to bottom. This sequence defines the RNN adjacency connection patterns possible. The application of the MDRNN in this case [725, 726] is image segmentation into regions of similar texture patterns, the textures are known

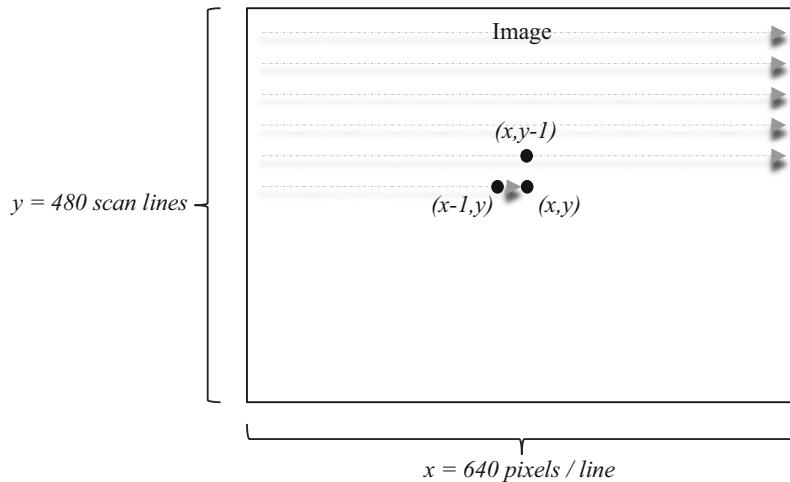


Figure 10.61 This figure illustrates frame-based scanline input to an MDRNN

and labeled in the training data, so adjacent pixels from the current line and prior line are useful to detect and define the segmentation. However, Graves et al. [733] also developed a method for RNN classification of unlabeled data via the Connectionist Temporal Classification method (CTC).

As shown in Fig. 10.61, the image is scanned a frame and line at a time into the RNN, therefore the RNN sequence state at any one time includes *prior pixels only*, including the adjacent pixel from the prior scan line, and the previous pixel on the current line, so only two directions of connectedness are known at once. So for this example, the RNN architecture should at least be large enough to hold two lines of pixels, the current and previous lines, and perhaps the input buffer can be arranged as a *circular pixel buffer*, or a *line buffer pool* for previous, current and next lines, with pointers to each which change as the previous line becomes the current line, and so on.

In summary, the MDRNN can be applied to 2D images or higher dimensional arrays, and has been used effectively in image segmentation. However, the type of feature generated by the MDRNN is a very primitive set of 1D weight templates, forming a 1D intersection kernel around the center pixel. It is not clear if the MDRNN approach has enough foundation to be extended into more powerful hierarchical feature sets, which have been demonstrated in the CNN approach to be very effective for classification.

Next, we survey selected examples of MDRNNs.

C-RNN, QDRNN

The Convolutional-RNN (C-RNN), or Quad-Directional RNN (QDRNN), introduced by Zuo et al. [731] learns spatial dependencies between convolutional features in local regions, which has been a glaring weakness of the simple CNN feature model. (To be fair, we note that *the CNN model was not designed to handle the problem of spatial dependencies*). The C-RNN is one of the few examples of an RNN being directly applied to computer vision feature representation. First, the C-RNN uses a CNN to capture features across the image. Second, the 2D CNN features are fed into four RNNs operating in parallel to capture *feature sequence signatures* in *quad-directional* compass directions, forming a cross-shaped feature dependency pattern. The quad-directional sequences are collected in a global hidden layer, as shown in Fig. 10.62. Note that the C-RNN is different from the MDRNN and MDSLSTM developed by Graves [725, 732] for image segmentation, since the C-RNN operates on CNN features rather than pixels. The relationship between the features themselves

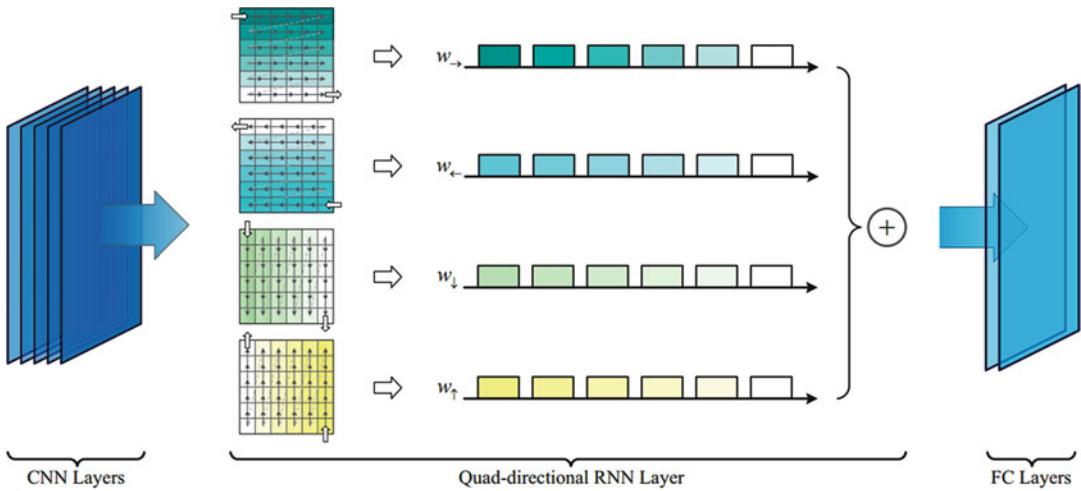


Figure 10.62 This figure illustrates the C-RNN method, where (left) CNN style correlation template features are collected, (center) the image is scanned in four directions using an RNN to learn the directional connected sequences of features, and (right) fully connected layers are used to learn the classification of the features and associated sequences. Image from CVPR Zuo et al. [731], © Springer-Verlag, used by permission

is sequenced. However, both the MDRNN and QDRNN use the same bidirectional scanning pattern for each axis (x, y). Note that an image labeling application of the C-RNN is discussed by Shuai et al. [730].

The C-RNN uses five feature layers composed of convolution and pooling, a recurrent layer for computing the four compass direction RNN sequences, and two fully connected layers for classification, and a softmax layer. The five C-RNN CNN layers include $11 \times 11-96$, $5 \times 5-256$, $3 \times 3-384$, $3 \times 3-384$, and $3 \times 3-256$. Input window stride is four for the first layer, and one otherwise. 3×3 max pooling is used, with a stride of 2, at the first, second, and fifth layers. The summary filter sizes fed into the RNN are thus $6 \times 6 \times 256$.

As shown in the center of Fig. 10.62, there are four RNNs computing sequences over all the 6×6 (36) features, and the RNN sequence length is 36 for each feature map (one of 256). The four sequences weight vectors w_x are concatenated into vector h_s as input to the FC layers, similar to the MDRNN formulation, as follows:

For notational convenience we convert symbols from Fig. 10.62:

$$[\uparrow, \downarrow, \rightarrow, \leftarrow] = [n, s, e, w]$$

⋮

$$h_s = (w_n + w_s + w_e + w_w)$$

In summary, the C-RNN explores a novel feature learning architecture, combining a CNN to learn the features with an RNN to learn the spatial relationships between the features. This author expects to see a trend towards more research into heterogeneous CNN/RNN architectures for computer vision applications.

RCL-RCNN

The Recurrent Convolutional Layer (RCL) proposed by Liang and Hu [751] is one of the first promising applications of RNNs to computer vision, and combines several novel features together.

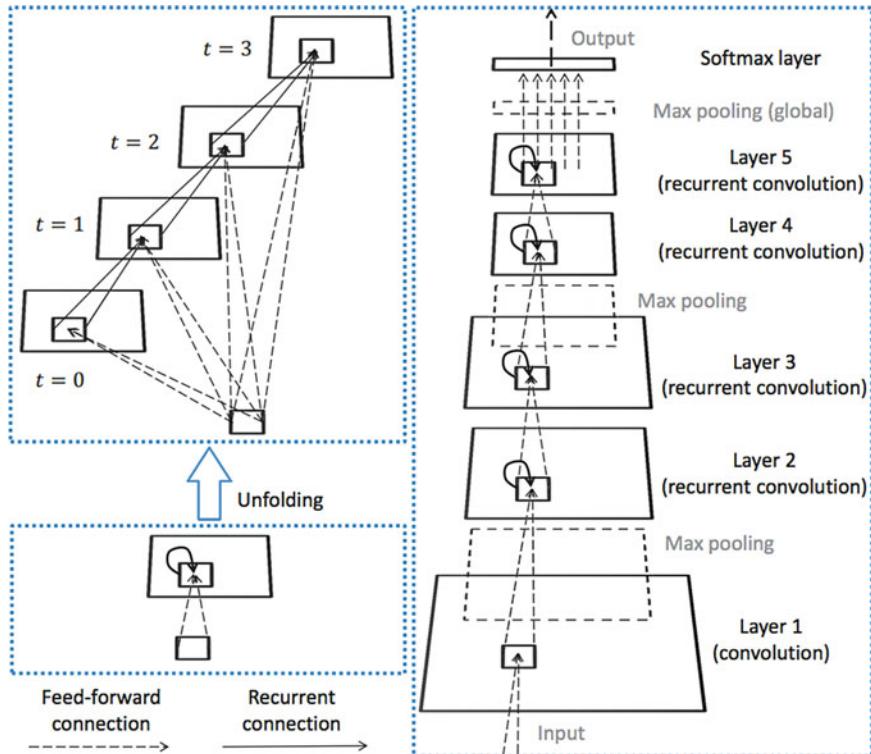


Figure 10.63 This figure illustrates the Recurrent Convolutional Layer (RCL), which uses an RNN in place of the convolutional function. Compare to the NiN method, which uses an MLP in place of the convolutional function. Image from CVPR Liang and Hu [751], © Springer-Verlag used by permission

The novel RCL (*Recurrent Convolutional Layer*) substitutes an RNN for the convolutional function normally used in CNNs. The RCL is used in each convolutional layer to build an FNN, which the authors refer to as an RCNN (*Recurrent Convolutional Neural Network*). It could be said that the RCL-RCNN is more like a CNN than an RNN and should be covered in the CNN survey section earlier; however, we cover it here instead since the RNN is incorporated. Additionally, the RCL-RCNN is quite novel in the direct use of *multi-scale inputs* to each RCL layer, as shown in Fig. 10.63, where the original image at *full scale* is fed into each RCL layer to compute the features, along with the feature map image from the previous layer which has been convolved and possibly pooled, *reducing the image scale*. This arrangement requires some scaling and alignment to get the images to line up for feature computations, and the exact method used by Liang and Hu is not specified in the paper [751].

The RCL-RCNN creates novel CNN-style feature patches by combining input patches recurrently from previous layers, for example using identically sized input windows from the current layer with input windows from a previous layer (which is shown in Fig. 10.63 as the original input layer), resulting in a deep recurrent feature representation. For example, a 3×3 feature for the current feature map k (from the set of 96 feature maps) using both the current c layer and a previous p layer, is computed as follows:

$$f_k(t) = \sum_{i=1}^3 \sum_{j=1}^3 w_{i,j}^c(c_{i,j}) + w_{i,j}^p(p_{i,j}) + b$$

where:

- (i, j) are coordinates in the image
- k is the index into the feature map images [1...96]
- w is the weight matrix (3×3 window) for a layer
- c is the current layer input window, $t(k)$ (3×3 window)
- p is previous layer input, $t - 1(k)$ (3×3 window)
- t is time, so t is the current layer, $t - 1$ previous layer
- b is the bias for the feature map k

As shown in Fig. 10.63, layer 1 is a CNN layer using 5×5 features, and layers 2–5 are RCL layers using 3×3 features. The network is expressed as RCL-RCNN- n , where n is the number of features per layer—*all layers use the same number of features*, in contrast to most other CNNs which vary the number of features at each layer. The authors report good results using 96, 128 and 160 features per layer, and report that the results only vary by about .5 % depending on the number of features per layer. Max pooling is used between some layers, with a region size of 3×3 and stride of 2 for some overlap. Local response normalization (LRN) is also used, similar to the method used in AlexNet surveyed above in the FNN section. A novel *Global Max Pooling* layer (GMP) combines the strongest activations from feature maps at each layer, which feeds into a softmax classifier.

Note that what we see in the RCL-RCNN and the NiN surveyed earlier is *both similar and novel*: both methods replace the convolutional layer used in standard CNNs with another function (MLP + RCL), and both methods replace the fully connected classifier with global, inclusive models (GAP + GMP).

dasNET

The Deep Attention Selective Network (dasNet) developed by Stollenga et al. [612] uses a novel recurrent feedback mechanism to adjust the sensitivity of convolutional feature weights in a CNN to boost the feature weights to optimize for misclassified samples. The central idea includes a method to *evaluate multiple hypothesis* in the form of slightly different feature weights, similar to the visual reasoning process a human expert might use, to arrive at the best feature weights overall. Some of the inspiration comes from the research of Branson et al. [755] and Cadieu et al. [617], showing how human experts perform complex visual evaluations by testing multiple hypothesis sequentially, similar to a *20-question game* where the number of questions is intelligently minimized via multi-class classification, using separate classifiers such as color, size, and shape to narrow down the scope of the solution space. One benefit of dasNet is that each feature is *automatically evaluated and optimized*, one at a time, by the built-in boosting process, reducing the need to visualize features as images for visual quality inspection [641].

The dasNet combines a slightly modified Maxout architecture (see Goodfellow et al. [610]) with a recurrent feedback mechanism to implement a variant of reinforcement learning [756] to optimize the feature weights. The Maxout network modifications allow for the weights to be modified one at a time as a single image is fed into the network multiple times in a tuning loop to compute the best weights over a range of possible values. Note that the Maxout Z-pooling parameters are set to reduce every n consecutive feature map images down to a single feature map image by taking the maximum value of all the maps in the Z pool (see the NiN, Maxout survey above for more details).

The dasNet operates in two phases: *training phase*, followed by the *boosting phase*. First, the Maxout network enters a normal training phase using labeled training samples in a supervised

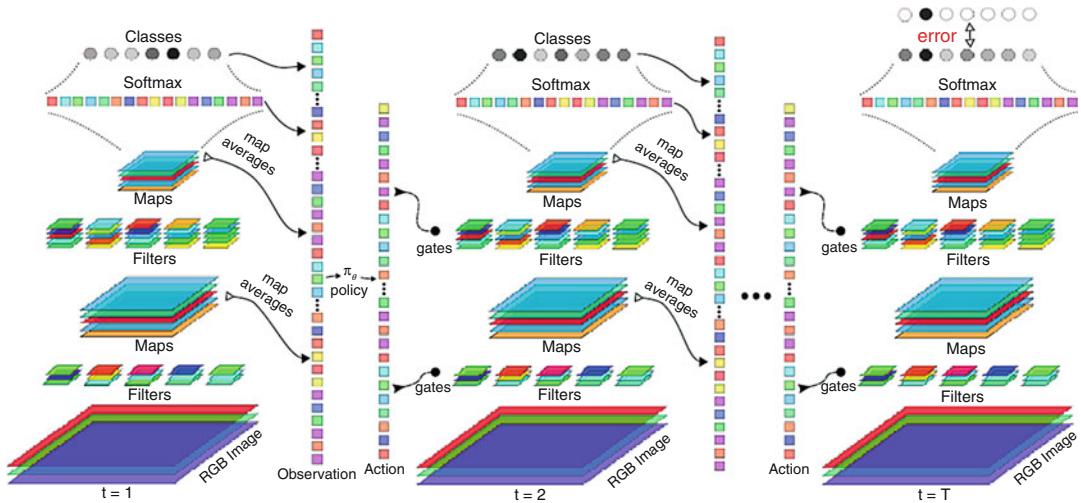


Figure 10.64 This figure illustrates the dasNet boosting phase, where the network parameters are composed into an observation vector (vertical bar) which is analyzed to produce an Action Vector used to optimize the feature weight parameters, implementing a form of reinforcement learning. Image from [612], © Marijn Stollenga used by permission

learning manner. Next, a *boosting phase* is performed to learn a *control policy* to improve the trained network by intelligently boosting the weights for misclassified samples or weakly classified samples. Note that the control policy is embodied in learned weights, trained during the boosting phase. The control policy uses a recurrent feedback mechanism, which iterates the network n times over a single image. During the boosting iterations, feature weight parameters and softmax classification scores are collected into an *observation vector*, see Fig. 10.64. The control policy takes input from the observation vector to implement a probabilistic optimization for the feature weights, reducing the error gradients by inhibiting or exciting the feature weights, using multiple passes over the same image to find the best weights. The output of the control policy is an *Action Vector* used to modulate the feature weight parameters. The end result is a form of *attentional learning*, which selectively enhances and inhibits features which were not effectively trained during normal supervised training. The boosted feature weight changes are applied before the maxout feature map reductions.

In essence, the observation vector provides a way for all parameters to influence each other in a form of *coadaptation*. The observation vector and action vector are the parameter space used to implement the control policy for weight optimizations.

The boosting phase involves the following process:

- Select a random batch of images from the training set
- Run the random batch through the network n times:
 - Run each image through network m times:
 - Run image through network
 - Collect the observation vector
 - Compute error term (*misclassified images are highest)
 - Update control policy from error term and observation vector
 - Produce action vector via control policy
 - Change weights according to action vector

In summary, dasNet is one of the most elegant combinations of RNNs and CNNs, leveraging the strengths of both methods, and in addition incorporating reinforcement learning to select and boost

misclassified and weak features to optimize the feature weights. In addition, dasNet points the way towards future research into more intelligent feature weight optimization and boosting methods, including selective image feature focus (which may bring us back full circle to earlier research on local feature descriptors and interest points to find candidate features to focus on within a CNN or RNN).

NAP—Neural Abstraction Pyramid

The Neural Abstraction Pyramid (NAP) proposed by Behnke [548] is a novel hierarchical recurrent network, combining lateral locally recurrent, feed-forward, and backward feedback connections into a single network. The NAP takes a unique approach to neural network design, guided by *iterative refinement* objectives to form the network and processing architecture, following neuroscience research. The human visual system iterates, and saccades, at specific interesting locations, comparing features and higher-level concepts together at the same level and across levels to verify key details. Neither local features relationships nor multilevel feature relationships alone can model the human visual system, so NAP supports a synergistic classification based on local lateral as well as hierarchical feature relationships. NAP addresses a major gap in existing CNN architectures in which features are independent, and the classification model is more like a probability distribution of features, with no notion of local or hierarchical feature dependence.

The NAP classification process is iterative, focusing on one feature, then another, using separate sets of *excitatory features* and *inhibitory features* to build confidence, inspired by neurons in the visual cortex which excite and inhibit in the same local region and across spatial regions, described later. Reliable and trusted features excite confidence and inhibit unreliable features. Short-distance correlations, or lateral correlations, are most important in the saccadic iteration process, and long distance correlations between different scale feature layers is less important. The iterative approach assumes that an FNN captures primitive features in the low layers, and then lateral and local spatial feature relationships within layers can *resolve* most partial representations, and decisions that cannot be made locally are *deferred* to higher layers. The highest layers contain abstract feature representations.

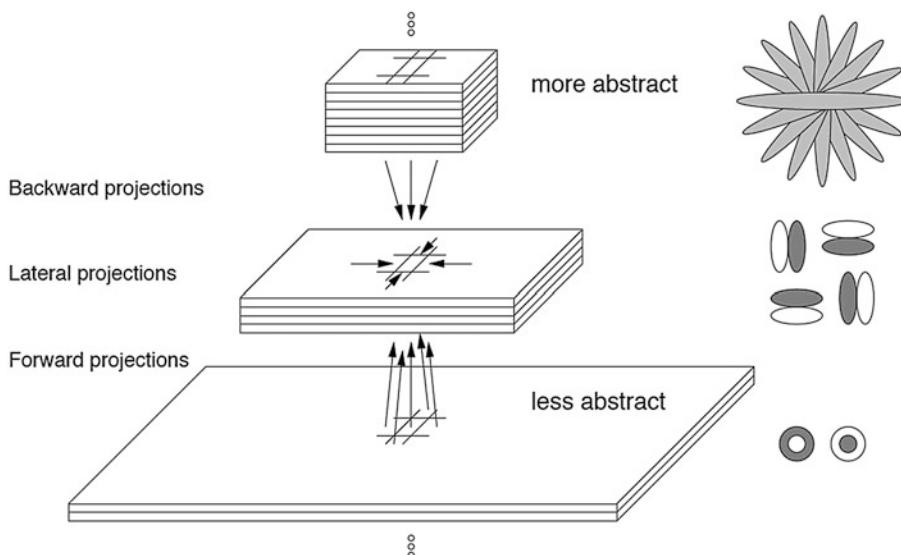


Figure 10.65 This figure illustrates the Neural Abstraction Pyramid, showing low-level feature layers, mid-level feature layers, and higher-level concept layers. Image © Springer-Verlag, from Hierarchical Neural Networks for Image Interpretation, Sven Behnke [548], Draft submitted to Springer-Verlag Published as volume 2766 of Lecture Notes in Computer Science

As shown in Fig. 10.65, the basic architecture consists of a hierarchical pyramid of features, where the lower layer features are primitive concepts such as edges, and higher levels represent mid-level and higher level concepts. The features are referred to as *Feature Cells*, and the number of feature cells increases with higher levels of abstraction. Like CNNs, NAP low level features are taken at a higher spatial resolution, and higher-level features are taken at reduced spatial resolution. The number of features increases up the NAP layer hierarchy, and the higher layer features represent more abstract concepts. The features are motivated by the success of CNNs and weight sharing is used at each layer.

The NAP pyramid abstraction is novel, and allows all features in the hierarchy to be *associated* laterally and across levels with regard to spatial x,y coordinates, so that a higher level concept spanning a large pixel window can be decomposed down the pyramid into mid level or low level features within the same window. Spatial relationships are thus preserved and used to guide classification. As shown in Fig. 10.65, NAP associates features using *Hypercolumns* and *Hyperneighborhoods*. Each feature retains an x,y coordinate which locating each hypercolumn and hyperneighborhood. The pyramid structure allows features to be *correlated* via projection across scale layers, for example allowing for analysis of a high level abstract feature and its contents of mid level and low level features. The feature relationships in the hypercolumns and hyperneighborhoods are described in a convolutional *weight vector*, representing associative excitation and inhibition. Weights represent local, lateral contextual influences within the same layer, and bias influences from other layers via forward or backward connections within the hierarchy. The NAP connection organization within and across layers can be advantageous for various coding optimizations and caching mechanisms that exploit locality, such as parallel processing, SIMD, and SIMD.

Like a CNN, an NAP feature is computed as a weighted sum and bias. However, the feature inputs are much more complex. Feature cells are composed over a lateral region + recurrent inputs + inputs from other layers, followed by a transfer function such as a zero-centered sigmoid. For details, see Behnke [548] section 4.2. NAP provides for both excitatory features and inhibitory features. The inhibitory features are composed of the sum of features within a layer, following the Neocognitron model [570, 571, 679], where the feature sum is used like an inhibitory gain factor. Global inhibition and excitation can be used to implement winner-take-all classification or boosting. Local inhibition or excitation can be used similar to max pooling in a CNN. The NAP operates in a recurrent fashion, recomputing features at discreet time steps. Depending on the specific NAP architecture, features can be computed bottom-up, layer by layer, as in an FNN, and computed as groups within layers, for example excitatory feature groups and inhibitory feature groups.

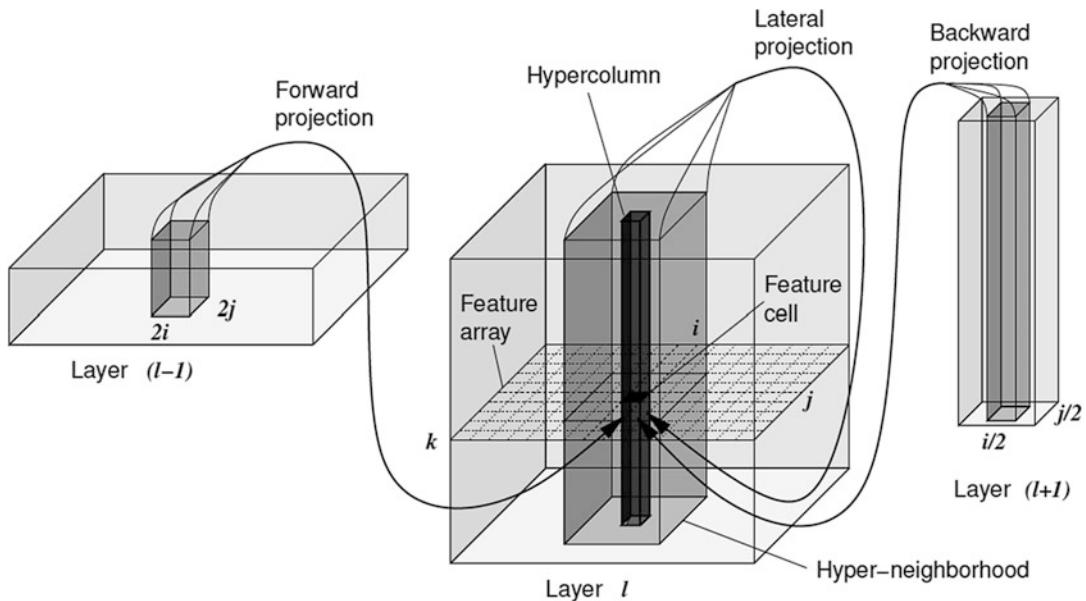


Figure 10.66 This figure illustrates the concept of using *feature coordinates* to allow for location-based hierarchical feature associations across layers in the NAP pyramid, Image © Springer-Verlag, from Hierarchical Neural Networks for Image Interpretation, Sven Behnke [548], Draft submitted to Springer-Verlag Published as volume 2766 of Lecture Notes in Computer Science

NAP implements a novel set of learning rules to optimize the response of each sparse feature, ensuring each feature is unique and necessary. The learning rules follow Hebbian [758] principles for weight updates and MAX pooling, rather than following gradient descent methods which uniformly distribute gradients across all weights indiscriminately for weight updates. In NAP learning, each feature is adjusted *individually* during training to respond more specifically, rather than more generally. *Competition* is used at each level to find features which should be inhibited or excited via weight updates, with the goal that all features respond to specific rather than general features, so that no feature wins too frequently. As the training image is sent through the network, the two most responsive features are selected and the top responding feature is adjusted relative to the second most responsive, which decorrelates the feature from all other features. Individual weights are adjusted differently within the feature kernel neighborhood to optimize maxima and ridge detection.

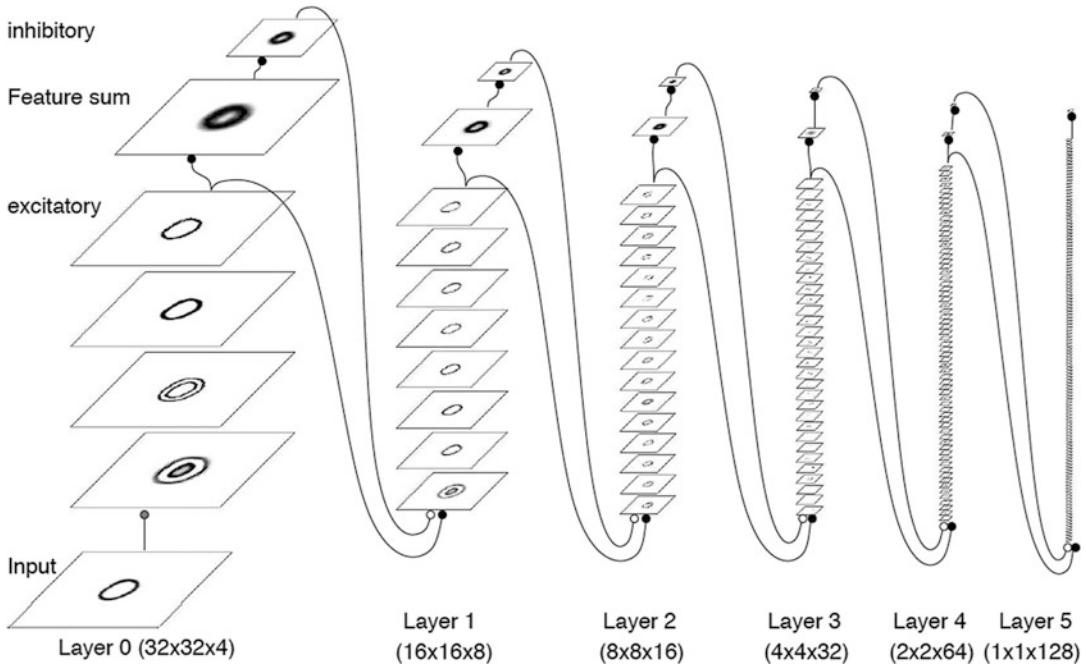


Figure 10.67 This figure illustrates the NAP architecture for an unsupervised, sparse learning approach to handwritten digits, Image © Springer-Verlag, from Hierarchical Neural Networks for Image Interpretation, Sven Behnke [548], Draft submitted to Springer-Verlag Published as volume 2766 of Lecture Notes in Computer Science

Behnke demonstrates, at a high level, how the NAP architecture may be adapted to a wide range of applications including local contrast normalization, handwriting binarization (i.e., skeletonization morphology), and translation invariant feature extraction using Gabor filters implemented using a DFT over Gaussian localized windows (similar to Wavelets). In addition, Behnke develops novel NAP architectures and performance analysis for more demanding applications including postage meter stamp recognition, binarization of 1D barcodes and 2D barcodes, face localization, local iterative image reconstruction on MNIST handwriting digits with injected occlusion defects and injected noise defects. Fig. 10.66 shows a NAP architecture for a six-layer sparse hierarchical feature learning feature system using an unsupervised learning approach applied to handwritten digits.

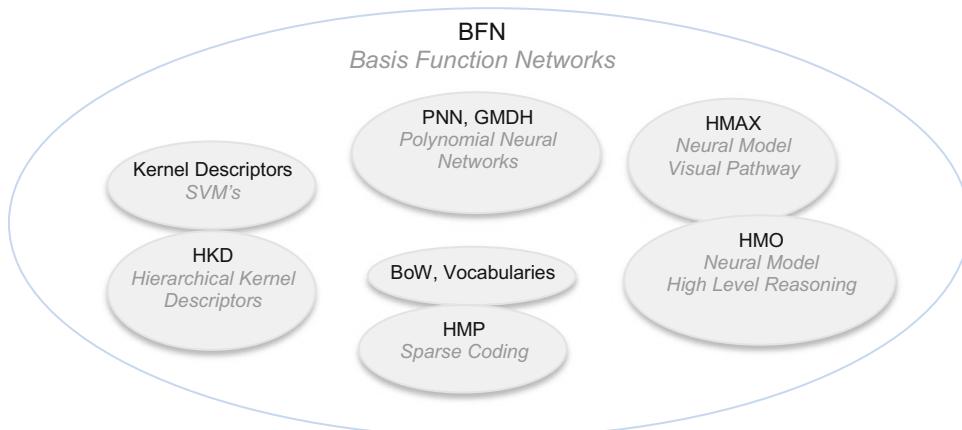


Figure 10.68 This figure illustrates the Basis Function Networks covered in the survey

BFN Architecture Survey

This BFN survey will cover a lot of ground, touching on most of the feature learning architectures used in computer vision that are not considered in the ANN style deep-learning architectures as covered in the FNN and RNN survey sections earlier. In other words, we consider BFNs as a catch-all category to contain all other non-neural network DNN architectures, which is a large category indeed, so we can only select representative architectures for this survey, and highlight common underlying concepts and components.

We distinguish basis functions from CNN-style weight features *generated* from training data. For example, a Gabor function or a Fourier Series component are basis functions generating basis features under this taxonomy. We define a BFN as a *Basis Function Network*, a network using *basis features* rather than purely convolutional features and artificial neuron models common in CNNs and RNNs. In a BFN, *basis functions* are used rather than *convolutional neural layers*. In many systems, hybrid feature models are used, combining BFN, CNN, RNN, and several styles of features and classifiers, which we refer to as *ensemble methods* or *hybrid networks*. Basis functions and CNN style functions are used together in some networks in this survey.

First we will survey several key *background concepts* to lay the groundwork for the BFN architecture surveys including:

- Feature Models, Classification models
- Basis Sets
- Vocabulary Learning
- Kernel Descriptor Learning
- Sparse Coding And Codebook Learning
- Other classifiers, Trees, Boosting

After exploring the background concepts, we survey representative BFN Architectures:

- Polynomial Features—PNN
- Kernel Descriptor Features—HKD
- Local Feature Descriptors—HMP
- CNN + Basis Features—HMAX

Concepts for Machine Learning and Basis Feature Networks

In this section, we discuss some key background concepts used in feature learning networks using basis functions including vocabulary methods, codebooks and sparse coding, and statistical classification models such as kernel machines and SVMs, which are distinct from ANN-style convolutional filter features and ANN-style classification models such as FC layers. The goal of this section is to lay the groundwork at a high level to appreciate the various BFN *architecture* examples in the survey.

Feature Models, Classification Models, Decision Models

As shown in Fig. 10.69, the basic models used to define BFNs and CNNs are illustrated at a high level for comparison. Note that neural network models are a subset of statistical models, for example neural classifiers such as FC layers and simple MLPs are *equivalent* to statistical methods such as regression and SVMs. However, BFNs are typically not based on neural models, but rather on a different set of

Decision Model

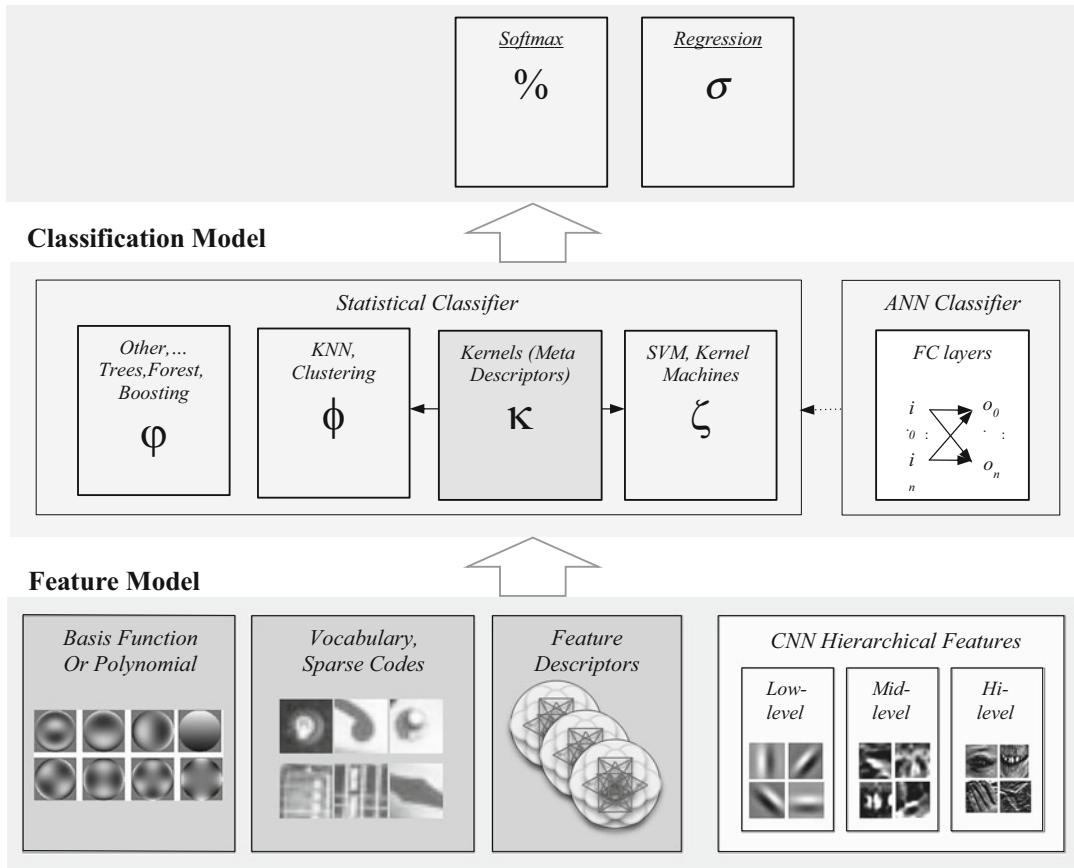


Figure 10.69 This figure illustrates feature learning architecture concepts for basis function networks, compared to CNNs. Note: Zernike polynomials (bottom left) are one of many mathematical basis feature alternatives. Depending on the features chosen, the classifier stage may be different, for example requiring kernel methods to project features into a linearly separable space for an SVM. CNNs commonly use SVMs in place of FCs

feature models and classification models. As shown in Fig. 10.69, feature learning networks *roughly correspond* to the following basic model parts:

- **A Feature Model:** any feature descriptor, such as pixel patches, local, regional, basis, or ANN style features. The features may be organized into a fixed-size feature hierarchy as used in CNNs, and encoded into a sparse codebook or a visual vocabulary (bag of words).
- **A Classification Model:** we divide classification models as follows:
 - *Convolutional Classifier Models (ANN, CNN, RNN)*
FC classifiers (discussed earlier) are common in CNNs and ANNs, implementing layers of simple linear classifiers using layered neural models of weights and bias trained using backprop.
 - *Statistical Classifier Models (many methods exist, discussed later)*
Statistical regression and clustering methods such as KNN (see Chap. 4).

SVM (Support Vector Machines), Kernel Machines, a framework for regression analysis and classification, mapping linearly unseparable features into a linearly separable space using kernel methods.

- **A Decision Model:** (*sometimes the classifier performs this function*): performing some decision function on the classified data, for example a probabilistic softmax %.

The *feature model* in large part determines the feature learning architecture, and especially influences the *classifier model*. Note that the items in shaded gray boxes on the left in Fig. 10.69 are typically associated in the same style of architecture, such as statistical classifiers and non-convolutional style features, but many variations are used in practice. The style of features also determines the *training protocol*, since convolutional features are tuned in fully convolutional networks via gradient descent methods, while BFNs use a wide range of other classifiers and *training protocols*. Neural style classification models were introduced earlier in this chapter under the *Fully Connected (FC) Layers, Flatten, Reduction, Reshape* section. Statistical classifier models, SVMs, and Kernel Methods were briefly introduced in Chap. 4, and are discussed in more detail later.

Function Basis vs. CNN Basis vs. Other Models

What is a basis function or basis feature? We take a wide and inclusive view of basis functions, defining a basis as *a base set of features generated by a function*. A basis function is typically a mathematical function or polynomial (see Chap. 3 and Fig. 3.19 for a summary of mathematical basis functions), but by our wider definition may also be any of the local feature descriptor methods discussed in Chaps. 4–6, as well as global and regional feature descriptors discussed in Chap. 3. For example, in frequency space, the Fourier Series components represent Sine and Cosine basis functions. *Gabor Basis Functions* (see Chap. 3) describing oriented edge-like features have been used in several DNNs for the lower level features, since they resemble the types of low-level edge features detected in the V1–V2 regions of the visual pathway as suggested by neuroscience research. Using basis features as low-level features eliminates the need to learn low-level features from scratch. Then, mid-level and higher level features can be learned on top of the Gabor functions, and expressed as convolutional filter masks for CNN style feature learning of mid-level and high-level features, for example the HMAX model surveyed later is a good example. While most ANNs use the basic convolutional *dot-product of inputs against weights + bias followed by a nonlinear function*, it is possible to create an ANN using purely basis functions such as the PNN or *Polynomial Neural Network* surveyed later in this section. (Note: A CNN feature hierarchy of features could be considered a CNN basis set under our loose definition; however, we separate CNN models from BFN models for the sake of the taxonomy.)

While the CNN tunes the feature weights to represent a group of similar features, the statistical methods tune feature groups to represent clusters of similar features. In other words, the CNN tunes each feature weight, contrasted against the statistical methods that tune the feature space.

We can compare the CNN feature models and architectures with other feature models and architectures to gain some insight:

- **The CNN Feature Model:** CNN features are generated using a complex backpropagation process analogous to averaging, tuning each feature weight in each feature to represent groups of similar features. The CNN classification stage typically uses an FC layer (discussed earlier in this chapter)

composed of a single large *1D feature weight vector* taking inputs from all the feature weights, trained in a similar manner to the other lower-level feature weight matrices. A CNN can be designed using purely convolutional features end-to-end by incorporating feature weight layers and FC layers. Some CNNs use an SVM in place of the FC, or on top of the FC. The softmax layer can be used as the last layer in a CNN as a probabilistic method for generating a confidence score for the classification decision.

- **Other Feature Models:** Other feature models such as basis features, vocabulary features, sparse codebooks, and local feature descriptor sets, require some amount of training to learn the feature set or tune the feature descriptor parameters, for example tuning and selecting Gabor functions, boiling down a vocabulary of features into a sparse set, and learning and tuning local feature descriptors (*local feature descriptor learning*) as discussed in Chaps. 4–6. Basis features other than CNN features typically use some sort of statistical classification model, like an SVM, or a clustering method such as KNN, and the final decision layer may be a softmax.

Visual Vocabularies, Bag of Words (BoW) Model, Alternative Encodings

Similar to a word vocabulary, a *Visual Vocabulary* [768] or *Bag of Words* (BoW) model allows for an image or image region to be classified based on the visual words detected. A visual vocabulary may also be referred to as a *visual dictionary* or a *codebook of code words*. Many types of feature descriptors can be used as visual words. The vocabulary can be used to form a *histogram descriptor*, as shown in Fig. 10.70, binning the total count of detected features against the closest feature in the vocabulary. Depending on the descriptor format of the visual words, different types of distance functions are used to perform the feature matching, and a variety of methods are used to reduce the vocabulary set via clustering similar features together, analogous to sparse coding methods.

The *vocabulary* is the basis feature set. The features in the basis set may be *encoded* to change the representation of the feature, typically to a smaller representation or alternative encoding, in order to reach design goals such as reducing the memory footprint and increasing searching and matching performance. We will survey several encoding methods later in this section.

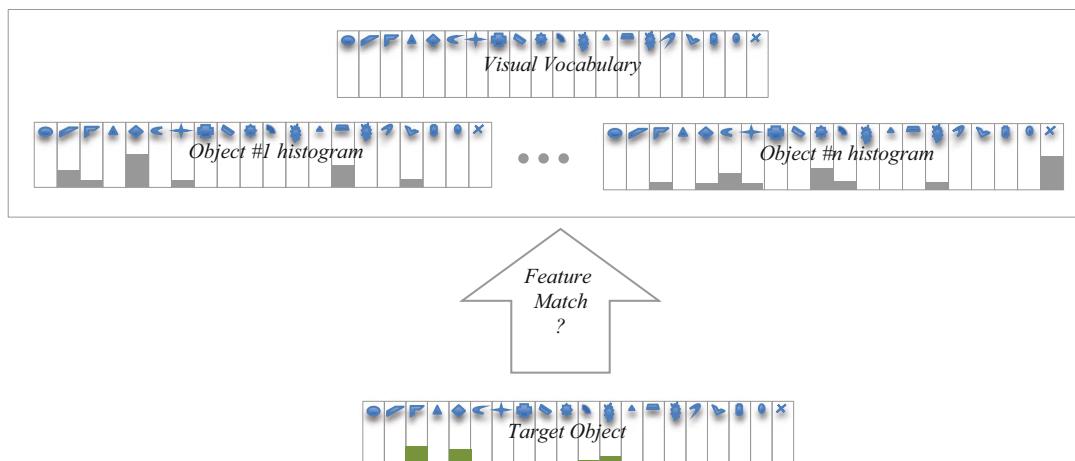


Figure 10.70 This figure illustrates a visual vocabulary and simple histogram-based feature vector representation, each object is a histogram vector counting the features detected in the object. Target features may be matched via reconstructing weighted combinations of a few vocabulary features

Visual dictionaries typically contain a few thousand or a few tens of thousands of visual words. Vocabulary methods are flexible and allow for (1) powerful local feature descriptors such as FREAK, ORB, and SIFT, (2) vocabulary feature comparisons using a wide range of distance functions, (3) a wide range of clustering methods to boil down the feature set into a lower dimensional set, and (4) high dimensional feature set topologies such as aggregated sets, and hierarchical sets. Search time for feature distance measurements and memory usage for the feature set can be limiting factors.

The vocabulary is typically encoded in a sparse manner, boiling down the features to the appropriate sized set using a range of clustering methods (see Chap. 4). In the literature, several related terms are used to describe a visual vocabulary, such as a *bag of words* or *bag of features* model, also called a *dictionary* or *codebook* (some practitioners make very fine distinctions within this terminology). One of the first vocabulary methods based on *texture patches* was pioneered in 1981 by Julez [764], using selected *textons* or texture patches for the feature vocabulary, and other research has continued along these lines [21, 40–42] including research to create a dataset of preexisting texture samples from real images in the CUReT dataset, see Appendix B. Vocabulary methods are applicable to generating image statistics and for scene recognition, see Jurie and Triggs [768]. Often, vocabulary methods are applied to image recognition, image and scene classification, and visual search engines.

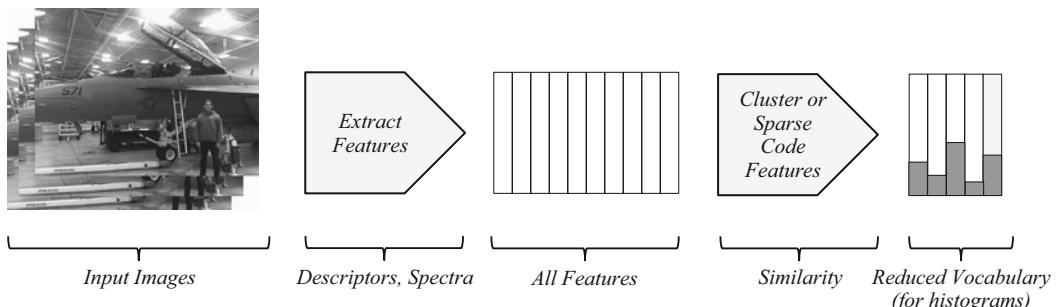


Figure 10.71 This figure illustrates the process of developing a visual vocabulary. As shown on the right, the vocabulary is a clustered or reduced codebook set of prototype features, which is used as a feature vector to contain a histogram binning of the occurrence of each vocabulary feature in a given image

As shown in Fig. 10.71, any type of feature can be used as the vocabulary basis. The vocabulary feature descriptor is typically a *histogram of features* from the codebook. Basic vocabulary models are primitive, and rely on *orderless* presence or absence of features, ignoring sequences, spatial associations, and feature invariance criteria such as rotation and scale. However, orderless vocabulary methods have been enhanced several ways, for example by aggregating multiple descriptors such as color, shape and texture [773], or color, gradients and LBP [774], incorporating feature descriptor locality to cluster nearby descriptors into a metadescriptor group by Ionescu [776], or by incorporating geometric scale (see the Pyramid Match Kernel [212] in Chap. 6).

The number of features in the vocabulary is a design choice. One might include all features detected in the training set into an exhaustive vocabulary, but for efficiency it is common to boil down the vocabulary into a smaller representative set of features using sparse coding methods, clustering methods or soft quantization methods [842]. We note that both BoW methods and CNN methods use hard-coded fixed numbers of features. How many features are needed? The feature count is typically based on the experience and intuition of the practitioner, along with some experimentation during training. Usually CNNs use a different number of features at each layer, maybe a few hundred features per layer, and the features are classified using one or more fully connected (FC) layers or

perhaps using an SVM. However, for basis features, descriptors, visual vocabularies, and the optimal number of features can be learned from the training data, taking guidance by finding the *minimal reconstruction error*, see Chap. 4 for some discussion on image reconstruction from local feature descriptors.

Vocabulary Encodings

The vocabulary items may be *encoded* in a variety of ways for convenience. For example a feature descriptor like SIFT or a spectra such as pixel intensity may be encoded or projected into another representational format to enable more uniform treatment in a particular classification algorithm. The encoding may be a kernel-based meta-descriptor, or simply a distance from a prototype feature in the vocabulary. Early work by Jakkola and Haussler [841] on deriving kernel functions has been extended by many researchers, which we cover in this section. Many encodings have been devised to optimize feature encodings for searching and matching. One of the key ideas used to make visual vocabulary representation and matching more efficient is to *aggregate* sets of feature descriptors into *compact encodings*, and then the encoded visual words become smaller and faster to process for classification. We highlight a few alternative visual word optimization methods here, and refer to the reader to Jegou [842], Chatfield et al. [844] and Tolias [840] and van de Sande [550] to dig deeper.

- **Histogram Encoding** (*Visual word dictionary basis, hard assignment to a single codeword, simple SSD or SAD distance*)—Simply creates a histogram *counting* the number of features matching each *closest descriptor* in the dictionary. The histogram bin assignments are hard assignments to the single nearest visual word to a single descriptor. The histograms may be normalized or weighted [842]. Correspondence can be computed using Euclidean distance, SSD, SAD, and other simple methods.
- **Kernel Codebook Encoding** (*Visual word dictionary basis, soft-assignment to multiple codewords, simple distance to nearest matches*)—Uses a soft assignment or *distributed assignment* of a single feature descriptor to multiple nearest features in the vocabulary, such as the nearest five features. See Gemert et al. [846].

Note that Fisher Vectors, VLAD and Super Vector encodings are similar and efficient since they encode *differences between basis shapes in feature centroid space* rather than encoding entire descriptor vectors for computing distance. As a result, distance is strongly affected by the clustering method used and the cluster initialization process. For example, see Fig. 10.73 illustrating cluster center anomalies using K-MEANS.

- **Fisher Vectors** (*HMM Gaussian basis, residual vector difference of feature descriptor from nearest HMM Gaussian basis*) - Fisher Vectors developed by Perronnin et al. [545, 838, 839] are based on creating a vocabulary set of *Gaussian basis features* to encode features as a Gaussian Mixture Model (GMM). The Gaussian basis features are created from a training set of feature descriptors such as SIFT. The vocabulary is created by clustering all the feature descriptors from the training set into several thousand clusters. Perronnin [848] optimized the vocabulary set by reducing the cluster count to 256 by first pooling the SIFT descriptor set using PCA. The Gaussian basis features are defined by each cluster shape's centroid, mean, and variance. The Fisher Vector encoding is a concatenation of all the *distances* between the GMM basis features. The *distance* from target features to the GMM basis shapes is a scaled directional gradient, which reduces the size of the descriptor to a small scalar value, rather than a large set of individual distances

matching the SIFT descriptor structure. Perrorin [847] developed a method to optimize Fisher Vectors for more efficient large-scale searching, computing similarity using the dot-product of Fisher Vectors, and also a binarized method of representing Fisher Vectors with Hamming encoding.

- **VLAD, MultiVLAD** [840, 842] (*Feature descriptor basis over visual word dictionary, Cosine difference of residual angles*)—VLAD (*Vectors of Locally Aggregated Descriptors*) encodings are optimized to be compact and fast for feature matching. VLAD builds a vocabulary of local descriptors such as SIFT, and clusters the descriptors using K-MEANS or a similar method. VLAD is based on a coarse vocabulary set of perhaps 256 features. For each basis cluster center, the residual distance between clusters is accumulated and concatenated into a vector matching the size of the SIFT descriptor, typically 128 bytes. So each VLAD is a vector aggregation of residual distances. VLAD encodes the *distance* of a descriptor to the basis *cluster center* using Cosine distance residual angles. VLADs are designed to be very low dimensional descriptors containing 16 bytes to describe an entire image, which is ideal for large-scale image classification tasks. For classification, each new target descriptor is evaluated based on the distance to the cluster center. Arandjelović and Zisserman [843] provide a very readable summary of VLAD and make several improvements to the original method, notably the Multi-VLAD method to use multiple VLAD descriptors at multiple scales and multiple tilings across the image.
- **Super Vectors** (*Centroid of codeword cluster basis, simple distance, residual distance*)—Proposed by Zhou et al. [845] are similar to Fisher encodings combined with a histogram encoding. One variant is assignment to the closest codeword, and another variant uses soft assignment distributed across a set of five nearest neighbors. Assignment is based on a novel distance criteria incorporating a cluster normalization step, the first order difference between individual features and cluster centers, and factoring in the mass of the clusters.
- **Hamming Encoding** (*Hamming Code Basis, fast Hamming Distance*)—Proposed by Douze [849] extends the BoW model by a combined encoding for each visual word including (1) a binary signature generation and encoding for use in quickly computing Hamming Distance between descriptors, since Hamming distance is extremely fast and is often implemented in CPU and GPU instruction sets, and (2) weak geometric consistency constraints providing matching penalties for rotation and scale mismatches. A novel soft-assignment to a variable number of closest visual words is also proposed.
- **FLAIR** (*Visual word basis + integral images for each codeword, various distance functions*)—FLAIR (*Fast Local Area Independent Representation*) Developed by van de Sande [550] encodes the image in a multidimensional integral image space containing codeword indexes for each pixel into the dictionary. One integral image is encoded for each visual word. Integral images are extremely efficient and fast to compute in constant time, so matching performance is very fast and predictable. FLAIR can embed one or more encodings into the integral image framework, with one integral image dimension per encoding. FLAIR is most effective on more complex encodings, such as Fisher Vectors and Flair, compared to BoW representations. Van de Sande provides an excellent survey of relevant search optimization approaches.

Next we discuss *sparse coding methods*, which are used to encode vocabulary features into an overcomplete sparse set for classification.

Sparse Coding and Codebook Learning Overview, K-MEANS, K-SVD

Sparse coding methods are used in a wide range of applications such as noise removal, inpainting, signal encoding, and general compression. For computer vision applications, a sparse codebook

contains a *sparse overcomplete set of codewords (features)*, from which unknown features are matched to the codebook by approximately reconstructing the unknown feature using linear weighted combinations of a small set of codewords. An overcomplete set allows for *more than one way* to approximate a feature from the basis set of codewords. For example, consider Fourier series based signal reconstruction.

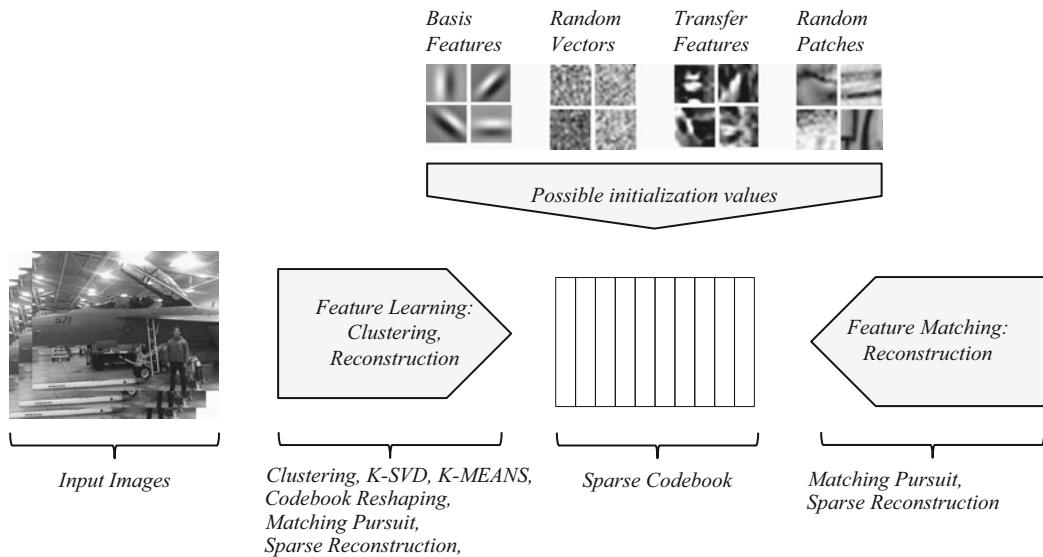


Figure 10.72 This figure illustrates the process of sparse codebook initialization, feature learning, and feature matching. Note that the same types of reconstruction algorithms are used for learning the features, and reconstruction of feature matches from the sparse codebook features

Image reconstruction from a sparse codebook of image patches is illustrated in Fig. 10.72. The sparse codebook becomes the *basis feature set* for a given feature domain, and may be chosen from some preexisting basis set which can be further tuned to fit the training data, or an entirely new basis set can be learned from a training set. *Overcompleteness* of the codebook is required for sufficient reconstruction from combinations of codewords. We provide a fundamental introduction to sparse codebook feature learning here, along with references to dig deeper.

Sparse coding can be divided into the following tasks:

1. *Codebook Initialization* from some basis set or random values
2. *Feature Learning* to modify the codeword entries from the training data
3. *Feature Matching* via reconstruction from small sets of codebook features

A sparse codebook does not contain a complete vocabulary of all possible codes needed to perfectly represent or reconstruct the training data, but rather sparse codes are a *compressed set of features*, defined to the desired level of sparsity desired for signal reconstruction and pattern matching accuracy. Sparse coding is a form of signal compression. One goal of signal compression can be to represent a signal using a smaller number of signals than normally required by the Nyquist Frequency sampling rate, which states that the signal must be oversampled by at least $>2\times$. Sparse coding dramatically reduces the reconstruction requirements. Sparse coding is analogous to coalescing scalar values in a *1D feature space* using some form of *quantization criteria*, such as varying bit resolution

from 8 bits to 4 bits to collapse similar values into fewer values, reducing the level of detail. However, sparse coding operates in a multidimensional feature space. For computer vision, sparse coding addresses the trade-off between *classifier discrimination vs. classifier compute efficiency*, since more features provides more discrimination, at the cost of compute efficiency.

Several approaches are taken to learn a sparse codebook of features, for example single-layer SIFT feature codebooks and pixel-patch codebooks. Aharon et al. [779] provides a good survey of various sparse coding methods, see also Feng et al. [797]. See Yu et al. [796] for a comparative survey of SIFT vs. pixel patch methods used for sparse coding, which are shown to be equally effective. See Candes et al. [795] for more on theoretical sparse coding optimizations. See Olhausen [2127] for details on neurobiological theories about sparse coding. See Coates and Ng [618], who find that the encoding scheme and architecture are more critical than any specific feature descriptor method used. In addition, the preponderance of researchers note that many weak features in a feature hierarchy can be as effective as powerful local feature descriptors for sparse coding. Mairal has developed several methods for feature learning and sparse coding, see [803, 804], including novel hierarchical methods [805, 806]. Bo et al. [132, 234] implement a novel hierarchical sparse codebook using a *Hierarchical Matching Pursuit* (HMP) to learn features from raw pixel patch features, surveyed in detail later. Ranzato et al. [551] developed a model of sparse coding convolutional features similar to an RBM network called SESM (*Sparse Encoding Symmetric machine*).

We can illustrate sparse coding by following Fig. 10.72. First, the sparse codebook is initialized via one of many methods such as (1) transfer learning using pretrained features from another sparse codebook, (2) from a basis set such as DCT or Gabor feature, or (3) from randomly sampled image patches or random valued feature vectors, or (4) from a set of candidate local feature descriptors computed at interest points which have first been clustered and reduced to a smaller set. For example, Bo [233] initializes the codebook to a set of overcomplete DCT basis features, and in another case initializes a codebook to a random set of image patches taken from the training images. Since it is impossible to know in advance precisely how to initialize the codebook, using an *overcomplete set* with more than enough detail is common, rather than an undercomplete set. Both the range of feature variation and the number of features in the set are inter-related in this respect.

Next, feature learning is accomplished tuning the initial sparse codebook features to *fit* the training data. To illustrate the concepts of sparse codebook building, we focus on the K-SVD method developed by Aharon et al. [779]. K-SVD iteratively *recomputes the feature codebook* during training. K-SVD is a singular value decomposition method, implementing a parameterized generalization of K-means, which iteratively recomputes the codebook to minimize the error between the new sample and the closest codewords. K-SVD will recursively reshape the codebook as it learns, to balance the uniformity of feature distribution within the feature space. K-SVD is one of many parameterized generalizations of K-MEANS clustering [779].

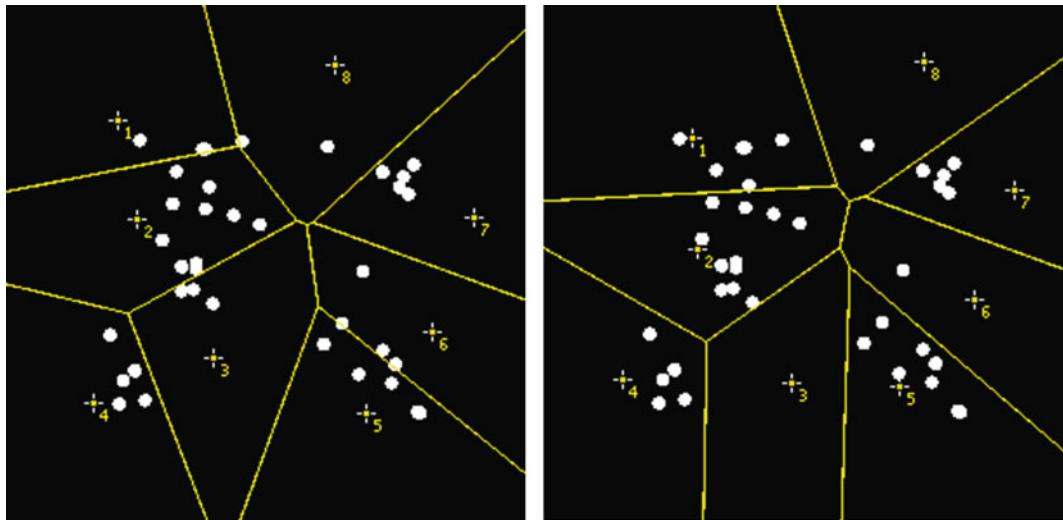


Figure 10.73 This figure illustrates K-MEANS clustering into eight groups using Euclidean distance, with Voronoi Tessellation lines dividing each cluster. The *left* and *right images* contain the same data; however, the centroid starting positions have been assigned to be at slightly different coordinates for each *left* and *right image* (see the cross hair positions), illustrating how K-MEANS produces different clusters depending on the starting centroid selected

K-MEANS learns a *centroid cluster set* from the training samples, and then test samples are classified by matching to the nearest cluster in the centroid cluster set. The cluster set positions are therefore the codebook entries for the feature vocabulary. K-MEANS iteratively clusters samples into a chosen number of cells around a centroid. We refer the interested reader to more detailed references, see the original paper by Lloyd [801], a good text by Hartigan [100], and a larger survey text by Hastie [347]. As shown in Fig. 10.73, the K-MEANS cell boundaries are often illustrated via the *Voronoi Tessellation Diagram* proposed in 1908 by Voronoi [802], which partitions the space into polygon cells with polygon boundaries equidistant to each neighboring cell centroid. K-MEANS allows the number of centroids and the coordinates of each centroid to be chosen in advance, and then the algorithm assigns each sample to the nearest centroid. The centroid distance to each sample may be computed simply in *Euclidean space* as the average of all x,y coordinates in a region. K-MEANS results vary depending on the distance function used to determine the cluster centroids, for example Euclidean distance vs. Manhattan distance will produce different results. Also, K-MEANS clustering will vary depending on the precise number of centroids chosen, and the coordinates of the centroid positions, as shown in Fig. 10.73.

K-MEANS clustering can be summarized as follows:

Initialization:

- Choose the number of clusters.
- Assigning a *centroid starting position* for each cluster, perhaps by choosing samples points from the dataset at random, or by some other method [100, 347].

Iteratively learn and refine clusters:

- Assign each sample to the nearest centroid using some distance function, such as Euclidean Distance (see distance functions in Chap. 4).
- Recompute each centroid to refine fit to nearest samples.
- Repeat 1–3 until stopping condition reached: the difference between the new centroid and previous centroids is computed and compared to a threshold; when the values are lower than the threshold, K-MEANS stops.

The weakness of K-MEANS and similar methods is that clustering results vary depending on the number of centroid points, the distance function used, and the coordinates of each centroid point. Therefore, improvements and variations have been devised such as K-SVD to solved for other specific objective functions, see Chap. 4 for a survey of clustering methods and clustering objectives.

K-SVD developed by Aharon et al. [779] is a combination of *K-Means (K)* and *Singular Value Decomposition (SVD)*—thus the name K-SVD. The goal of K-SVD is to generalize and parameterize K-MEANS in the context of SVD. SVD is equivalent to Principal Component Analysis (PCA), since both methods decorrelate values to identify the principal component values with the highest variance, which is conceptually similar to finding cluster centroids.

The K-SVD algorithm can be summarized as follows:

Initialization:

- Choose the number of basis set items.

- Initialize basis items from basis functions, randomly, PCA on training samples, other.

Iteratively learn and refine sparse codebook:

- Recompute basis items (sparse coding):

- Add new sample using basis pursuit to approximate value.
- Recompute all basis items.

- Recompute dictionary (dictionary optimization):

- Prune basis items that are not used often.
- Remove basis items that are too similar (mutually coherent).
- Replace seldom used codewords with underrepresented codewords.

K-SVD reconstructs each new target feature as an *approximation* composed of a linear combination of a few basis features from the codebook. K-SVD adds the new approximation to the dictionary, updates each dictionary item, and finally updates the entire dictionary to prune and remove mutually coherent (i.e., similar) items. For example, Fourier Series approximation is a similar method for reconstructing a target feature from the basis features, see Fig. 2.15.

K-SVD can be used with any *matching pursuit* method to approximate new sample features from the sparse codebook features. A matching pursuit method composes a linear combination of sparse codes to *approximately reconstruct* a sample feature, similar to a series reconstruction from the familiar Fourier Series sine and cosine basis waves (see Figs. 2.14 and 2.15). Depending on the feature descriptor representation details, different matching pursuit algorithms are chosen. Matching pursuit methods have been pioneered in signal processing by Pati et al. in 1993 [778] for wavelet signal dictionary composition, which Pati calls *Orthogonal Matching Pursuit*, i.e., looking for the closest features *orthogonally* across the basis set. Several matching pursuit approaches are surveyed by Aharon [779], including the more common *Orthogonal Matching Pursuit* (OMP) and *Basis Pursuit* (BP). More sophisticated matching pursuit methods are also used, which evaluate several possible reconstructions using multiple-path codeword combinations [779, 800] and batch methods [811].

An over-simplified algorithm for a matching pursuit could be devised as follows:

```
// the sample feature to match
input: sampleFeature
// record the weighted sum of best matching codewords
output: reconstruction = 0
// keep a running total of the match residual range [0-1]
Residual = 1
// find the n closest matching codewords
Iterate n times:
    closestCodeword = findClosestCodeword(sampleFeature * residual)
    //reconstruct the best fit of n weighted codewords
```

```

Reconstruction += residual * closestCodeword
// the difference between the sampleFeature and the best match
residual = residual - closestCodeword

```

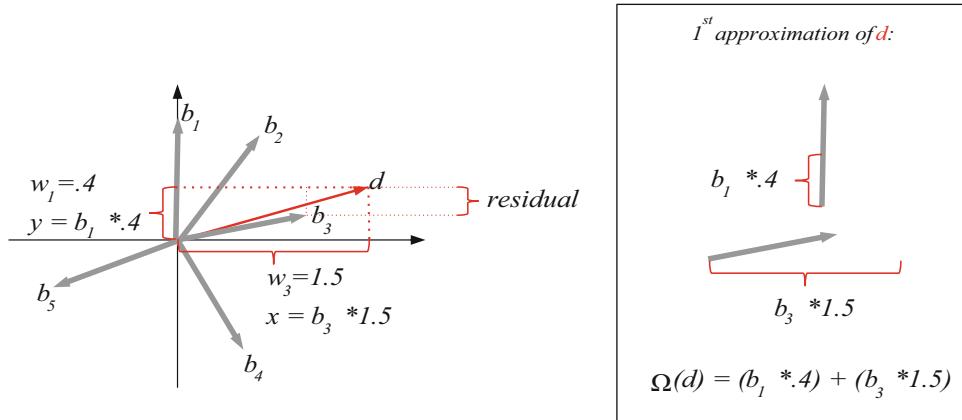


Figure 10.74 This figure illustrates a simplified reconstruction of feature d by projection of vector d into the basis vector space to determine the two best matching basis vectors. Note the weights w_1 and w_3 are determined by projecting vector d onto closest matching basis vectors b_1 and b_3

As illustrated in the algorithm, and in Fig. 10.74, a *basis pursuit* approach can be used to approximate a target feature d using a vector *projection* of the target against all basis codewords, to find the match with the greatest magnitude. The magnitude is the strength of the match, and is saved as the weighting coefficient. The *residual* is recorded by subtracting the weighting coefficient from the sample feature. The basis pursuit repeats recursively n times to find the best n sparse codewords to represent the remaining n residuals. The final approximation is reconstructed from the weighted sum of the n best matching vectors. A stopping criteria can be devised, such as a reconstruction error threshold, or a fixed number of iterations.

Here are the simplified basis pursuit and reconstruction details from Fig. 10.74.

(Sparse Codebook basis vectors):

$$\mathbb{R}_c = \{b_0, \dots, b_n\}$$

(First approximation of feature d from weighted sum of codebook features):

$$\hat{x}(d) = \forall \mathbb{R}_c : \sum_{i=0}^n w_i b_i(d)$$

(Residual to minimize):

$$r = d - \hat{x}$$

(Second approximation of residual r from weighted sum of codebook features):

$$\hat{y}(r) = \forall \mathbb{R}_c : \sum_{i=0}^n w_i b_i(r)$$

(Reconstruction using two basis vectors *NOTE: more residuals may be computed and combined):

$$\Omega(d) = \hat{x} + \hat{y}$$

where:

x —the signal to approximate

w_i —weight determined via projection of feature $[d,r]$ onto closest matching basis atom, see Fig. 10.74

The sparse coding feature space deserves special consideration, especially the distribution of features in the space, and the distance between features. Measuring similarity between features in a feature space is typically a multidimensional problem, and methods vary among matching pursuit methods. For example, we discuss *Kernel Methods* in the next section, which are sometimes used to project features into a higher dimensional feature space in a matrix representation to enable efficient similarity solutions, and could also be used within a basis pursuit algorithm. Some of the considerations for creating a good sparse coded feature space include ensuring *a uniform distribution of features in the feature space* with no distance distortions around common features, which leads to distance distortions around uncommon but necessary features which may be otherwise quantized out via the encoding scheme. We discuss sparse code feature space uniformity compensations and reshaping later in the HMP method survey.

To verify that the sparse codebook is sufficient, one method is to reconstruct images from the sparse codebook by sampling the image patches, matching each sample image patch into the sparse codebook to find the best matching sparse code combinations, and then reconstructing the image from the sparse code combinations. A *sufficient feature set* should allow for decent image reconstruction to the expected level of detail, as illustrated with several examples in Chap. 4 based on local feature descriptors such as SIFT, HOG, and FREAK. See Fig. 10.75 illustrating how the original images patches are reconstructed, one by one, from combinations of either two or five codebook feature combinations in the HMP method. Thus, the codebook is trained to sparsely represent the feature vocabulary for the application domain, and is used to reconstruct pattern matches from combinations of sparse codebook entries.



Figure 10.75 This figure illustrates image reconstruction from sparse codes, each codeword is a 5×5 image patch. (Left) original, (center) reconstruction using two codewords and looking blocky, (right) reconstruction using five codewords and working well. Image from Bo et al. [234] from ISER Springer Tracts in Advanced Robotics, © Springer used by permission

For more details on sparse coding for feature learning, see Grimes [535], Grosse [536], and Bergstra [537]. Signal processing literature and video compression literature are also good sources. See Wang et al. [783] for an example of locality-based sparse encoding, as well as [618, 624]. See Candes et al. [795] for more on sparse coding optimizations. See also Bourdeau et al. [699] for more on preserving 2D locality or position, as well as preserving feature space locality to improve results on smaller dictionaries.

Kernel Functions, Kernel Machines, SVM

A vocabulary or codebook of features may provide invariance, robustness, and the feature description and extraction compute efficiency, yet be difficult to *untangle* in the feature space to perform classification. To allow for optimal feature classification, *kernel methods* are often used to *prepare* features for classification of vocabularies using a range of machine learning methods. The term kernel has various meanings within mathematics. For example, in the literature regarding data mining [698, 789], the term *kernel* is used in several different contexts, for example within the domain of *kernel regression* methods, which treat the kernel function like a bump-map or windowing function along the regression line, to weight the local data points within the window prior to distance measurements. In statistical classification discussions on *Kernel Machines* and *Support Vector Machines* (SVMs), *kernel functions* are used to map feature vectors into a *kernel matrix* (*kernel*) to perform dot-product similarity measures between kernels in a higher dimensional space. Computing similarity between kernel matrix pairs is much faster than first converting the features to a *higher dimensional space* prior to computing similarity, and also allows for feature aggregations for multivariate descriptors, as illustrated below.

The kernels operate in a *Hilbert space*, analogous to a multidimensional Euclidean space, where familiar vector operations can be used to search for linear relationships, using dot products to compute angles and distances. A kernel matrix is also referred to as a *Gram matrix*, which is symmetric and positive semi definite, solved via the dot-product between all points in the kernel matrix pairs to completely define the coordinates in the multidimensional Hilbert space. Many application-specific *kernel functions* have been devised to create kernel matrices, and methods exist to learn a kernel matrix referred to as Multiple Kernel Learning (MKL), see Lanckriet et al. [793]. We provide a high-level overview here, and refer the reader to better references in machine learning and statistical analysis texts as we go.

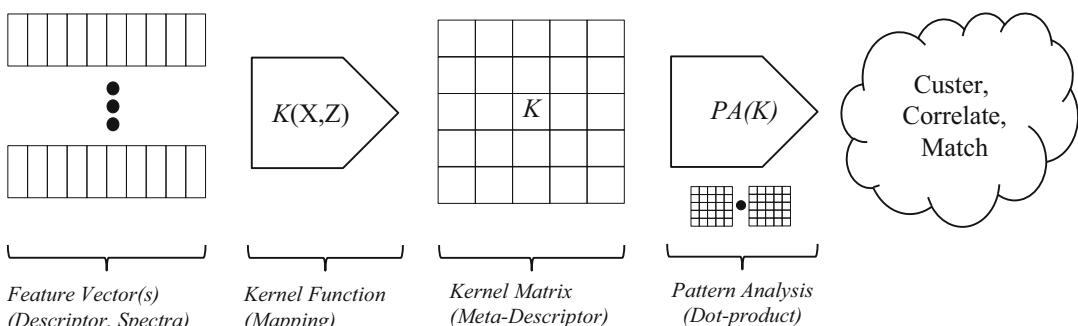


Figure 10.76 This figure illustrates how a kernel function is used to map feature vector data or raw pixel patch spectra into a kernel matrix (i.e., meta-descriptor) in a different feature space for optimal pattern analysis and classification via simple dot-products between kernel matrices, see also Christiani and Shawe-Taylor [586, 785]

Kernel methods are used to perform either or both:

- Aggregation of one or more features into a new *meta-feature representation*
- Projection of features or *spectra* into another representational *space* for classification (Figs. 10.77 and 10.78)

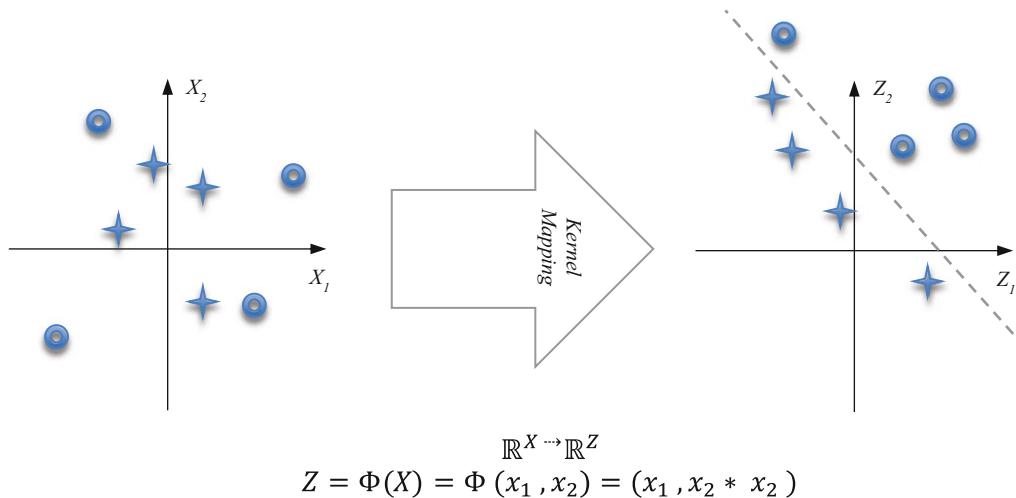


Figure 10.77 This figure illustrates how a kernel function maps linearly unseparable data in space X (left) into linearly separable data in space Z (right), after Boswell [775]

The need for kernel methods arises from the nonlinear distribution of data in some applications, which complicates analysis. As shown in Fig. 10.78, data that is not linearly separable *becomes separable* using the right kernels. So a kernel function is like a mapping function, or a projection function, to move features into another feature space. Thus, the kernel matrix is a powerful method for *unifying*, *normalizing*, and *combining* heterogeneous feature descriptor or spectra combinations into a common feature space for pairwise similarity computations for classification. We will survey examples in this section showing how features descriptors, such as SIFT features and RGB spectra pixel patches, are converted into kernel matrices for classification.

Kernel methods (see Hoffmann [786]) are used with *Kernel Machines*, such as *Support Vector Machines* (SVMs). Kernel Machines allow for various kernels to be substituted and tried during training to find the optimal kernels. When kernels are used with SVMs, the kernel projects the data from a space of dimension n into a higher dimensional space $n + 1$ (i.e., *the VC-dimension*, $n + 1$). As shown in Fig. 10.78, in a 1D space, the maximum number of points that are guaranteed to be separable is 2. In a 2D space, the maximum number of points guaranteed to be separable is 3 (unless the points are colinear). In a 3D space, the nonseparable 2D points may become separable by hyperplanes. The goal of kernel function design is to find the best kernel to map the data into some higher dimensional space for optimal analysis, which involves a combination of *stretching and compressing the feature space* for optimal clustering and separation, so many kernels are used in practice.

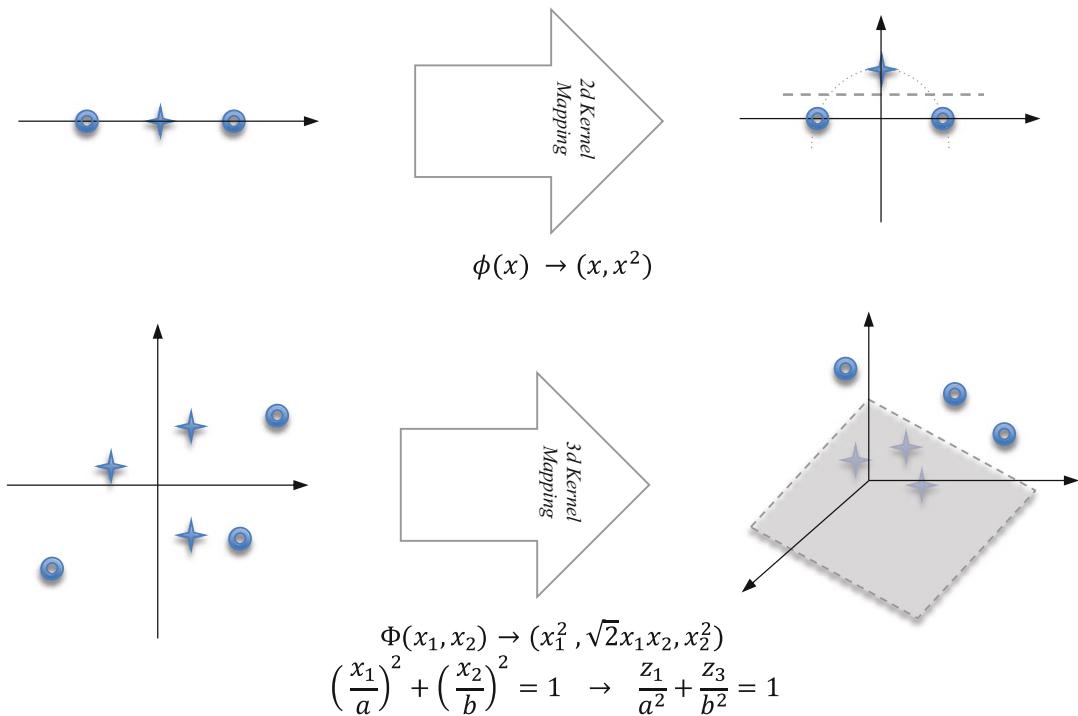


Figure 10.78 This figure illustrates how kernel mappings can yield linearly separable data in the VC-dimension (i.e., a higher dimensional space), (top) kernel mapping of 1D data into 2D space, and (bottom) kernel mapping of 2D data into 3D space, separable with a *hyperplane*

Instead of using coordinates of features to compute distance for pattern matching, kernel methods compute the distance between two *kernel matrices* via the dot-product of kernel matrix pairs, which projects one feature onto the other to reveal similarity. The kernel matrix acts as a *meta-descriptor*. The kernel functions which create the kernel matrices are problem-specific, and details on the derivation of specific kernel functions is beyond the scope of this work, and is covered in various texts, see Vapnik [597] and also Shawe-Taylor and Nello Cristianini [586, 785]. Kernel solutions return a single value, the *dot product* of one kernel matrix with another kernel matrix. A range of standard statistical analysis methods are performed using kernel methods, such as pair-wise distance analysis, principal component analysis, and cluster analysis. Complex and irregular data such as character strings and data structures like trees must first be converted to a suitable matrix form in order for kernel methods to apply. For example, structured-data can be run through a hash-like function to produce a matrix representation of the data for kernel methods, see also Hausler [777] for more references and details on his method of generating kernels to represent abstract structures. Gaussian kernels, RBF kernels, and simple polynomial kernels are commonly employed in most cases, although a wide range of kernels are employed for different applications. Finding the right kernel for the data is also important to make kernel methods work well, as covered in the references. The NIPS⁷ community contains a large body of research papers and other information on the subject of kernel machines, SVMs and kernel functions.

⁷ <http://www.nips.cc> - Neural Information Processing Systems.

Kernel matrices can also make classification much faster, acting as a *shortcut* to avoid direct computations in the higher dimensional space to compute distance. As shown in the simplified example below, the kernel function uses a simple *dot product* between two *kernel matrices*, rather than taking a complete algebraic solution to first move the kernel matrix features into the higher dimension prior to computing the distances (often referred to as the *kernel trick* <vulgar> in the literature). Note that for large feature vectors and large feature sets, the algebraic solution is computationally prohibitive compared to the dot product.

A verbose example using a hypothetical kernel function $K(x, y) = (\langle x, y \rangle)^2$ is compared to the equivalent algebraic solution below, illustrating the computational differences.

Slow Algebraic Method (full projection into higher dimension space):

$$\begin{aligned}\phi(x, y) &= \langle f(x), f(y) \rangle \\ f(n) &= (x_1x_1, x_1x_2, x_1x_3, x_2x_1, x_2x_2, x_2x_3, x_3x_1, x_3x_2, x_3x_3) \\ x &= (1, 3, 5) \\ y &= (2, 1, 3) \\ f(x) &= (1, 3, 5, 3, 9, 15, 5, 15, 25) \\ f(y) &= (4, 2, 6, 2, 1, 3, 6, 3, 9) \\ \langle f(x) | f(y) \rangle &= (4 + 6 + 30 + 6 + 9 + 45 + 30 + 45 + 225) = 400\end{aligned}$$

Faster Kernel Method (no projection into higher dimensional space, use dot product instead):

$$\begin{aligned}K(x, y) &= (\langle x, y \rangle)^2 \\ K(x, y) &= ([x_1, x_2, x_3] \cdot [y_1, y_2, y_3])^2 \\ K(x, y) &= ([1, 3, 5] \cdot [2, 1, 3])^2 \\ K(x, y) &= (2 + 3 + 15)^2 = 20^2 = 400\end{aligned}$$

Kernel functions *map* the feature data or spectra into another feature space of higher dimension, where the features can be disentangled into *linearly separable* clusters. Recall from Fig. 5.1 that we define *spectra* as any representation of data derived from pixels, such as a basis set, pixel patch intensity values, RGB colors, depth information, local region histograms, or LBPs. *Kernel functions* create a new feature representation (i.e., Meta-Descriptor) from combinations of various data (i.e., spectra and features), by *projecting the data* into a *kernel matrix* representing the data in a vector space. A range of problem-specific kernel functions have been designed. For a survey kernel method applications, see Mueller et al. [772], Cho et al. [769], Lampert [767], Zhang et al. [765], and Vedaldi et al. [770]. A good tutorial on SVMs is provided by Teknomo [788]. SVMs are also applied in DNNs. In some CNNs no FC layers are used, and an SVM is used instead for classification. In other CNN systems, the FC layers are replaced by an SVM after training, and fine-tuned from the learned CNN features.

Vocabularies or codebooks may be represented as collections or hierarchies of kernel matrices (i.e., *meta-descriptors*) built from other feature descriptors and spectra. As demonstrated by Bo et al. in the Hierarchical Kernel Descriptor (HKD) and Hierarchical Matching Pursuit (HMP) methods [108, 131, 233, 684, 659], RGB-D spectra and LBP feature descriptors are used as input into kernel functions used to create the kernel matrices or meta-descriptors, which we survey later in this section. A novel application of kernel methods using optimized match kernels is provided by Bo et al. [763], which we survey later as well. Multiple descriptors may be *aggregated* and encoded

together into a single kernel matrix, for example see Tolias et al. [840] regarding their *Selective Match Kernel* (SMK) and *Aggregated SMK* (ASMK), compared to similar methods such as *Vector And Locally Applied Descriptors* (VLAD), and *Hamming Encoding* (HE). An algebraic method for aggregating set kernels is also found in Shashua and Hazan [791]. See Gehler and Nowozin [792] for more details about multivariate descriptors composed using kernel methods. For a good survey of kernel methods with several references see Lampert [767] and Zhang et al. [765].

Other Statistical Classification Methods, Decision Trees, Forests, Boosting

We have only briefly surveyed three classification methods: (1) CNN FC layers, (2) Kernel Methods and SVMs, and (3) Vocabulary And Sparse Coding, and have ignored the vast majority of other interesting classification models. The author believes that the topic of statistical classification is far larger than the field of computer vision and feature learning, which is the primary reason why the focus of this work is the pixel side of computer vision and especially feature descriptors, rather than the mathematical and statistical classification methods that are borrowed and applied.

A few other notable classification approaches we do not survey include *Tree* and *Forest-Based* classifiers such as FERNS [298], which organize the features into a hierarchy of feature similarity. Also the Viola-Jones method [138, 178] for feature learning is noteworthy, combining multiple features into a hierarchy or funnel of features, trained and optimized by *boosting weak feature* [363], see also Chaps. 4 and 6 for more on Viola Jones.

For more information on classification methods applied in machine learning, see the standard texts by Hartigan [100] and Hastie [347]. See also the NIPS community resources.

Next, we will begin a survey of representative BFNs to illustrate all the background concepts we have covered including various feature models and classifier models.

PNN—Polynomial Neural Network, GMDH

We begin the BFN survey with perhaps the world’s first DNN, the *Group Method Of Data Handling* (GMDH), otherwise known as a *Polynomial Neural Network* (PNN), developed by Ivakhenko and Lapa in 1965 [564–566]. The PNN uses tunable *polynomials* as the basis features, rather than convolutional filters as in the CNN architecture. A good overview of PNN, along with neural network implementation details, is provided by Zjavka [762]. PNN departs from the McCulloch and Pitts work (1943), so rather than defining neurons as binary two or three state equilibrium systems, GMDH defines neurons as complex, nonlinear functions. Note that neuroscience has not yet discovered a verifiable electrochemical model of a physical neural function; however, the PNN polynomials appear to offer flexibility to model a wide range of possibilities compared to simple weight templates as used in CNNs. Ivakhenko’s work appeared frequently in *Avtomatika* and other Soviet publications unknown outside the USSR. The GMDH model is inductive and self-organizing, and has continued to be popular [760] especially in Russia, and other parts of the world as well, with a significant community of researchers. GMDH implementations are found in several commercial software packages and systems world-wide. Most of the GMDH applications are statistical in nature including *general multidimensional mathematical modeling*, data mining and forecasting, but some computer vision and pattern recognition applications have been developed [760].

GMDH is inspiring and unique. The basic ideas embodied in GMDH sound too good to be true compared to ad hoc methods for designing CNNs and RNNs used by many practitioners. Here is a summary of a few key GMDH concepts:

- Creates an optimal mathematical model of the data
- Self-organizing network, learned inductively by sorting the data
- Polynomials used to describe features, instead of CNN style templates

- Number of neurons and layers determined automatically
- Automatic structuring of network model
- Automatic learning of inter-relationships and patterns in data

In 1971, Ivakhneko [567] refined the GMDH using eight layers to learn the optimal number of layers, optimal number of neural units per layer, and prune neural units as needed. For example, the GMDH neural activation functions used second order polynomials, and self-adjusting thresholds, and could take advantage of Kolmogorov-Gabor polynomials, providing more control than other simple activation functions used in later systems such as sigmoids. In fact, the original paper mentions over 20 algorithms (*similar to neural activation functions in today's parlance*) that had been proposed within GMDH. The Ivakhnenko system could learn and train features from a validation set, much like today, and each layer could be trained differently for the given data and application. In fact, the validation set for the GMDH was conditioned using a variant of *drop-out* which is parameterized to eliminate unwanted data samples which can lead to over-fitting during training.

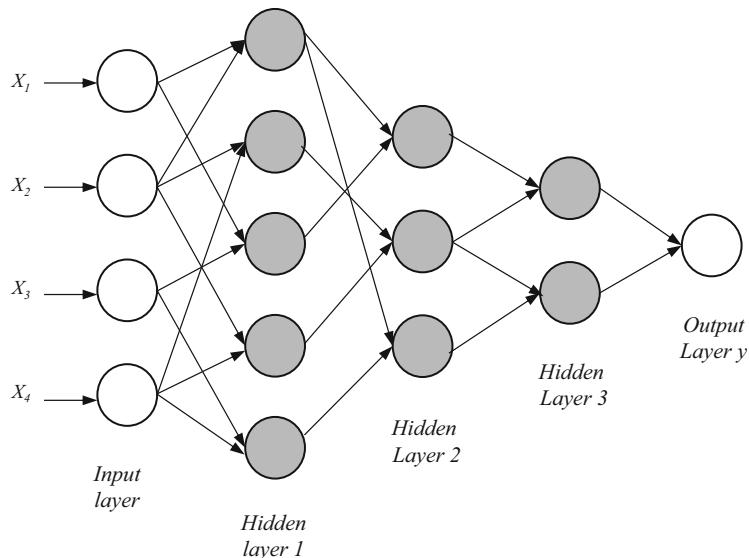


Figure 10.79 This figure illustrates a GMDH PNN, after Zjavka [762]. Note the computation of each PNN neuron takes two inputs, uses a polynomial neural function with six weights, and produces a single output

The GMDH polynomials are modeled after Kolmogorov-Gabor polynomials, a type of Gabor function. The PNN neural model takes two inputs, and produces a single output via a *quadratic function* of inputs using a total of six weights, combining the polynomials as a *multinomial* to produce the final output, as shown in Fig. 10.79. See Zjavka [762] for a good overview.

Since Ivakhnenko introduced dropout in 1971, several other researchers [569] have rediscovered dropout variants. Recently, [568] another drop-out method was introduced, *random drop-out*, to drop random training samples by setting them to zero to prevent over-fitting. Apparently, nobody yet has compared random dropout to Ivakhnenko's work, so perhaps Ivakhnenko's work will be revisited. More work has continued on GMDH in Russia and the Ukraine, see the detailed website [760] summarizing historical and continuing GMDH research [761, 762].

HKD—Kernel Descriptor Learning

Kernel descriptor methods learn features by converting feature vectors into kernel matrices, such as simple pixel patches, local region gradients, color patches, LBPs, Z depth information, and other feature descriptors. The kernel matrix is the *kernel descriptor*, or *meta-feature*, suitable for use in kernel machines. Sometimes in the literature a kernel descriptor is referred to as a *match kernel*. Kernel descriptors can represent vocabulary feature vector histograms, and turn such histograms into kernel matrices for use in a kernel machine. Kernel methods may be considered to be more mathematically sound and common in statistical analysis, compared to DNN methods that rely on ad hoc models of neurons trained in artificial connection topologies.

For background on kernel methods mentioned in this HKD survey, review the section above on Kernel Functions, Kernel Machines, SVMs.

In this section we survey a few architectures using *kernel-based* feature learning to produce *meta-descriptors* from spectra such as RGB-D patches and LBP features in a hierarchical architecture.

The Hierarchical Kernel Descriptor method (HKD) developed by Bo et al. [759] learns kernel descriptors from various pixel patch spectra such as gradient color, depth, and LBPs. HKD is based on the earlier work of Bo et al. on Kernel Descriptors [774] and extends the descriptors into a hierarchy by computing *kernel descriptors over kernel descriptors* in a layered hierarchy. Thus, HKD extends the basic kernel descriptor method *to take input from the output of other kernel descriptors in a hierarchy*, combining features from local receptive fields into higher level features, rather than taking input from pixel patches or other features. See Fig. 10.80.

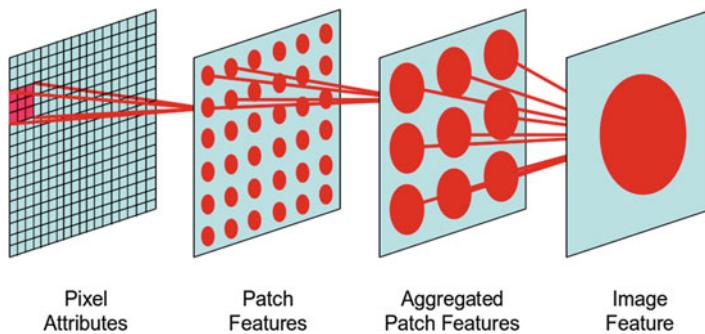


Figure 10.80 This figure illustrates the HKD method of computing kernel descriptors recursively over kernel descriptors into a feature hierarchy, image from Bo et al. [759] in CVPR, © Springer, used by permission

The goal of HKD is to provide a principled, uniform method for learning kernel descriptors from pixel spectra or other feature descriptors, inspired by the earlier work on efficient match kernels (EMK) developed by Bo et al. [763] which produces kernel descriptors from SIFT features and Fourier spectra. However, HKD learns kernels from raw pixel patches and other spectra from rectangular patches including:

- KDES-G: Gradient Match Kernel, composed of a normalized gradient histogram weighted using gradient magnitudes, with contributions from the gradient orientations and a Gaussian pixel position kernel within the patch. The KDES-G *learns* a kernel representation of a HOG or SIFT style features.

- KDES-C: Color Match Kernel, composed of individual color channel components such as RGB, combined with a Gaussian pixel position kernel within the patch.
- KDES-S: Shape Match Kernel, using the LBP to represent local patch shape information, with a Gaussian pixel position kernel within the patch.
- KDES-D: Depth Match Kernel, composed of Z depth channel scalars, combined with a Gaussian pixel position kernel within the patch.

The HKD gradient kernel descriptors learn a *representational view* of gradient orientation histograms from of a pixel patch, similar to SIFT and HOG descriptors, but HKD gradient descriptors are claimed to be slightly more accurate. The HKD gradient kernel descriptor includes three components: (1) a pixel-by-pixel attribute comparison for gradient magnitude, (2) orientation, and (3) a Gaussian weighted pixel position comparator.

HKDs address the computational problems associated with large feature matrices. If a kernel matrix is large, and the feature set is large, the compute cost of kernel methods grows to be prohibitive. To address the compute cost of larger kernel matrices, HKD first maps the features into a lower dimensional space (*reduction of feature set*), and then shrinks the size of each feature (*compaction of each feature*) via a convex quadratic approximation method.

In HKD, the kernel descriptors are learned from the training data from *pixel patches*, for example patches of size 16×16 taken across a dense 8×8 sampling grid. Next the patches are reduced into a sparse set (*lower dimensional set*). For example, if 1,000,000 patches are collected across the sampling grid, the vocabulary is reduced down to 1000 words using K-MEANS clustering. Next the final kernel descriptors are computed from the reduced patch set into the dictionary vocabulary. The kernel descriptors are individually *compacted* in size to reduce the compute workload, since larger feature vectors require more compute, which is especially apparent as the feature set size grows. The compaction is performed using a modified KPCA (Kernel-PCA) operating on joint basis vectors in the feature set. The goal is to approximate the kernels over a finite dimension, to make the kernel descriptors smaller, and reduce redundancy in the feature set. Bo finds that patch sizes of 16×16 are sufficient to approximate the basis vectors.

Various patch sizes and spectra can be used together. For example, intensity channels represent gradient information, RGB color channels to represent color appearance, depth information from a depth camera for z spatial information, and LBPs to represent local x,y spatial relationships or *local intensity shape*. The HKD research confirms some interesting findings noted by other researchers, namely that using a variety of patch sizes together, rather than only a single patch size, increases accuracy slightly. For example, patch sizes of 8×8 , 16×16 , 25×25 , and 31×31 were tried to confirm that multi-size patches increases accuracy. Another finding is that accuracy is improved by using *multivariate feature descriptor kernels*, concatenating different types of feature vectors such as gradients and color together prior to mapping the descriptors into kernel matrices. See Gehler and Nowozin [792] and Vedaldi et al. [887] for more details about multivariate descriptors.

Another interesting architecture using kernel descriptors is developed by Mairal et al. [794] as a *Convolutional Kernel Network* (CKN), incorporating kernel descriptors in a CNN framework, instead of using convolutions and associated functions to model the artificial neuron. CKN is claimed to be a generalization of HKD methods, with additional kernel variations and optimizations. A useful overview and comparison of HMP and HKD is found in Reubold [784] who analyzes the details and trade-offs between each method.

In summary, we note that HKD is yet another example illustrating the point that feature hierarchies and the sheer number of features supported in an architecture seem to be the key to best results, rather than attributing success to any specific feature descriptor, learned or otherwise crafted. And a

corollary observation is that strong local feature descriptors such as SIFT, FREAK, and ORB can be rivaled and sometimes surpassed by large sets of individually weak features in a deep and wide hierarchy, such as HKD and CNNs.

HMP—Sparse Feature Learning

In this section we illustrate a sparse feature learning architecture via a survey of the Hierarchical Matching Pursuit (HMP) method by Bo et al. [132]. HMP learns and encodes a multilevel feature hierarchy as a sparse dictionary of *pixel-patch* features in an unsupervised framework from unlabeled data. HMP has been extended by Bo [234] to incorporate data from RGB-D color channels, depth maps (D), and surface normal vectors (N). More HMP enhancements were made in the multi-path extensions MP-HMP [109] to use three or more layers of features in the hierarchy with multiple sized feature patches.

For background on sparse coding, codebook learning, K-SVD, K-MEANS, and matching pursuit methods mentioned in this HMP survey, review the Sparse Coding And Codebook Learning Overview section above.

We provide observations in this survey on a few of the key innovations across HMP versions including:

- PSC—Pyramid Sparse Code Features [234] encode sparse-coded features from pixel patches using a novel method call *Spatial Max Pooling*.
- Multivariate RGB-D-N data [234] descriptors are introduced.
- MI-KSVD [132] uses a variation of K-SVD called MI-KSVD to learn a hierarchical sparse codebook with optimized distance between features.
- M-HMP [109] M-HMP uses multisized feature patches [109] encoded through multiple paths.

Note that Yu et al. [796] previously developed a method similar to HMP, using hierarchical sparse coding (HSC) which *jointly encodes* low-level feature codes in a local region from layer 1 into higher-level feature codes in layer 2, demonstrating that hierarchical sparse coding methods can provide *spatial encoding in the higher layers* to spatially associate local features together into higher-level sparse codes, which is lacking in single-layer sparse coding methods using orderless BoW feature vocabularies. HMP follows the same approach as HSC to encode higher-level sparse code features from lower level features using local region spatial dependence.

HMP Pyramid Sparse Code (PSC) Feature Descriptor

HMP’s *Pyramid Sparse Code* feature descriptor is novel, encoding of a set of 21 sparse codes concatenated together. In the M-HMP extensions to PSC, the positive and negative sparse codes are split out into separate features to allow for separate weighting and responses. PSC is therefore one of the most complex features in this survey.

As shown in Fig. 10.81, HMP encodes *features-in-features* using *Spatial Pyramid Max Pooling* to select the MAX sparse code from each region of the spatial pyramid. Each sparse code is computed from a 5×5 pixel region as shown in Fig. 10.81. Note that there are 21 regions defined in the spatial pyramid of Fig. 10.81 (right), and the MAX sparse codes from each region are concatenated together into the 21-element PSC descriptor. The resulting HMP PSC descriptor is

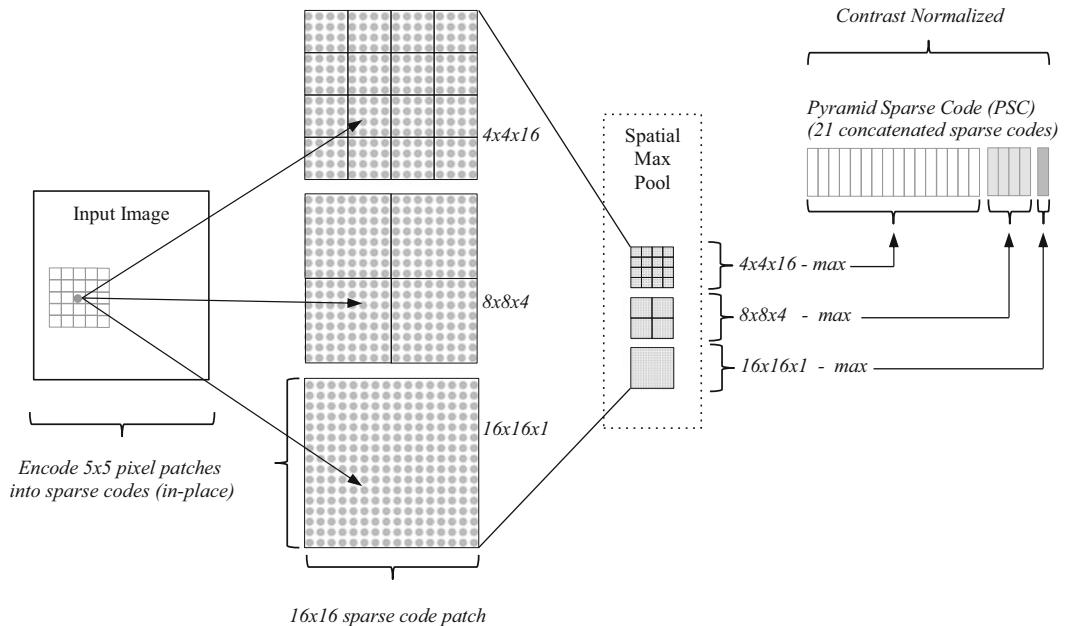


Figure 10.81 This figure illustrates the composition of Pyramid Sparse Code features composed of 21 sparse codes taken from the MAX sparse codes in a spatial pyramid of 21 regions of size 4×4 (16 codes), 8×8 (4 codes), and 16×16 (a code)

similar to Pyramid Match Kernel (PMK) developed by Grauman and Darrell [517] discussed in Chap. 6, except that the PMK descriptor is computed over the entire image, using simple *histogram features* of each subdivided image region, and the histograms for each region are concatenated together. However, the HMP pyramid sparse code feature is composed over a 16×16 patch of sparse codes where the MAX values are concatenated together to represent the scale pyramid in the 16×16 region.

The M-HMP extension uses the same 5×5 sparse codes and 16×16 patches for the first level, but extends the patch sizes and pooling region counts. Each layer in the feature hierarchy takes input from the lower level spatially max pooled and contrast normalized features. The second level uses 36×36 patches to learn mid-level features, and the third layer uses 36×36 patches to learn whole image features. Bo reports M-HMP a range of results using one, two and three layer networks, and variable feature counts ranging from 300–1000.

HMP Dictionary Learning with MI-KSVD

The first version of HMP [132] uses the K-SVD dictionary learning method developed by Aharon et al. [779] to encode features in the sparse codebook. K-SVD recursively optimizes and recomputes the entire codebook as each new feature is added, as explained earlier. However, Bo [109] later made key enhancements to K-SVD referred to as MI-KSVD (*Mutually Incoherent—KSVD*) [234] to create a balance between common and uncommon features by encoding basis features with a *more uniform relative distance* to include common and uncommon features, rather than clustering the codebook around the most common features. To illustrate the problem, imagine building the sparse codebook from only the most commonly observed pixel patches—this would yield a codebook overfit to the

most commonly observed patches, not likely covering the entire feature space. M-HMP balances mutual incoherence with reconstruction error to incorporate common and uncommonly observed patches to optimize the codebook for a more uniform feature space distribution. The MI-KSVD method is therefore novel.

To add features or find features in the sparse codebook, MI-KSVD uses an *orthogonal matching pursuit (OMP)* method, as discussed earlier, to compose a linear combination of sparse codes to *approximately reconstruct* pixel patches from combinations of codebook features, similar to a series reconstruction of a wave using the familiar Fourier Series (see Fig. 2.15). During codebook learning, the basic idea is to ideally find the single closest feature in the sparse codebook matching the sample feature, and if nothing close exists, add in a new feature. A matching pursuit will reconstruct a feature from multiple basis codewords if needed. HMP uses an optimization method called Batch-Tree OMP (BTOMP) which subdivides the basis set dictionary into smaller dictionaries (i.e., *batches*) using K-means clustering, so that the matching pursuit is computed more quickly in parallel over each batch, see Rubinstein [811].

We compare the K-SVD and MI-KSVD objectives here:

$$\begin{aligned} \min_{D,X} & D, X \quad D, X \|Y - DX\|_F^2 && \text{(KSVD objective)} \\ \min_{D,X} & D, X \quad D, X \|Y - DX\|_F^2 + \lambda \sum_{i=1}^M \sum_{j=1, j \neq i}^M |d_i^\top d_j| && \text{(MI-KSVD objective)} \end{aligned}$$

where:

$$\begin{aligned} D &= [d_k, \dots] \text{ Codebook} \\ X &= [x_k, \dots] \text{ Sparse codes} \\ Y &= [y_k, \dots] \text{ Pixel patch matrix observations} \\ \lambda &= \text{mutual coherence trade-off parameter} \end{aligned}$$

As shown in the equations, for each new patch sample matrix Y , the sparse code matrix X is computed from codebook D to approximately reconstruct Y using BT-OMP to find the set of closest matching codebook items. Next each item in the codebook is recomputed using the *MI-KSVD objective function*, parameterized by λ to balance reconstruction error against mutual incoherence (i.e., codebook entry similarity avoidance) resulting in a reshaped codebook. The codebook D is recomputed repeatedly in the same manner for all new samples Y .

Fig. 10.82 illustrates the basic method for building HMP layer 1 features.

As shown in Fig. 10.82 (1), the first layer sparse codebook is initialized to an overcomplete DCT basis set converted to sparse codes. The second level codebook is initialized from a reasonably random set of perhaps 1,000,000 5×5 image patches mean-zero normalized and converted to sparse codes. HMP builds sparse codes from randomly sampled 5×5 pixel patches, and then assembles the sparse codes into a sparse code image.

As shown in 10.82 (2) HMP uses *spatial pyramid pooling* over a 16×16 patch of sparse codes, using three levels of spatial resolution, rather than the simple subsampling pooling over a single spatial resolution region typically used in CNNs. The three pooling levels used in HMP are 1×1 (the current feature pixel patch), a 2×2 image division into four regions, and a 4×4 image division into 16 regions, and one 16×16 region, for a total of 21 regions in the spatial max pool concatenated into the Pyramid Sparse Code (PSC). The max feature from each region is selected and encoded. All features are normalized in the range 0–1.

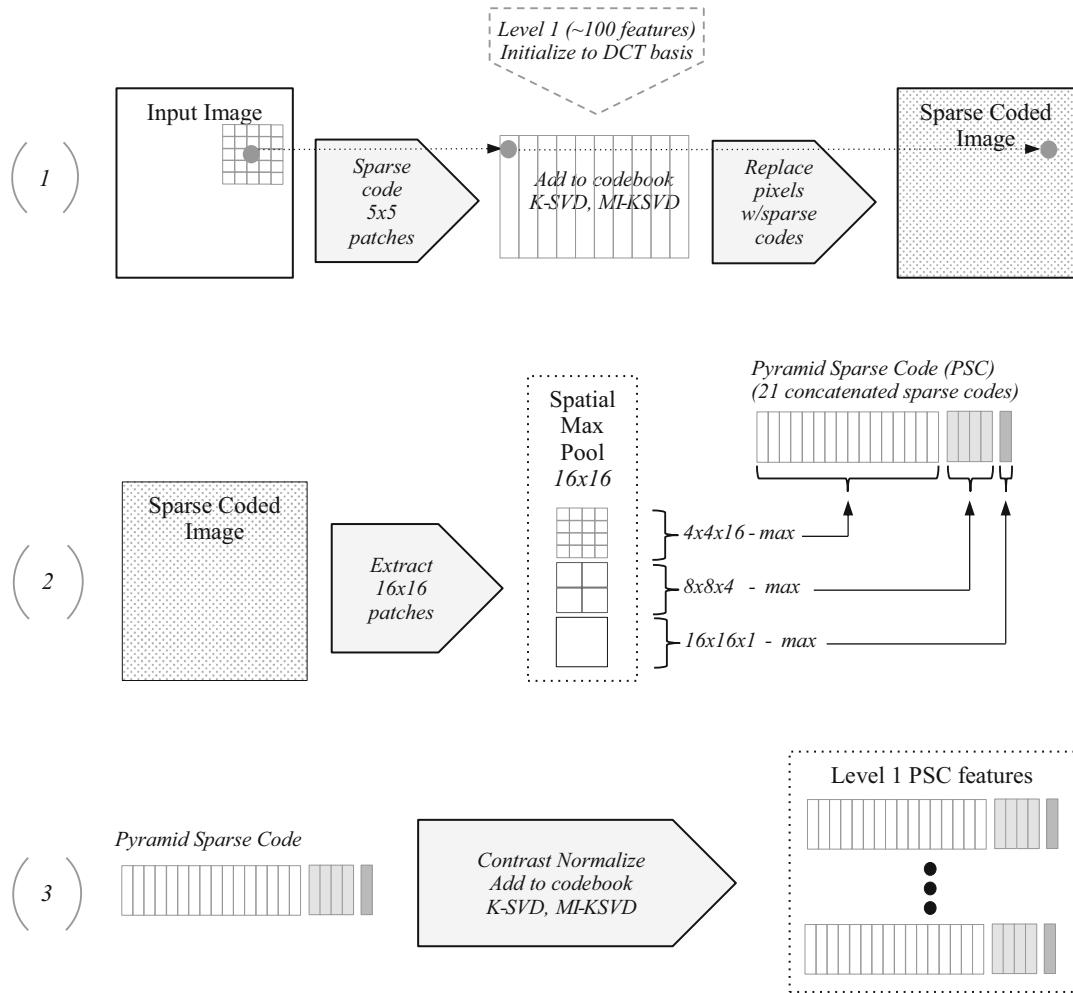


Figure 10.82 This figure illustrates the sparse coding for the first layer (1) encoding pixels into sparse codes over 5×5 regions, (2) combining sparse codes into the 21-element Pyramid Sparse Code feature vector using spatial pyramid max pooling, (3) adding each feature into the sparse codebook

Next in 10.82 (3), MI-KSVD is used to add each feature into the codebook. HMP is trained in a layer-wise fashion, starting from the lower layers. Steps (2) and (3) are repeated for each layer of the codebook. However, Bo also tries a method for *joint pooling over multiple layers* [234] with the RGB-D-N data, and reports improved results.

The HMP training protocol is novel and involves training the first layer to record sparse codes within 16×16 spatial pyramid pools, and then in intermediate layers (2 – n) refines the sparse codes by using 16×16 patches across the whole image. To train, 16×16 pixel patches are taken at a stride of 4 across the input training images. The 16×16 pixel patch is first encoded into 16×16 sparse code patch from the sparse codebook, using a 5×5 pixel local region around each pixel to compute the sparse codes. Each sparse code is then composed from the codebook using B-OMP. Next for each 16×16 region of sparse codes, a spatial pyramid subdivision is built consisting of a 2×2 subdivision and a 4×4 subdivision and the entire 16×16 region. The maximum sparse code from each region is selected, and concatenated together into a regional sparse code.

In one test, Bo freezes the first layer DCT basis and does not perform feature learning at all on the first layer. When the DCT basis is compared against the features learned via K-SVD, both the DCT basis and the K-SVD sparse coded basis perform within a few percentage points of accuracy, again illustrating that the feature descriptor itself is not as important as the sheer number of features in the feature hierarchy.

HMP Multivariate I-RGB-D-N Features

HMP is extended [234] to use features from a depth camera ([234] is one of the first methods of encoding depth information into feature descriptors), as well as color information, with four channels as input patches: (1) *Intensity*, (2) *RGB*, (3) *Depth camera Z pixels*, and (4) *Surface Normal vectors*. The method follows basically the same sparse coding and K-SVD methods as employed in HMP. However, the final set of four feature vectors can be associated together for a stronger feature via concatenation of the features from each channel, which forms an 188,300 dimensional feature descriptor. Bo concludes that this method of learning separate smaller features for each channel works better than learning a single larger feature learned from all channels combined into a single larger channel. See Fig. 10.83 for an illustration of the learned I-RGB-D-N features.

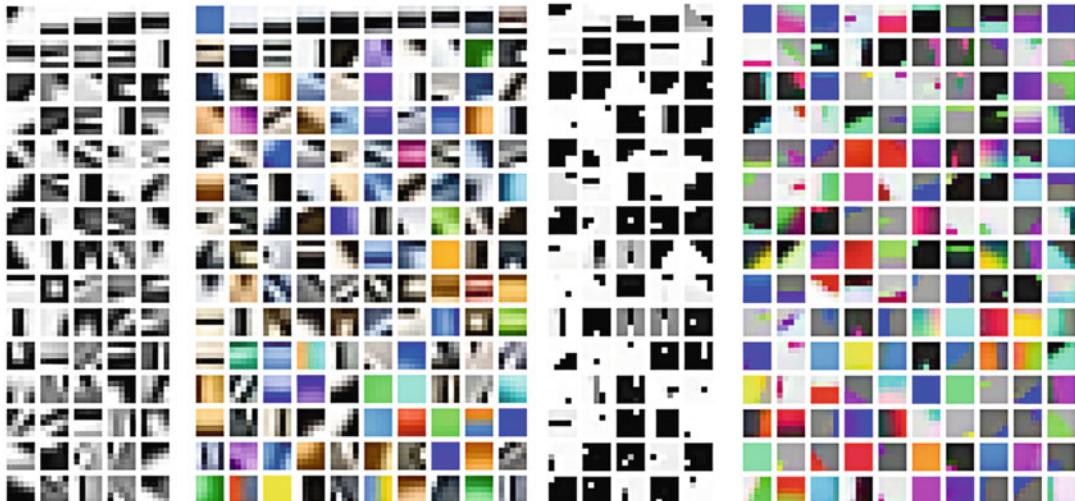


Figure 10.83 This figure illustrates the different sparse codebooks learned by HMP methods, including (*left to right*) RGB-I intensity, RGB, Depth channel, and 3D surface normal encoded using RGB. Image from Bo et al. [234] from ISER Springer Tracts in Advanced Robotics, © Springer used by permission

M-HMP Multiscale Features

The Multipath Hierarchical Matching Pursuit extends the basis HMP method across several patch sizes (16×16 and 32×32) and a three-level feature hierarchy, resulting in a deep network with unusually large features compared to many CNNs using 3×3 and 5×5 features. The Multipath matching pursuit extends the single-path encoding each patch across multiple paths to extract a wider and deeper range of features. The Multipath HMP method using several patch sizes creates a richer feature set, increasing accuracy over mono-sized patches as in the original HMP method.

We note that many local feature descriptors such as FREAK (31×31 or other large size) and SIFT (16×16) also use larger feature sizes as well, with good success.

In summary, HMP variants show that learning features from simple pixel patches can rival the performance of SIFT feature sets. Bo et al. find that two level feature hierarchies perform better than a single level, and two-level hierarchies perform about as well as three layer hierarchies. Multivariate I-RGB-D-N features used together are shown to be more effective than mono I-channel features. Multipath coding using multiple feature patch sizes is demonstrated to increase effectiveness.

HMAX and Neurological Models

We will survey the original HMAX work here as introduced by Riesenhuber and Poggio in 1999 [812], and also survey related neurovision models including a few subsequent variations from the basic HMAX architecture. The interested reader should consult the collection of historical papers and continuing research within the HMAX community at Riesenhuber’s MAXLab at Georgetown [837] which includes online source code resources and references. To dig deeper into neuroscience research, see the neuroscience journals listed in [Appendix C](#).

To understand HMAX, we provide some brief background on the visual pathway here, which influenced the HMAX model. HMAX is one of the *first models* of the entire visual pathway hierarchy based on neuroscience. The neuroscience community is quite active in developing models of the visual pathway, and their work does not often overlap with the computer vision community since the research goals are different. So this section serves as a brief introduction, via HMAX, to a neuroscience model for vision and feature learning which is different and perhaps more complex than the CNNs and RNNs surveyed earlier.

The Standard Model of the Visual Pathway

The foundations of HMAX lie in a so-called *standard model* of the visual pathway described by Ungerleider in 1994 [834], Riesenhuber [833], and others. HMAX is based on a *standard model* of the visual pathway as shown in Fig. 10.84. The standard model includes a hierarchy of receptive fields following the general Hubel and Wiesel model discussed earlier in this chapter. (See also the discussion on the visual pathway and Figs. 9.10 and 9.12). Many of the HMAX concepts are biologically inspired, some are not. HMAX is more widely used as a research tool in the neuroscience community, compared to CNNs being increasingly used in commercial systems and within the computer vision community. This author notes that neuroscience research is increasingly driving computer vision towards synthetic vision models like HMAX, rather than ad hoc computer vision models designed as trade-offs to solve real problems under primitive compute and memory constraints. As computing power increases, synthetic vision models are becoming more complex and realistic, often driven by the best neuroscience.

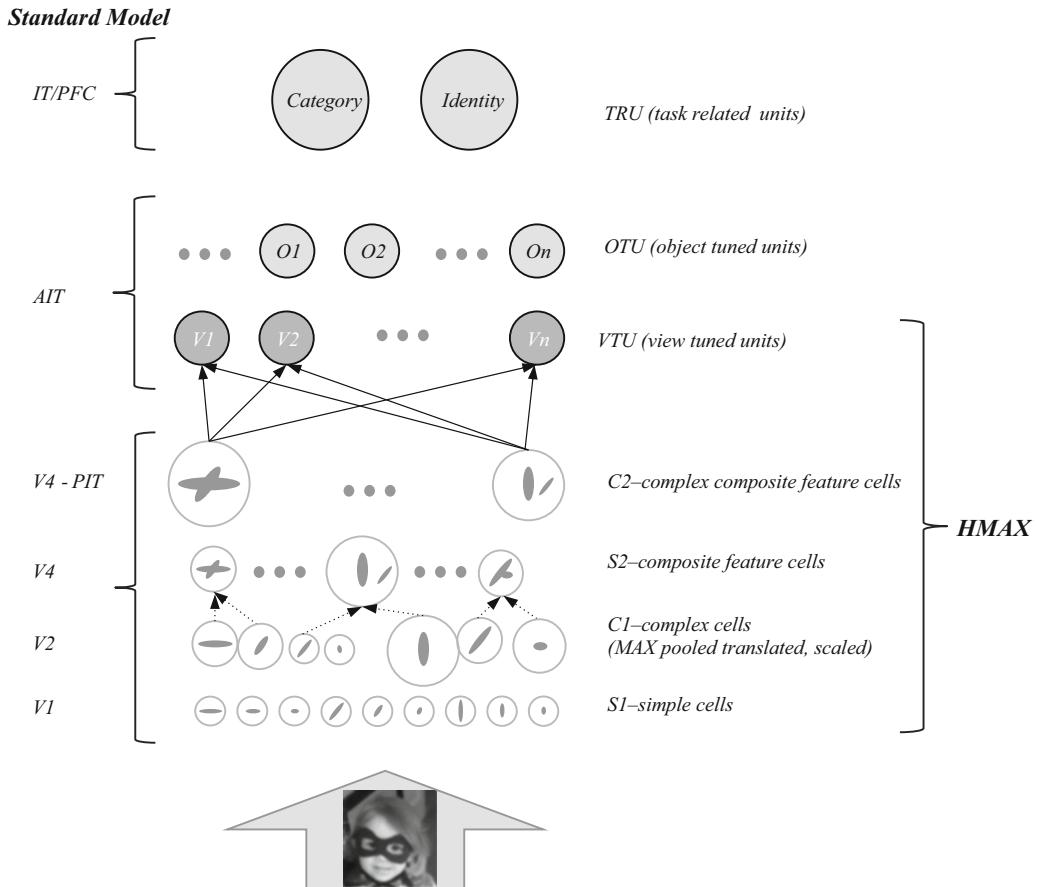


Figure 10.84 This figure illustrates the standard visual pathway and the HMAX model after Riesenhuber [833]

HMAX [812] models the low-level features as a small set of oriented *Gaussian Radial Basis Functions* (RBFs) which are similar in appearance to Gabor functions and oriented edge patterns. The *simple* Gaussian RBFs can be composed together into *complex* features, where combinations of scaled and translated features may overlap and compose together, see Fig. 10.85. HMAX is largely based on the experimental data of Logothetis et al. [813] who measured responses to shapes across the visual pathway in monkeys. Logothetis found that groups of neurons along the feed-forward hierarchy respond to specific shapes like edges at the low levels, and higher level concepts such as faces in higher levels, which is to be expected since each neuron in the ventral stream V1 V2 takes some input from lower-level local receptive fields, so lower-level neurons are taking input from small regions where edge and blob features predominate. Neurobiology research by Logothetis and others indicates that the visual pathway has *a huge hierarchical feature memory containing billions of features*, processed via extremely massive amounts of parallel and simultaneous processing, where some neurons are dedicated to a single low-level feature, and other neurons are dedicated to part of a larger feature, and other neurons are dedicated to making high level classification decisions to form and test hypothesis. The implications are that future synthetic vision models may be designed with the assumption that huge numbers of features are useful and advantageous, in contrast to models using smaller number of features and statistical classifiers which tend to be better at generalization, compressed, and suitable for implementation on modest computer systems.

Viewpoint Invariance Models

HMAX provides a *bridge model* that partially reconciles theories on viewpoint invariance in human vision. In other words, how does the human visual system identify the same object from different viewpoints? According to Tarr [814] there are two fundamental visual representation theories to account for viewpoint invariance:

1. *Hierarchical Parts Models (mostly viewpoint independent)*: hierarchies of parts, each of which are wholly or partially viewpoint invariant. Most viewpoint invariance is assumed to be recorded in the features hierarchy, likely at the higher levels. It is theorized that some interpolation between view-dependent features is performed in the visual pathway to reconcile viewpoints.
2. *Appearance Models (mostly viewpoint dependent)*: each viewpoint is represented by separate neural feature memories, perhaps with higher-level concepts sharing some features between viewpoints. Viewpoint representations are theorized to *record* new neural memory impressions from the original viewing, perhaps with new neural growth and interconnects forming based on the *novelty* of the impressions, and the *importance* of the impressions.

*HMAX synthesizes both models (1) and (2): HMAX uses a hierarchy of parts for the lower level features, and viewpoint-dependent models (view-tuned units or VTUs) for higher level concepts.

HMAX introduced the term *view-tuned cells* to represent higher level concepts in the *View Tuned Units* (VTUs), as shown in Figs. 10.84 and 10.86. Each view-tuned cell is based on a hierarchy of lower-level features. The higher-level *viewpoint-dependent* view-tuned cells are pattern recognizers composed of lower-level *viewpoint-independent* features. HMAX has proven to model scale and translation invariance well, with some mirroring invariance about the x or y axis also, leveraging the view-tuned cells and low-level viewpoint independent features.

HMAX Feature Hierarchy

HMAX composes an *overcomplete* set of lower level features into a representation of higher-level concepts, which mitigates feature invariance problems. As shown in Fig. 10.85, note that the lower-level features are similar to Gabor functions, which can model an object such as an edge segment as a collection of 3D rotations, equivalent to 3D rotation and scale invariance. We consider viewpoint invariance to consist primarily of affine transformations. The higher-level features are thus built on 3D invariant edge-like features, see also the SYMNETS survey earlier regarding the affine-invariant symmetry group of features.



Figure 10.85 This figure illustrates the apparent 3D rotational and scale *appearance* of a set of *multiscale* Gabor features. By building the same Gabor features at multiple scales, the set appears to contain a 3D set of scaled and rotated Gabor features, since scaled versions of a feature can be interpreted by the eye as a rotated version of the cell in 3D. Using multiscale low-level features allows HMAX to compose higher-level *view-tuned* features using combinations of lower-level apparently 3D invariant features

HMAX is based on neurobiology concepts and some hypothetical models as well. HMAX uses hard-wired feature for the lower levels such as Gabor or Gaussian functions, which resemble the oriented edge response of neurons observed in the early stages of the visual pathway as reported by Tanaka [825], Logothetis [827], and others. HMAX builds higher level concepts on the lower level

features, following research showing that higher levels of the visual pathway (such as IT) are receptive to highly view-specific patterns such as faces, shapes and complex objects, see Perrett [821, 822] and Tanaka [823]. In fact, clustered regions of the visual pathway IT region were shown by Tanaka [826] to respond to similar clusters of objects, suggesting that neurons grow and connect to create semantically associated view-specific feature representations as needed for increased discrimination. And Connectome research is also providing evidence that related feature concepts are stored in adjacent areas [852, 859]. HMAX provides a viewpoint-independent model that is invariant to scale and translation, leveraging a MAX pooling operator over scale and translation for all *inputs* feeding the higher-level S2, C2, and VTU units, resembling *lateral inhibition* which has been observed between competing neurons, allowing the strongest activation to shut down competing lower strength activations. HMAX also allows for sharing of low-level features and interpolations between them as they are combined into higher-level viewpoint-specific features.

Under HMAX assumptions, each neuron is more like a very simple memory cell with a *neural correlator*. Indeed, neurobiological research [816–818] provides ample evidence that specific groups of neurons act as independent memory cells, each one containing *different views of the same object*, rather than encoding viewpoint invariance in a single neuron. Under the view-dependent assumptions, smaller number of neurons are needed to remember the higher-level viewpoint dependent differences between groups of neurons [819–821] as noted in research on monkeys, where specific neurons in the anterior IT respond to view-specific features. Viewpoint dependence of specific neural structures seems likely, for example where it is necessary for features learned in the top-level IT region of the visual pathway to be very specific and even viewpoint-specific to recognize a human face such a family member.

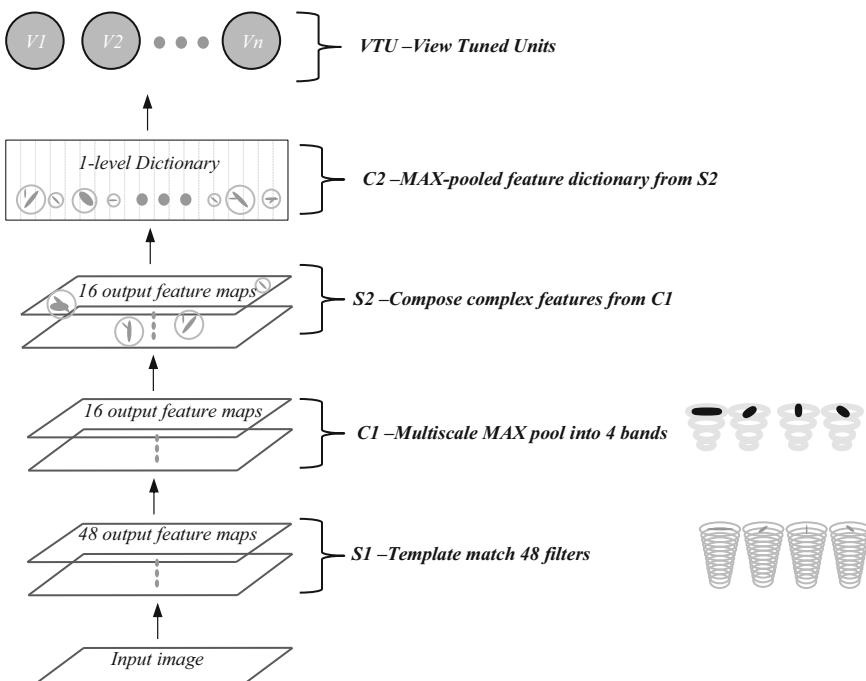


Figure 10.86 This figure illustrates the HMAX model of view-tuned features representing higher-level concepts, built on top of a hierarchy of view dependent lower-level features

The original HMAX concept is illustrated in Fig. 10.86 as a way to visualize the architecture of a real system. HMAX does not learn the bottom layer features in S1, C1, S2, and C2, but instead uses basis functions instead [543]. The low-level features in HMAX are hard coded or supplied using transfer learning, based on the findings from see Serre et al. [835] who implemented real HMAX systems, showing that learning features as is common in CNN models is no more effective than providing basis features or features derived from transfer learning which can adapt to the training set. The high-level View-Tuned Units (VTUs) are where the features are learned. The S1 and C1 units produce feature maps, and the S2 and C2 units produce dictionaries of unordered features (BoW model). More and more research shows that DNA may contain memory impressions or *genetic memory* such as instincts and character traits, analogous to basis functions rather than learned functions (see [824], many more references can be cited). Other research shows that DNA can be modified via *memory impressions and experiences* [825] which can be passed on to subsequent generations via the DNA. So the HMAX model using preexisting features at the low levels is neurobiologically plausible, with specialization and higher-level concept learning occurring at higher levels of the visual cortex.

The key layers in the HMAX model are as follows:

- S1: Simple cells compute oriented multiscale filter responses via template matching.
- C1: Complex C cells perform a multiscale MAX pooling operation.
- S2: Compose combinations of scaled and translated features into *prototype features*.
- C2: a 1D feature dictionary of MAX pooled S2 cells.
- VTU: the view-independent units (VTUs) composed of high-level features.

HMAX Layers

Here we survey Reisenhuber's original HMAX version [812] here following Reisenhuber's MAXLab open-source code [837], layer by layer, and point out more recent HMAX enhancements and innovations by other researchers as we go.

S1 Layer

The S1 layer is for *multiscale filtering* of the input image into a set of multi-scale output feature maps containing filter responses across the entire input image which are subsampled according to the size of the filter region.

As shown in Fig. 10.87, S1 composes four oriented Gaussian edge-like functions computed across 12 scales ($4 \times 12 = 48$) into weight template matrices for correlation. Each output feature map contains the filter response for a different scale of each feature, using a *retinal coordinate system* to preserve the spatial grid location of features after subsampling. The original HMAX versions use second derivative Gaussian filters. Mutch and Lowe [543, 828], Serre [544], and later versions of HMAX [829, 831] use Gabor Filters instead of Gaussians, and Hu [830] uses PCA learning to create convolutional filters from patches. Gabor filters are more commonly used in more recent versions of HMAX since the Gabor filters can be tuned more precisely than the Gaussians. Sharpee et al. tried curved Gabor Filters [653] and found that curved Gabor filters are still insufficient to describe the types of features detected.



Figure 10.87 This figure illustrates the S1 layer features, containing four oriented filter functions, and the 12 scales for each filter, yielding 48 filters. Each filter is rendered as a circular template weight matrix

Original HMAX second derivative of Gaussian 2D filter shape:

$$G_{x,y} = \frac{(-x \cos \theta + y \sin \theta)^2}{\sigma^2(\sigma^2 - 1)} \exp\left(\frac{(x \cos \theta + y \sin \theta)^2 + (-x \cos \theta + y \sin \theta)^2}{2\sigma^2}\right)$$

where orientation = θ and width = σ

Gabor filter shape (Mutch & Lowe, and Serre):

$$G(x,y) = \exp\left(\frac{X^2 + \gamma^2 Y^2}{2\sigma^2}\right) \cos\left(\frac{2\pi}{\lambda} X\right)$$

where:

$$X = x \cos \theta - y \sin \theta$$

$$y = x \sin \theta + y \cos \theta$$

x and y range [-5 ... 5]

θ range [0 ... π]

γ (aspect ratio), σ (effective width), λ (wavelength)

Pixel patch X response to Gabor filter:

$$R(X, G) = \left| \frac{\sum X_i G_i}{\sqrt{\sum X_i^2}} \right|$$

Each filter is replicated into four orientations of 0, 45, 90 and 135, and each orientation is replicated into 12 scaled versions ranging from 7×7 to 29×29 pixels in scale increments of 2: $[7 \times 7, 9 \times 9, \dots, 29 \times 29]$. So the total number of S1 features is $4 \times 12 = 48$. Later versions of HMAX use slightly different arrangements of filter sizes [829–831]. The S1 features are contained in a circular region, rather than a rectangular region, which increases rotational invariance and is more biologically plausible than a simple rectangle.

S1 produces filter responses via template matching, so the filters are rendered into weight matrices for template matching against pixel regions, similar to CNNs. Each filter is centered over each pixel in the input image for filtering, and all filter responses are collected into 48 output feature maps to feed into layer C1. Instead of creating scaled features ranging from $7 \times 7 \dots 29 \times 29$, Mutch and Lowe [543, 828] scale the input image instead and use a monoscale filter, see Fig. 10.88. Serre et al. [835] extend the filter scale range from 7×7 to 37×37 at spacing of 2 leading to 16 scales at four orientations for 64 feature types.

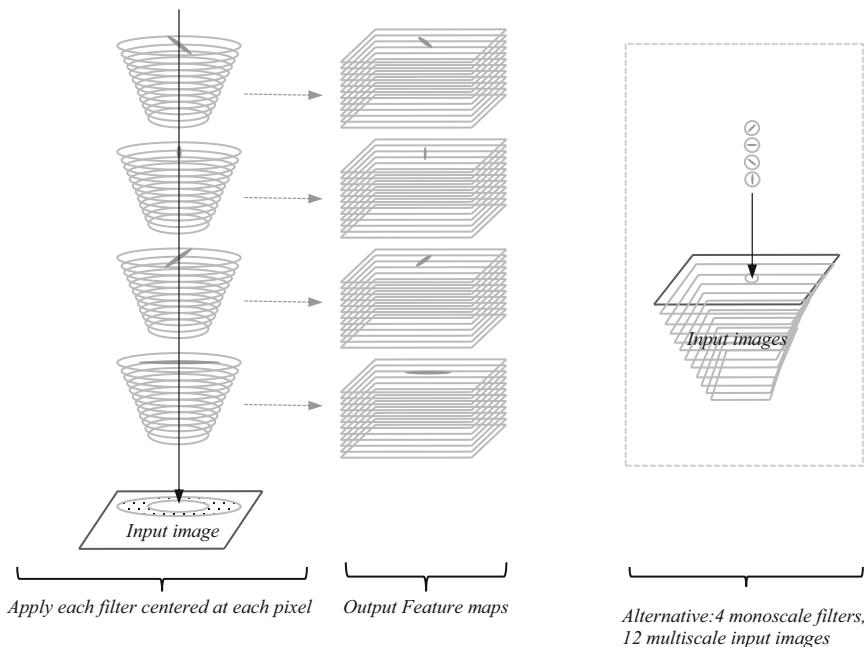


Figure 10.88 This figure illustrates the HMAX S1 layer. (Left, center) Each filter is applied centered at each pixel location in the input image, and filter responses collected as output feature maps for input to the C1 units. (Right) an alternative method using monoscale features applied to multiscale input images after Mutch and Lowe [543]

C1 Layer

The C1 layer performs *Multiscale MAX pooling of all filter responses at all orientations and scales* to creates V2 complex cells which are scale invariant within a small scale band, and position invariant within local regions. Each oriented filter set is pooled independently. As shown in Fig. 10.89, C1 pools and subsamples the 48 input feature maps from S1 into 16 output feature maps corresponding to *scale bands*. The scale bands contain filters of similar size, for example band4: $[7 \times 7, 9 \times 9]$, band 3: $[11 \times 11, 13 \times 13, 15 \times 15]$, band 2: $[17 \times 17, 19 \times 19, 21 \times 21]$, and band 1: $[23 \times 23, 25 \times 25, 27 \times 27, 29 \times 29]$. The idea of using bands is to increase scale invariance by pooling

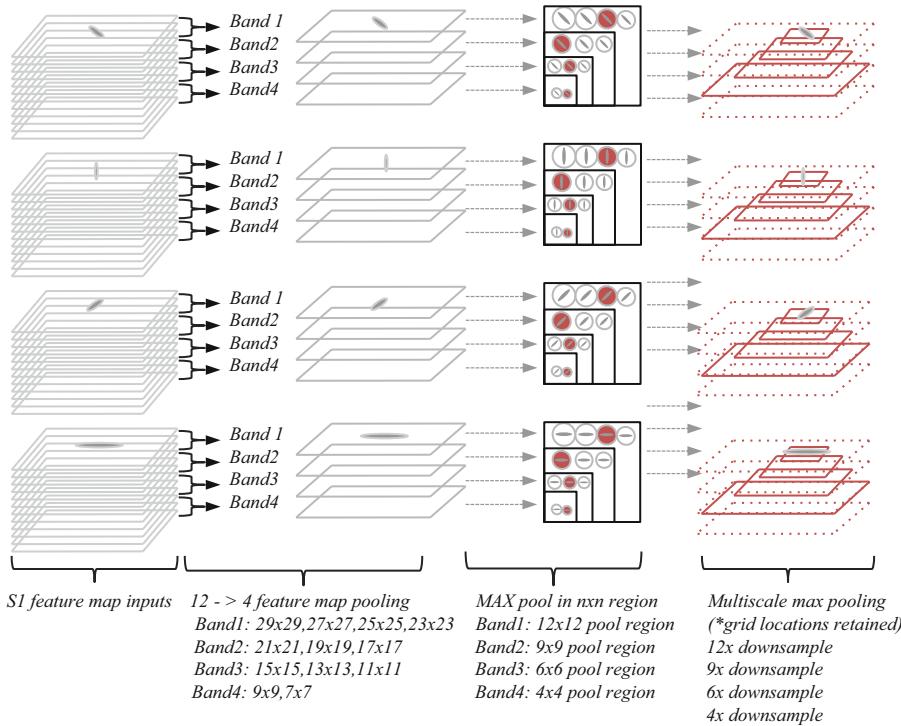


Figure 10.89 This figure illustrates the HMAX C1 layer. The S1 filter responses are pooled into four bands, then MAX pooled through $n \times n$ regions. The pooled features contain scale and position variations

the response of similar sized filter responses. The MAX pooling region sizes are 4×4 , 6×6 , 9×9 and 12×12 to be large enough for the features in each band, which provides some invariance for feature position, and the regions may overlap by a variable stride. Alternatively, features of each polarity may be generated and used instead. As shown in Fig. 10.89, the MAX pooling region size is proportional to the size of the features in each band, for example using smaller regions such as 4×4 for smaller features, and larger regions such as 12×12 for larger features. MAX pooling region overlap can be adjusted, for example using a stride factor of for dense overlapped region sampling, or higher stride factors to reduce region overlap. The absolute value of each filter response is used for MAX pooling to provide *feature polarity invariance* in the case of contrast inversion.

S2 Layer

The S2 layer corresponds to the V4 or posterior IT layer in the standard model. The S2 units pool afferent inputs from filter responses in C1 2×2 local regions for each scale band at each of four orientations, then compose a *complex combined filter response* against a *prototype* $n \times n$ patch P taken from a training image at the C1 layer. S2 cells respond to co-activation of C1 feature combinations of orientations and scales over larger receptive field sizes. As shown in Fig. 10.91, each S2 unit pools and combines input from a local 2×2 region at each orientation in the C1 feature maps to create the complex filter. Mutch and Lowe [543] improve the feature composition by only using the dominant orientation of each filter response as shown in Fig. 10.90.

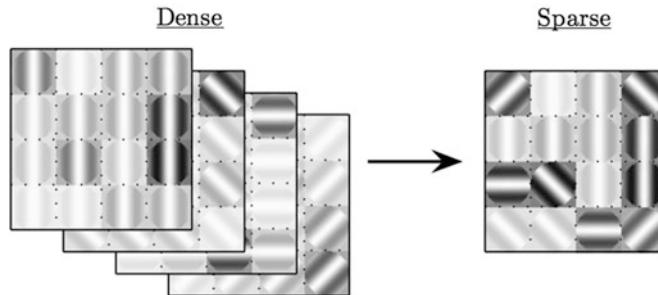


Figure 10.90 This figure illustrates the Mutch and Lowe [543] approach for sparse feature composition (tuning) using only the MAX orientation from all filter band responses, instead of using all filter responses combined, image from [543] IJCV 2008, © Springer and used by permission

S2 units find the response of C1 patches X to prototype patches P from the training image. Each patch region has a depth of 4 (one for each of the four C1 scale bands), and contains four possible values. C1 feature maps contain four feature orientations \times four pooled size bands, so the possible number of scale and orientation invariant feature combinations is $4^4 = 256$. All combinations are considered at the S2 layer across the entire image in local 2×2 regions. Note that the feature count can be increased by including a wider region than 2×2 and increasing the number of C1 feature orientations included. Cadieu [831] extends the region size to 3×3 , and includes features from all orientations in the composition instead of just a single orientation. For information and visualizations regarding S2 unit shape representation see Cadieu [831].

The composition and tuning method varies across implementations, and the original HMAX version [829] uses a response function with a weighted Gaussian summation $\{1,1,1,1\}$ of four features with standard deviation 1. In the HMAX-S derivative, Theriault et al. [832] use Cosine similarity between C and C1 patches, and use a normalized dot product between P and X which is invariant to illumination intensity, while the RBF is intensity sensitive. Other similarity functions are used as well in HMAX variants [828, 836].

The complex features contain several scales and orientations of each filter combined together, so the S2 output is a single 2D feature map dictionary of *complex prototype features* P , as shown in Fig. 10.91.

The S2 feature is taken by comparing the *difference* between the current image patch X from the training image against the current C1 prototype feature X, and the *difference* corresponds to the strength of the feature match. To compose each S2 prototype, random image patches X are taken for each image at the C1 level, and a response function $R(X,P)$ is used to compose a feature Y from the current image patch X in C1 and the prototype P_i patch at all positions for all image patches across each band and orientation as follows:

$$Y = R(X, P) = \exp(-\gamma(\|X - P_i\|^2)) \text{ (RBF from Serre & Reisenhuber [544, 835])}$$

*where γ is a sharpness tuning parameter, distance is Euclidean.

$$Y = R(X, P) = \exp\left(-\left(\frac{\|X - P_i\|^2}{2\sigma^2}\right)\right) \text{ (RBF—Mutch and Lowe [543, 828])}$$

*where $\alpha = (n/4)^2$ is a normalizing factor, n is the filter size dimension $n \times n$, distance is Euclidean.

$$Y = R(X, P) = \frac{P|_X}{\|P\|\|X\|} \text{ (Norm. Dot Product—Theriault et al. [832])}$$

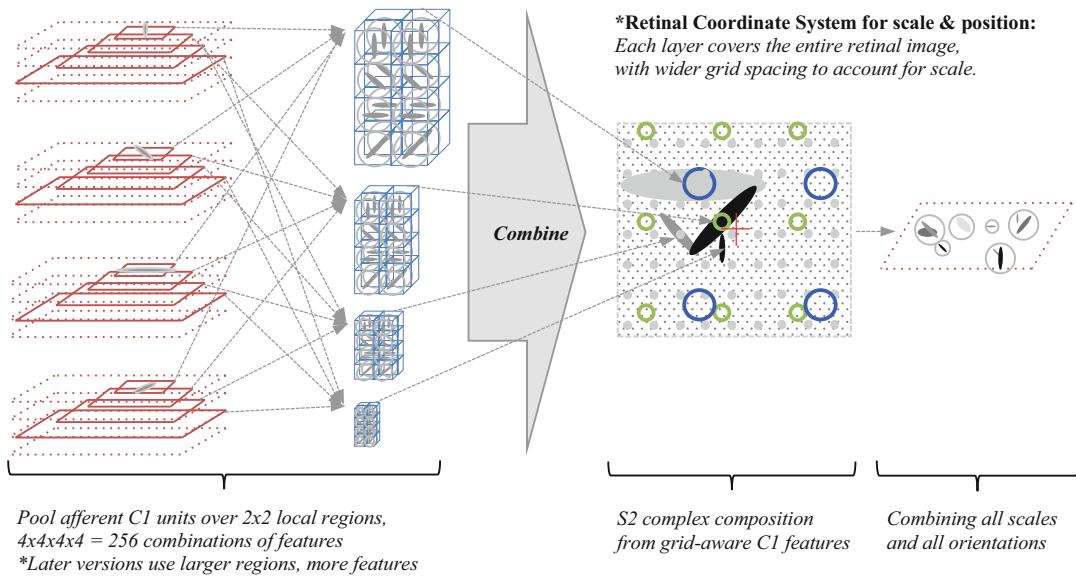


Figure 10.91 This figure illustrates the S2 unit complex feature composition from C1 feature maps, which contains all combinations ($4^4 = 256$) of multiscale and translated features

The S2 unit activation function is a tunable Gaussian function, and subsequent HMAX versions use a Gabor function. The S2 units are used to hold the feature dictionary which is a combination of bar-like features contained in the C2 cells at four orientations.

Note that Cadieu [831] also provides extensions to S2 and C2 to relax the method for combining features by allowing some feature learning at S2 to tune features to better match target patterns, and the C2 layer pooling parameters are also more flexible.

C2 Layer

C2 cells combine different sizes of S2 cells to respond to larger receptive field sizes. The C2 layer takes the S2 units as input, and produces output as an unordered 1D dictionary. C2 units can be constructed simultaneously while the S2 layer is constructed by taking the MAX response at all positions and scale bands across the whole image. C2 units feed into the VTUs to compose higher level concepts. The C2 dictionary contains *composite oriented-bar features*. The C2 units MAX pool across all the composite features from the S2 units of a specific orientation from all four filter bands together across the whole image, which provides a large amount of rotational and translation invariance. The S2 units compose the composite features, and the C2 units essentially just choose the MAX values. There is no spatial arrangement to the pooling, so at this point the features are simply collected as an unordered dictionary. The patterns contained in the 256 C2 units can combine to encode arbitrary object shapes (Fig. 10.92).

Note that the HMAX C2 dictionary is entirely based on the training data prototypes; however, Serre [835] also provides research regarding much larger feature counts in a *Universal Feature Dictionary* incorporating other features besides the Gabor-like oriented edges, using larger feature counts ~ 5000 features, and larger numbers of training examples than earlier HMAX implementations.

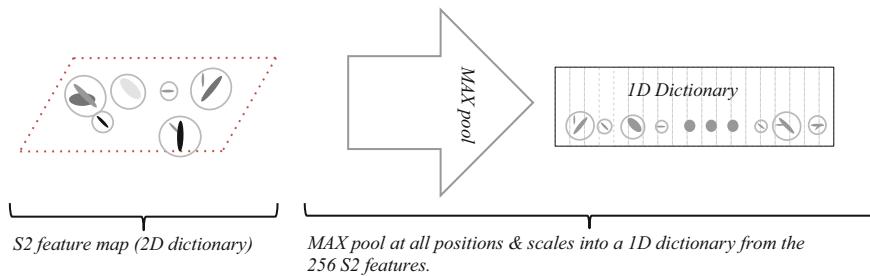


Figure 10.92 This figure illustrates the C2 layer, which performs max pooling across the S2 feature maps over all positions and scales into a 1D dictionary containing no scale or position information

VTU Classification

The VTU layer is the final classification layer where C2 features and perhaps C1 features are combined to form higher-level objects. Finally, the actual feature learning in HMAX occurs at the VTU layer. The VTU units (View-Tuned Units) take input from C2 units and mimic response to a 2D view of a 3D object, which closely resemble VTUs observed in monkeys by Logothetis [813]. The VTUs may use a Gaussian response function tuned to respond to a smaller width to focus the response, to achieve a maximum response of unity for strongest matches, and zero for no match. C2 units may be filtered out and ignored if the activation strength is too weak, or alternatively selected as afferents if strong enough.

Note that Serre and Riesenhuber [829] later extend the basic architecture to use more features (17 instead of 12), as well as providing parameters for filter band grouping and pooling region size tuning. Mutch and Lowe [543] introduce sparsity into the feature set using several methods including (1) suppressing weak activations for a particular feature orientation if the activation is <50 %, (2) computing matches for only the strongest orientations of each feature at a given location (a form of lateral inhibition), and (3) discarding weight templates with low values instead of passing them forward to the VTUs. Hu et al. [830] also make enhancements to the HMAX feature basis by learning S-layer features via sparse coding, transfer learning, and PCA/ICA PCA, instead of starting with the hard-wired Gabor filters. Serre et al. [544] uses an SVM classifier at the VTU layer taking inputs from C1 and C2 features, and experimented with boosting [486].

Training Protocols

HMAX training protocols vary, with early versions using smaller training sets of unmodified images, compared to the massive scaled, rotated, and contrast-modified training sets often used for CNNs. The HMAX training process involves extracting a set of size N randomly located pixel patches X from the training images. Typically, several thousand patches are extracted all together. The pixel patch X sizes match the region pooling sizes for C1 filters (i.e., 4×4 , 6×6 , 9×9 , 12×12 for the original HMAX version). Each of the patches is filtered to compute the response at the C1 level for all orientations and scales, and then is considered a *prototype P*. The number of C1 filter responses varies with the region pooling size, so for a 4×4 patch, there are 16 positions, and for each position there are four oriented filter units, so a 4×4 patch contains $4 \times 4 \times 4 = 64$ C1 unit responses. Each prototype is centered and filtered against all the S2 unit features, and the response is measured using a distance function such as Euclidean distance [835] or Cosine distance [832]. Lau et al. [851] and Theriault [832] use a dot product to measure distance. The HMAX-S method [832] uses multiple local scales for deeper prototype feature responses, resulting in much more detailed and sensitive prototypes. The prototypes are then MAX pooled to create the C2 feature dictionary.

In summary, the HMAX model is one of the most detailed models of neurobiology, primitive though it is, and is popular in the neuroscience research community. As compute power and memory increase, models like HMAX will become more common and be extended.

HMO—Hierarchical Model Optimization

Another neurological model is the *Hierarchical Model Optimization* (HMO) developed by Yamins et al. [646, 649] which models the high level reasoning centers in the IT cortex. By comparison, CNN models deal with the lower-level through higher-level features in the visual pathway, and local feature descriptor methods focus mostly on modeling the retina and eye with some other processing, yet in both CNNs and local descriptor models the higher level IT region and learning centers are not well addressed, and instead are posed as a classification problem solved using FC layers and SVM approaches. So HMO fills a unique niche within computer vision models at the highest level.

HMO optimizes an ensemble of lower-level hierarchical models using boosting and other optimization methods to model higher-level reasoning, and achieves remarkable equivalence with the human visual pathways under some tests, see Fig. 10.93. The HMO model development process involved basic research connecting electrodes to 168 neurons in a subject, and measuring the neural electrode response to a labeled training set. Then, thousands of possible response models such as CNNs and HMAX were evaluated and tuned in a hierarchical model optimization process (HMO) to discover the closest matching classifications from the candidate models (i.e., *reverse engineering*). The HMO response model set is optimized starting with random combinations of tuning parameters, such as feature counts, layers and momentum, with training protocol variations on image rotations, scales, and intensity. HMO discovers and combines the best performing models together to both predict and achieve high classification scores, using adaptive boosting and parameter optimizations.

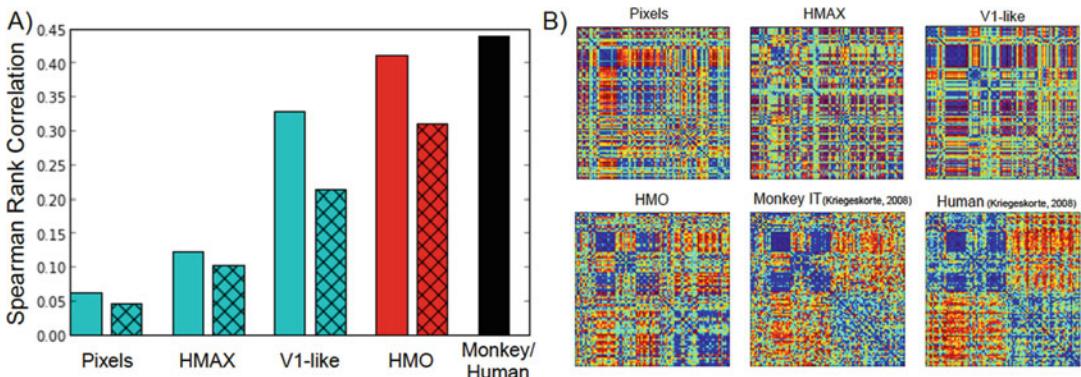


Figure 10.93 This figure illustrates the impressive results of the Hierarchical Model Optimization (HMO) model, image © by Yamins et al. [646], used by permission

Ensemble Methods

Ensemble methods are combinations of networks working together, which may build features in parallel, classify in parallel, and vote on the combined results. The ensemble of networks may be heterogeneous or homogeneous, for example using multiple CNNs exclusively, or perhaps using

CNNs and vocabulary methods together, or using CNNs and RNNs together as surveyed earlier (See C-RNN, QDRNN, RCL_RCNN, and dasNET). A simple ensemble network might include five CNNs each using slightly different training protocols and learning parameters (see Inception), each yielding labeled features for classification, with a final voting mechanism at the end incorporating all results.

Ensembles can provide some advantages such as:

- Reduce overfitting
- Add more degrees of invariance to the feature set
- Increase overall accuracy
- Speed up training, as suggested in 1989 by Waibel [688]

One disadvantage of ensembles is the increased computational requirements; however, methods for optimizing ensembles have been explored as explained next.

As noted by Bucilla et al. [614] many of the best performing models are *ensembles of hundreds or thousands of models*. This is intuitive since no single model can be optimized for all types of data, and models are trained to specialize on a given training set. Bucilla developed a method for compressing a set of models into a smaller set of models. The key concepts include identifying smaller faster models to approximate the slower larger models, and their work focuses on identifying the best training protocols and training data to enable the model comparisons. See also Hinton et al. [620] for more on distilling ensembles on neural networks.

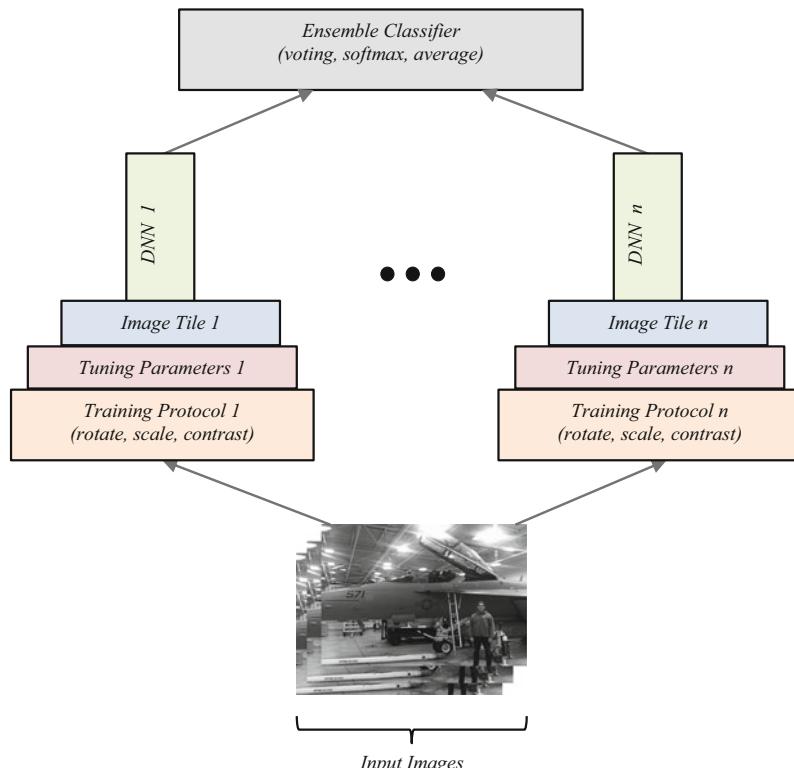


Figure 10.94 This figure illustrates an ensemble of networks working together, each with different training protocol, tuning parameters, and image tile regions, feeding into a final classifier

The dasNet method [612] surveyed earlier uses an ensemble of CNNs in parallel to allow for parallel testing of different hypothesis via modifying the weights in each network, similar to the way an expert might test a set of hypothesis. DasNet incorporates an automatic feature optimization and evaluation process.

The AlexNet method [640] surveyed earlier in the CNN section splits the feature learning into two parallel networks: one network learning low-level features, and the other network learning the higher-level features. Although the motivation for AlexNet's dual feature learning network is performance, a side-effect is that the feature learning in each parallel network follows a slightly different route and leads to slightly different results than if the features were learned in the same network.

The Multicolumn DNN (MCDNN) developed by Ciresan Meier and Shmidhuber [790] uses several DNNs in parallel and provides different image input to each DNN, and then averages the results for the final classification score. Each column of DNNs uses shared weights, and allows for parallel evaluation of features. Each DNN is trained on overlapping columns of the input images in a winner-take-all manner, and only the winning DNN features are trained. This preserves features that have already been learned, and allows each network to potentially focus on a different level in the feature hierarchy, so each DNN may be the winner for a different level of the hierarchy. The input image columns may be processed in different ways.

The HMO methods by Yamins et al. [649] surveyed earlier evaluates thousands of candidate models, such as CNNs and HMAX models, and optimizes the final results across all the models via a boosting and hyper parameter optimization and tuning process. HMO is another model driven from the neuroscience community rather than the computer vision community.

The downside of ensembles is the *design by committee* syndrome, where nobody is right, nobody is wrong, everybody contributes, and nobody is 100 % happy. Even if one of the committee members are correct, the design by committee approach ensures that they will be muted. As evidenced by MAX pooling (choosing the strongest activation), the highest confidence response has proven in many cases to be better than average pooling response (i.e., the ensemble approach).

For more references and historical developments in the area of ensembles and committees, see Schmidhuber [552].

Deep Neural Network Futures

The current state of the art of DNNs has been surveyed in this chapter, so here we will explore a few areas for future research to expand the boundaries of DNNs. Future research into more complex neuron models continues, such as Spiking Neural Networks which provide feedback paths between neuron groups to influence neuron group firing. Other areas for future research include (1) how to increase network depth to the maximum useful level in a compute efficient manner (network depth optimization), (2) refactoring and compressing a single deep network or a complex ensemble network into a smaller single network as an approximation (model compression), (3) decomposing complex classifiers into a set of simpler classifiers (classifier decomposition and recombination), and (4) training protocols will be a key future research area, we should expect additional breakthroughs incorporating better preparation of the training set and better selection of images, combined with better segmentation of the correct regions of interest from the training set images by the DNN, propelling classification accuracy to higher levels. Finally, we should expect to see a proliferation of special-purpose DNN related processors, such as the Baidu data bandwidth accelerators for their cloud-based systems surveyed earlier in the MINWA section, the Google TPU cloud accelerator for increased compute performance, and special purpose accelerators for endpoint devices to perform inferencing on local images and trained DNN models.

Increasing Depth to the Max—Deep Residual Learning (DRL)

Deep Residual Learning (DRL) was developed by He et al. [872] to explore methods of increasing DNN network depth as far as possible to improve accuracy, and borrows from He's earlier work on SPP [542] and Fast-R-CNN [707] architectures discussed earlier in this chapter. He et al. report that DRL networks are the deepest networks to date, and architectures using 100–1000 layers have been implemented using a very efficient parameterization comparable to much smaller networks. The DRL method is driven by the goal to solve a key problem observed in DNN research, namely that when the depth of layers increase, a point is reached where accuracy stops converging, and then accuracy begins to become worse, ending up inferior to DNNs with fewer layers. In addition, He et al. shows another benefit of DRL concepts, namely the mitigation of problems associated with improper initialization of the weights and network parameters. See also earlier related work by Deng [891], and more recent developments of DRL as Resnet by Targ [892] and Szegedy [893].

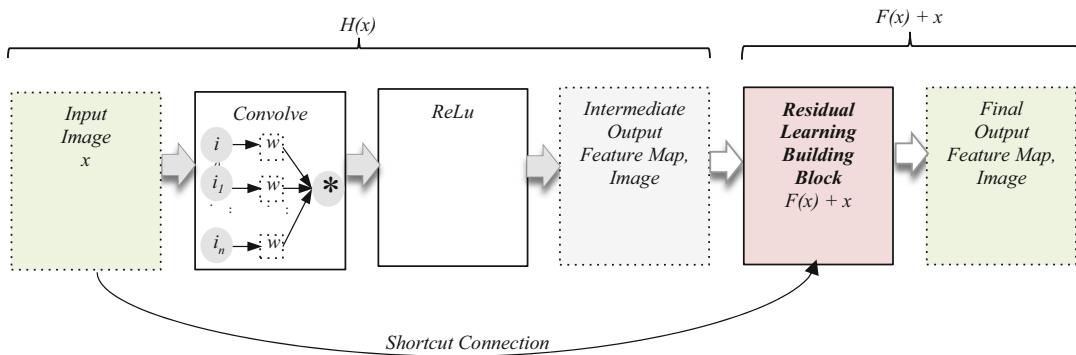


Figure 10.95 This figure illustrates the residual learning building block (RLBB) formulation using shortcut connections

One of the basic concepts used by DRL to increase the network depth is *shortcut connections* as shown in Fig. 10.95, which feed the input forward to reuse later. (Note Inception Net surveyed earlier in this chapter also uses shortcut connections. Many historical MLPs have also used shortcut connections in an *ad hoc* manner, see Schmidhuber [552]). The DRL network reformulates the basic FNN into *layer groups* separated by novel *Residual Learning Building Blocks* (RLBBs) as shown in Fig. 10.95, which are inserted typically after each filter layer in He's work. RLBBs take input from the skip connection to compute a residual, or difference, between the input and a processed input, using *residual functions* to combine the input x to a group of layers $F(x)$ with the output of the group of layers, yielding the *residual difference*. The key idea is that training on the numerically smaller magnitude residuals makes true fluctuations in the gradients easier to spot during training, and enables more accurate weight updates to be made. The residual function is defined on a group of layers as a *mapping function* expressed as follows:

$$H(x) : \text{underlying mapping function}$$

$$F(x) := H(x) - x : \text{stacked non-linear mapping}$$

$$F(x) + x : \text{recast mapping using residuals}$$

He et al. demonstrate networks using RLBBs which can effectively eliminate the need for most of the compute and parameter intensive *fully connected* classification layers, discussed earlier in this

chapter. Since large FC layers represent most of the parameters in a typical DNN, by eliminating FCs the total network parameter count can be greatly reduced even with the increased layer count by incorporating RLBBs. Thus, the RLBB formulation does not add complexity or additional parameters to the network, is trainable using backpropagation, and shown to solve the accuracy divergence anomalies of deeper networks.

To illustrate the anomalies observed in DNNs of varying layers, He et al. provide results showing a 22 layer CNN architecture that yields better accuracy than a similar 50 CNN layer architecture which exhibits the training accuracy divergence anomalies. While the root cause of the accuracy divergence and degradation is not explored in He's work and is noted as an area for future research, a likely cause of the divergence seems to be the method of gradient descent itself, which may create transient gradient spikes across the network due to anomalous gradient error combinations, similar to transient spikes in electrical circuits. Perhaps, when a given gradient is corrected, the correction may lead to related oscillations in other gradients, which at specific training epoch intervals may align and sum together into an objectionable gradient transient spike, which is then adjusted by corresponding anomalous weight updates, subsequently contributing to increased gradient transients. The very method of gradient descent itself is not well understood, and resembles a massive, serendipitous averaging and dithering procedure, controlled using ad hoc learning rate and momentum parameters to smooth out the gradient descent, as introduced earlier in this chapter.

Approximating Complex Models Using A Simpler MLP (Model Compression)

One approach to increasing DNN accuracy is to employ an ensemble of DNNs together, each architected and trained slightly differently, and the results of the ensemble are averaged to get a synergistic *design by committee style classification*. However, since the ensemble is a large compute workload, researchers have developed methods to *approximate the ensemble* using a single DNN to reduce the compute workload, which seems intuitive since an MLP is a general function approximator. The end result is a smaller and compact approximation of the much larger ensemble. In a similar fashion, a single complex and deep DNN may be approximated by a simpler DNN to provide compute benefits. One obvious application for model reductions is to first train a much larger cloud-based ensemble or complex network, and then reduce the model to a single smaller and faster DNN for deployment on a small embedded or portable device. We discuss some relevant research here.

One approach to approximating an ensemble via a single model is the two-stage approach taken by Bucila et al. [875] which *first* trains an ensemble to label the training set, and *second* takes the labeled output of the ensemble and the test set as the input to train a single DNN to *approximates* the ensemble. Using a single DNN to approximate a large ensemble vastly decreases the compute workload, both at training time and deployment time. In this respect, model compression is related to transfer learning in DNNs as discussed earlier in this chapter, where a DNN is first trained, and then the weights are transferred into another DNN which is further trained for a similar knowledge domain, gaining faster training times, feature refinement and specialization, and perhaps better accuracy. The *compressed model* thus approximates both the classification labels and the weights. Of course, instead of an ensemble, a very deep network could also be approximated by a smaller and more efficient compressed model in a similar fashion.

To generalize Bucila's work, Hinton et al. [873] define a Knowledge Distillation model (KD) using a *teacher model* and a *student model*. A parameterized softmax relaxation function is applied to the teacher model outputs to allow the student model to generalize a new model within a general range of accuracy. The student model is then further optimized using an *objective function* on the classifier that compares the student and teacher model results.

A further refinement to the KD model is the FitNets architecture developed by Romero et al. [874], which uses *hints* derived from the teacher model to guide the training of the student model. A hint is taken from the hidden layers of the teacher network, and used to guide the approximation of a hidden layer in the student network. Thus, the hint process guides and optimizes the student to reproduce the weights of a certain layer of the teacher, which is intended to optimize the student network in the right direction towards the correct classification output. FiNets are shown to increase approximation accuracy over both KD and Bucila's method. Hint-based training is considered to be a relative of Bengio's earlier work on Curriculum Learning [876], which trains the network using a simple to complex training protocol, similar to transfer learning (i.e., *we may term curriculum learning as a successive refinement transfer learning protocol*). First, simple training examples are used to train the network, and then the network weights are progressively retrained by using more expressive and complex training examples in series. Thus, curriculum learning successively refines the same network to generalize from simple to complex training examples.

Classifier Decomposition and Recombination

A related method for model compression is developed by Hinton et al. [873] to *decompose* the classifier to produce one or more *specialist models* or *fine grained classifiers*. Hinton models the teacher classifier as a set of smaller fine-grained classifiers, since the larger teacher models may confuse classification in some cases into a *coarse grain classification*. The idea of decomposing a coarse grained classifier into a set of fine-grained classifiers is novel, and is a promising area of future research to increase classification accuracy.

Contrary wise, future work is expected to find optimal ways to recombine a set of fine-grained classifiers together to produce a stronger hybrid classifier which is more generic or application-specific, for example by training a set of smaller DNNs on smaller training sets, and then combining the smaller DNNs together into a *classification bank*, similar to a filter bank. The classification bank may be implemented as a very wide FC layer, parallel FC layers, or as an ensemble network for specific applications.

Summary

We explore feature learning architectures and deep learning using both ad hoc and neuroscience inspired methods. In most feature learning systems, a hierarchy of features is learned, ranging from low-level edge and texture features, through mid-level motif concepts, up to higher level object parts and whole objects. Some use an ensemble of classifiers to evaluate the features, while other approaches use a hierarchy of classifiers together to reach a conclusion. In the future, we will see an increase in *better feature representations* beside the simple correlation templates used today in most DNNs, taking advantage of local, regional, and global features.

The neural network approaches used in feature learning point to a future merger of *synthetic intelligence* and *synthetic vision*, using the same underlying neural network architecture standardized into silicon, which can be tuned for a wide range of analytic problems including computer vision, speech, investing, marketing, and surveillance.

Science and technology are like waves, forming power as they rise, carrying the best research minds surfing on the crest, and as the waves rapidly approach the shoreline, applications become more widespread and tower over older methods, then become commercialized or militarized, which changes societies and nations. Then the waves crash on the sand as the technology is commoditized approaching a zero price point (i.e., free and expected), then the research is perhaps uninteresting to many, and often the best minds and researchers who have spent the prime years of their life

researching the technology are *stranded on the beach*, with obsolete knowledge, several past successes, and not enough of a lifetime left to switch to a new discipline and ride the next big wave. While the present wave crashes, new waves of technology are approaching on the back of the previous waves, surfed by a new crop of researchers and bright minds, imbued with all the excitement and power of the new tidal wave, *and so the cycle repeats*. We are nearing the crest of the new wave of *synthetic vision systems* based on visual neuroscience concepts: *synthetic brains*. We will soon see synthetic brains, synthetic vision, intelligent prosthetics, and robotics change society forever. The current wave of AI and feature learning has already left earlier waves of computer vision researchers washed up on the beach, since they are not riding the surging wave of neurological vision research.

The early crest of the synthetic brain and synthetic vision wave is already here, pushing along new innovations such as smart cars, visual surveillance, smart advertising, and smart analytics, but there is much more to come. When the synthetic vision and synthetic brain waves onto the shore of commercial markets in a big way, we will see a flood of inexpensive and ubiquitous smart devices that save time and energy, similar to apps and inexpensive appliance-like devices. The products will be complete physical appliances, which will likely use a common core architecture of ANNs, robotic motor controllers, and synthetic vision, allowing an ecosystem of intelligent devices to be built and customized for a wide range of commercial applications. This will change the nature of society. However, the ANNs cannot create the human spirit and soul, so we will live on as before, but with more synthetic assistance.

Chapter 10: Learning Assignments

1. Discuss the Perceptron (P) architecture, and the Multilayer Perceptron (MLP), including the learning rules and training protocols used.
2. Discuss the Neocognitron architecture.
3. Describe how to create a feature map.
4. Describe a hierarchical feature map volume (stack).
5. Describe the layers in the basic LeNet architecture.
6. Describe a forward pass through a CNN based on the basic LeNet architecture.
7. Describe a backward pass through a CNN based on the basic LeNet architecture.
8. What is stored in memory during the forward pass through the CNN?
9. Describe how a fully connected layer (FC layer) is used for classification in a CNN, discuss the FC layer feature weights, and provide a hypothetical FC layer design.
10. Describe the common layers in a CNN.
11. Describe how a convolution kernel is applied across input images and feature maps.
12. Describe the difference between the dot product, convolution, correlation, and normalized correlation, in the context of CNNs.
13. Compare at least two methods for initializing CNN feature weights.
14. Discuss sliding windows and how a stride factor is used, and compare the advantages and disadvantages of small vs. large windows.
15. Describe the bias input to the artificial neuron, and how it is used.
16. Describe the information collected during the forward pass of the CNN which is used in the backpropagation step.
17. Describe backpropagation using gradient descent.
18. Describe how the total gradient error is computed at the classifier to begin the backpropagation step, and describe how the total gradient error is proportionally split apart into partial derivatives and distributed backwards through the network at each neuron.
19. Describe how the partial derivatives of the total gradient error passed backwards, and the neural state derivative, are used together at each neuron to adjust the feature weights.
20. Describe learning rate and momentum parameters used during backpropagation to tune the feature weights.
21. Discuss considerations for determine the number of layers in a CNN, and the number of features per layer. Include considerations for compute performance and memory.
22. Describe the advantages and limitations of using stacked convolutions with small kernel windows vs. using convolutions over larger windows.
23. Discuss separable convolution, and provide an example algorithm.
24. Discuss fused convolution, and provide an example algorithm.
25. Discuss how to estimate architecture parameters to measure CNN complexity.
26. Describe the VGGnet architecture, including variations.
27. Describe the artificial neuron model in the NiN architecture, which computes the features.
28. Describe cross channel parametric pooling (CCCP) in the NiN architecture.
29. Describe cross channel pooling (CCP) in the Maxout architecture.
30. Describe advantages of using Z-columns for 1×1 convolutions across feature map volume.
31. Compare global average pooling (GAP) as used in the NiN architecture against FC-layers used in typical CNNs.
32. Describe the composition of a GoogLenet feature layer (i.e., the Inception module).
33. Describe the feature vector format of a GoogLenet inception module.

34. Describe the feature model of the SMYNETS architecture.
35. Discuss the Polynomial Neural Network (PNN) model, otherwise referred to as the Group Method for Data Handling (GMDH).
36. Describe vanishing gradients and exploding gradients in the context of backpropagation.
37. Describe an RNN neuron model and discuss short-term memory in RNNs.
38. Discuss how to unroll an RNN into an FNN, and motivations for doing so.
39. Describe the Long Short-Term Memory (LSTM) enhancement to RNNs.
40. Discuss why an RNN is suited to sequence processing and spatiotemporal pattern matching.
41. Describe a bidirectional RNN, draw a diagram also.
42. Describe a 2D RNN, draw a diagram, and discuss applications of a 2D RNN to computer vision.
43. Describe the K-MEANS clustering algorithm at a high level, and discuss practical considerations and pitfalls for using K-MEANS clustering to build a dictionary or visual vocabulary.
44. Name and describe at least two feature encoding methods.
45. Describe the K-SVD sparse coding algorithm, and compare it to the K-MEANS method.
46. Describe a kernel function, kernel projection, kernel encoding, and why kernels are used in classification (HINT: the kernel trick).
47. Describe the general concept and advantages of kernel machines, such as the Support Vector Machine.
48. Describe how a linear classifier works, and how a logistic classifier works.
49. Describe the basic HMAX architecture.
50. Discuss the types of features used in the lower layers of the HMAX architecture.
51. Discuss feature learning in the HMAX architecture.
52. Discuss view-tuned units in the HMAX architecture.
53. Discuss viewpoint dependent vs. viewpoint independent models of vision, and explain the difference in terms of the types of features stored in memory.
54. Discuss the concepts behind ensemble architectures.
55. Discuss the HMO architecture ensemble approach to higher-level reasoning.

Appendix A: Synthetic Feature Analysis

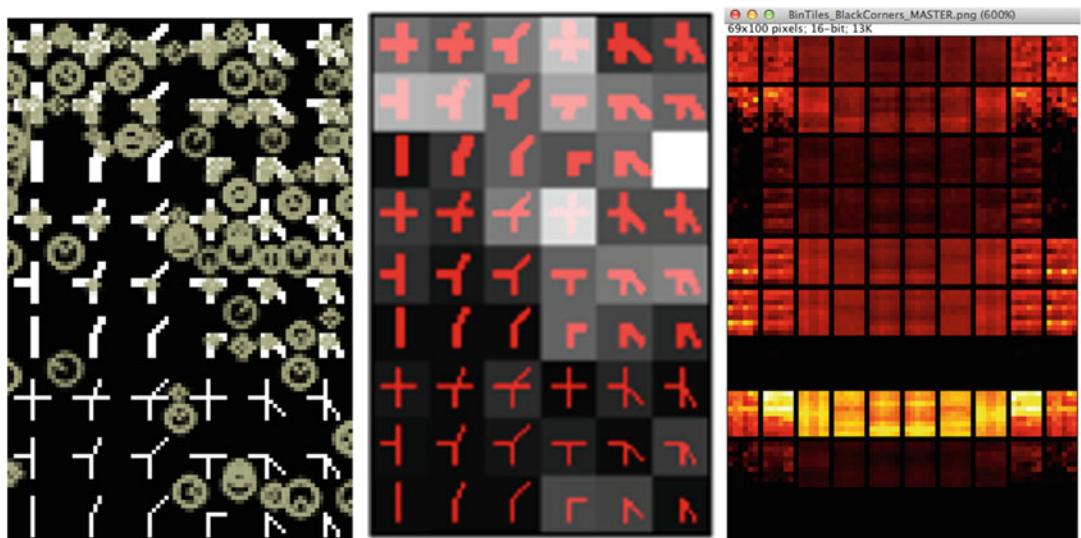


Figure A.1 Example analysis results from Test #4 below, (*left*) annotated image showing detector locations, (*center*) count of each alphabet feature detected, shown as a 2D shaded histogram, (*right*) set of 2D shaded histograms for rotated image sets showing all ten detectors

This appendix provides analysis of several common detectors against the synthetic feature alphabets described in Chap. 7. The complete source code, shell scripts, and the alphabet image sets are available from Springer Apress at: <http://www.apress.com/source-code/ComputerVisionMetrics>.

This appendix contains:

- Background on the analysis, methodology, goals, and expectations.
- Synthetic alphabet ground truth image summary.
- List of detector parameters used for standard OpenCV methods: SIFT, SURF, BRISK, FAST, HARRIS, GFFT, MSER, ORB, STAR, SIMPLEBLOB. Note: No feature descriptors are

computed or used, only the detector portions of BRISK, SURF, SIFT, ORB, and STAR are used in the analysis.

- Test 1: Interest point alphabets.
- Test 2: Corner point alphabets.
- Test 3: Synthetic alphabet overlays onto real images.
- Test 4: Rotational invariance of detectors against synthetic alphabets.

Background Goals and Expectations

The main goals for the analysis are:

- To develop some simple intuition about human vs. machine detection of interest point and corner detectors, to observe detector behavior on the synthetic alphabets, and to develop some understanding of the problems involved in designing and tuning feature detectors.
- To measure detector anomalies among white, black, and gray versions of the alphabets. A human would recognize the same pattern easily whether or not the background and foreground are changed; however, detector design and parameter settings influence detector invariance to background and foreground polarity.
- To measure detector sensitivity to slight pixel interpolation artifacts under rotation.

Note Experienced practitioners with well-developed intuition regarding capabilities of interest point and corner detector methods may not find any surprises in this analysis.

The analysis uses several well-known detector methods as implemented in the OpenCV library; see Table A.1. The analysis provides detector information only, with no intention to compare detector goodness against any criteria. Details on which features from the synthetic alphabets are recognized by the various detectors are shown in summary tables, counting the number of times a feature is detected with each grid cell. For some applications, the synthetic interest point alphabet approach could be useful, assuming that an application-specific alphabet is designed, and detectors are designed and tuned for the application, such as a factory inspection application to identify manufactured objects or parts.

Test Methodology and Results

The images in the ground truth data set are used as input for a few modified OpenCV tests:

- opencv_test_features2d
(BRISK, FAST, HARRIS, GFFT, MSER, ORB, STAR, SIMPLEBLOB)
- opencv_test_nonfree
(SURF, SIFT)

The tuning parameters used for each detector are shown in Table A.1; see the OpenCV documentation for more information. Note: no attempt is made to tune the detector parameters for the synthetic alphabets. Parameter settings are reasonable defaults; however, the maximum keypoint feature count is bumped up in some cases to allow all the detected features to be recorded.

Table A.1 Tuning parameters for detectors

Detector	Tuning Parameters
BRISK	octaves = 3 threshold = 30
FAST	threshold = 10 nonMaximalSuppression = TRUE
HARRIS	maxCorners = 60000 (to capture all detections) qualityLevel = 1.0 minDistance = 1 blockSize = 3 useHarrisDetector = TRUE k = .04
GFFT	maxCorners = 60000 (to capture all detections) qualityLevel = .01 minDistance = 1.0 blockSize = 3 useHarrisDetector = FALSE k = .04
MSER	Delta = 5 minArea = 60 maxArea 14400 maxvariation = .25 minDiversity = .2 maxEvolution = 200 areaThreshold = 1.01 minMargin = .003 edgeBlurSize = 5
ORB	WTA_K = 2 edgeThreshold = 31 firstLevel = 0 nFeatures = 60000 (to capture all detections) nLevels = 8 patchSize = 31 scaleFactor = 1.2 scoreType = 0
SIFT	contrastThreshold = 4.0 edgeThreshold = 10.0 nFeatures = 0 nOctaveLayers = 3 sigma = 1.0
STAR	maxSize = 45 responseThreshold = 30 lineThresholdProjected = 10 lineThresholdBinarized = 8
SURF	Extended = 0 hessianThreshold = 100.0 nOctaveLayers = 3 nOctaves = 4 upright = 0

(continued)

Table A.1 (continued)

Detector	Tuning Parameters
SIMPLEBLOB	<pre> thresholdStep = 10 minThreshold = 50 maxThreshold = 220 minRepeatability = 2 minDistBetweenBlobs = 10 filterByColor = true blobColor = 0 filterByArea = true minArea = 25 maxArea = 5000 filterByCircularity = false minCircularity = 0.8f maxCircularity = std::numeric_limits<float>::max() filterByInertia = true minInertiaRatio = 0.1f maxInertiaRatio = std::numeric_limits<float>::max() filterByConvexity = true minConvexity = 0.95f maxConvexity = std::numeric_limits<float>::max()</pre>

Each test produces a variety of results, including:

1. Annotated images showing location and orientation (if provided) for detected features.
2. Summary count of each detected synthetic feature across the grid in text files, including interest point coordinates, detector response strength, orientation if provided by the detector, and the number of total detected synthetic features found.
3. 2D histograms showing bin count for each feature in the alphabet.

Detector Parameters Are Not Tuned for the Synthetic Alphabets

No feature detector tuning is attempted here. Why? In summary, feature detector tuning has very limited value in the absence of (1) a specific feature descriptor to use the keypoints, and (2) an intended application and use-cases. Some objections may be raised to this approach, since detectors are designed to be tuned and must be tuned to get best results for real applications. However, the test results herein are only a starting point, intended to allow for simple observations of detector behavior compared to human expectations.

In some cases, a keypoint is not suitable for producing a useful feature descriptor, even if the keypoint has a high score and high response. If the feature descriptor computed at the keypoint produces a descriptor that is too weak, the keypoint and corresponding descriptor should both be rejected. Each detector is designed to be useful for a different class of interest points, and tuned accordingly to filter the results down to a useful set of good candidates for a specific feature extractor.

Since we are not dealing with any specific feature descriptor methods here, tuning the keypoint detectors has limited value, since detector parameter tuning in the absence of a specific feature description is ambiguous. Furthermore, detector tuning will be different for each detector–descriptor pair, different for each application, and potentially different for each image.

Tuning detectors is not simple. Each detector has different parameters to tune for best results on a given image, and each image presents different challenges for lighting, contrast, and image

preprocessing. For typical applications, detected keypoints are culled and discarded based on some filtering criteria. OpenCV provides several novel methods for tuning detectors; however, none are used here. The OpenCV tuning methods include:

- **DynamicAdaptedFeatureDetector** class will tune supported detectors using an *adjusterAdapter()* to only keep a limited number of features, and to iterate the detector parameters several times and redetect features in order to try and find the best parameters, keeping only the requested number of best features. Several OpenCV detectors have an *adjusterAdapter()* provided while some do not, and the API allows for adjusters to be created.
- **AdjusterAdapter** class implements the criteria for culling and keeping interest points. Criteria may include KNN nearest matching, detector response or strength, radius distance to nearest other detected points, removing keypoints for which a descriptor cannot be computed, or other.
- **PyramidAdaptedFeatureDetector** class can be used to adapt detectors that do not use a scale-space pyramid, and this adapter will create a Gaussian pyramid and detect features over the pyramid.
- **GridAdaptedFeatureDetector** class divides an image into grids, and adapts the detector to find the best features within each grid cell.

Expectations for Test Results

The reader should treat these tests as information only to develop intuition about feature detection. The test results do not prove the merits of any detector. Interpretation of the test results should be done with the following information in mind:

1. One set of detector tuning parameters is used for all images, and detector results will vary widely based on tuning parameters. In fact, the parameters are deliberately set to over-sensitive values for ORB, SURF, and other detectors to generate the maximum number of possible keypoints that can be found.
2. Sometimes an alphabet feature generates multiple detections; for example, a single corner alphabet feature may actually contain several corner features.
3. The detection results may not be repeatable over the distribution of replicated features in the image feature grid. In other words, identical patterns, which look about the same to a human, are sometimes not recognized at different locations. Without looking in detail at each algorithm, it is hard to say what is happening.
4. Detectors that use an image pyramid such as SIFT, SURF, ORB, STAR, and BRISK may identify keypoints in a scale space that are offset or in between the actual alphabet features. This is expected, since the detector is using features from multiple scales.

Summary of Synthetic Alphabet Ground Truth Images

The ground truth dataset is summarized here. Note that rotated versions of each image file in the set are provided from 0 to 90° at 10° intervals. The 0° image in each set is 1024 × 1024 pixels, and the rotated images in each set are slightly larger to contain the entire rotated 1024 × 1024 pixel grid.

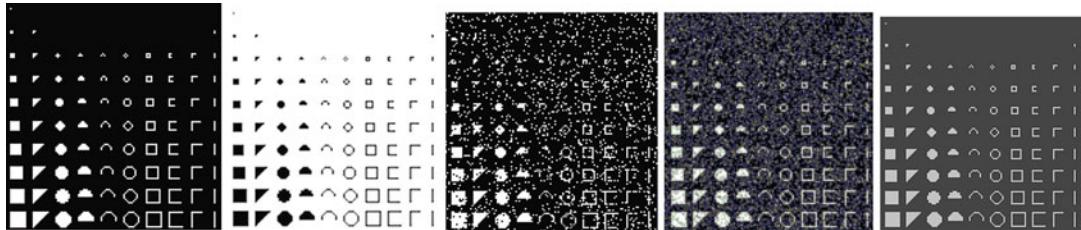


Figure A.2 Synthetic interest points

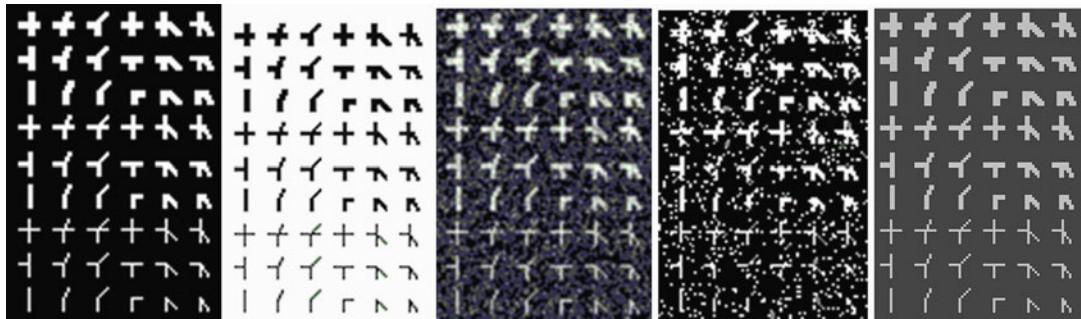


Figure A.3 Synthetic corner point

Synthetic Interest Point Alphabet

The synthetic interest point alphabet contains multiples of the 83 unique patterns, as shown in Fig. A.2. A total of 7×7 sets of the 83 features fit within the 1024×1024 image. Total unique feature count for the image is $7 \times 7 \times 83 = 4116$, with $7 \times 7 = 49$ instances of each feature. The features are laid out on a 14×14 pixel grid composed of 10 rows and 10 columns, including several empty grid locations. Gray image pixel values are 0×40 and $0 \times c0$, black and white pixel values are 0×0 and $0xff$.

Synthetic Corner Point Alphabet

The synthetic corner point alphabet contains multiples of the 63 unique patterns, as shown in Fig. A.3. A total of 8×12 sets of the 63 features fit within the 1024×1024 image. Total unique feature count is $8 \times 12 \times 63 = 6048$, with $8 \times 12 = 96$ instances of each feature. Each feature is arranged on a grid of 14×14 pixel rectangles, including 9 rows and 6 columns of features. Gray image pixel values are 0×40 and $0 \times c0$, black and white pixel values are 0×0 and $0xff$.

Synthetic Alphabet Overlays

A set of images with the synthetic alphabets overlaid is provided, including rotated versions of each image, as shown in Fig. A.4.

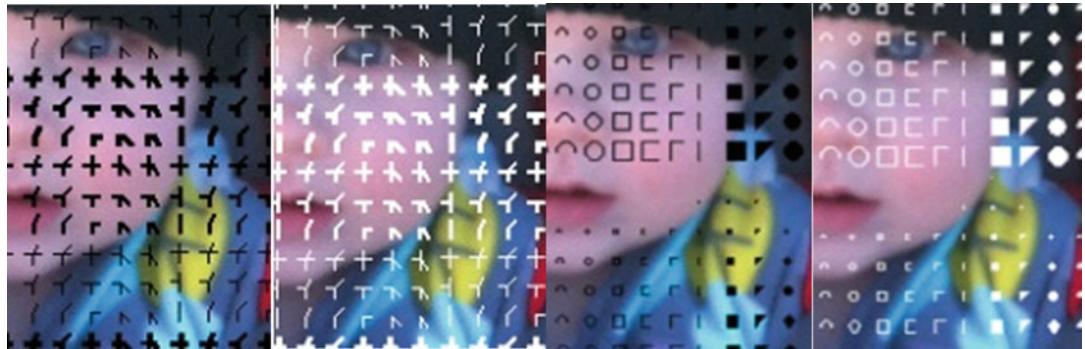


Figure A.4 Synthetic alphabets overlaid on real images

Table A.2 Summary count of detected features found in the synthetic interest point alphabet, 0° rotation

Detector	Interest Points White On Black	Interest Points Black On White	Interest Points White On Black Salt /Pepper Noise	Interest Points White On Black Gaussian Filtered	Interest Points Lt. Gray on Dk. Gray
SURF	18178	19290	33419	22951	13526
SIFT	11672	15208	18323	19054	8519
BRISK	823	4634	25070	9075	550
FAST	343	4971	41265	50711	2112
HARRIS	14833	14217	47025	23473	14854
GFFT	16296	14069	52415	58804	15876
MSER	0	1	2758	2289	0
ORB	32414	42675	56653	55044	27996
STAR	3486	5847	3692	4336	2277
SIMPLEBLOB	441	1201	68	385	441

Test 1: Synthetic Interest Point Alphabet Detection

Table A.2 provides the total detected synthetic interest points. Note: total detector counts include features computed at each scale of an image pyramid. For detectors, which report feature detections at each level of an image pyramid, individual pyramid level detections are shown in Table A.3.

The total number of features detected in each alphabet cell is provided in summary tables from the annotated images. Note that several features may be detected within each 14×14 cell, and the detectors often provide non-repeatable results, which are discussed at the end of this appendix. The counts show the total number of alphabet features detected across the entire image, as shown in Fig. A.5.

Annotated Synthetic Interest Point Detector Results

For ORB and SURF detectors, the annotated renderings using the *drawkeypoints()* function are too dense to be useful for visualization, but are included in the online test results.

Table A.3 Octave count of detected features found in the synthetic interest point alphabet, 0° rotation

Detector	Interest Points White On Black	Interest Points Black On White	Interest Points White On Black Salt/ Pepper Noise	Interest Points White On Black Gaussian Filtered	Interest Points Lt. Gray on Dk. Gray
SURF total:	18178	19290	33419	22951	13526
Octave 0:	9044	9807	24820	15667	8176
Octave 1:	4392	4505	5199	3936	2801
Octave 2:	4623	4862	3270	3226	2435
Octave 3:	119	116	130	122	114
BRISK total:	823	4634	25070	9075	326
Octave 0:	258	3482	24686	8256	143
Octave 1:	21	170	2	226	0
Octave 2:	402	851	315	555	179
Octave 3:	136	101	54	31	4
ORB total	32414	42675	56653	55044	27996
Octave 0:	330	4924	13030	13030	330
Octave 1:	5507	9467	10859	10859	5126
Octave 2:	7437	8519	9049	9049	7003
Octave 3:	6114	6333	7541	7541	5704
Octave 4:	4575	4625	6284	6284	3922
Octave 5:	3390	3495	4744	3869	2787
Octave 6:	2988	3150	3173	2793	2061
Octave 7:	2073	2162	1973	1619	1063

The diameter of the circle drawn at each detected keypoint corresponds to the “diameter of the meaningful keypoint neighborhood,” according to the OpenCV KeyPoint class definition, which varies in size according to the image pyramid level where the feature was detected. Some detectors do not use a pyramid, so the diameter is always the same. The position of the detected features is normalized to the full resolution image, and all detected keypoints are drawn.

Entire Images Available Online

To better understand the detector results for each test, the entire image should be viewed to see the anomalies, such as where detectors fail to recognize identical patterns. Figure A.5 is an entire image showing BRISK detector results, while others are available online. Test results shown in Figs. A.6 through A.15 only show a portion of the images.

Test 2: Synthetic Corner Point Alphabet Detection

Table A.4 provides the total detected synthetic corner points at all pyramid levels; some detectors do not use pyramids. Note: for detectors that report features separately over image pyramid levels, individual pyramid-level detections are shown in Table A.5.

Each feature exists within a 14×14 pixel region, and the total number of features detected in each cell is provided in summary tables with the annotated images. Note that several features may be detected within each 14×14 cell, and the detectors often provide non-repeatable results, which are discussed at the end of this appendix.

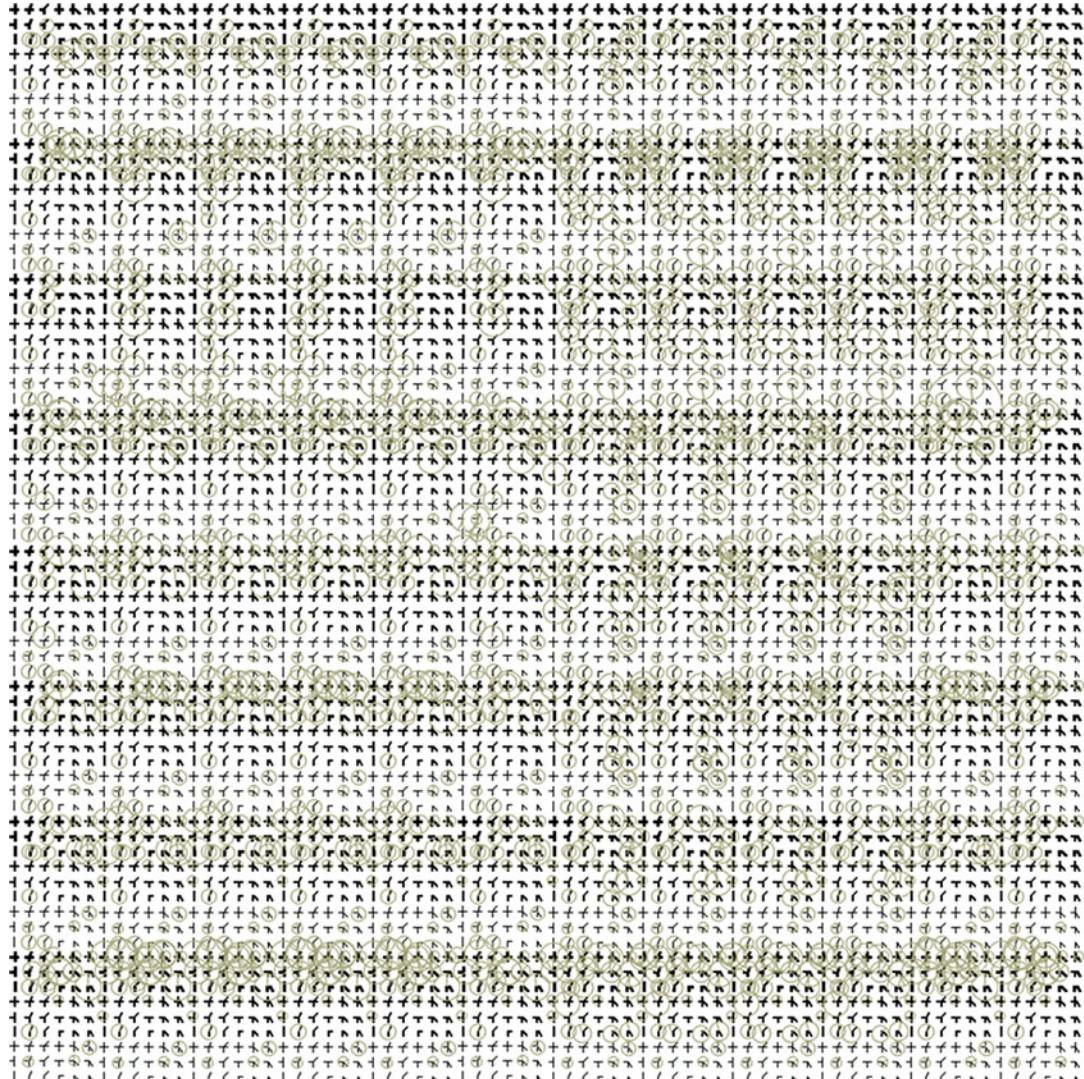


Figure A.5 Annotated BRISK detector results . NOTE: there are several non-repeatability anomalies

Annotated Synthetic Corner Point Detector Results

Test 2 is exactly like the interest point detector results in Test 1. As such, for ORB and SURF detectors, the annotated renderings using the *drawkeypoints()*function are too dense to be useful, but are included in the online test results.

The diameter of the circle drawn at each detected keypoint corresponds to the “diameter of the meaningful keypoint neighborhood,” according to the OpenCV KeyPoint class definition, which varies in size according to the image pyramid level where the feature was detected. Some detectors do not use a pyramid, so the diameter is always the same. The position of the detected features is normalized to the full resolution image, and all detected keypoints are drawn.

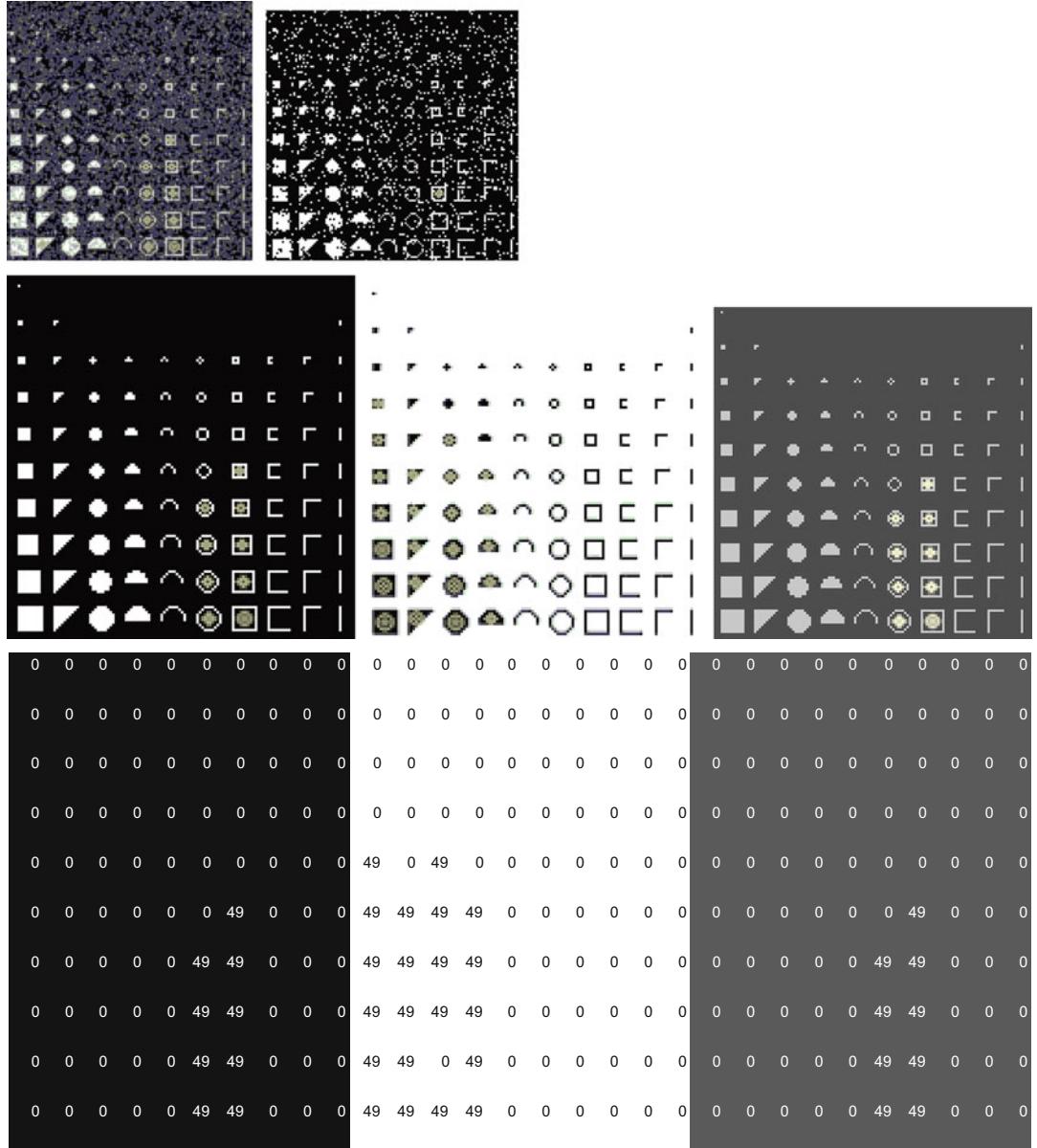


Figure A.6 SIMPLEBLOB detector, with results shown for a single alphabet grid set. (*Top row*) Gaussian and salt/pepper response. (*Middle row*) Black, white, and gray response. (*Bottom row*) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, *black*, *white* and *gray* images, color-coded tables

Entire Images Available Online

To better understand the detector results for each test, the entire image should be viewed to see the anomalies, such as where detectors fail to recognize identical patterns. Test results shown in Figs. A.16 through A.25 only show a portion of the images.

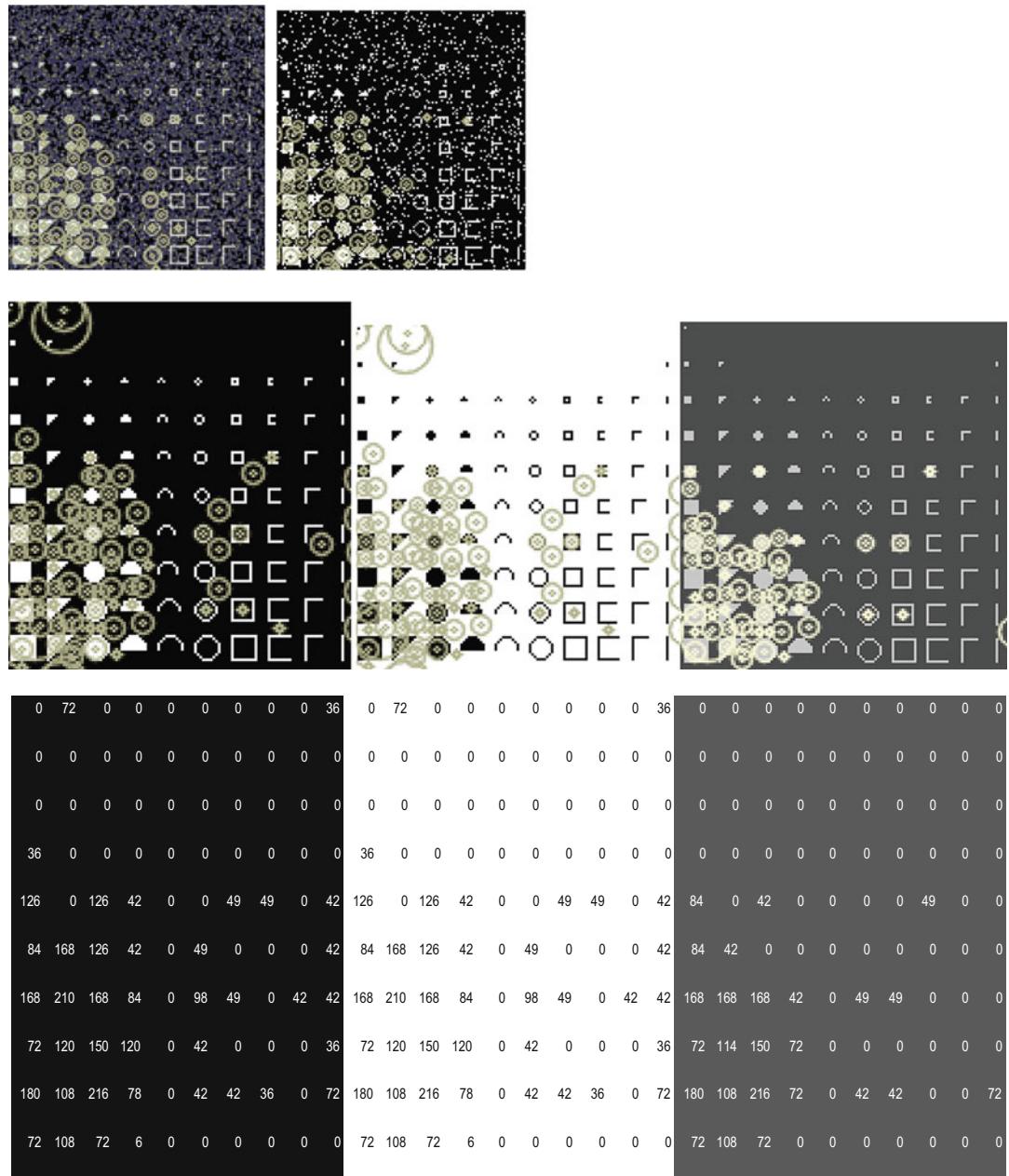


Figure A.7 STAR detector, with results shown for a single alphabet grid set. (Top row) Gaussian and salt/pepper response. (Middle row) Black, white, and gray response. (Bottom row) Summary count of individual alphabet feature detections across all the alphabets in the grid across each 1024 × 1024 image, black, white and gray images, color-coded tables

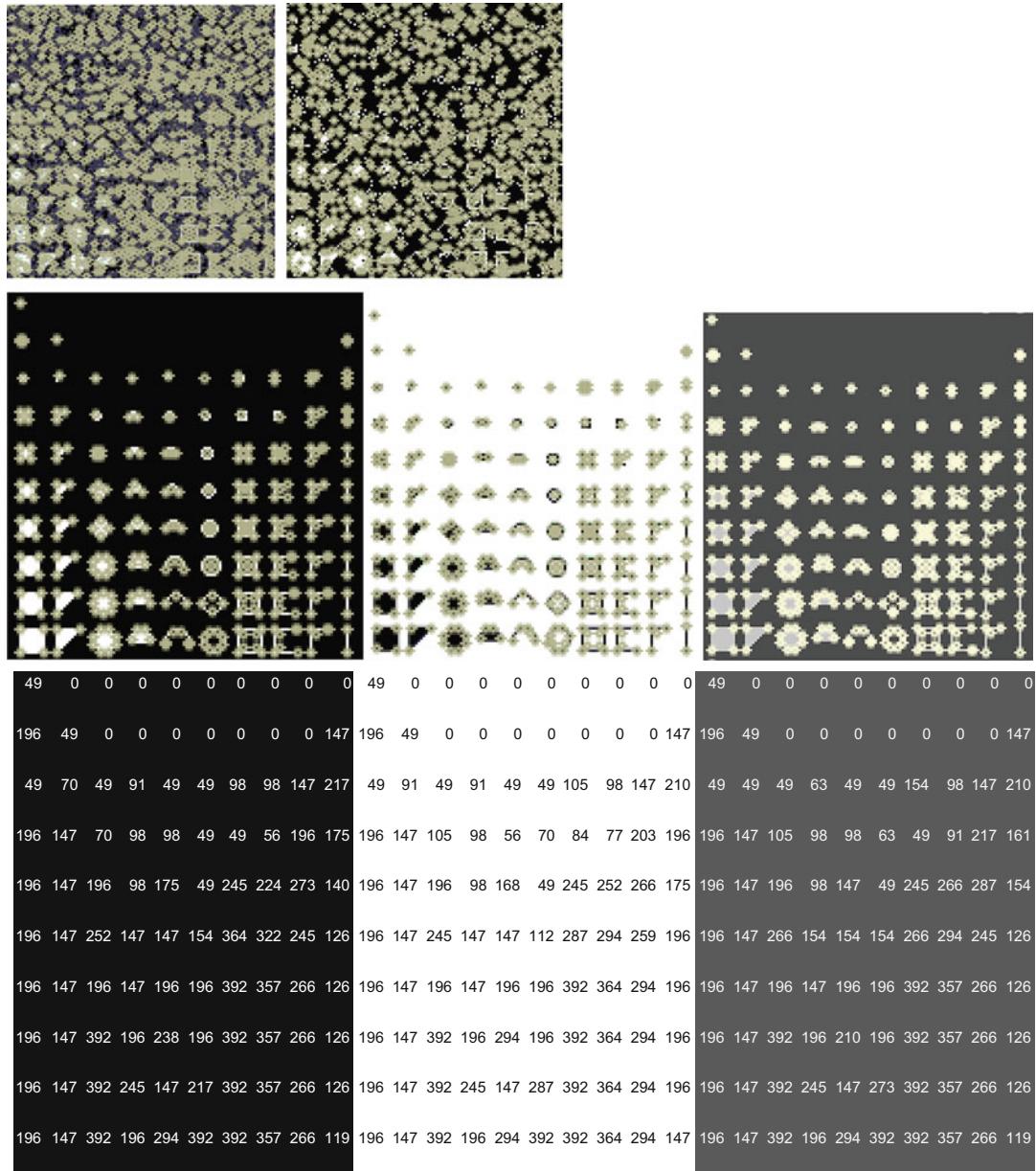


Figure A.8 GFFT detector, with results shown for a single alphabet grid set. (*Top row*) Gaussian and salt/pepper response. (*Middle row*) Black, white, and gray response. (*Bottom row*) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, black, white, and gray images, color-coded tables

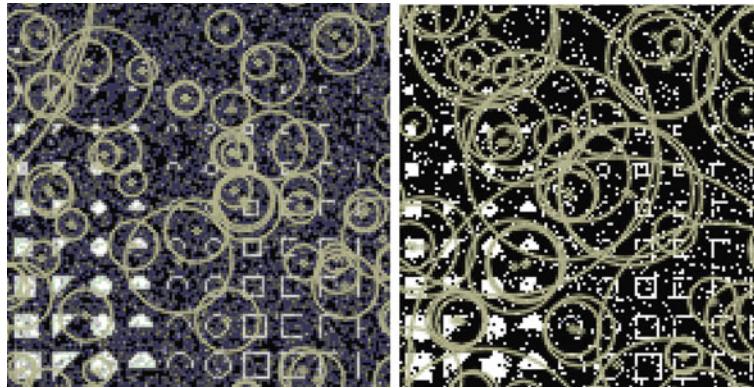


Figure A.9 MSER detector (*black on white, white on black, and light gray on dark gray have no detected features*)

150	52	0	0	0	0	0	7	17	156	54	0	0	0	0	0	0	8	21	93	25	0	0	0	0	0	0	3		
224	195	31	47	37	27	52	27	49	186	223	200	33	46	38	26	52	28	50	189	192	159	15	15	14	6	24	13	27	137
193	212	220	228	228	243	244	241	235	158	194	213	220	229	228	247	244	240	233	164	177	195	195	194	183	206	225	220	201	130
198	233	223	245	326	316	374	312	278	169	197	236	224	245	331	307	380	311	280	176	197	222	215	232	282	274	343	284	240	140
194	209	180	208	283	334	496	340	273	154	201	209	183	210	290	333	509	333	270	156	185	204	177	201	265	297	461	310	245	110
317	285	198	235	441	529	671	396	354	150	315	286	202	231	434	518	680	393	355	156	307	276	188	225	396	497	628	359	298	112
422	376	320	234	482	634	721	471	426	171	421	377	329	238	477	643	728	461	433	178	403	369	308	221	426	580	653	397	353	130
571	448	503	346	483	707	801	567	451	194	572	452	509	348	488	704	821	565	450	194	531	428	469	333	411	566	632	450	389	148
613	505	909	384	504	715	751	656	415	221	614	505	912	384	510	717	770	649	417	220	553	433	860	359	412	544	606	444	366	157
571	512	558	369	489	760	757	599	427	222	566	511	562	371	492	765	770	587	420	227	543	442	535	358	371	612	564	421	353	174

Figure A.10 ORB detector (annotations using default parameters not useful, images provided online), with results showing summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024 × 1024 image, *black*, *white*, and *gray* images, color-coded tables

Test 3: Synthetic Alphabets Overlaid on Real Images

Table A.6 provides the total detected synthetic features found in the test images of little girls, shown in Fig. A.3. Note that only the 0° version is used (no rotations), and both the black versions and the white versions of each alphabet are overlaid. In general, the white feature overlays produce more interest points and corner-point detections.

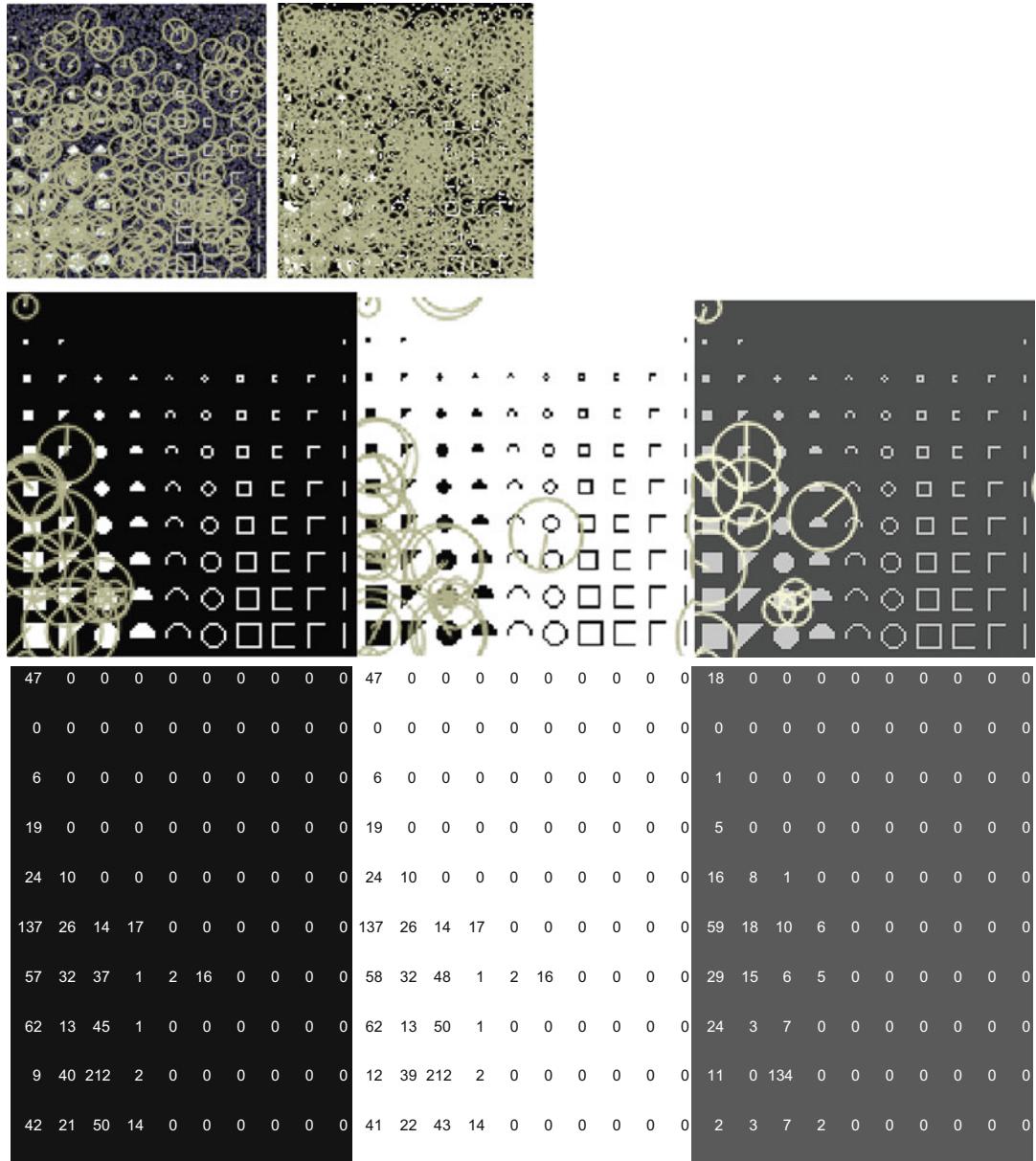


Figure A.11 BRISK detector, with results shown for a single alphabet grid set. (*Top row*) Gaussian and salt/pepper response. (*Middle row*) Black, white, and gray response. (*Bottom row*) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, black, white, and gray images, color-coded tables

Annotated Detector Results on Overlay Images

Annotated images are available online.

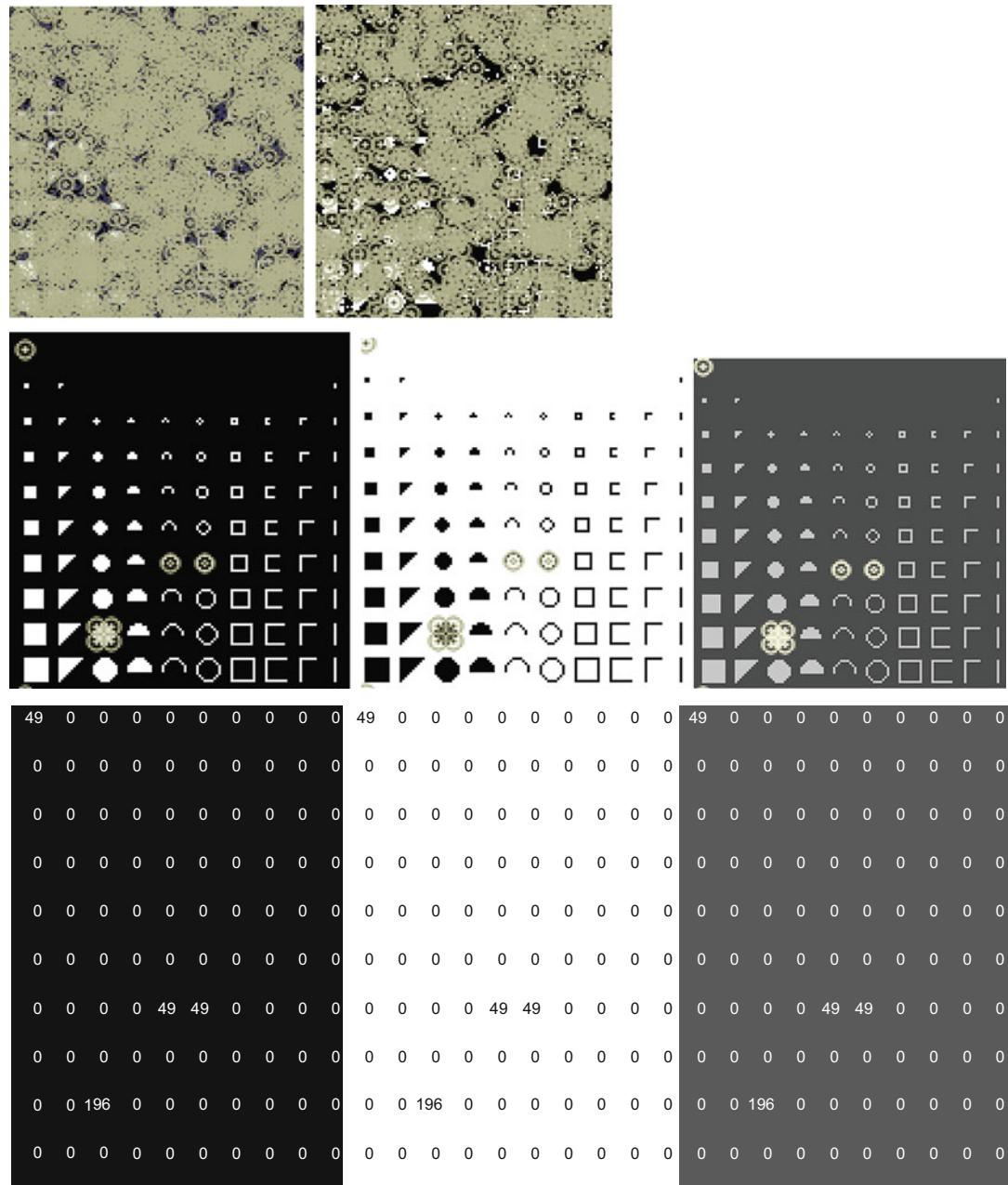


Figure A.12 FAST detector, with results shown for a single alphabet grid set. (*Top row*) Gaussian and salt/pepper response. (*Middle row*) Black, white, and gray response. (*Bottom row*) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, black, white, and gray images, color-coded tables

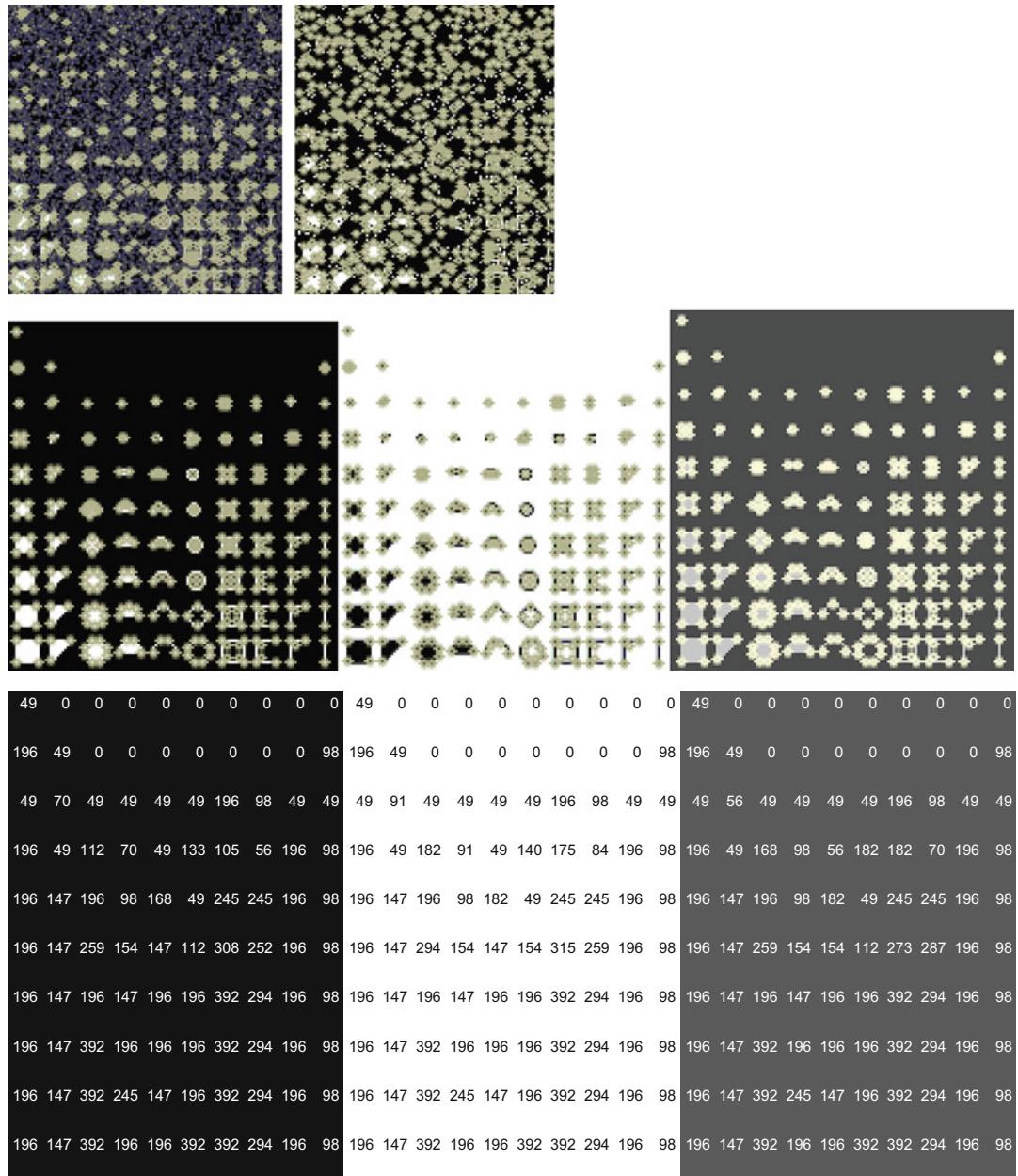


Figure A.13 HARRIS detector, with results shown for a single alphabet grid set. (*Top row*) Gaussian and salt/pepper response. (*Middle row*) Black, white, and gray response. (*Bottom row*) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024 × 1024 image, black, white, and gray images, color-coded tables

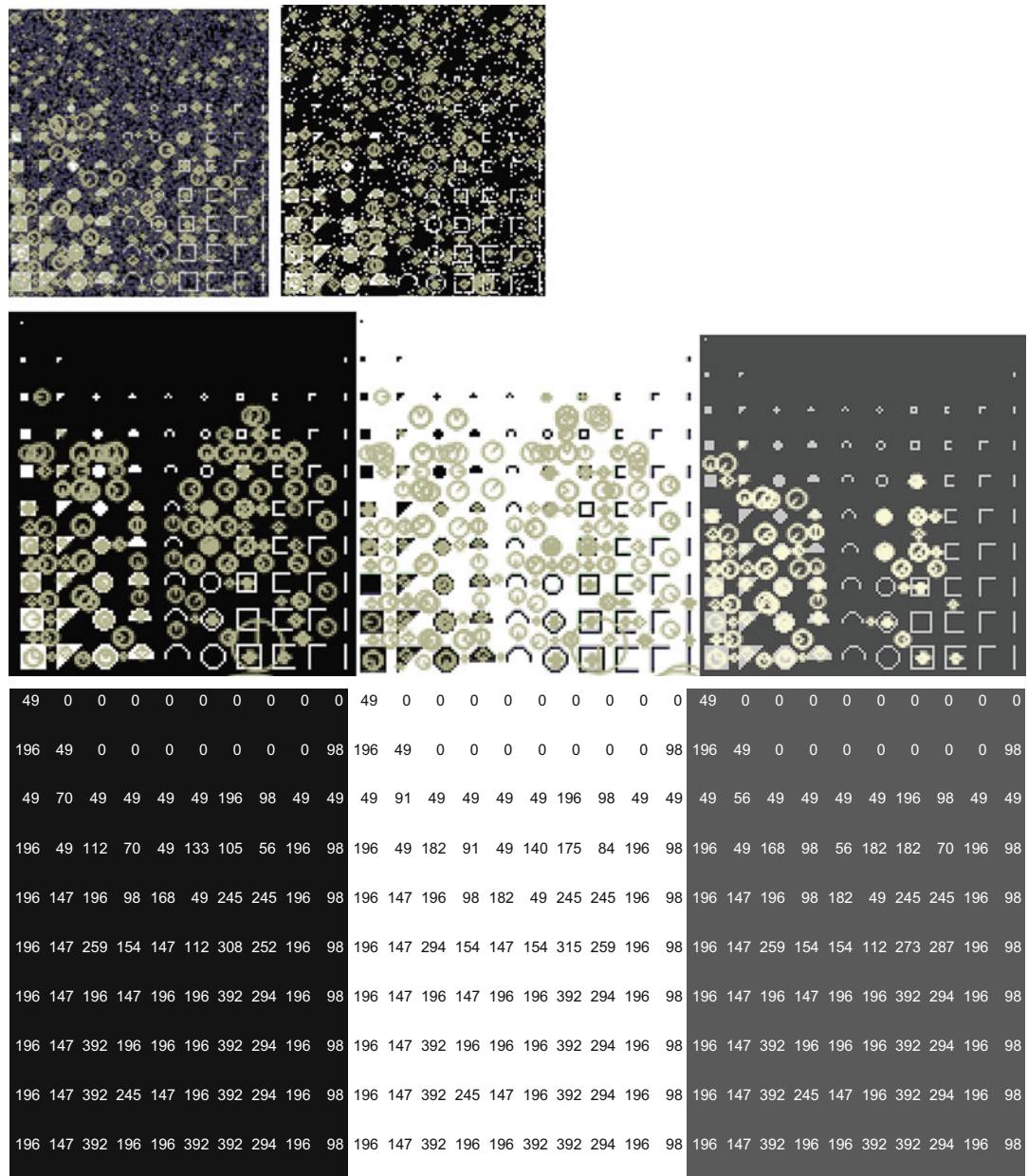


Figure A.14 SIFT detector, with results shown for a single alphabet grid set. (Top row) Gaussian and salt/pepper response. (Middle row) Black, white, and gray response. (Bottom row) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024 × 1024 image, black, white, and gray images, color-coded tables

98 0 126 0 84 49 49 0 91 42	98 0 126 0 84 49 0 0 85 42	42 0 42 0 84 49 0 0 85 0
` 134 212 133 0 98 156 188 201 162	251 134 212 133 0 98 156 188 201 162	196 0 36 42 0 0 58 6 54 84
140 140 196 183 147 163 165 110 238 126	140 140 196 183 147 163 165 110 238 126	98 98 98 134 49 58 116 110 232 42
182 189 245 183 150 98 98 147 127 126	182 189 245 183 150 98 98 147 127 126	140 147 196 98 98 49 0 49 49 84
140 231 245 189 245 245 196 203 287 168	140 231 245 189 245 245 196 203 287 168	140 231 245 147 196 147 98 147 287 168
133 182 245 189 245 196 98 196 343 210	133 182 245 189 245 196 98 245 343 210	91 182 245 189 196 98 49 98 287 126
224 231 294 196 245 245 238 343 301 168	224 231 294 147 196 245 238 343 301 168	140 231 294 98 245 196 147 147 287 168
140 241 98 165 245 147 245 244 238 84	140 241 98 165 245 147 343 293 238 84	140 241 98 165 196 98 196 55 147 84
140 176 281 147 147 294 441 378 245 84	140 176 281 147 196 245 441 378 245 84	140 176 281 147 196 196 441 336 203 84
98 196 98 0 196 245 287 441 350 6	98 196 98 0 196 245 287 441 350 6	98 196 98 0 196 245 238 294 294 0

Figure A.15 SURF detector (annotations using default parameters not useful, images provided online), with results showing summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, *black*, *white*, and *gray* images, color-coded tables

Table A.4 Summary count of detected features found in the synthetic interest point alphabet, 0° rotation

Detector	Corner Points White On Black	Corner Points Black On White	Corner Points White On Black Salt/ Pepper Noise	Corner Points White On Black Gaussian Filtered	Corner Points Lt. Gray on Dk. Gray
SURF	28579	28821	32637	26806	26406
SIFT	17996	17515	22377	28624	16122
BRISK	1852	2286	22472	12522	550
FAST	2112	2304	37283	51266	2112
HARRIS	28616	29210	45615	30868	29760
GFFT	32720	31578	51969	55069	32597
MSER	0	0	3751	2446	0
ORB	40162	40373	59549	58693	37665
STAR	5932	6178	5589	7473	4251
SIMPLEBLOB	0	96	1	1	0

Test 4: Rotational Invariance for Each Alphabet

This section provides results showing detector response as *rotational invariance* across the full $0\text{--}90^\circ$ rotated image sets of black, white, and gray alphabets. Key observations:

- **Black on white, white on black:** Rotational invariance is generally less using black and white images with the current set of detectors and parameters, mainly owing to (1) the maxima and minima values of 0×0 and 0xff used for pixel values, and (2) un-optimized detector tuning

Table A.5 Octave count of detected features found in the synthetic corner point alphabet, 0° rotation

Detector	Interest Points White On Black	Interest Points Black On White	Interest Points White On Black Salt Pepper Noise	Interest Points White On Black Gaussian Filtered	Interest Points Lt. Gray on Dk. Gray
SURF total:	28579	28821	32637	26806	26406
Octave 0:	16122	16217	20494	15402	16120
Octave 1:	2327	2315	2925	2008	1692
Octave 2:	9989	10141	9062	9297	8582
Octave 3:	141	148	156	99	12
BRISK total:	1852	2286	22472	12522	550
Octave 0:	1356	1223	21913	11686	426
Octave 1:	172	278	2	183	0
Octave 2:	324	727	535	644	124
Octave 3:	0	57	22	8	0
ORB total	40162	40373	59549	58693	37665
Octave 0:	1932	2105	13030	13030	1932
Octave 1:	6752	6653	10859	10859	6594
Octave 2:	9049	9049	9049	9049	9049
Octave 3:	6870	6920	7541	7541	6664
Octave 4:	4334	4343	6284	6284	4140
Octave 5:	4072	4181	5237	5010	3751
Octave 6:	3909	3919	4364	4080	3316
Octave 7:	3244	3203	3185	2840	2219

parameters. The detectors each seem to operate in a similar manner on images at orientations of 0° and 90° that contain no rotational anti-aliasing artifacts on each alphabet pattern; however, for the other rotations of 10–80°, pixel artifacts combine to reduce rotational invariance for these alphabet patterns—each detector behaves differently.

- **Light gray on dark gray:** Rotational invariance is generally better for the detectors using the reduced-range gray scale image alphabet sets using pixel values of 0 × 40 and 0xc0, rather than the full maxima and minima range used in the black and white image sets. The gray alphabet detector results generally show the most well-recognized alphabet characters under rotation. This may be due to the less pronounced local curvature of closer range gray values in the local region at the interest point or corner.

Methodology for Determining Rotational Invariance

The methodology for determining rotational invariance is illustrated in Figs. A.26 through A.28, and illustrated via pseudo-code as follows:

```

For (degree = 0; degree < 100; degree += 10)
    Rotate image (degree)
    For each detector (SURF, SIFT, BRISK, . . .):
        Compute interest point locations
        Annotate rotated image showing interest point locations
        Compute bin count (# of times) each alphabet feature is detected
        Create bin count image: pixel value = bin count for each alphabet character
    
```

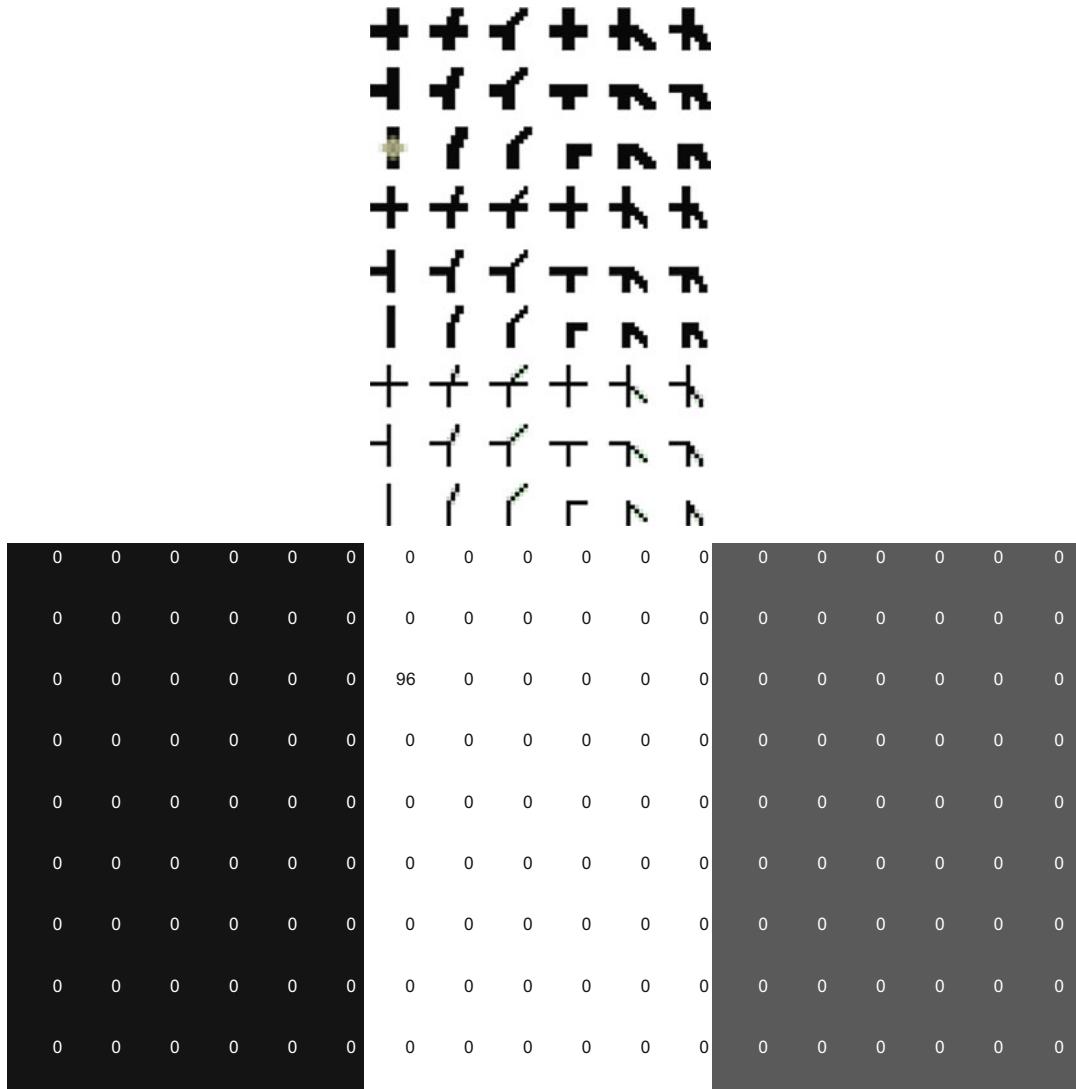
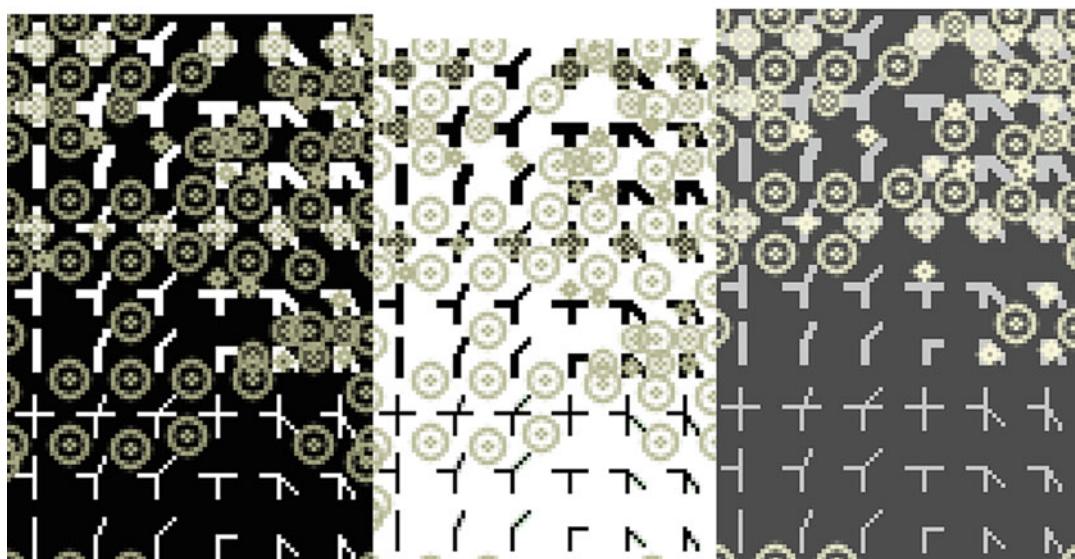
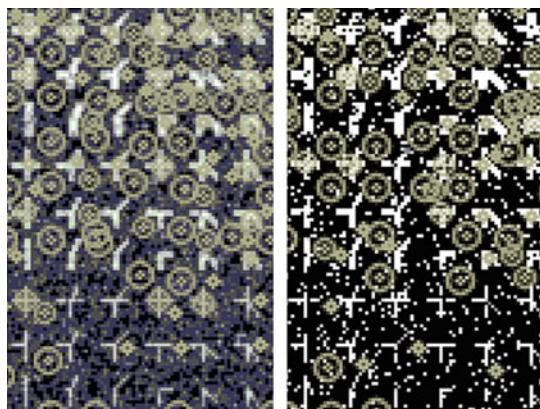


Figure A.16 SIMPLE BLOB detector (*black on white* is the only image with detected features), with results showing summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, *black*, *white*, and *gray* images, color-coded tables

Figures A.29 and A.30 show the summary bin counts of synthetic corner point detections across $0\text{--}90^\circ$ rotations. The ten columns in each image show, *left to right*, the $0\text{--}90^\circ$ rotated image final bin counts displayed as images.

Analysis of Results and Non-repeatability Anomalies

Complete analysis results are online, including annotated images showing detected keypoint locations and text files containing summary information on each detected keypoint.



154	147	20	108	158	279	127	138	30	163	183	224	154	147	20	108	158	273
231	142	50	70	238	308	220	143	70	70	247	265	231	140	50	70	238	308
82	82	70	305	154	226	104	126	80	295	134	199	82	77	70	165	154	226
154	147	105	185	154	159	113	133	85	220	165	202	154	147	75	185	154	159
232	70	40	210	77	103	205	83	110	210	88	71	155	70	0	70	0	103
34	101	27	100	308	231	12	40	32	33	265	282	0	0	0	0	154	154
46	39	78	40	20	77	42	72	76	77	148	92	0	0	0	0	0	0
74	70	28	0	71	126	74	73	89	0	46	140	0	0	0	0	0	0
0	0	0	0	0	2	31	0	0	0	0	51	0	0	0	0	0	0

Figure A.17 STAR detector, with results shown for a single alphabet grid set. (*Top row*) Gaussian and salt/pepper response. (*Middle row*) Black, white, and gray response. (*Bottom row*) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, black, white, and gray images, color-coded tables

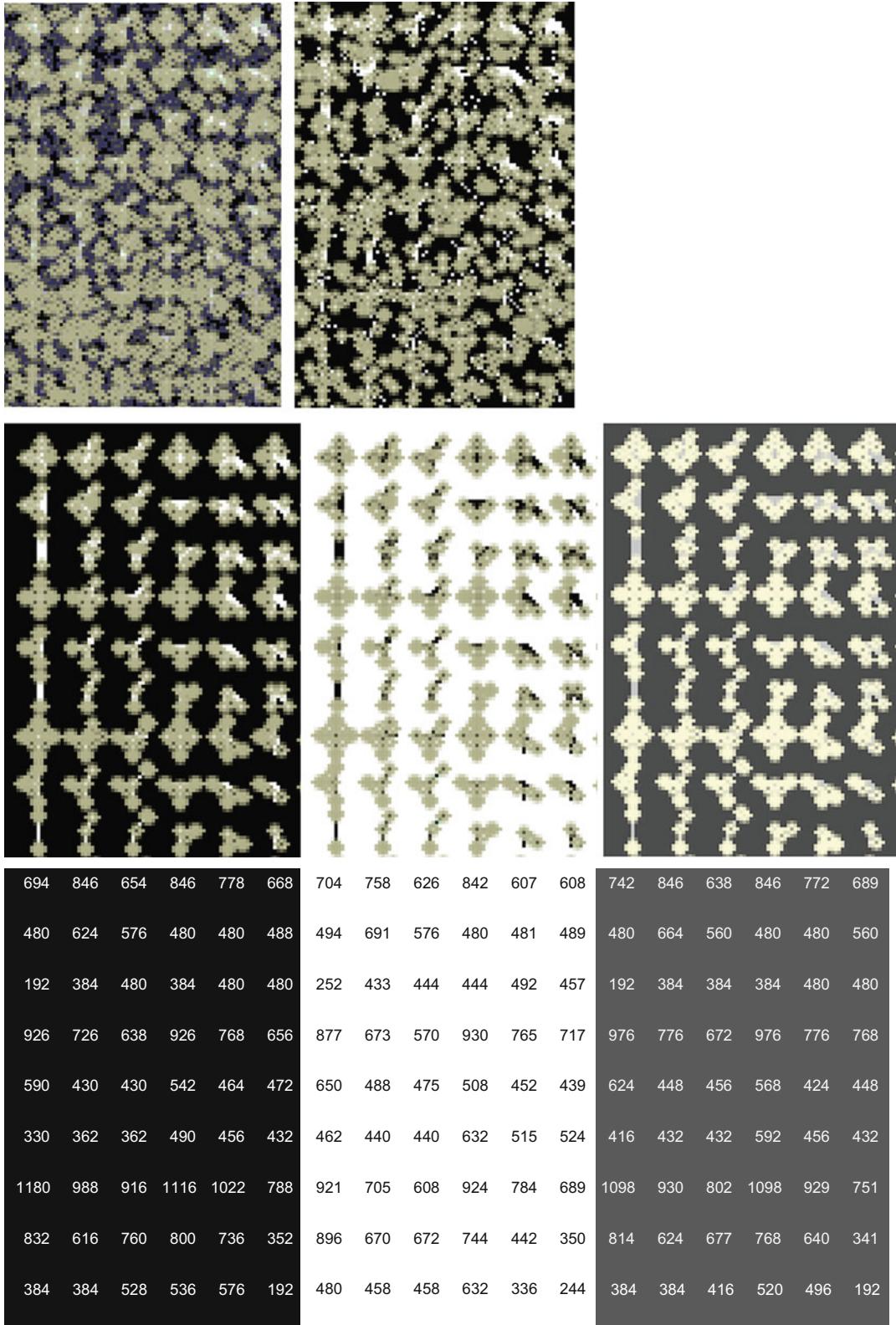


Figure A.18 GFFT detector, with results shown for a single alphabet grid set. (*Top row*) Gaussian and salt/pepper response. (*Middle row*) Black, white, and gray response. (*Bottom row*) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, black, white, and gray images, color-coded tables

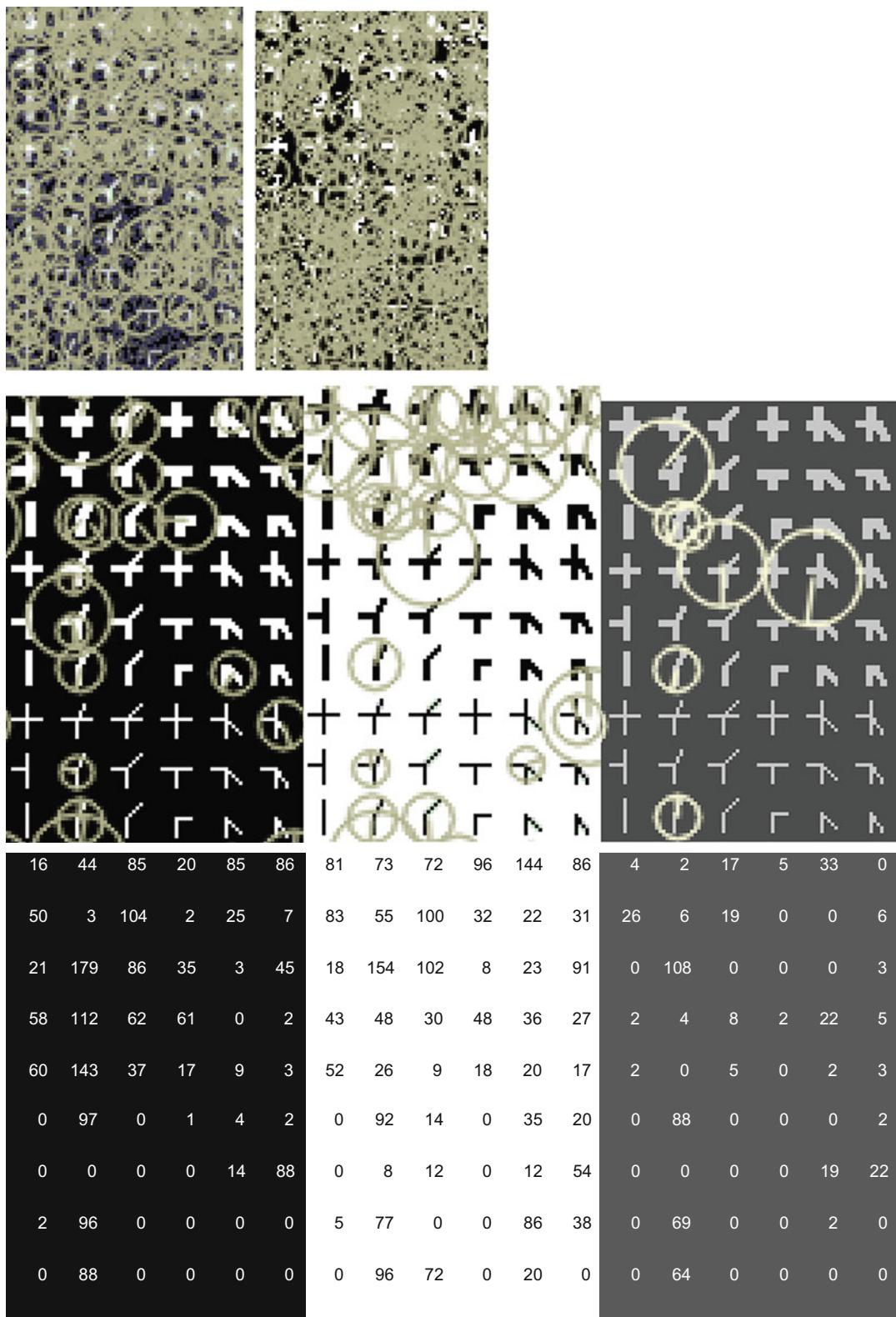


Figure A.19 BRISK detector, with results shown for a single alphabet grid set. (Top row) Gaussian and salt/pepper response. (Middle row) Black, white, and gray response. (Bottom row) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024 × 1024 image, black, white, and gray images, color-coded tables

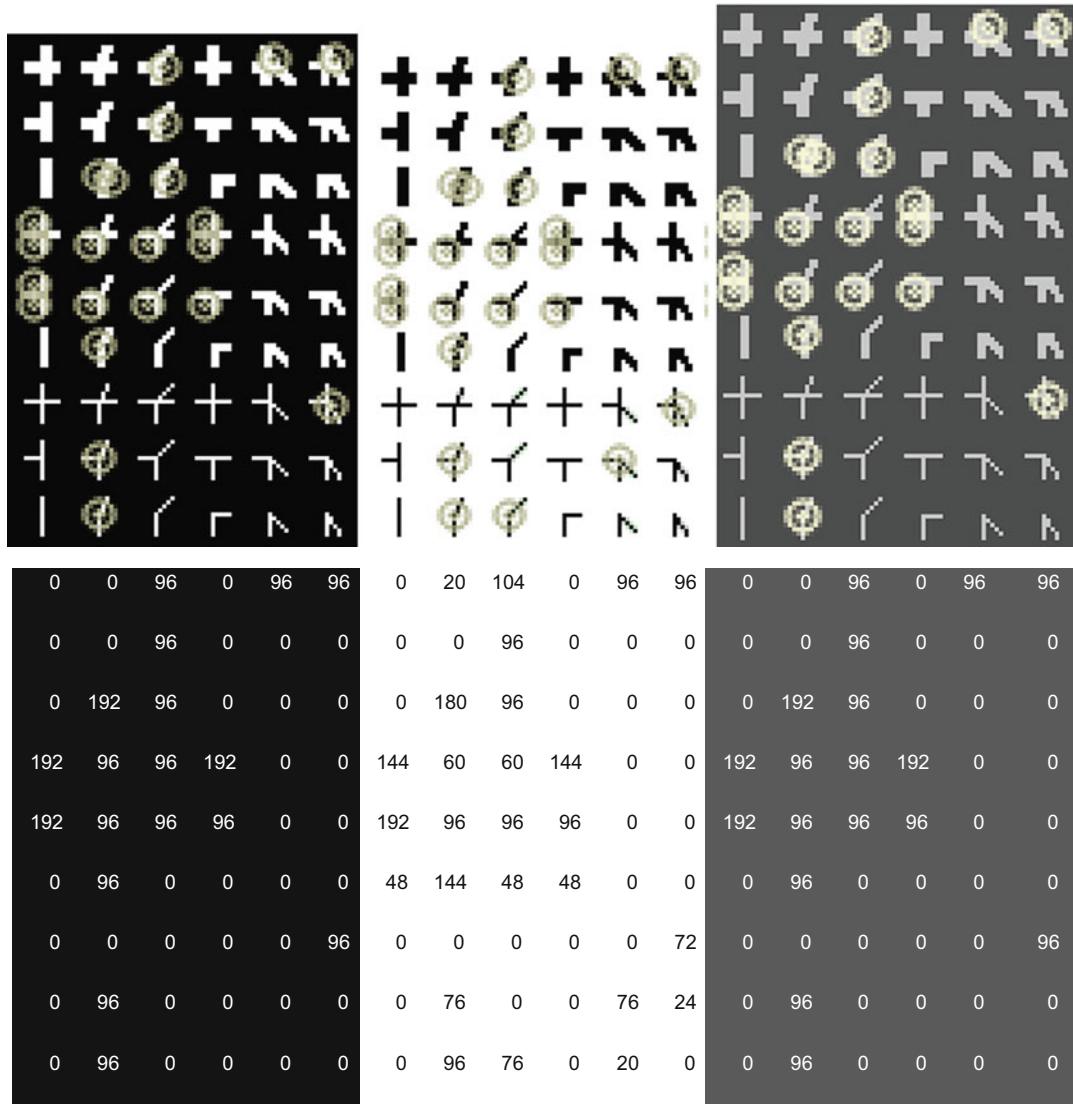
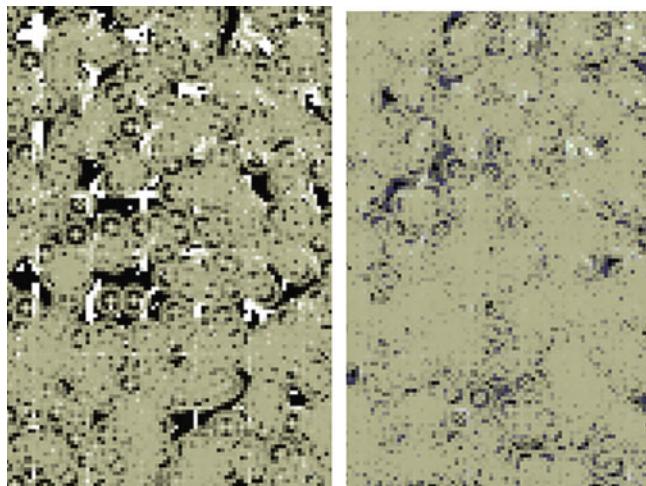


Figure A.20 FAST detector, with results shown for a single alphabet grid set. (*Top row*) Gaussian and salt/pepper response. (*Middle row*) Black, white, and gray response. (*Bottom row*) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, black, white, and gray images, color-coded tables

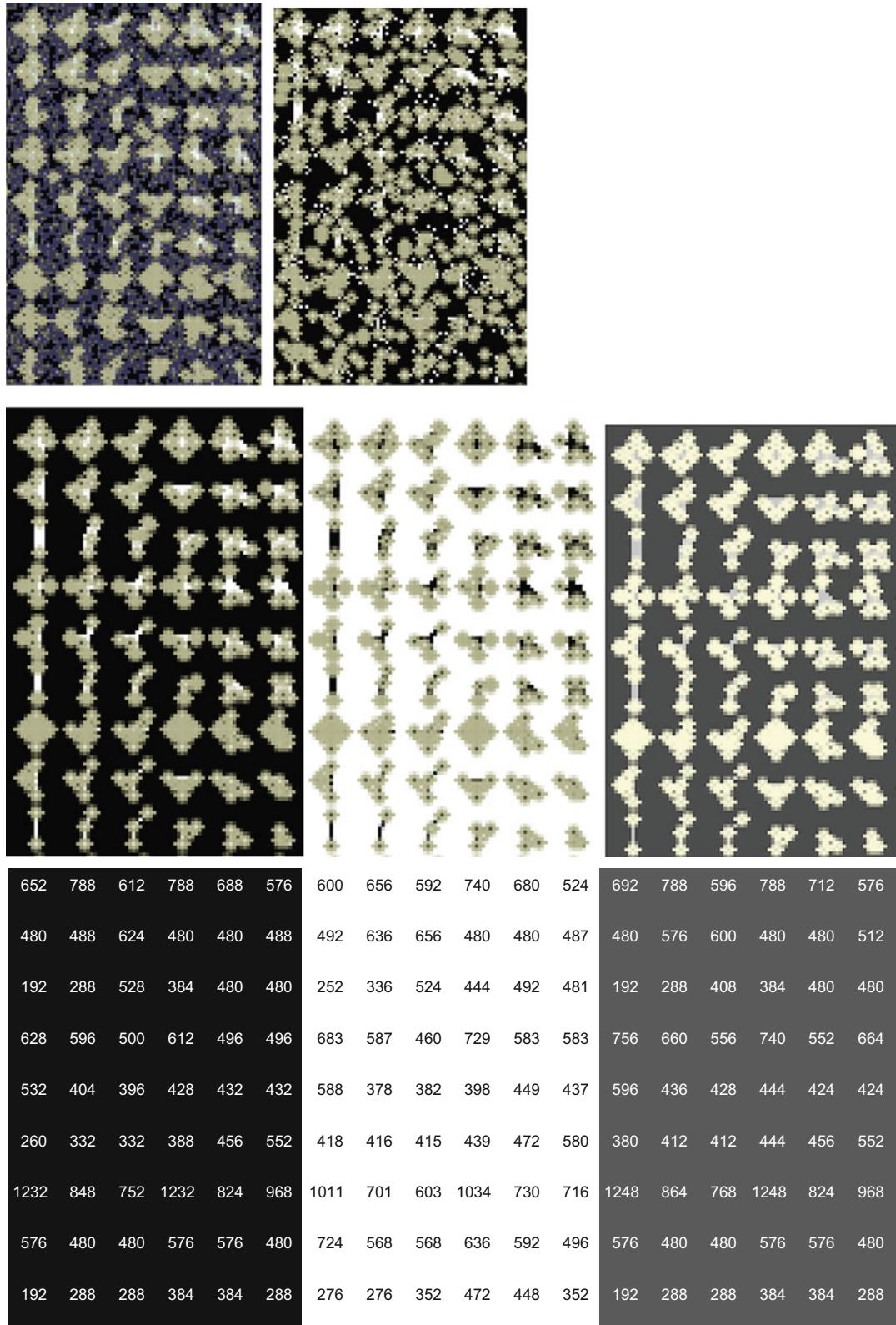


Figure A.21 HARRIS detector, with results shown for a single alphabet grid set. (*Top row*) Gaussian and salt/pepper response. (*Middle row*) Black, white, and gray response. (*Bottom row*) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, black, white, and gray images, color-coded tables

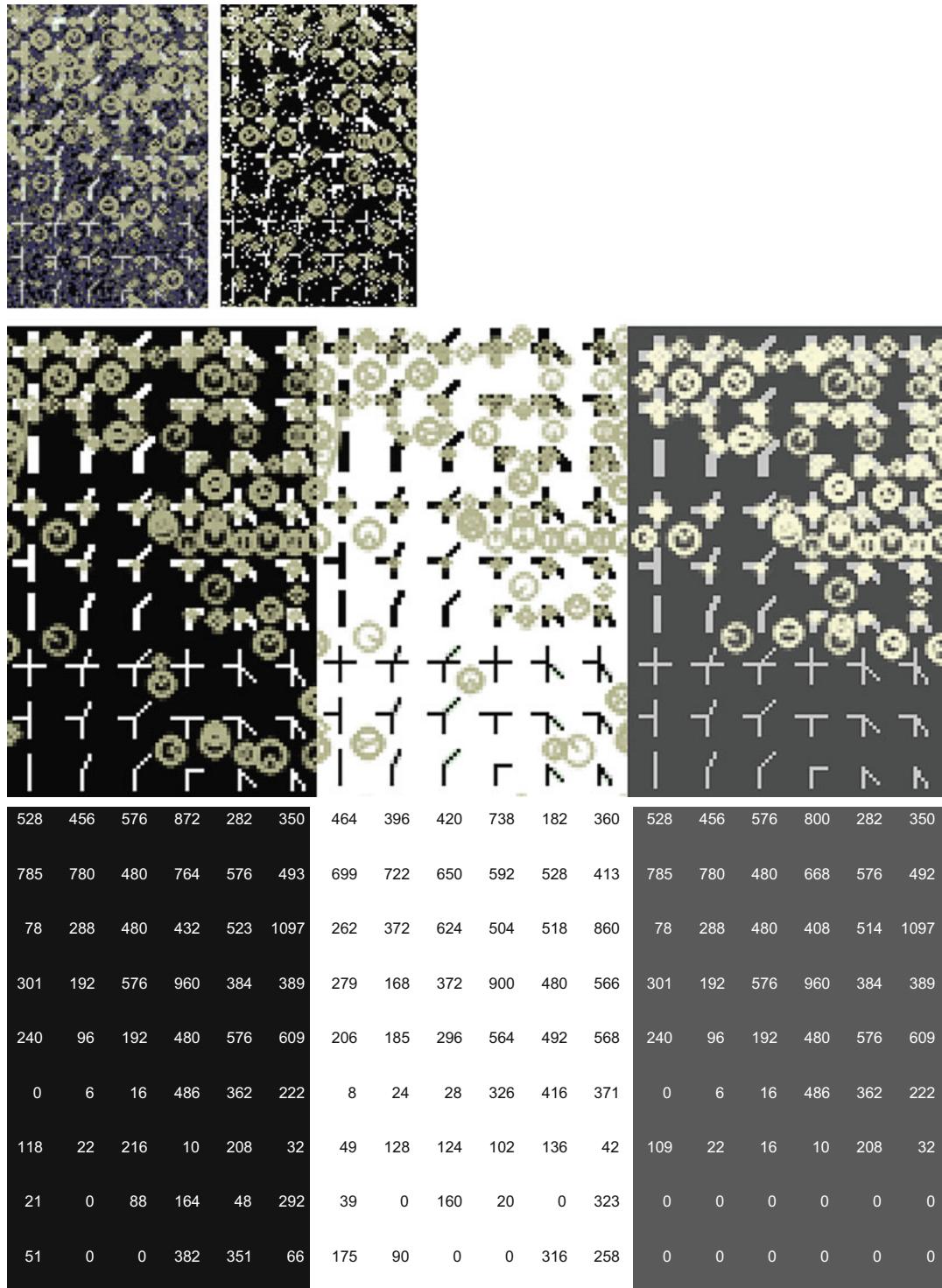


Figure A.22 SIFT detector, with results shown for a single alphabet grid set. (*Top row*) Gaussian and salt/pepper response. (*Middle row*) Black, white, and gray response. (*Bottom row*) Summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024×1024 image, black, white, and gray images, color-coded tables

523	516	433	574	381	468	516	531	502	634	422	458	520	485	384	509	381	463
664	361	428	607	523	484	627	391	486	597	465	444	664	361	427	607	518	401
346	368	375	487	525	388	369	410	426	503	497	451	344	368	371	487	525	385
690	470	622	672	442	505	574	512	558	611	509	578	681	414	526	643	442	505
692	348	654	750	417	351	688	385	601	886	445	344	580	294	512	650	417	351
513	647	436	426	451	434	441	559	466	401	466	515	395	606	333	379	348	377
704	567	501	566	591	577	676	650	619	537	544	548	694	551	461	524	530	418
829	660	738	620	565	492	860	662	643	595	596	523	750	562	651	543	612	346
540	404	697	531	504	522	584	454	653	441	507	461	520	381	677	530	494	509

Figure A.23 SURF detector (annotations using default parameters not useful, images provided online), with results showing summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024 × 1024 image, *black*, *white*, and *gray* images, color-coded tables

841	887	874	812	1160	1022	868	880	845	821	1114	986	811	833	820	781	1025	939
776	820	876	597	827	941	777	849	899	593	901	879	710	764	838	581	815	831
617	919	925	521	645	753	584	877	899	528	685	746	555	877	880	491	626	646
1012	898	1034	1044	1041	1037	993	943	1006	1064	1179	1014	981	866	1001	1012	987	988
862	851	890	696	889	843	865	792	891	672	768	735	835	830	871	684	885	820
473	856	783	454	633	722	563	923	887	584	802	828	471	834	746	416	622	697
510	683	793	498	608	784	471	714	792	479	603	808	492	642	754	483	588	750
425	673	591	378	606	678	385	608	562	352	663	726	358	604	518	331	565	640
391	686	594	423	466	544	341	616	617	403	465	528	275	566	487	340	399	504

Figure A.24 ORB detector (annotations using default parameters not useful, images provided online), with results showing summary count of individual alphabet feature detections across all the alphabets in the grid, across each 1024 × 1024 image, *black*, *white*, and *gray* images, color-coded tables

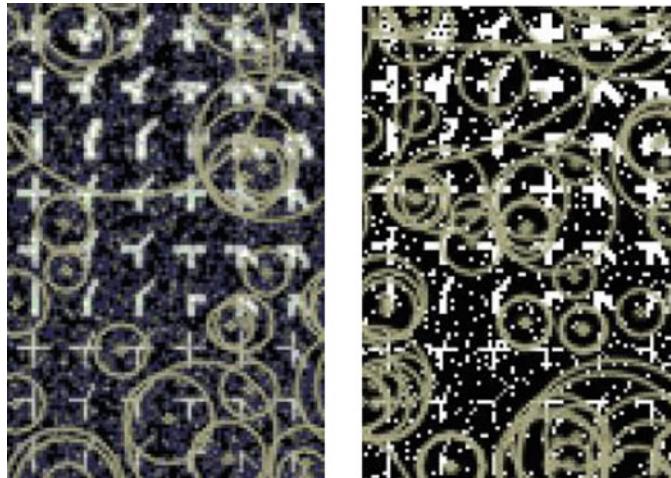


Figure A.25 MSER detector (*black on white, white on black, and light gray on dark gray have no detected features*)

Table A.6 Summary count of detected features found in the synthetic overlay images of little girls

Detector	Normal Image, no overlays	Black Corners Overlay	White Corners Overlay	Black Interest Points Overlay	White Interest Points Overlay
SURF	3945	16458	20809	10134	14196
SIFT	1672	12417	15347	8017	11551
BRISK	600	7919	10351	5914	8741
FAST	9026	25463	24952	17770	17995
HARRIS	475	9393	22201	4408	11097
GFFT	4474	23009	25120	11632	13872
MSER	1722	174	163	309	209
ORB	7325	53080	57016	41300	50946
STAR	477	3135	5558	2728	4756
SIMPLEBLOB	19	45	10	551	405

Caveats

There are deliberate reasons why each interest point detector is designed differently; no detector may be considered superior in all cases by any absolute measure. A few arguments against loosely interpreting these tests results are as follows:

- Unpredictability:** Interest point detectors find features that are often unpredictable from the human visual system standpoint, and they are not restricted by design into the narrow boundaries of synthetic interest points and corners points shown here. Often, the interest point detectors find features that a human would not choose.
- Pixel aliasing artifacts:** The aliasing artifacts affect detection and are most pronounced for the rotated images using maxima and minima alphabets, such as *black on white* or *white on black*, and are less pronounced for *light gray on dark gray* alphabets.

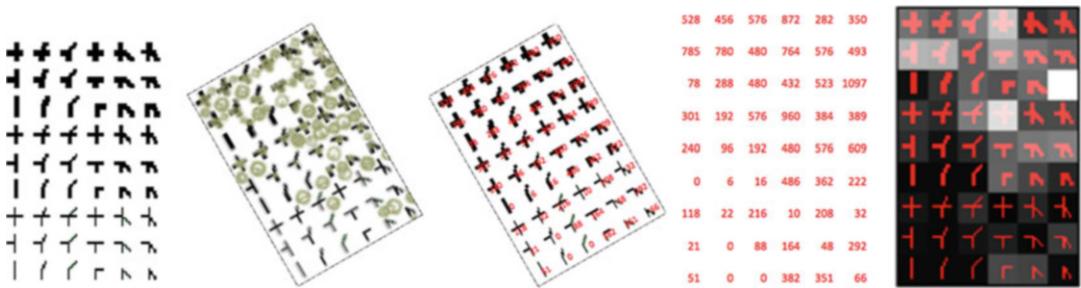


Figure A.26 Method of computing and binning detected alphabet features across rotated image sets, mocked-up SIFT data for illustration. (*Left*) original image. (*Center left*) Rotated image annotated with detected points. (*Center*) count of all detected points across entire image superimposed on alphabet cell regions. (*Center right*) Summary bin counts of detected alphabet features in grid cells. (*Right*) 2D histogram rendering of bin counts as an image; each pixel value is the bin count. Brighter pixels in the image have a higher bin count, meaning that the alphabet cell has a higher detection count

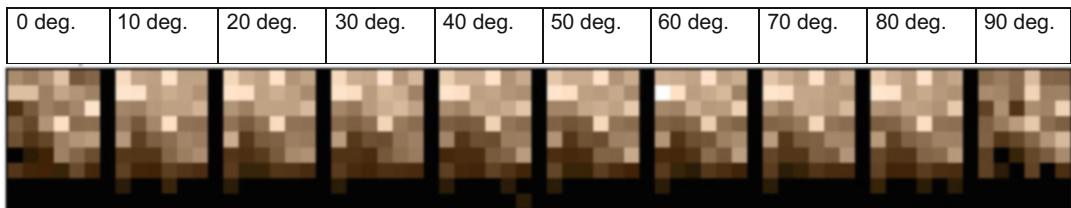


Figure A.27 Group of 10 SIFT gray scale corner alphabet feature detection results displayed as a 2D histogram image, sepia LUT applied, with pixel values set to the histogram bin values. The histogram for each rotated image is shown here: left image = 0° rotation; left-to-right sequence: $0, 10, 20, 30, 40, 50, 60, 70, 80, 90^\circ$ rotations. Note that the histogram bin counts are computed across the entire image, summing all detections of each alphabet feature

3. **Scale Space:** Not all the detectors use scale space, and this is a critical point. For example, SIFT, SURF, and ORB use a scale-space pyramid in the detection process. The scale-space approach filters out synthetic alphabet features that are not visible in some levels of a scale-space pyramid.
4. **Binary vs. scalar values:** FAST uses a binary value comparison to build up the descriptor, while other methods use scalar values such as gradients. Binary value methods, such as FAST, will detect the same feature regardless of polarity or gray value range; however, scalar detectors based on gradients are more sensitive to pixel value polarity and pixel value ranges.
5. **Pixel region size:** FAST uses a 7×7 patch to look for connected circle perimeter regions, while other features like SIFT, SURF, and ORB use larger pixel regions that bleed across alphabet grid cells, resulting in interest points being centered between alphabet features, rather than on them.
6. **Region shape:** Features such as MSER and SIMPLEBLOB are designed to detect larger connected regions with no specific shape, rather than smaller local features such as the interest point alphabets. An affine-invariant detector, such as SIFT, may detect features in an oval or oblong region corresponding to affine scale and rotation transformations, while a non-affine detector, such as FAST, may only detect the same feature as a template in a circular or square region with some rotational invariance at scale.
7. **Offset regions from image boundary:** Some detectors, such as ORB, SURF, and SIFT, begin detector computations at an offset from the image boundaries, so features are not computed across the entire image.
8. **Proven value:** Each detector method used here has proved useful and valuable for real applications.

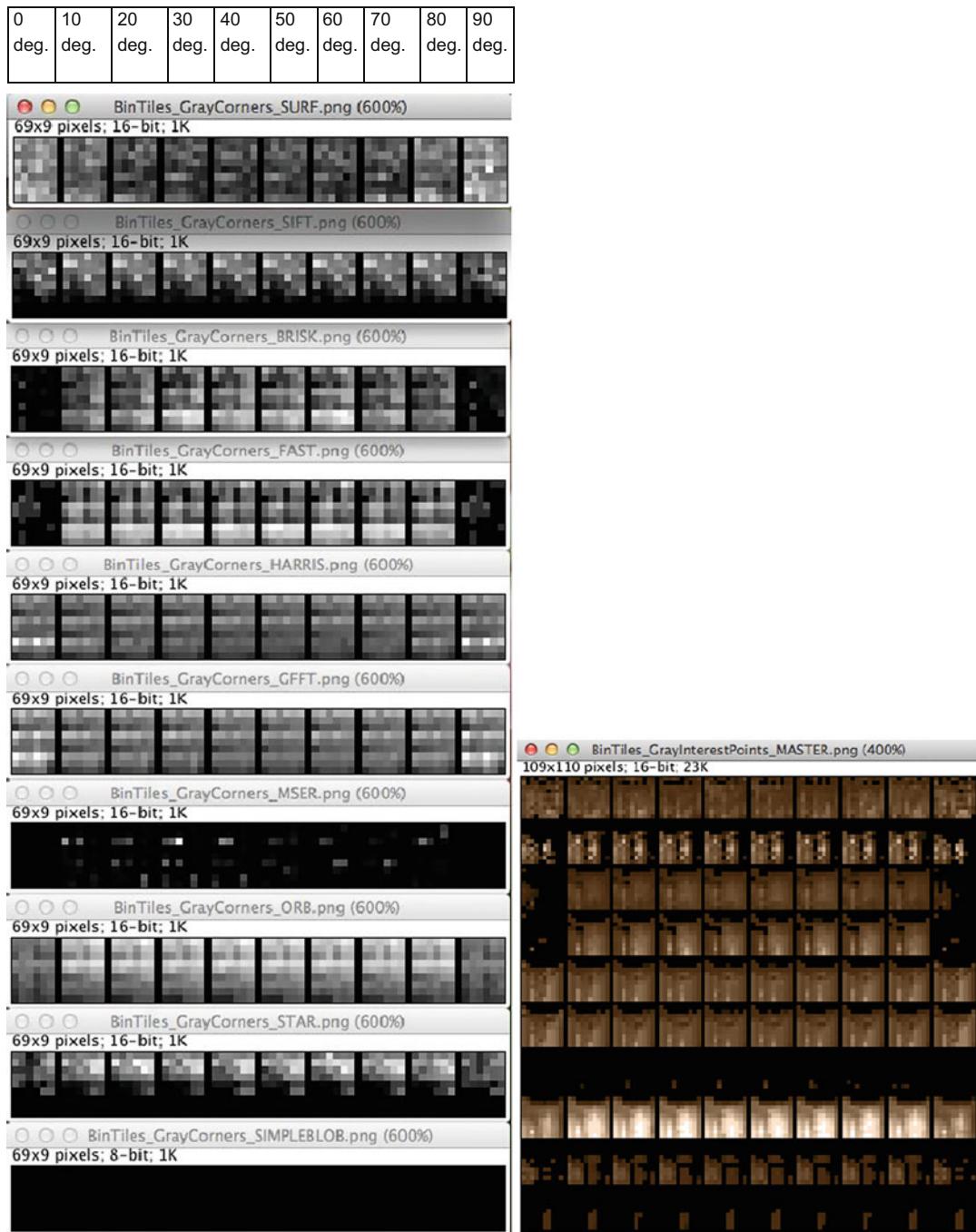


Figure A.28 (Left) Gray corner points 2D histogram bin images. *Left to right:* 0–90° rotations, gray scale LUT applied, and light gray on dark gray interest points alphabet 2D histogram binning image, contrast enhanced, sephia LUT applied

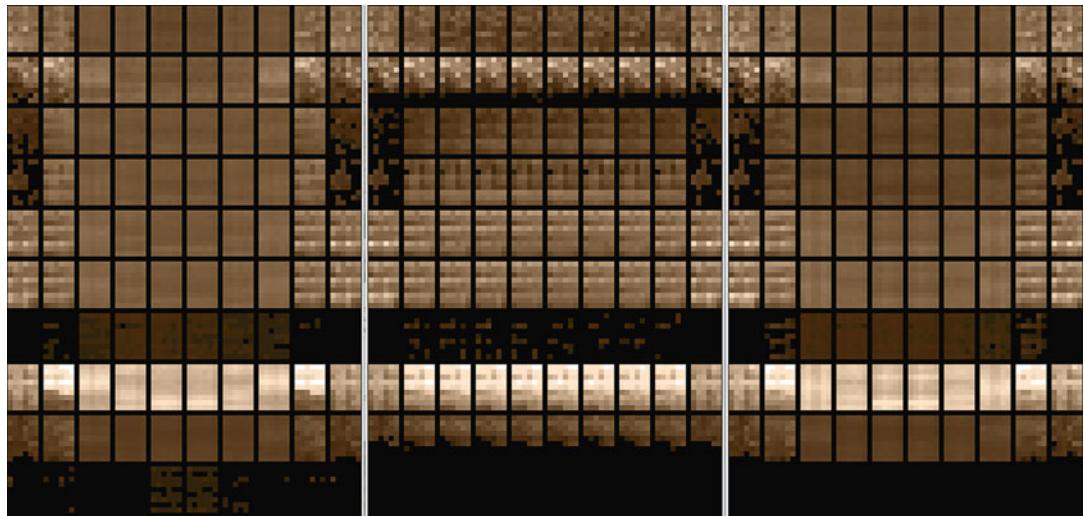


Figure A.29 Summary bin counts of detected corner alphabet features displayed as a set of 6×9 pixel images, where each pixel value is the bin count. (Left 10×10 image group) Black on white corners. (Center 10×10 image group) Light gray on dark gray corners. (Right 10×10 image group) White on black corners. Note that the gray alphabets are detected with the best rotational invariance. The columns are left to right $0\text{--}90^\circ$ rotations, and rows are top to bottom, SURF, SIFT, BRISK, FAST, HARRIS, GFFT, MSER, ORB, STAR, SIMPLEBLOB. Sepia LUT applied

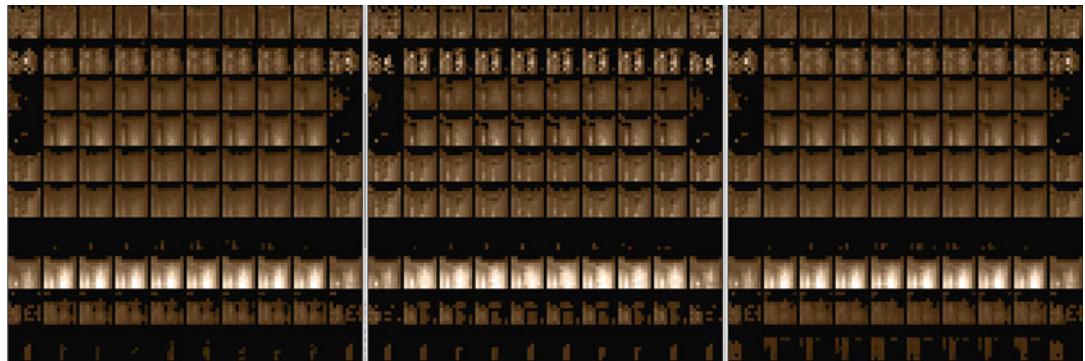


Figure A.30 Summary bin counts of detected interest point alphabet features displayed as a set of 10×10 pixel images, where each pixel value is the bin count. (Left 10×10 image group) Black on white corners. (Center 10×10 image group) Light gray on dark gray corners. (Right 10×10 image group) White on black corners. Note that the gray alphabets are detected with the best rotational invariance. The columns are left to right $0\text{--}90^\circ$ rotations, and rows are top to bottom, SURF, SIFT, BRISK, FAST, HARRIS, GFFT, MSER, ORB, STAR, SIMPLEBLOB. Sepia LUT applied

With these caveats in mind, the test results can be allowed to speak for themselves.

Non-repeatability in Tests 1 and 2

One interesting anomaly visible in Tests 1 and 2 appears in the annotated images, illustrating that detector results are not repeatable on the synthetic interest point and corner alphabets. In some cases,

the nonlinearity is striking; see the annotated images for Tests 1 and 2. The expectation of a human is that identical interest points should be equally well recognized. Here are some observations:

1. A human would recognize the same pattern easily whether or not the background and foreground are changed; however, some detectors do not have much invariance to extreme background and foreground polarity. The anomalies between detector behavior across white, black, and gray versions of the alphabets are less expected and harder to explain without looking deeper into each algorithm.
2. Some detectors compute over larger region boundaries than the 14×14 alphabet grid, so detectors virtually ignore the alphabet feature grid and use adjacent pieces of alphabet features.
3. Some detectors use scale space, so individual alphabet features are missed in some cases at higher scale levels, and detectors such as SIFT DoG use multiple scales together.

In summary, interest point detection and parameter tuning are analogous to image processing operators and their parameters: there are endless variations available to achieve the same goals. It is hoped that, by studying the test results here, intuition will be increased and new approaches can be devised.

Other Non-repeatability in Test 3

We note non-repeatability anomalies with Test 3 using little girl images with synthetic overlays, but there is less expectation of repeatability in this test. Some analysis of the differences between the positive (white) and negative (black) feature overlays can be observed in the annotated synthetic overlay images online.

Test Summary

Take-away analysis for all tests includes the following:

1. **Non-repeatability:** some non-repeatability anomalies detecting nearly identical features, differing only under rotation by local pixel interpolation artifacts. Some detectors also detect the *black*, *white* and *gray* alphabets differently.
2. **Gray level alphabets (lt. gray on dk.gray)** are detected generally most similar to human expectations. The results show that detectors, with the current tuning parameters, respond more uniformly across rotation with gray level patterns, rather than maxima black and white patterns.
3. **Real images overlaid with synthetic images tests** provide interesting information to develop intuition about detector behavior—for illustration purposes only.

Future Work

Additional analysis should include devising and using alternative alphabets suited for a given type of application, including a larger range of pixel sizes and scales, especially alphabets with closer gray level value polarity, rather than extreme maxima and minima pixel values. Detector tuning should also be explored across the alphabets.

Appendix B: Survey of Ground Truth Datasets

Table B.1 is a brief survey of public domain datasetsPublic domain datasets in various categories, in no particular order. Note that many of the public domain datasets are freely available from universities and government agencies.

Table B.1 Public domain datasets

Name	Labelme
Description	Annotated scenes and objects
Categories	Over 30,000 images; comprehensive; hundreds of categories, including car, person, building, road, sidewalk, sky, tree
Contributions	Open to contributions
Tools and apps	Labelme app for iPhone to contribute to database
Key papers	[59] [60]
Owner	MIT CSAIL
Link	http://new-labelme.csail.mit.edu/Release3.0/

Name	SUN
Description	Annotated scenes and objects
Categories	908 scene categories, 3,819 object categories, 13,1072 objects, and growing
Contributions	Open to contributions
Tools and apps	Image classifier source code + API, iOS app, Android app
Key papers	[62]
Owner	MIT CSAIL
Link	http://groups.csail.mit.edu/vision/SUN/

Name	UC Irvine Machine Learning Repository
Description	Very useful; huge repository of many categories of images
Categories	Too many to list; very wide range of categories, many attributes of the data are specifically searchable and designed into the ground truth datasets
Contributions	Ongoing
Tools and apps	Online assistant to search for specific ground truth datasets
Key papers	[532]
Link	http://archive.ics.uci.edu/ml/datasets.html

Name	Stanford 3D Scanning Repository
Description	High-resolution 3D scanned images with sub-millimeter accuracy, including XYZ and RGB datasets
Categories	Several scanned 3D objects with 3D point clouds, resolution ranging from 3,400,000 scanned point to 750,000 triangles and upwards
Link	http://graphics.stanford.edu/data/3Dscanrep/

Name	KITTI Benchmark Suite, Karlsruhe Institute of Technology
Description	Stereo datasets for various city driving scenes
Categories	KITTI benchmark suite covers optical flow, odometry, object detection, object orientation estimation; Karlsruhe sequences cover grayscale stereo sequences taken from a moving platform driving through a city; Karlsruhe objects cover grayscale stereo sequences taken from a moving platform driving through a city
Link	http://www.cvlibs.net/datasets/index.html

Name	Caltech Object Recognition Datasets
Description	Old but still useful; objects in hundreds of categories, some annotated with outlines
Categories	Over 256 categories, animals, plants, people, common objects, common food items, tools, furniture, more.
Key papers	[63]
Link	http://www.vision.caltech.edu/Image_Datasets/Caltech101/ http://www.vision.caltech.edu/Image_Datasets/Caltech256/ http://authors.library.caltech.edu/7694/ (latest versions of 101 and 256)
Name	Imagenet + Wordnet
Description	Labeled, annotated, bounding-boxed, and feature-descriptor marked images; over 14,197,122 images indexed into 21,841 sets of similar images, or synsets, created using sister app Wordnet
Categories	Categories include almost anything
Contributions	Images taken from Internet searches
Tools and apps	Online controls: http://www.image-net.org/download-API Source Code: ImageNet Large Scale Visual Recognition Challenge (ILSVRC2010) http://www.image-net.org/challenges/LSVRC/2010/index
Key papers	[64]; several see http://www.image-net.org/about-publication
Owner	Images have individual owners; website is © Stanford and Princeton
Link	http://www.image-net.org/ http://www.image-net.org/challenges/LSVRC/2012/
Name	Middlebury Computer Vision Datasets
Description	Scholarly and comprehensive datasets, and algorithm comparisons over most of the datasets
Categories	Stereo vision (excellent), multi-view stereo (excellent), MRF, Optical Flow (excellent), Color processing
Contributions	Algorithm benchmarks over the datasets can be submitted
Key papers	Several; see website
Owner	Middlebury College
Link	http://vision.middlebury.edu/
Name	ADL Activity Recognition Dataset
Description	Annotated scenes for activity recognition of common living scenes
Categories	Daily life
Tools and apps	Activity recognition code available (see link below)
Key papers	[65]
Link	http://deeptought.ics.uci.edu/ADLdataset/adl.html

Name	MIT Indoor Scenes 67, Scene Classification
Description	Annotated dataset specifically containing diverse indoor scenes
Categories	15,620 images organized into 67 indoor categories, some annotations in Labelme format
Key papers	[66]
Link	http://web.mit.edu/torralba/www/indoor.html
Name	RGB-D Object Recognition Dataset, U of W
Description	Dataset contains RGB and corresponding depth images
Categories	300 common household objects, 51 categories using Wordnet similar to Imagenet style (Imagenet dataset reviewed above), each object recorded in RGB and Kinect depth at various rotational angles and viewpoints
Key papers	[67]
Link	http://www.cs.washington.edu/rbgd-dataset/
Name	NYU Depth Datasets
Description	Annotated dataset of indoor scenes using RGB-D datasets + accelerometer data
Categories	Over 500,000 frames, many different indoor scenes and scene types, thousands of classes, accelerometer data, inpainted and raw depth information
Tools and apps	Matlab toolbox + g++ code
Key papers	[68]
Link	http://cs.nyu.edu/~silberman/datasets/nyu_depth_v2.html
Name	Intel Labs Seattle - Egocentric Recognition of Handled Objects
Description	Annotated dataset for egocentric handled objects using a wearable camera
Categories	Over 42 everyday objects under varied lighting, occlusion, perspectives; over 6GB total video sequence data
Key papers	[69, 70]
Link	http://seattle.intel-research.net/~xren/egovision09/
Name	Georgia Tech GTEA Egocentric Activities - Gaze(+)
Description	Annotated dataset for egocentric handled objects using a wearable camera
Categories	Many everyday objects under varied lighting, occlusion, perspectives
Tools and apps	Code library of vision functions and mathematical functions
Key papers	[71]
Link	http://www.cc.gatech.edu/~afathi3/GTEA_Gaze_Website/
Name	CUReT: Columbia-Utrecht Reflectance and Texture Database
Description	Extensive texture sample and illumination datasets directions
Categories	Over 60 different samples with over 200 viewing and illumination combinations, BRDF measurement database, more
Key papers	[72]
Link	http://www.cs.columbia.edu/CAVE/software/curet/

Name	MIT Flickr Material Surface Category Dataset
Description	Dataset for identifying material categories including fabric, glass, metal, plastic, water, foliage, leather, paper, stone, wood
Categories	Contains images of materials for surface property analysis, in contrast to object or texture analysis; 10 categories of materials + 100 images in each category
Key papers	[73]
Link	http://people.csail.mit.edu/celiu/CVPR2010/index.html
Name	Faces in the Wilds
Description	Collection of over 13,000 images of faces annotated with names of people
Categories	Faces
Key papers	[74]
Link	http://vis-www.cs.umass.edu/lfw/
Name	The CMU Multi-PIE Face Database
Description	Annotated face and emotion database with multiple pose angles
Categories	750,000 face images are taken over a period of several months for each of 337 subjects over 15 viewpoints and 19 illuminations, annotated facial expressions
Key papers	[75]
Link	http://www.multipie.org/
Name	Stanford 40 Actions
Description	People actions image database
Categories	People performing 40 actions, bounding-box annotations, 9,532 images, 180-300 images per action class
Key papers	[76]
Link	http://vision.stanford.edu/Datasets/40actions.html
Name	NORB 3D Object Recognition from Shape
Description	NYU object recognition benchmark
Categories	Stereo image pairs; 194,400 total images of 50 toys under 36 azimuths, 9 elevations, and 6 lighting conditions
Tools and apps	EABLEARN C++ learning and vision library, LUSH programming language, VisionGRader object detection tool http://www.cs.nyu.edu/~yann/software/index.html
Key papers	[77]
Link	http://www.cs.nyu.edu/~yann/research/norb/
Name	Optical Flow Algorithm Evaluation
Description	Tools and data for optical flow evaluation purposes
Categories	Many optical flow sequence ground truth datasets
Tools and apps	Tool for generating optical flow data, some optical flow code algorithms
Key papers	[78]
Link	http://of-eval.sourceforge.net/

Name	PETS Crowd Sensing Dataset Challenge
Description	Multi-sensor camera views composed into a dataset containing sequences of crowd activities
Categories	Challenge goals include crowd estimation, density, tracking of specific people, flow of crowd
Key papers	[86]
Link	http://www.cvg.rdg.ac.uk/PETS2009/a.html
Name	I-LIDS
Description	Security-oriented challenge ground truth dataset to enable competitive benchmarking including scenes for locating parked vehicles, abandoned baggage, secure perimeters, and doorway surveillance
Categories	Various categories in the security domain
Contributions	No, funded by UK government
Tools and apps	n.a.
Key papers	n.a.
Link	http://computervision.wikia.com/wiki/I-LIDS
Name	TRECVID, NIST, US Government
Description	NIST-sponsored public project spanning 2001-2013 for research in automatic segmentation, indexing, and content-based video retrieval
Categories	1. Semantic indexing (SIN) 2. Known-item search (KIS) 3. Instance search (INS) 4. Multimedia event detection (MED) 5. Multimedia event recounting (MER) 6. Surveillance event detection (SER), natural scenes, humans, vegetation, pets, office objects, more
Contributions	Annually by U.S. Government
Tools and apps	The Framework For Detection Evaluations (F4DE) tool, story evaluation tool, and others
Key papers	[87]
Link	http://www-nlpir.nist.gov/projects/trecvid/
Name	Microsoft Research Cambridge
Description	Pixel-wise labeled or segmented objects
Categories	Several hundred objects
Link	http://research.microsoft.com/en-us/projects/objectclassrecognition/
Name	Optical Flow Algorithm Evaluation
Description	Volume-rendered video scenes for optical flow algorithm benchmarking
Categories	Various scenes for optical flow; mainly synthetic sequences generated via ray tracing
Contributions	n.a.
Tools and apps	Yes, Tcl/Tk
Key papers	[88]
Link	http://of-eval.sourceforge.net/

Name	Pascal Object Recognition VOC Challenge Dataset
Description	Standardized ground truth data for a research challenge spanning 2005-2013 in the area of object recognition; competitions include classification, detection, segmentation, and actions over each of 20 classes of data
Categories	Consists of over 20 classes of objects in scenes including persons, animals, vehicles, indoor objects
Contributions	Via the Pascal conference
Tools and apps	Includes a developer kit and other useful software for labeling data and database access, and tools for reporting benchmarks results
Key papers	[89]
Link	http://pascallin.ecs.soton.ac.uk/challenges/VOC/

Name	CRCV
Description	Very extensive; University of Central Florida's Center for Research in Computer Vision hosts a large collection of research data covering several domains
Categories	Comprehensive set of categories (aerial views, ground views) including dynamic textures, multi-modal iPhone sensor ground truth data (video, accelerometer, gyro), several categories of human actions, crowd segmentation, parking lots, human actions, much more
Contributions	n.a.
Tools and apps	n.a.
Key papers	[90]
Link	http://vision.eecs.ucf.edu/datasetsActions.html

Name	UCB Contour Detection and Image Segmentation
Description	U.C. Berkeley Computer Vision group provides a complete set of ground truth data, algorithms, and performance evaluations for contour detection, image segmentation, and some interest point methods
Categories	500 ground truth images on natural scenes containing a wide range of subjects and labeled ground truth data
Contributions	n.a.
Tools and apps	Benchmarking code (<i>globalPB</i> for CPU and GPU)
Key papers	[91]
Link	http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html#bench

Name	CAVIAR Ground Truth Videos for Context-Aware Vision
Description	Project site containing labeled and annotated ground truth data of humans in cities and shopping centers, including 52 videos with 90K frames total including people in indoor office scenes and shopping centers
Categories	Both scripted and real-life activities in shopping centers and offices, including walking, browsing, meeting, fighting, window shopping, entering/exiting stores
Contributions	n.a.
Tools and apps	n.a.
Key papers	[92]
Link	http://homepages.inf.ed.ac.uk/rbf/CAVIAR/caviar.htm

Name	Boston University Computer Science Department
Description	Image and video database covering a wide range of subject categories
Categories	Video sequences for head tracking and sign language; some datasets are labeled; still images for hand tracking, multi-face tracking, vehicle tracking, more
Contributions	Anonymous FTP
Tools and apps	n.a.
Key papers	[93]
Link	http://www.cs.bu.edu/groups/ivc/data.php

Appendix C: Imaging and Computer Vision Resources

This appendix contains a list of some resources for computer vision and imaging, including commercial products, open-source projects, organizations, and standards bodies.

Commercial Products

Name	Matlab
Description	Industry standard math package with many scientific package options for various fields including imaging and computer vision. Includes a decent software development environment, providing add-on libraries for computer vision, image processing, visualization, more. Suited well for code development.
Library API	Extensive API libraries Internal to the SDE.
SDE	Includes software development environment for coding.
Open Source	Not for the product, but possibly for some code developed by users.
Link	http://www.mathworks.com/products/matlab/

Name	Mathematica
Description	Industry standard math package with many scientific package options for various fields, including image processing and computer vision. Excellent for creation of publication-ready visualizations and math notebooks. Add-on libraries for computer vision, image processing, visualization, more.
Library API	Extensive API libraries Internal to the SDE.
SDE	Includes a default function-based script development environment, and some code development add-ons.
Open Source	Not for the product, but possibly for code developed by users.
Link	http://www.wolfram.com/mathematica/

Name	Intel TBB, Intel IPP, Intel CILK++
Description	Intel provides libraries, languages, and compilers optimized for the IA instruction set. Intel TBB is a multi-threading library for single and multi-core processors, Intel IPP provides imaging and computer vision performance primitives optimized for IA and SIMD instructions and in some cases GPGPU, and Intel CILK++ is a language for writing SIMD/SIMT parallel code.
Library API	Extensive API libraries.
SDE	No, but Intel CILK++ is a programming language.
Open Source	No.
Link	http://software.intel.com/en-us/intel-tbb http://software.intel.com/en-us/intel-ipp

Open Source

Name	OpenCV
Description	Industry standard computer vision and image processing library, used worldwide by major corporations and others.
Library API	Extensive API library.
SDE	No.

Name	OpenCV
Open Source	BSD license.
Link	http://opencv.org/
Name	ImageJ - FIJI
Description	Application for image processing, visualization, and computer vision. Developed by the USG National Institutes of Health [484], available for public use. Extensive. FIJI is a distribution of ImageJ with many plug-ins submitted by the user community.
Library API	No.
SDE	No.
Open Source	Public domain use.
Link	http://rsbweb.nih.gov/ij/index.html http://rsb.info.nih.gov/ij/plugins/ http://fiji.sc/Fiji
Name	VLFeat
Description	C library containing a range of common computer vision algorithms for feature description, pattern matching, and image processing.
Library API	Extensive API library.
SDE	No.
Open Source	BSD license.
Link	http://vlfeat.org
Name	VTK
Description	C++ library containing a range of common image processing, graphics, and data visualization functions. Includes GUI widgets. VTK also provides consulting.
Library API	Extensive API library.
SDE	No.
Open Source	BSD license.
Link	http://vtk.org/
Name	Meshlab
Description	Application for visualizing, rendering, annotating, and converting 3D data meshes such as point clouds and CAD designs. Extensive. Uses the VCG library from ISTI – CNR.
Library API	No.
SDE	No.
Open Source	BSD license.
Link	http://meshlab.sourceforge.net/
Name	PfeLib
Description	Library for image processing and computer vision acceleration.
Library API	Yes.

Name	PfeLlib
SDE	No.
Open Source	No.
Link	See reference [477].

Name	Point Cloud Library (PCL)
Description	Extensive open-source library for dealing primarily with 3D point clouds, including implementations of many cutting-edge 3D descriptors from the latest academic research and visualization methods.
Library API	Yes.
SDE	No.
Open Source	Yes.
Link	http://pointclouds.org/downloads/ http://pointclouds.org/documentation/ http://docs.pointclouds.org/trunk/a02944.html

Name	Shogun Machine Learning Toolbox
Description	Library for machine learning and pattern matching. Extensive.
Library API	Yes.
SDE	No.
Open Source	GPL.
Link	http://shogun-toolbox.org/page/features/

Name	Halide High-Performance Image Processing Language
Description	C++ language classes optimized for SIMD, SIMD, and GPGPU.
Library API	Yes.
SDE	No.
Open Source	Open-source MIT license.
Link	http://halide-lang.org/

Name	REIN (Recognition INfrastructure) Vision Algorithm Framework
Description	Framework for computer vision in robotics; uses ROS operating system. See references [379, 485].
Library API	Yes.
SDE	No.
Open Source	Open-source MIT license.
Link	http://wiki.ros.org/rein

Name	ECTO –Graph Network Construction for Computer Vision
Description	Library for creating directed acyclic graphs of functions for computer vision pipelines, supports threading. Written in a C++/Python framework. Can integrate with OpenCV, PCL and ROS.
Library API	Yes.
SDE	No.

Name	ECTO –Graph Network Construction for Computer Vision
Open Source	Apparently.
Link	http://plasmodic.github.io/ecto/

Organizations, Institutions, and Standards

Microsoft Research http://academic.research.microsoft.com/	Microsoft Research has one of the largest staff of computer vision experts in the world, and actively promotes conferences and research. Provides several good resources online.
CIE http://www.cie.co.at/	International Commission on Illumination, abbreviated CIE after the French name, provides standard illuminant data for a range of light sources as it pertains to color science, as well as standards for the well-known color spaces CIE XYZ, CIE Lab and CIE Luv.
ICC http://www.color.org/index.xalter	International Color Consortium provides the ICC standard color profiles for imaging devices, as well as many other industry standards, including the sRGB color space for color displays.
CAVE Computer Vision Laboratory http://www.cs.columbia.edu/CAVE/	Computer Vision Laboratory at Columbia University, directed by Dr. Shree Nayar, provides world-class imaging and vision research.
RIT Munsel Color Science Laboratory http://www.rit.edu/cos/colorscience/	Rochester Institute of Technology Munsel Color Science Laboratory is among the leading research institutions in the area of color science and imaging, provides a wide range of resources, and has strong ties to industry imaging giants such as Kodak, Xerox, and others.
OPENVX KHRONOS http://www.khronos.org/openvx	OPENVX is a proposed standard for low-level vision primitive acceleration, operated with the KHRONOS standards group.
SPIE Society for Optics and Photonics <i>Journal of Medical Imaging</i> <i>Journal of Electronic Imaging</i> <i>Journal of Applied Remote Sensing</i> http://spie.org/	Interdisciplinary approach to the science of light, including photonics, sensors, and imaging; promotes conferences, publishes journals.
IEEE CVPR, Computer Vision and Pattern Recognition PAMI, Pattern Analysis and Machine Intelligence ICCV, International Conference on Computer Vision IP, Trans. Image Processing http://ieee.org	Society for publication of journals and conferences, including various computer vision and imaging topics.
CVF Computer Vision Foundation http://www.cv-foundation.org/	Promotes computer vision, provides dissemination of papers.
NIST – Image Group (USG) National Institute Of Standards http://www.nist.gov/itl/iad/ig/	Promotes computer vision and imaging grand challenges; covers biometrics standards, fingerprint testing, face, iris, multimodal testing, next-generation test bed.
I20 - Darpa information innovation office (USG) http://www.darpa.mil/Our_Work/I2O/Programs http://www.darpa.mil/OpenCatalog/index.html	Extensive array of computer vision and related program research for military applications. Some work is released to the public via the <i>OpenCatalog</i> .

Journals and Their Abbreviations

- CVGIP *Graphical Models/graphical Models and Image Processing/computer Vision, Graphics, and Image Processing*
 CVIU *Computer Vision and Image Understanding*
 IJCV *International Journal of Computer Vision*
 IVC *Image and Vision Computing*
 JMIV *Journal of Mathematical Imaging and Vision*
 MVA *Machine Vision and Applications*
 TMI—IEEE *Transactions on Medical Imaging*
-

Conferences and Their Abbreviations

- 3DIM International Conference on 3-D Imaging and Modeling
 3DPVT 3D Data Processing Visualization and Transmission
 ACCV Asian Conference on Computer Vision
 AMFG Analysis and Modeling of Faces and Gestures
 BMCV Biologically Motivated Computer Vision
 BMVC British Machine Vision Conference
 CRV Canadian Conference on Computer and Robot Vision
 CVPR Computer Vision and Pattern Recognition
 CVRMed Computer Vision, Virtual Reality and Robotics in Medicine
 DGCI Discrete Geometry for Computer Imagery
 ECCV European Conference on Computer Vision
 EMMCVPR Energy Minimization Methods in Computer Vision and Pattern Recognition
 FGR IEEE International Conference on Automatic Face and Gesture Recognition
 ICARCV International Conference on Control, Automation, Robotics and Vision
 ICCV International Conference on Computer Vision
 ICCV Workshops
 ICVS International Conference on Computer Vision Systems
 ICWSM International Conference on Weblogs and Social Media
 ISVC International Symposium on Visual Computing
 NIPS Neural Information Processing Systems
 Scale-Space Theories in Computer Vision
 VLSM Variational, Geometric, and Level Set Methods in Computer Vision
 WACV Workshop on Applications of Computer Vision
-

Online Resources

Name	CVONLINE
Description	Huge list of computer vision software and projects, indexed to Wikipedia
Link	http://homepages.inf.ed.ac.uk/rbf/CVonline/environ.htm

Name	Annotated Computer Vision Bibliography
Description	Huge index of links to computer vision topics, references, software, more
Link	http://www.visionbib.com/bibliography/contents.html

Name	NIST Online Engineering Statistics Handbook(USG)
Description	Handbook for statistics, includes examples and software
Link	http://www.itl.nist.gov/div898/handbook/

Name	The Computer Industry (David Lowe)
Description	Includes links to major computer vision and imaging product companies
Link	http://www.cs.ubc.ca/~lowe/vision.html

Artificial Intelligence and Computer Vision-Key Research

Dalle Molle Institute for Artificial Intelligence Research, Juergen Schmidhuber

The Courant Institute of Mathematical Sciences, Center for Neural Science, Yann LeCun

Department of Computer Science and Operations Research Canada Research Chair in Statistical Learning Algorithms, Yoshua Bengio and Geoffrey E. Hinton

Stanford Computer Science Department, Andrew Ng

Neuroscience Journals and Research

Nature—International weekly journal of science

Nature Reviews Neuroscience

Nature Neuroscience Journal

Brain (A journal of Neurology Oxford University)

Annals of Neurology

Behavioral and Brain Sciences

NeuroImage (Elsevier)

NeuroComputing (Elsevier)

Neuroscience (Elsevier)

Neuron (Elsevier)

The Journal of Neuroscience

European Journal of Neuroscience

PLOS Computational Biology

Neural Information Processing Systems (NIPS)

Vision Research (Elsevier)

Brain Research (Elsevier)

International Conference on Machine Learning

Journal of Cognitive Neuroscience

The Journal of Machine Learning Research

Selected Deep Learning Resources

TORCH open source code for machinelearning

<http://torch.ch>

FANN Fast Artificial Neural Network Library

<http://leenissen.dk/fann/wp/>

Minerva: deep learning toolkit for multi-GPU acceleratioib

<https://github.com/dmlc/minerva>

caffee—CNN deep learning open source

<http://caffe.berkeleyvision.org>

cuDNN—Optimized NVIDIA deep learning library, works with cafee

<https://developer.nvidia.com/cudnn>

DeepLearnToolbox—Matlab deep learning tools

<https://github.com/rasmusbergpalm/DeepLearnToolbox>

Neon—Python based deep learning library

<http://neon.nervanasys.com/docs/latest/index.html>

Graphlab Create—Machine learning toolkit

<https://dato.com/products/create/>

Appendix D: Extended SDM Metrics

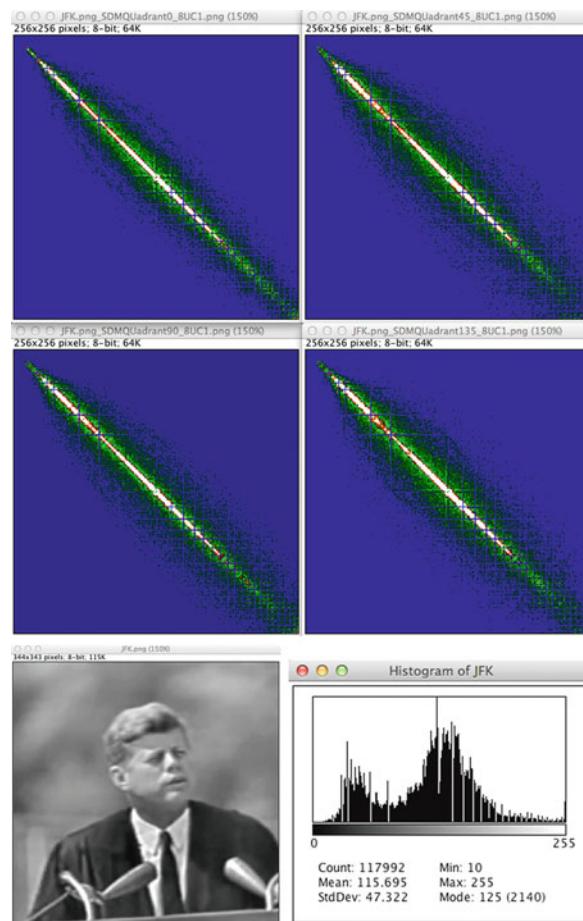


Figure D.1 SDM extended metrics

Listing D.1 illustrates the extended SDM metrics from Chap. 3. The code is available online at <http://www.apress.com/source-code/ComputerVisionMetrics>

Listing D.1 Extended SDM metrics from Chap. 3

```
/*
**  CREATED 1991 (C) KRIG RESEARCH, SCOTT KRIG - UNPUBLISHED SOFTWARE
**  PORTED TO MAC 2014
**
**          ALL RIGHTS RESERVED
**
**  THIS SOFTWARE MAY BE USED FREELY FOR ACADEMIC AND RESEARCH PURPOSES.
**  REFERENCE THIS BOOK AND PROVIDE THIS NOTICE WHEN USING THE SOFTWARE.
*/
using namespace std;
#include <math.h>
#include <stdio.h>
#include <opencv2/opencv.hpp>
#include "/usr/local/include/opencv/cv.h"
#include "/usr/local/include/opencv2/core/core.hpp"
#include "/usr/local/include/opencv2/highgui/highgui.hpp"
#include <iostream>

using namespace cv;
#define TINY 0.0000000001
#define F6U "%6f.3"
#define F6F "%.6f"
#define F3F "%.3f"
#define FXF "%.0f"
#define FALSE 0
#define TRUE 1

typedef struct area {
    int x;
    int y;
    int dx;
    int dy;
} area_t;

typedef struct {
    double t0;
    double t90;
    double t135;
    double t45;
    double tave;
} ctab;

typedef struct {
    double median;
    double ave;
    double adev;
    double sdev;
    double svar;
    double skew;
    double curt;
    int min;
    int max;
    ctab xcentroid;
    ctab ycentroid;
    ctab _asm;
}
```

```

ctab low_frequency_coverage;
ctab total_coverage;
ctab corrected_coverage;
ctab total_power;
ctab relative_power;
ctab locus_length;
ctab locus_mean_density;
ctab bin_mean_density;
ctab containment;
ctab linearity;
ctab linearity_strength;
ctab autocorrelation;
ctab covariance;
ctab inertia; /* haralick contrast */
ctab absolute_value;
ctab inverse_difference; /* haralick */
ctab entropy; /* haralick */
ctab correlation; /* haralick */
} glob_t;

glob_t gt;
/* FUNCTIONS */
int i_sort(
int *x,
int n,
int parm)
{
    int k,i,ii;
    int y,found;
    int xi;
    int n2, n2p;
    x--;
    for (k=1; k<n+1; k++) {
        y = x[k];
        for (i=k-1, found = FALSE; i>=0 && !found;) {
            xi = x[i];
            ii = i+1;
            if (y < xi) {
                x[ii] = xi;
                i--;
            } else {
                found = TRUE;
            }
        }
        x[ii] = y;
    }
    if (parm == 0) return 0;
    n2p = (n2=(n>>1))+1;
    return (n % 2 ? x[n2p] : (x[n2] + x[n2p]) >> 1);
}
int lmoment(
int *data,
int n,
double *median,
double *ave,
double *adev,

```

```

double *sdev,
double *svar,
double *skew,
double *curt)
{
    int j;
    double s,p,ep=0.0;
    if (n <= 1) return 0;
    s=0.0;
    for (j=1; j<=n;j++) s += (double)data[j];
    *ave=s/n;
    *adev=(*svar)=(*skew)=(*curt)=0.0;
    for (j=1;j<=n;j++) {
        *adev += abs(s=(double)data[j]-(*ave));
        *svar += (p=s*s);
        *skew += (p *= s);
        *curt += (p *= s);
    }
    *adev /= n;
    *svar = (*svar - ep*ep / n) / (n-1);
    *sdev=sqrt(*svar);
    if (*svar) {
        s = (n>(*svar)*(*sdev));
        if (s != 0) *skew /= s;
        else *skew = 0;
        s = (n>(*svar)*(*svar))-3.0;
        if (s != 0) *curt = (*curt) / s;
        else *curt = 0;
    } else {
        *skew = *curt = 0.0;
    }
    *median = 0;
    if (n > 20000) return 0;
    *median = (double)i_sort(data, n, 1);
    return 0;
}
int mean_sdev(
int xp,
int yp,
int *xdata,
double *xmean,
double *xsdev,
double *ymean,
double *ysdev)
{
    double u_x1, a_x1;
    int mx, my,v,t,x,y,z, offset;
    int dif[256];
    /* first calculate mean */
    offset = 256 * yp;
    x = y = 0;
    for (z=0; z < 256; x += xdata[offset+z], z++);
    for (z=0; z < 256; y += xdata[xp + (z*256)], z++);
    mx = x / 256.;
    *xmean = (double)mx;
    my = y / 256.;
```

```
*ymean = (double)my;
/* now calculate standard deviation */
x = y = 0;
z=0;
while (z < 256) {
    v = mx - xdata[offset+z];
    x += v*v;
    v = my - xdata[xp + (z*256)];
    y += v*v;
    z++;
}
*xsdev = x / 256;
*ysdev = y / 256;
return 0;
}
int lohi(
int n,
int *cv,
int *lo,
int *hi)
{
    int x;
    int lv, hv;
    lv = 0x1fffff;
    hv =0;
    x=0;
    while (x < n) {
        if (cv[x] < lv) lv = cv[x];
        if (cv[x] > hv) hv = cv[x];
        x++;
    }
    *lo = lv;
    *hi = hv;
    return 0;
}
int savegt(
ctab *ctp,
double dv1,
double dv2,
double dv3,
double dv4)
{
    ctp->t0 = dv1;
    ctp->t90 = dv2;
    ctp->t135 = dv3;
    ctp->t45 = dv4;
    ctp->tave = (dv1 + dv2 + dv3 + dv4) / 4;
    return 0;
}
int gtput(
char *prompt,
char *fs,
ctab *ctp,
FILE *fstream)
{
    char str[256];
```

```

char form[256];
fputs(prompt, fstream);
sprintf(form, "%s %s %s %s %s \n", fs, fs, fs, fs, fs);
sprintf(str, form, ctp->t0, ctp->t90, ctp->t135, ctp->t45, ctp->tave);
fputs(str, fstream);
return 0;
}
int put_txfile(
FILE *fstream)
{
    char str[256];
    sprintf(str, "gray value moments: min:%u max:%u mean:%u\n", gt.min, gt.max, (int)gt.ave);
    fputs(str, fstream);
    sprintf(str, "moments: adev:%.4f sdev:%.4f svar:%.4f skew:%.6f curt:%.6f \n",
        gt.adev, gt.sdev, gt.svar, gt.skew, gt.curt);
    fputs(str, fstream);
    fputs("\n", fstream);
    fputs(" ----- \n", fstream);
    fputs(" 0deg 90deg 135deg 45deg ave\n", fstream);
    fputs(" ----- \n", fstream);
    gtput("xcentroid ", FXF, &gt.xcentroid, fstream);
    gtput("ycentroid ", FXF, &gt.ycentroid, fstream);
    gtput("low_frequency_coverage ", F3F, &gt.low_frequency_coverage, fstream);
    gtput("total_coverage ", F3F, &gt.total_coverage, fstream);
    gtput("corrected_coverage ", F3F, &gt.corrected_coverage, fstream);
    gtput("total_power ", F3F, &gt.total_power, fstream);
    gtput("relative_power ", F3F, &gt.relative_power, fstream);
    gtput("locus_length ", FXF, &gt.locus_length, fstream);
    gtput("locus_mean_density ", FXF, &gt.locus_mean_density, fstream);
    gtput("bin_mean_density ", FXF, &gt.bin_mean_density, fstream);
    gtput("containment ", F3F, &gt.containment, fstream);
    gtput("linearity ", F3F, &gt.linearity, fstream);
    gtput("linearity_strength ", F3F, &gt.linearity_strength, fstream);
    return 0;
}
int texture(
char *filename)
{
    char str[256];
    int pmx[256], pmy[256];
    int x,y,z,dx,dy,dz,sz,bpp;
    int accum, tmin, tmax;
    int tmin2, tmax2, yc;
    int *data;
    int mval0, mval90, mval135, mval45;
    double median, ave, adev, sdev, svar, skew, curt;
    double median2, ave2, adev2, sdev2, svar2, skew2, curt2;
    int *dm0, *dm90, *dm135, *dm45;
    FILE *fstream;
    int i0, i90, i135, i45, iave, n;
    int c0, c90, c135, c45, cave;
    int p0, p90, p135, p45, pave;
    double d0, d90, d135, d45, dave;
    double f0, f90, f135, f45;
}

```

```
*****
/* READ THE INPUT IMAGE, EXPECT IT TO BE 8-bit UNSIGNED INT */
/* Mat type conversion is simple in openCV, try it later */
Mat imageIn = cv::imread(filename);
dx = imageIn.rows;
dy = imageIn.cols;
unsigned char *pixels = imageIn.data;
cout << "dx " << dx << " dy " << dy << " elemSize() " << imageIn.elemSize() << endl;
data = (int *)malloc(dx * dy * 4);
if (data == 0)
{
    cout << "malloc error in texture()" << endl;
}
for (y=0; y < dy; y++) {
    for (x=0; x < dx; x++) {
        int pixel = (int)*(imageIn.ptr(x,y));
        if (pixel > 255) {pixel = 255;}
        data[(y * dx) + x] = pixel;
    }
}
*****
/* PART 1 - get normal types of statistics from pixel data */
Imoment(data, sz, &median, &ave, &adev, &sdev, &svar, &skew, &curt);
lohi(sz, data, &tmin, &tmax);
gt.median = median;
gt.ave = ave;
gt.adev = adev;
gt.sdev = sdev;
gt.svar = svar;
gt.skew = skew;
gt.curt = curt;
gt.min = tmin;
gt.max = tmax;
fstream = fopen("SDMExtended.txt", "w");
if (fstream <= 0) {
    cout << "#cannot create file" << endl;
    return 0;
}
sprintf(str, "texture for object: %s\n", filename);
fputs(str, fstream);
sprintf(str, "area: %u, %u \n", dx, dy);
fputs(str, fstream);
*****
/* PART 2 - calculate the 4 spatial dependency matrices */
dm0 = (int *)malloc(256*256*4);
dm90 = (int *)malloc(256*256*4);
dm135 = (int *)malloc(256*256*4);
dm45 = (int *)malloc(256*256*4);
if ((dm0==0) || (dm90==0) || (dm135==0) || (dm45==0)) {
    cout << "malloc error in texture2" << endl;
    return 0;
}
x=0;
while (x < 256*256) {
    dm0[x] = dm90[x] = dm135[x] = dm45[x] = 0;
    x++;
}
```

```

}

y=0;
while (y < dy-1) {
    yc = dx * y;
    x=0;
    while (x < dx-1) {
        dm0[(data[yc + x]&0xff) + (((data[yc + x + 1]<< 8)&0xff00)]++;
        dm0[(data[yc + x + 1]&0xff) + (((data[yc + x])<< 8)&0xff00)]++;
        dm90[(data[yc + x]&0xff) + (((data[yc + x + dx])<< 8)&0xff00)]++;
        dm90[(data[yc + x + dx]&0xff) + (((data[yc + x])<< 8)&0xff00)]++;
        dm135[(data[yc + x]&0xff) + (((data[yc + x + dx + 1])<< 8)&0xff00)]++;
        dm135[(data[yc + x + dx + 1]&0xff) + (((data[yc + x])<< 8)&0xff00)]++;
        dm45[(data[yc + x + 1]&0xff) + (((data[yc + x + dx])<< 8)&0xff00)]++;
        dm45[(data[yc + x + dx]&0xff) + (((data[yc + x + 1])<< 8)&0xff00)]++;

        x++;
    }
    y++;
}
//********************************************************************* CALCULATE TEXTURE METRICS ******/
/* centroid */
pmx[0] = pmx[1] = pmx[2] = pmx[3] = 0;
pmy[0] = pmy[1] = pmy[2] = pmy[3] = 0;
i0 = i90 = i135 = i45 = 0;
y=0;
while (y < 256) {
    x=0;
    while (x < 256) {
        z = x + (256 * y);
        pmx[0] += (x * dm0[z]);
        pmy[0] += (y * dm0[z]); i0 += dm0[z];
        pmx[1] += (x * dm90[z]);
        pmy[1] += (y * dm90[z]); i90 += dm90[z];
        pmx[2] += (x * dm135[z]);
        pmy[2] += (y * dm135[z]); i135 += dm135[z];
        pmx[3] += (x * dm45[z]);
        pmy[3] += (y * dm45[z]); i45 += dm45[z];
        x++;
    }
    y++;
}
pmx[0] = pmx[0] / i0;
pmy[0] = pmy[0] / i0;
pmx[1] = pmx[1] / i90;
pmy[1] = pmy[1] / i90;
pmx[2] = pmx[2] / i135;
pmy[2] = pmy[2] / i135;
pmx[3] = pmx[3] / i45;
pmy[3] = pmy[3] / i45;
x = (pmx[0] + pmx[1] + pmx[2] + pmx[3]) / 4;
y = (pmy[0] + pmy[1] + pmy[2] + pmy[3]) / 4;
gt.xcentroid.t0 = pmx[0];
gt.ycentroid.t0 = pmy[0];
gt.xcentroid.t90 = pmx[1];
gt.ycentroid.t90 = pmy[1];
gt.xcentroid.t135 = pmx[2];
gt.ycentroid.t135 = pmy[2];

```

```
gt.xcentroid.t45 = pmx[3];
gt.ycentroid.t45 = pmy[3];
gt.xcentroid.tave = x;
gt.ycentroid.tave = y;
/* low frequency coverage */
i0 = i90 = i135 = i45 = 0;
c0 = c90 = c135 = c45 = 0;
x=0;
while (x < 256*256) {
    if ((dm0[x] != 0) && (dm0[x] < 3)) i0++;
    if ((dm90[x] != 0) && (dm90[x] < 3)) i90++;
    if ((dm135[x] != 0) && (dm135[x] < 3)) i135++;
    if ((dm45[x] != 0) && (dm45[x] < 3)) i45++;
    if (!dm0[x]) c0++;
    if (!dm90[x]) c90++;
    if (!dm135[x]) c135++;
    if (!dm45[x]) c45++;
    x++;
}
d0 = (double)i0 / 0x10000;
d90 = (double)i90 / 0x10000;
d135 = (double)i135 / 0x10000;
d45 = (double)i45 / 0x10000;
savegt(&gt.low_frequency_coverage, d0, d90, d135, d45);
d0 = (double)c0 / 0x10000;
d90 = (double)c90 / 0x10000;
d135 = (double)c135 / 0x10000;
d45 = (double)c45 / 0x10000;
savegt(&gt.total_coverage, d0, d90, d135, d45);
d0 = (c0-i0) / (double)0x10000;
d90 = (c90-i90) / (double)0x10000;
d135 = (c135-i135) / (double)0x10000;
d45 = (c45-i45) / (double)0x10000;
savegt(&gt.corrected_coverage, d0, d90, d135, d45);
/* power */
i0 = i90 = i135 = i45 = 0;
c0 = c90 = c135 = c45 = 0;
p0 = p90 = p135 = p45 = 0;
y=0;
while (y < 256) {
    z = y * 256;
    x=0;
    while (x < 256) {
        n = x-y;
        if (n < 0) n = -n;
        if (dm0[x+z] != 0) {i0 += n; c0++;}
        if (dm90[x+z] != 0) {i90 += n; c90++;}
        if (dm135[x+z] != 0) {i135 += n; c135++;}
        if (dm45[x+z] != 0) {i45 += n; c45++;}
        x++;
    }
    y++;
}
d0 = (i0 / 0x10000);
d90 = (i90 / 0x10000);
d135 = (i135 / 0x10000);
```

```

d45 = (i45 / 0x10000);
savegt(&gt.total_power, d0, d90, d135, d45);
d0 = (i0 / c0);
d90 = (i90 / c90);
d135 = (i135 / c135);
d45 = (i45 / c45);
savegt(&gt.relative_power, d0, d90, d135, d45);
/* locus density */
d0 = d90 = d135 = d45 = 0.0;
c0 = c90 = c135 = c45 = 0;
p0 = p90 = p135 = p45 = 0;
y=0;
while (y < 256) {
    z = y * 256;
    i0 = i90 = i135 = i45 = 0;
    x=0;
    while (x < 256) {
        n = x-y;
        if (n < 0) n = -n;
        if ((dm0[x+z] != 0) && (n < 7)) {c0++; p0 += dm0[x+z];}
        if ((dm90[x+z] != 0) && (n < 7)) {c90++; p90 += dm90[x+z];}
        if ((dm135[x+z] != 0) && (n < 7)) {c135++; p135 += dm135[x+z];}
        if ((dm45[x+z] != 0) && (n < 7)) {c45++; p45 += dm45[x+z];}
        if ((dm0[x+z] == 0) && (n < 7)) {i0++;}
        if ((dm90[x+z] == 0) && (n < 7)) {i90++;}
        if ((dm135[x+z] == 0) && (n < 7)) {i135++;}
        if ((dm45[x+z] == 0) && (n < 7)) {i45++;}
        x++;
    }
    if (!i0) d0 += 1;
    if (!i90) d90 += 1;
    if (!i135) d135 += 1;
    if (!i45) d45 += 1;
    y++;
}
savegt(&gt.locus_length, d0, d90, d135, d45);
d0 = (p0/c0);
d90 = (p90/c90);
d135 = (p135/c135);
d45 = (p45/c45);
savegt(&gt.locus_mean_density, d0, d90, d135, d45);
/* density */
c0 = c90 = c135 = c45 = 0;
p0 = p90 = p135 = p45 = 0;
x=0;
while (x < 256*256) {
    if (dm0[x] != 0) {c0 += dm0[x]; p0++;}
    if (dm90[x] != 0) {c90 += dm90[x]; p90++;}
    if (dm135[x] != 0) {c135 += dm135[x]; p135++;}
    if (dm45[x] != 0) {c45 += dm45[x]; p45++;}
    x++;
}
d0 = c0 / p0;
d90 = c90 / p90;
d135 = c135 / p135;
d45 = c45 / p45;

```

```

savegt(&gt.bin_mean_density, d0, d90, d135, d45);
/* containment */
i0 = i90 = i135 = i45 = 0;
x=0;
while (x < 256) {
    if (dm0[x]) i0++; if (dm0[256*256 - x - 1]) i0++;
    if (dm90[x]) i90++; if (dm90[256*256 - x - 1]) i90++;
    if (dm135[x]) i135++; if (dm135[256*256 - x - 1]) i135++;
    if (dm45[x]) i45++; if (dm45[256*256 - x - 1]) i45++;
    if (dm0[x*256]) i0++; if (dm0[(x*256)+255]) i0++;
    if (dm90[x*256]) i90++; if (dm90[(x*256)+255]) i90++;
    if (dm135[x*256]) i135++; if (dm135[(x*256)+255]) i135++;
    if (dm45[x*256]) i45++; if (dm45[(x*256)+255]) i45++;
    x++;
}
d0 = 1.0 - ((double)i0 / 1024.0);
d90 = 1.0 - ((double)i90 / 1024.0);
d135 = 1.0 - ((double)i135 / 1024.0);
d45 = 1.0 - ((double)i45 / 1024.0);
savegt(&gt.containment, d0, d90, d135, d45);
/* linearity */
i0 = i90 = i135 = i45 = 0;
c0 = c90 = c135 = c45 = 0;
y=0;
while (y < 256) {
    z = y * 256;
    if (dm0[z + y] > 1) {i0++; c0 += dm0[z+y];}
    if (dm90[z + y] > 1) {i90++; c90 += dm90[z+y];}
    if (dm135[z + y] > 1) {i135++; c135 += dm135[z+y];}
    if (dm45[z + y] > 1) {i45++; c45 += dm45[z+y];}
    y++;
}
d0 = (double)i0 / 256.;
d90 = (double)i90 / 256.;
d135 = (double)i135 / 256.;
d45 = (double)i45 / 256.;
savegt(&gt.linearity, d0, d90, d135, d45);
/* linearity strength */
d0 = (c0/(i0+.00001)) / 256.;
d90 = (c90/(i90+.00001)) / 256.;
d135 = (c135/(i135+.00001)) / 256.;
d45 = (c45/(i45+.00001)) / 256.;
savegt(&gt.linearity_strength, d0, d90, d135, d45);
/* WRITE ALL STATISTICS IN gt.STRUCTURE TO OUTPUT FILE */
put_txfile(fstream);
/* clip to max value 255 */
mval0 = mval90 = mval135 = mval45 = 0;
x=0;
while (x < 256*256) {
    if (dm0[x] > 255) dm0[x] = 255;
    if (dm90[x] > 255) dm90[x] = 255;
    if (dm135[x] > 255) dm135[x] = 255;
    if (dm45[x] > 255) dm45[x] = 255;
    x++;
}

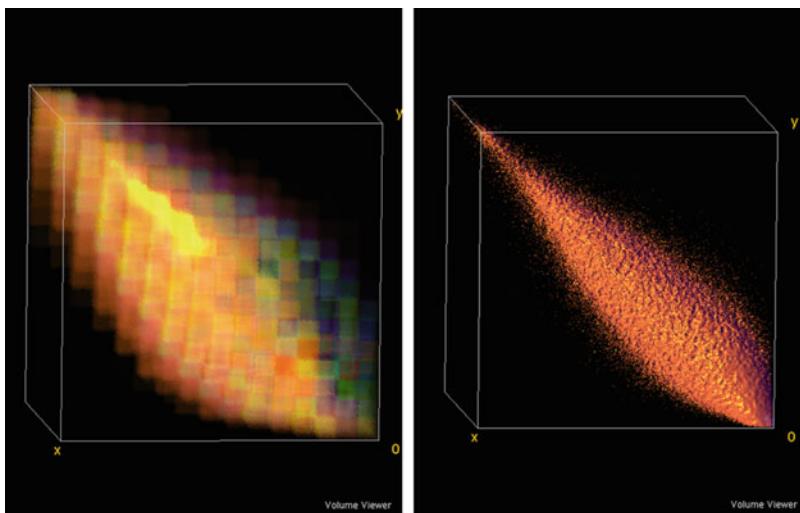
```

```
/*****************************************/
/* Convert data to unsigned char to write into png */
unsigned char *dm0b = (unsigned char *)malloc(256*256);
unsigned char *dm90b = (unsigned char *)malloc(256*256);
unsigned char *dm135b = (unsigned char *)malloc(256*256);
unsigned char *dm45b = (unsigned char *)malloc(256*256);
if ((dm0b==0) || (dm90b==0) || (dm135b==0) || (dm45b==0)) {
    cout << "malloc error in texture3" << endl;
    return 0;
}
x=0;
while (x < 256*256) {
    dm0b[x] = (unsigned char) (dm0[x] & 0xff);
    dm90b[x] = (unsigned char) (dm90[x] & 0xff);
    dm135b[x] = (unsigned char) (dm135[x] & 0xff);
    dm45b[x] = (unsigned char) (dm45[x] & 0xff);
    x++;
}
/*
* write output to 4 quadrants:  0=0, 1=90, 2=135, 3=145
*/
char outfile[256];
sprintf(outfile, "%s_SDMQUadrant_0deg_8UC1.png", filename);
Mat SDMQuadrant0(256, 256, CV_8UC1, dm0b);
imwrite(outfile, SDMQuadrant0);
sprintf(outfile, "%s_SDMQUadrant_90deg_8UC1.png", filename);
Mat SDMQuadrant90(256, 256, CV_8UC1, dm90b);
imwrite(outfile, SDMQuadrant90);
sprintf(outfile, "%s_SDMQUadrant_135deg_8UC1.png", filename);
Mat SDMQuadrant135(256, 256, CV_8UC1, dm135b);
imwrite(outfile, SDMQuadrant135);
sprintf(outfile, "%s_SDMQUadrant_45deg_8UC1.png", filename);
Mat SDMQuadrant45(256, 256, CV_8UC1, dm45b);
imwrite(outfile, SDMQuadrant45);
free(dm0);
free(dm90);
free(dm135);
free(dm45);
free(data);
free(dm0b);
free(dm90b);
free(dm135b);
free(dm45b);
fclose(fstream);
return 0;
}
int main (int argc, char **argv)
{
    cout << "8-bit unsigned image expected as input" << endl;
    texture (argv[1]);
    return 0;
}
```

Appendix E: The Visual Genome Model (VGM)

The memory impression is the feature

Scott Krig



Volume renderings of synthetic neural clusters represented as visual genome features

In this appendix we discuss *the Visual Genome Model VGM*), a view-based vision model, assuming virtually unlimited feature memory space to store features and concepts, rather than constraining and compressing the feature representation to a sparse or more computable set as is typical in common neural models such as CNNs. Visual Genomes record all the features detected in separate *virtual neurons* modeled as simple memory cells to record each feature, and a comparator to test input impressions presented to the neuron against the memory cell. Visual Genomes are composed together into sequences or *visual genomes*, similar to a DNA chain, to represent higher-level concepts in strands and bundles. The *Visual Genome Model* is inspired by the basic low-level structures of the visual pathway including the retina through, LGN, and V1-V4 layers. The higher-level reasoning centers of the visual pathway are not included in the VGM, and instead, high level reasoning centers are assumed to be a consciousness level which can be modeled as a special-purpose *proxy agent* process implemented in software, managing a suitable training protocol, classifier, and

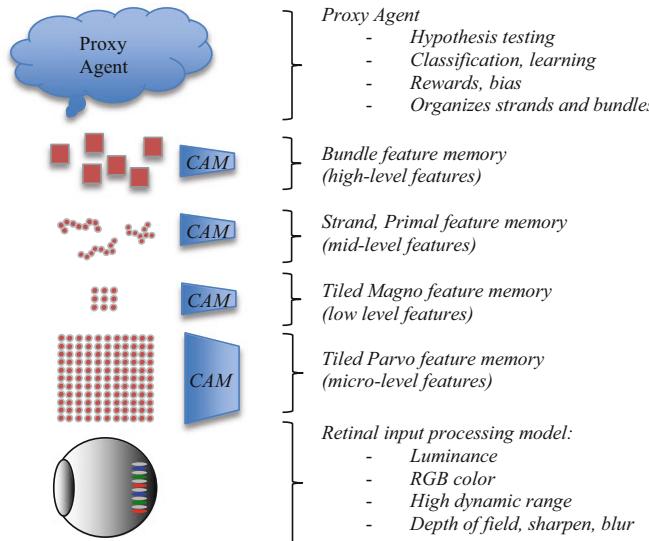


Figure E.1 This figure illustrates the visual genome architecture and feature memory concepts

hypothesis testing mechanism. In this respect, the VGM may be considered as a visual memory machine suitable for use under the control of a proxy agent within a larger visual processing machine. We present an overview of the VGM in this appendix, and additional details with more comprehensive results for classification and object recognition are provided in Krig [534] and <http://krigresearch.com>.

The VGM follows the neurobiological concepts of local receptive fields and a hierarchy of features, similar to the Hubel and Wiesel model [559, 560]. As shown in Fig. E.1, the hierarchy consists of parvo, magno, strand, and bundle features as discussed below. Each feature is simply a *memory record* of the visual inputs stored in groups of neuron memory. This is in contrast to the notion of designing a feature descriptor.

VGM stores low-level Parvo and Magno features as raw memory records or *impressions*, and groups the low-level memory records within contiguous segmented primal regions as *strands*, and groups of strands are associated together as *bundles* describing higher-level concepts. The raw input pixel values of local receptive fields are used to compose a feature *address vector* referencing a huge virtual multi-dimensional address space, see Fig. E.8. *The address is the feature*. The intent of using the raw pixel values concatenated into an address is to enable storage of the raw visual impressions directly in a virtually unlimited feature memory with no intervening processing, following the view based model of neurobiology. The bit precision of the address determines the size of the memory space. The bit precision and coarseness of the address is controlled by a *quantization parameter*, discussed later along with the VGM neural model. So the VGM operates in a *quantization space*, revealing more or less detail about the features as per the quantization level.

Why such a simple feature model? The VGM model assumes that *the sheer number of features is more critical than the type of feature chosen*, as evidenced by the feature learning architecture survey in Chap. 10, providing several examples of systems that all achieve similar accuracy, using a wide range of feature types and hierarchy levels. It is not clear from CNN research that the scale hierarchy itself is the major key to success, or if a large set of mono-scale multi-model features, rather than a scale hierarchy, would be equally effective. For example, large numbers of *simple image pixel regions* have been demonstrated by Gu et al. [706] to be very capable image descriptors for

segmentation, detection and classification. Gu organizes the architecture as a robust bag of overlapped features, using Hough voting to identify hypothesis object locations, followed by a classifier, to achieve state of the art accuracy on several tests.

Neuroscience Inspiration for VGM

Here we summarize the specific neuroscience research which informs and inspires the VGM, followed by the corresponding architecture and design details. We cover the basics of the visual pathway, memory mechanisms, neural models, and the retinal processing model.

Feature and Conceptual Memory Locality

Neuroscience research shows that the visual pathway stores related concepts in contiguous memory regions [852, 853], suggesting a *view-based model* [814] for vision. Under the view-based model, new memory records, *rather than invariant features*, are created to store variations of similar items for a concept. Related concepts are stored in a local region of memory proximate to similar objects. The mechanism for creating new memory features is likely based on *an unknown learning motivation or bias*, as directed by higher layers of reasoning in the visual pathway. Conversely, the stored memories do not appear to be individually invariant, but rather the invariance is built up conceptually by collecting multiple scene views together with geometric or lighting variations. *Brain mapping research* supports the view-based model hypothesis. Research using functional MRI scans (*fMRI*) shows that brain mapping can be applied to forensics, by mapping the brain regions that are activated while viewing or remembering visual concepts, as reported by Lengleben et al. [859]. In fact, Nature has reported that limited mind reading is possible [852, 853, 860] using brain mapping, revealing in MRI-type imaging modalities specific regions of the cerebral cortex that are electrically activated while viewing a certain subject, evaluating a certain conceptual hypothesis, or responding to verbal questions. (*Of related interest, according to some researchers brain mapping reveals cognitive patterns that can be interpreted to reveal raw intelligence levels, and also brain mapping has been used to record cognitive fingerprints which are currently fashionable within military and government security circles*).

New memory impressions will remain in *short-term* memory for evaluation of a given hypothesis, and may be subsequently forgotten unless classified and committed to *long-term* memory by the higher level reasoning portions of the visual pathway. The higher level portions of the visual pathway consciously direct classification using a set of hypothesis against either incoming data in short term memory, or to reclassify long term memory. The higher-level portions of the visual pathway are controlled perhaps independently of the biology by higher-level consciousness of the soul. The eye and retina may be directed by the higher level reasoning centers to adjust the contrast and focus of the incoming regions.

Attentional Neural Memory Research

Baddely [605] and others have shown that the human learning and reasoning process typically keeps several concepts at attention simultaneously at the request of the *central executive*, which is directing the reasoning task at hand. (*VGM models the central executive as a proxy agent, discussed below*). The central executive concept assumes that inputs may come in at different times, thus several

concepts need to be at attention at a given time. Perhaps up to seven concepts can be held at attention by the human brain at once, thus Bell Labs initially create phone numbers using seven digits. Selected concepts are kept at attention in a working memory or *short-term memory* (i.e., *attention memory*, or *concept-memory*), as opposed to a *long-term memory* from the past that is not relevant to the current task. As shown by Goldman-Rakic [606] the attention-memory or concepts may be accessed at different rates, for example checked constantly, or not at all, during delay periods while the central executive is pursuing the task at hand and accessing other parts of memory. The short-term memory will respond to various cues, and loosely resembles the familiar associative memory or content-addressable memory (CAM) used for caching in some CPUs. The VGM address feature model is similar to a CAM model, and allows the central executive to determine feature detection on-demand, and VGM does not distinguish short/long term memory or limit short term memory.

HMAX Model and Visual Cortex Models of the Visual Pathway

The HMAX model is designed after the visual pathway regions, which clearly shows a hierarchy of concepts. HMAX uses hardwired feature for the lower levels such as Gabor or Gaussian functions, which resemble the oriented edge response of neurons observed in the early stages of the visual pathway as reported by Tanaka [825], Logothetis [827], and others. Logothetis found that some groups of neurons along the hierarchy respond to specific shapes similar to Gabor-like basis functions at the low levels, and object-level concepts such as faces in higher levels. HMAX builds higher level concepts on the lower level features, following research showing that higher levels of the visual pathway (IT) are receptive to highly view-specific patterns such as faces, shapes and complex objects, see Perrett [821, 822] and Tanaka [823]. In fact, clustered regions of the visual pathway IT region are reported by Tanaka [826] to respond to similar clusters of objects, suggesting that neurons grow and connect to create semantically associated view-specific feature representations as needed for view-based discrimination. HMAX provides a viewpoint-independent model that is invariant to scale and translation, leveraging a MAX pooling operator over scale and translation for all *inputs* feeding the higher-level S2, C2, and VTU units, resembling *lateral inhibition* which has been observed between competing neurons, allowing the strongest activation to shut down competing lower strength activations. HMAX also allows for sharing of low-level features and interpolations between them as they are combined into higher-level viewpoint-specific features.

Virtually Unlimited Feature Memory

The brain contains perhaps 100 billion neurons or 100 *giga-neurons* (GN), (*estimates vary*), and each neuron is connected to perhaps 10,000 other neurons on average (*estimates vary*), yielding over 100 trillion connections [858] compared to the estimated 200–400 billion stars in the Milky Way galaxy. Apparently, there are plenty of neurons to store information in the human brain, so the VGM takes the assumption that there is no need to reduce the size of the feature set, and supports virtually *unlimited feature memory*. Incidentally for unknown reasons, the brain apparently only uses a portion of the available neurons, estimates range from 10 to 25 % (10GN–25GN). Perhaps with longer life spans of perhaps 1000 years, all the neurons could be activated into use.

VGM feature memory is represented in a quantization space where the bit resolution of the features is adjusted to expand or reduce precision, which is useful for practical implementations. In effect, the size of the virtual memory for all neurons is controlled by *the numeric precision of the pixels*. Visual genomes represent features at variable resolution to produce either coarse or fine results in a quantization space, discussed in subsequent sections on the VGM neural model and memory structure below.

Genetic Preexisting Memory

More and more research shows that DNA may contain memory impressions or *genetic memory* such as instincts and character traits (see [824], many more references can be cited). Other research shows that DNA can be modified via memory impressions [825] that are passed on to subsequent generations via the DNA.

Neuroscience suggests that some features are preexisting in the neurocortex at birth, for example memories and other learnings from ancestors may be imprinted into the DNA, while other behaviors are pre-wired in the basic human genome, designed into the DNA, and not learned at all. It is well known that DNA can be modified by experiences, for better or worse, and passed to descendants by inheritance. So the DNN training notion of feature learning by initializing weights to random values and averaging the response over training samples is primitive best, and a rabbit trail following the evolutionary assumptions of *time + chance = improvement*. In other words, we observe that visible features are both *recorded* and *created* by genetic design, not generated by random processes.

The VGM model allows for preexisting memory to be emulated using transfer learning to initialize the VGM memory space, which can be subsequently improved by recording new impressions from a training set or visual observation on top of the transferred features. Specifically, some of the higher level magno, strand, and bundle features can be initialized to *primal basis sets*, for example shapes or patterns, to simulate inherited genetic primal shape features, or to provide experience-based learning.

Neurogenesis, Neuron Size and Connectivity

As reported by Bergami et al. [861, 862] as well as many other researchers, the process of neurogenesis (i.e., neural growth) is *regulated by experience*. Changes to existing neural size and connectivity, as well as entirely new neuron growth, take place in reaction to real or perceived experiences. As a result, there is no fixed neural architecture for low-level features, rather the architecture grows. Even identical twins (i.e., DNA clones) develop different neurobiological structures based on experience, leading to different behavior and outlook.

Various high level structures have been identified within the visual pathway, as revealed by brain mapping [852, 853], such as conceptual reasoning centers and high-level communications pathways [686], see also Fig. 9.10. Neurogenesis occurs in a controlled manner within each structural region. Neurogenesis includes both growth and shrinkage, and both neurons and dendrites have been observed to grow significantly in size in short bursts, as well as shrink over time. Neural size and connectivity seem to represent memory *freshness*, and *forgetting*, so perhaps *forgetting* may be biologically expressed as neuronal shrinkage accompanied by disappearing dendrite connections. Neurogenesis is reported by Lee et al. [862] to occur throughout the lifetime of adults, and especially during the early formative years.

To represent neurogenesis, VGM represents neural size and connectivity by the number of times a feature impression is detected, which can be interpreted as (a) a new neuron for each single impression, or (b) a larger neuron for multiple impression counts (*it is not clear from neuroscience if either a OR b, or both a AND b are true.*) Therefore, neurogenesis is reflected in terms of the size and connectivity of each neuron in VGM.

Bias and Motivation for Learning New Memory Impressions

Neuroscience suggests that the brain creates new memory impressions of important items under the view-based theories surveyed in the HMAX section in Chap. 10, rather than dithering visual impressions together as in DNN backprop training. Many computer vision models are based on the notion that features should be designed to be invariant to specific robustness criteria, such as scale, rotation, occlusion, and other factors discuss in Chap. 5, which may be an artificial notion only partially expressed in the neurobiology of vision. Although bias is assumed during learning, VGM does not model a bias factor in the VGM neuron. Most artificial neural models include a bias factor for matrix method convenience, but usually the bias is ignored or fixed. Bias can account for the observation that people often see what they believe, rather than believing what they see, and therefore bias seems problematic to model.

Depth Processing

Depth processing in the human visual system is accomplished in at least two ways: (1) using stereo depth processing within a L/R stereo processing pathway in the visual cortex, and (2) using other 2D visual cues associated together at higher level reasoning centers in the visual pathway. As discussed in Chap. 1 and summarized in Table 1.1, the human visual system relies on stereo processing to determine *close range depth* out to perhaps 5–10 m, and then relies on other 2D visual cues like shadows and spatial relationships to determine *long range depth*, since stereo information from the human eye is not available at increasing distances due to the short baseline distance between the eyes (see Fig. 1.20). In addition, stereo depth processing is affected by a number of key problems including occlusion and holes in the depth map due to the position of objects in the field of view, and also within the Horopter region where several points in space may appear to be *fused together* at the same location, requiring complex approximations in the visual system. The VGM model does not attempt to model depth or the stereo pathway.

However, future work may include providing a depth map channel and surface normal vector images as input channels for magno and parvo features, but perhaps the better approach is to provide depth maps and surface normal images to the higher-level proxy agent for incorporation into strands, bundles, and a classifier.

Dual Retinal Processing Pathways: Magno and Parvo

As shown in Fig. E.3, there are two types of cells in the retina which provide ganglion cell inputs to the optic nerve: *magno cells* and *parvo cells*. Of the approximately one million ganglion cells leaving the retina, about 80–90 % are smaller parvo cells with smaller receptive fields, and about 10–20 % are larger magno cells with a larger receptive field. The magno cells track gross movement in 3D and are sensitive to contrast, luminance and coarse details (i.e., the receptive field is large). The parvo cells are slower to respond and represent color and fine details (i.e., the receptive field is small). Magno cells are spread out across the retina and provide the gross low-resolution outlines, and parvo cells are concentrated in the center of the retina and respond most to the saccadic dithering to increase effective resolution.

The magno and parvo cell resolution differences suggest a two-level spatial pyramid arrangement built into the retina for magno-subsampled low resolution, and parvo high resolution. In addition, the

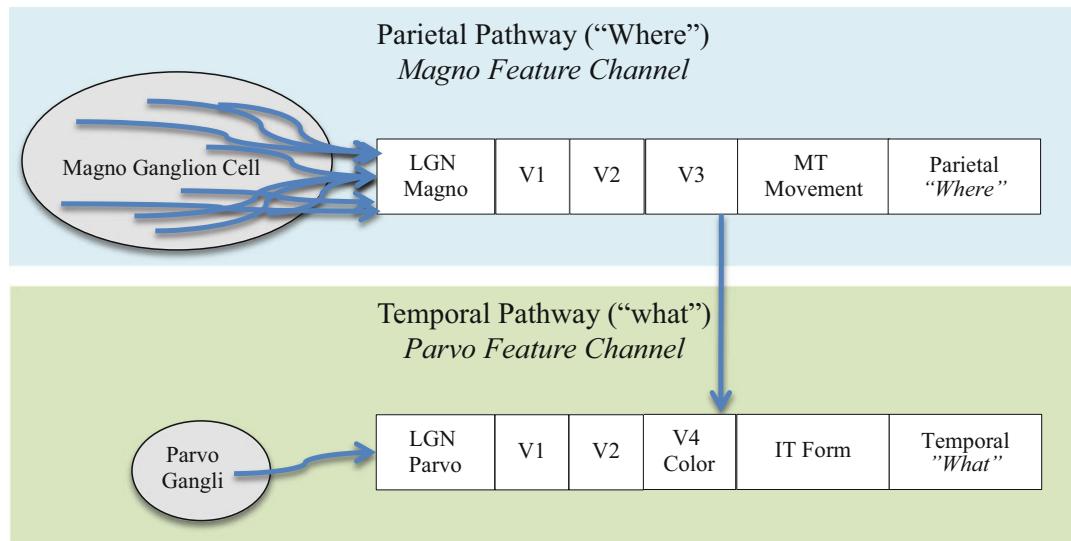


Figure E.3 This figure illustrates the partetal and temporal visual pathways composed of larger magno cells with lower resolution, and smaller parvo cells with higher resolution. Parvo cells are 10–20 % as large as magno cells

visual pathway contains two separate parallel processing pathways—a fast magno shape tracking monochrome pathway, and a slow parvo color and texture pathway. Following the magno and parvo concepts, Visual Genomes provides two classes of genomes: low-resolution luminance genomes for coarse shapes and segmentations (magno features), and higher resolution color and texture genomes (parvo features).

Following the dual parvo and magno pathways in the human visual system, Visual Genomes models *parvo features* as micro-level RGB color and texture tiles at higher resolution, and *magno features* as low-level luminance channels at lower resolution, such as primitive shapes with connectivity and spatial relationships. The magno features correspond mostly to the rods in the retina which are sensitive to luminance and fast-moving shapes, and the parvo features correspond mostly to the cones in the retina which are color sensitive to RGB, and capture low-level details with spatial acuity. The central foveal region of the retina is exclusively RGB cones, optimized to capture finer detail, and contains the highest density of cells in the retina, with retinal cell density becoming more sparse towards the edge of the field of view.

Retinal Processing Model

The retina can perform a wide range of processing, including dynamic range adjustments at each cell. The retina performs a saccadic dithering process to get more detail from a specific area by dithering the focal point around the area. The iris can open and close to control lighting, and the receptive rod, cone and ganglion cells together perform local contrast enhancement. In addition, the lense can be used to change the depth of field and focal plane (depth of field is a stereo process, and the visual pathway provides a separate L/R processing pathway for depth processing). Notice that the retinal model does not include geometric position or scale changes.

VGM provides a retinal input processing model consisting of a set of separate input images, which reflect the capabilities of actual vision biology at the eye:

- Luminance images
- RGB color images and separate color channel images
- High dynamic range contrast enhanced images using the biologically inspired Retinex method (*see Scientific American, May 1959 and December 1977*)
- Local contrast normalization
- Sharpened and blurred images

Visual Genomes assume a *retinal model for input processing, combined with simple neurons that do not perform any processing*. The retina provides depth of field and focus controls, contrast controls, dynamic range controls for compression and expansion. However, other variations such as rotation and scale are controlled by moving the body and eye position, rather than neural image processing. In this respect, the training protocol can be optimized by including prepared images of different views and perspectives for optimal learning.

Visual Genomes Model Concepts

Here we describe more details on each component of the VGM model.

Magno and Parvo Features

Following the biological region subsampling that occurs in magno cells, VGM defines two types of features and two types of images in a two-level feature hierarchy:

1. **Parvo features:** Parvo features are modeled as RGB features with high detail, following the design of parvo cells, take input images at full resolution (100 %), use RGB color, and represent color and texture features.
2. **Magno features:** Magno features are modeled as lower resolution luminance features, following the magno cell biology, chosen to be 20 % of full resolution (*as a default approximation to retinal biology*), following the assumption that the larger magno cells integrate and subsample a larger retinal area, therefore yielding a lower resolution image suited for the rapid tracking of shapes, contours and edges for masks and cues. Since actual magno cells use predominantly monochrome rod cells, VGM defines magno features to use a monochrome space and a *Retinex* processing algorithm (*see Scientific American, May 1959 and December 1977*) to model the low light-level rod response which provides *local contrast enhancement*, and also a *global contrast normalization* method similar to histogram equalization.

The parvo and magno features are collected in four genome shapes A, B, C, D within overlapped input windows, simulating the Hubel and Weiss [559, 560] primitive edges found in local receptive fields, see Figs. E.8, E.9, and E.10. The genome shapes are discussed in more detail later.

It is also observed that the visual pathway operates in two main phases:

- **Scanning phase (magno features):** Apparently the visual system first identifies regions of interest via the magno features, such as shapes and patterns, during a *scanning phase*, where the eye is looking around the scene and not focused on a particular area. During the scanning phase, the retinal model is not optimized for a particular object or feature, except perhaps for controlling gross lighting

and focus via the iris and lense. The Magno features are later brought into better focus and dynamic range is optimized when the eye focuses on a specific region for closer evaluation.

- **Saccadic phase (parvo features):** For closer evaluation, the visual system inspects interesting magno shapes and patterns to identify parvo features, and then attempts to identify larger concepts. The retinal processing optimizations may change several times during the parvo feature scanning in the saccadic dithering stage according to the current hypothesis under attention by the high-level proxy agent, for example focus and depth of field may be changed at a particular point to test a hypothesis.

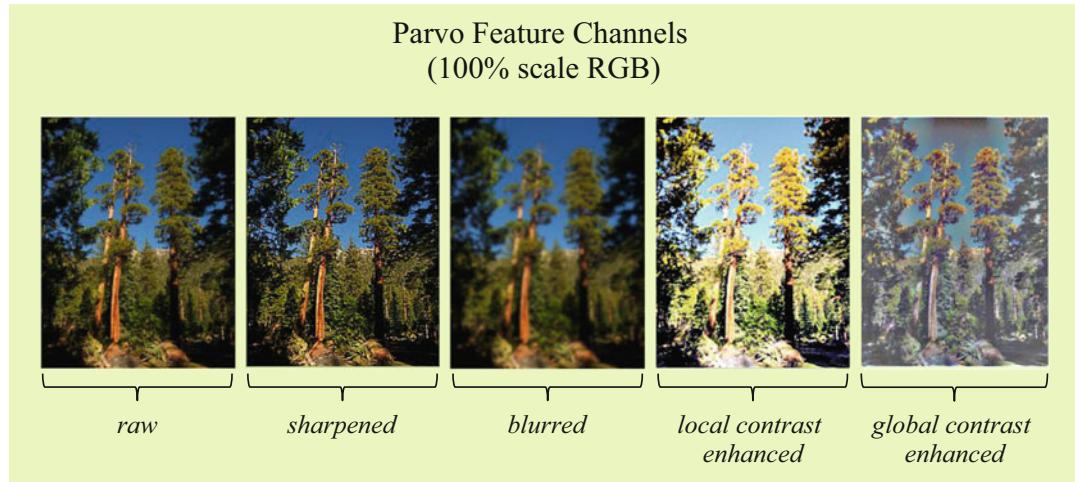


Figure E.4 This figure illustrates the five Parvo feature channels

Parvo Retinal Processing

Parvo cells are designed to capture color and texture with high detail, and operate at a higher resolution than the magno cells, and slower to respond to changes. To emulate the parvo cells in the VGM retinal model, four types of input processing are used to create full-resolution images for parvo features, corresponding to the biological capabilities of the eye:

1. Unprocessed raw first-pass RGB images.
2. Local contrast enhanced RGB images emulating saccadic dithering.
3. Global contrast enhanced RGB images to mitigate shadow and saturation effects and support high dynamic range contrast enhancements.
4. Sharpened RGB images supporting focus increase.
5. Blurred RGB images emulating depth of field effects.

The input images are all processed and combined by the VGM neuron into the same genome features (A, B, C, D as shown in Fig. E.8), so there is not a separate feature genome for blur, sharpen, raw and contrast enhanced images. Instead, the goal is to *integrate* the range of retinal features into each memory cell, making the assumption that there is short term memory in biological neurons, allowing the neuron to form and commit the feature memory as controlled by the proxy agent. Also, using separate genomes for each retinal processing function would explode the feature memory count and processing load without clear justification at this stage of the VGM prototype development.

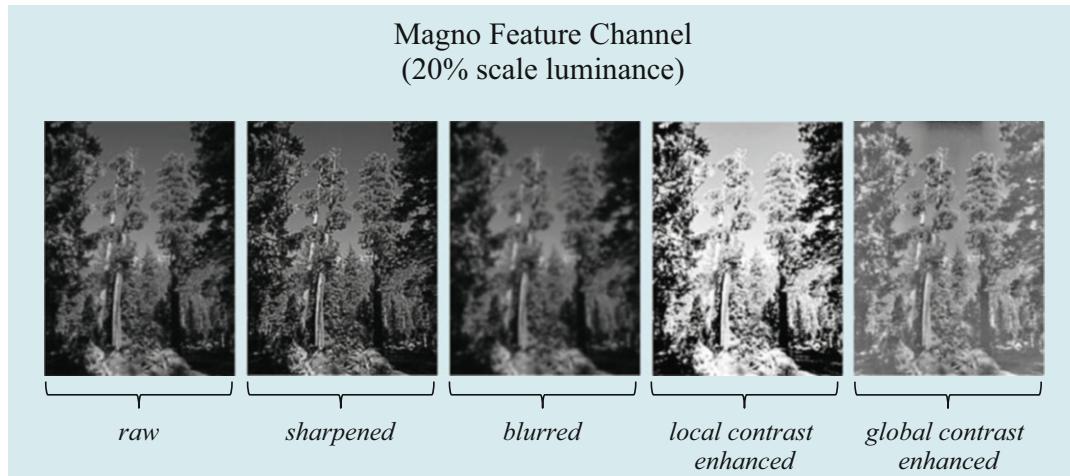


Figure E.5 This figure illustrates the magno feature channels

Magno Retinal Processing

For the magno features, we assume a much simpler model than the parvo cells, since the magno cells are lower resolution and are attuned to fast moving objects, which implies that the retinal model does not change the magno features as much from impression to impression, rather the retinal model for magno scanning is perhaps changed three times: (1) global scene scan at constant settings, (2) pause scan at specific location and focus, and (3) contrast enhance at paused position, and then hand off processing to the parvo stage. Therefore, we propose to model the magno features as raw luminance input subsampled to 20 % of full resolution, with prepared luminance images reflecting the eye processing biology. Since magno cell regions contain several cells within the magno region, and are predominantly attuned to faster-moving luminance changes, subsampling luminance 5:1 (i.e., 20 % scale) is a reasonable emulation of magno cell biology. Perhaps the larger size of the magno region is both (1) faster to accumulate a low light response over all the cells in the magno region, and (2) the magno cell output is lower resolution due to the subsampling of all the cells within the magno region.

Note that scale and other geometric changes are not controlled by the eye, but rather by the body containing the eye, or the position of the subject. Therefore, geometric variations such as scale, rotation, and warping can be accomplished best by using a range of carefully selected labeled training samples from various viewpoints, angles and distances, or else emulated by applying geometric transforms to the labeled training images (*not as desirable under the viewpoint model*).

Magno Primal Feature Segmentation

And it also seems necessary to carefully *mask out* extraneous information from the labeled training samples, for example masking out only the apple in a labeled image of apples, for recording the optimal genome impressions for the apple. To emulate selective region masking and attentional



Figure E.6 This figure illustrates the relative scale of parvo and magno image input. The scale difference corresponds to the magno cell area vs. parvo cell area, where parvo cells are 10–20 % as large as magno cells. The parvo cells are full resolution retinal images, and the magno cells are 5:1 down-sampled images

focus, the magno scanning phase incorporates an image segmentation pipeline to identify interesting regions, discussed later in the *Visual Genome Sequences, Tiles, Strands, Bundles, Primal Features* section.

VGM Neuron Model

Each magno and parvo neuron models a separate feature; however, features may be shared corresponding to neurons being connected to multiple other neurons. In the VGM, neural size and connection density are modeled as corresponding to the number of times the neuron is shared. The feature sharing is recorded as feature detection counts for each stored neural memory impression. (*Note: strand and bundle neurons and features are discussed in more detail in [534]*). As shown in Fig. E.7, neurons representing tiled magno and parvo features are composed of a CAM memory cell and a comparator δ , which operates at a bit precision quantization level θ input to the neuron. The neural input \mathbf{k} is either considered as a 3×3 matrix representing *tile features*, or as a 9×1 vector. Likewise, the CAM memory corresponds to a 3×3 tile or 9×1 vector. (See Krig [534] for information on other sizes besides 3×3). The contents of the CAM memory and the input \mathbf{k} are taken by the comparator δ to produce two items: (1) the Correlation Distance output \mathbf{k}' , and (2) the Motivation or Firing output Φ which is a TRUE or FALSE output, corresponding to a dendrite firing all or nothing. The correlation distance \mathbf{k}' can be used for implementing *inhibition* (a biologically plausible neural mechanism), when the distance is small. The *Motivation* firing activity and *Output Distance* of each neuron can be used by a higher level proxy agent for forgetting memory (discussed in [534]) or determining feature distance. To generate Φ , a quantization factor input θ is used with \mathbf{k}' to implement a *quantization space* for determining the firing threshold. The quantization space is based on binary quantization, so eliminating a bit of precision quantizes the space by a power of 2, for example $11111111 = 256$ space quantization, $11111110 = 128$ space quantization. The bit-level quantization simulates a form of *attentional level of detail*, which is biologically plausible.

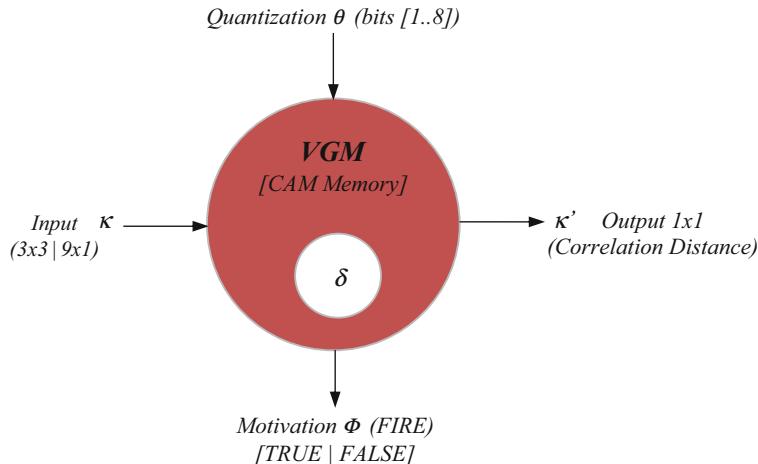


Figure E.7 This figure illustrates the VGM model for magno and parvo neurons

$$\kappa \sim [\kappa_0, \kappa_1, \kappa_2, \kappa_3, \kappa_4, \kappa_5, \kappa_6, \kappa_7, \kappa_8] \sim \begin{bmatrix} \kappa_0 & \kappa_1 & \kappa_2 \\ \kappa_3 & \kappa_4 & \kappa_5 \\ \kappa_6 & \kappa_7 & \kappa_8 \end{bmatrix}$$

$$\kappa' = f\left(\sum_{n=0}^9 |CAM_n - \kappa_n|, \theta\right)$$

$$\phi = f(\kappa', \theta)$$

In raw *Impression mode* (similar to a learning mode), all memory impressions are captured in CAM memory cells as *detection counts*. Common impressions have larger impression counts since they are detected more often. As shown in the Genome Renderings at the end of this appendix, commonly detected features are typically recorded in regular patterns follow the address format structure. The *Quantization input* θ bits can be used to shape the memory address by masking each pixel to *coalesce similar memory addresses* which focuses and groups similar features together, as explained in the VGM Memory Structures section below and Fig. E.11. Essentially, the Quantization input θ is used to mask off the lower bits of each pixel in an address to (1) *reduce* the level of detail and number of different features detected, and correspondingly (2) *increase* the detection count by coalescing similar features. We also refer to θ as *Quantization Distance* or *Quantization Space*. Quantization allows *variable precision* feature interpretation, or recording, allowing the same feature to be represented with a variable amount of detail depending on the task, for example lower detail for high-level passes to find candidate matches, and high detail for final classification passes.

To illustrate *Quantization Space*, consider two different feature addresses that become equal after coalescing using a quantization θ mask:

Feature 1 = 0x81A89D

Feature 2 = 0x83A19e

Quantization input θ mask = 0xFCFCFC

0x81A89D & 0xFCFCFC = 0x80A89C

0x83A19e & 0xFCFCFC = 0x80A89C

Each neuron represents a complete Magno or Parvo feature at some level of the feature hierarchy in the visual pathway.

VGM Feature Memory Structures

The VGM feature is a memory address composed from a 3×3 region of pixel values—the pixel values comprise the address. The idea is to represent each visual impression as a memory feature. Of course, this leads to a very large memory space, so several models were evaluated to come up with a reasonable memory address format to limit the size. The simplest format is to concatenate the pixels together into a memory address. For example using the nine 8-bit pixel values from the 3×3 pixel region concatenated into a 72-bit address yields a space of $9 \times 8 = 72 = 2^{72}$ (4 zetabytes) which is impractical for desktop computers. Note that while it is possible to reduce the pixel resolution to less than 8-bits, the trade-off of bit precision does not seem worthwhile, for example 4-bit pixel precision yields $9 \times 4 = 2^{36} = 68\text{GB}$ which is outside the per-process address space limitation of typical desktop-class systems, and the level of detail of the pixels is greatly reduced which may not be desirable. The current implementation uses a trade-off to segment the address space into four regions, as shown in Fig. E.8. Most desktop computers using 32-bit and 64-bit memory addressing with commercial operating systems support at least 2GB of address space per process (*note: for practical*

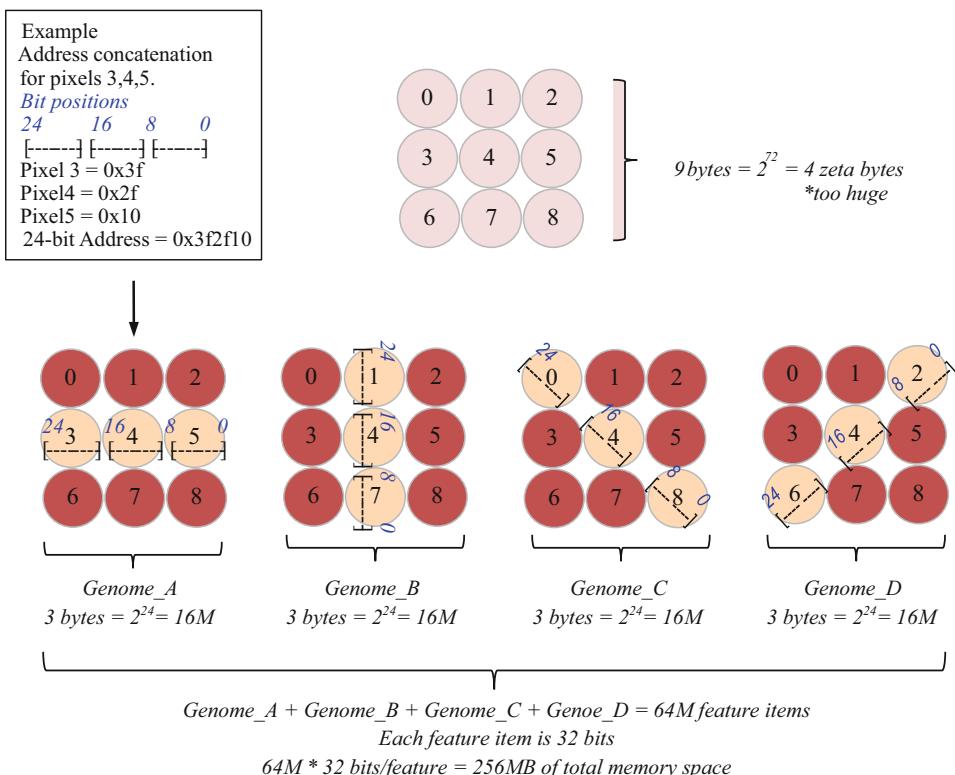


Figure E.8 This figure illustrates the method of defining four genomes from the 3×3 matrix, each genome is a set of three bytes forming a 24-bit address, which is the feature descriptor

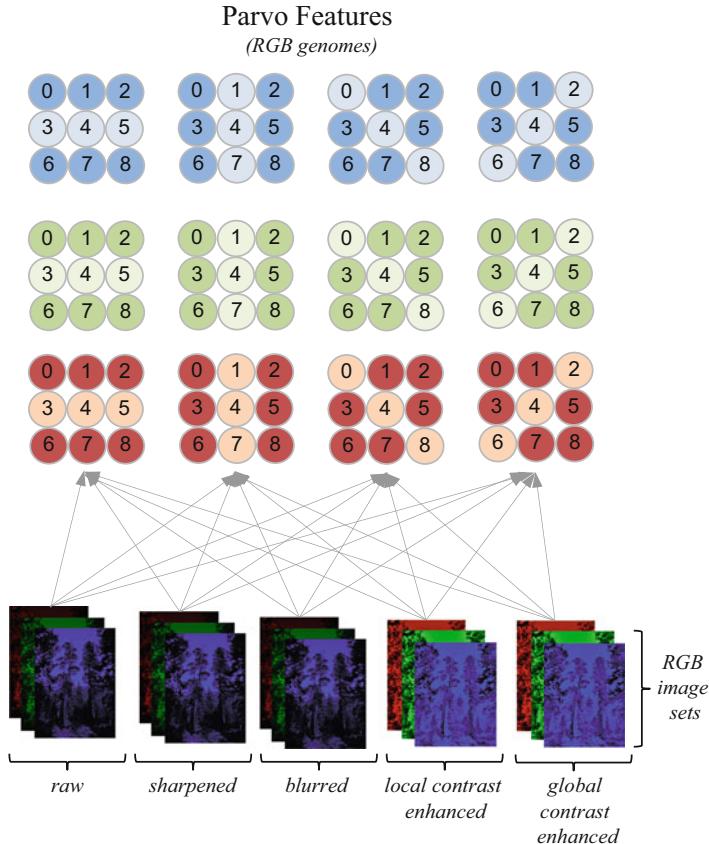


Figure E.9 This figure shows (*upper image*) parvo input processing of a total of 15 image inputs combined into separate RGB genomes

reasons, desktop computers and operating systems do not use all 64 bits of the CPU address lines to map against a contiguous 64-bit addressed memory space).

As shown in Fig. E.8, the address space is reduced by segmenting the address into four 16M feature segments for genomes A, B, C, and D. Each input 3×3 matrix of pixels is broken into the four genomes A, B, C, and D by combining three pixels from oriented line segments. Each genome memory unit contains a 32-bit unsigned int (four bytes) to record feature detection counts, so 16M 32-bit features consumes 64MB of memory. Using 5-bit pixels instead of 8-bit pixels yields genomes containing only 32k 32-bit features, and since 5-bit color images are realistic, 5-bit pixel values for the genome computations saves space.

In addition, the features can be coalesced together by using the *Quantization input θ* bits as shown in the code example in Fig. E.11 (note the quantization mask used to adjust each address).

As shown in Fig. E.9, parvo features are computed from five types of input images: raw, sharpened, blurred, local contrast enhanced, and global contrast enhanced, broken into 3 RGB channels, for a total of 15 input images combined into the four genomes A, B, C, D for each RGB color. The input images are combined together into the same genome, so each genome represents combined variations of raw, sharpened, blurred, and contrast enhanced impressions. Figure E.10 shows magno luminance channel input to compute the four magno genomes A, B, C, D, so for magno level segmentation into primal regions, any or all 5 types of retinal images may be used.

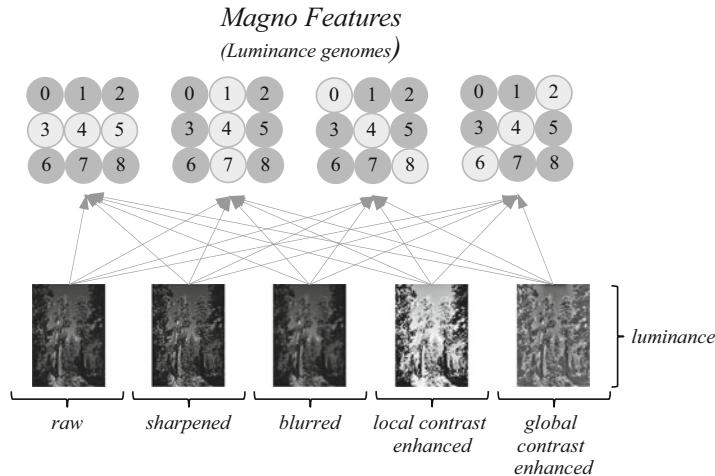


Figure E.10 This figure illustrates the luminance-channel magno image recorded into four genomes A, B, C, and D

For the parvo features as shown in Fig. E.9, for 8-bit pixels the total memory space occupied by each genome is $4\text{genomes} * 3\text{RGBcolors} * 16\text{Mfeatures} * 4\text{bytes} = 1.2\text{GB}$. For the magno features as shown in Fig. E.10, the total memory space occupied by each genome is $4\text{genomes} * 1\text{luminance_channel} * 16\text{Mfeatures} * 4\text{bytes} = 268\text{MB}$.

The parvo and magno feature genomes together comprise sixteen groups. As shown in the feature count details below, if all the tile feature genomes shown in Figs. E.9 and E.10 are concatenated into a contiguous address (for illustration purposes only), then the total virtual feature count for all magno and parvo genomes at 8-bit pixel resolution is 2^{384} .

Feature Count Details (8-Bit Pixels for Each RGBL Channel)

$$2^{24} = 16\text{M possible features for each genome}$$

$$4=3+1: 3 \text{ color genomes (R,G,B) for each parvo feature} + 1 \text{ magno luminance}$$

4 (A, B, C, D) genomes

$$2^{384} = \text{total possible features } (2^{(24 * 4 * 4)})$$

Feature Count Details (5-Bit Pixels for Each RGBL Channel)

$$2^{15} = 32\text{k possible features for each genome}$$

$$4=3+1: 3 \text{ color genomes (R,G,B) for each parvo feature} + 1 \text{ magno luminance}$$

4 (A, B, C, D) genomes

$$2^{240} = \text{total possible features } (2^{(15 * 4 * 4)})$$

*Note: the five parvo inputs (raw, sharp, blur, retinex, global contrast) are combined into the shape genomes (A, B, C, D) rather than separately recorded

It should be noted that the 2^{384} possible feature addresses for 8-bit pixels will not occur for real images, and therefore the entire address space will never be populated, and will be clustered around the center of the volume space like a 3D SDM, as illustrated in the Genome Renderings at the end of this appendix, due to the fact that maximally or widely diverging adjacent pixel values do not often occur in natural images, and instead, the adjacent pixels are usually closer together in value. Widely diverging adjacent pixel values are more characteristic of noise and saturation effects, while reasonable divergence corresponds to texture, and no divergence corresponds to no texture or a flat surface.

```

for(int y=0; y < ysize-2; y++)
{
    for(int x=0; x < xsize-2; x++)
    {
        getRegion3x3_u8((U8_PTR)filedata_8u_g, x, y, xsize, ysize, (U8_PTR)w3x3);
        //
        // [x x x] [x B x] [C x x] [x x D]
        // [A A A] [x B x] [X C x] [x D x]
        // [x x x] [x B x] [X X C] [D x x]
        //
        U32 genome_A_address = ((w3x3[1][1] & 0xff) | ((w3x3[0][1]<<8) & 0xff00) | ((w3x3[2][1]<<16) & 0xffff0000);
        U32 genome_B_address = ((w3x3[1][1] & 0xff) | ((w3x3[1][0]<<8) & 0xff00) | ((w3x3[1][2]<<16) & 0xffff0000);
        U32 genome_C_address = ((w3x3[1][1] & 0xff) | ((w3x3[0][0]<<8) & 0xff00) | ((w3x3[2][2]<<16) & 0xffff0000);
        U32 genome_D_address = ((w3x3[1][1] & 0xff) | ((w3x3[2][0]<<8) & 0xff00) | ((w3x3[0][2]<<16) & 0xffff0000);

        quantization_mask = 0xF8F8F8;
        magno_luminance_g[GENOME_A_0_DEGREES][genome_A_address & quantization_mask]++;
        magno_luminance_g[GENOME_B_90_DEGREES][genome_B_address & quantization_mask]++;
        magno_luminance_g[GENOME_C_135_DEGREES][genome_B_address & quantization_mask]++;
        magno_luminance_g[GENOME_D_45_DEGREES][genome_C_address & quantization_mask]++;

    }
}

```

Figure E.11 This figure illustrates the method of creating 24-bit addresses from 8-bit pixel values into the four genomes, using a quantization mask to coalesce and focus the features

So the extremes of the address space will likely never be populated for visual genome features, which will resemble sparse volumetric shapes.

Each time a given feature address is detected in the image, the count for the address is incremented, corresponding to feature commonality. The method for computing the feature addresses and counts is simple as illustrated in Fig. E.8, and relies on the quantization input to the VGM neuron as introduced in the *VGM Neuron Model* section above, see Fig. E.7. As a practical example using 8-bit pixel values for each RGB-L channel, the address can be quantized and focused by using a quantization space represented as an 8-bit hexadecimal mask value of 0xF8 (binary 1111 1000), and then each pixel value in the address is *bit-masked* into the desired quantization space to ignore the bottom 3 bits. This is illustrated in the following code snippet.

Visual Genome Sequences, Tiles, Strands, Bundles, Primal Features

VGM allows for a hierarchy of feature types, more details are provided in Krig [534]. Since DNA can apparently encode visual features into the visual neurons which then become biological defaults for subsequent learning, VGM allows for a set of *primal features* to be loaded into the model. The primal features correspond to *region shape masks*, corresponding to one of: (1) a segmentation mask derived from the actual Magno images (see Fig. E.12), (2) primal feature template masks containing some preexisting shape (See Appendix A), or (3) segmentations from another image set, which correspond to a form of transfer learning. The primal features are shape masks, and the masks comprise a region of the image over which a complete set of parvo features are computed and summed into a visual genome address space. Since the masks are computed at magno resolution, the masks are first scaled 5x prior to applying the masks to the full resolution parvo images.

The actual segmentation pipeline to create the shape masks is currently based on superpixel segmentations [256, 257] (*see also* Chap. 2 *regarding Morphology and Segmentation*). A range of superpixel size settings are used to collect a large set of candidate segmentations, with some amount of statistical criteria applied to select the optimal superpixel regions to use as shape masks, and which ones to ignore. The present statistical selection criteria uses a combination of Haralick feature metrics and the Krig Extended SDM metrics discussed in Chap. 3. The superpixels are the primal shape

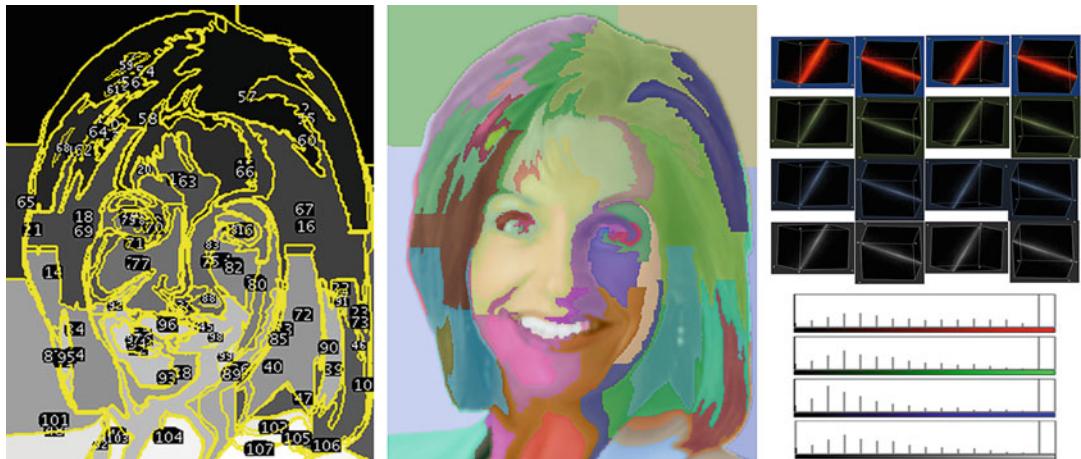


Figure E.12 (Left) one method of segmenting the magno image into mask regions, and (center) using each mask region as a template under which to define RGB-L parvo feature segmentation masks, and (right) the 16 visual genome parvo features, and RGB-L pixel histogram features for the mask region

masks for bounding the magno and parvo features. The current segmentation pipeline is based on heuristics and testing; however, a promising area for future work is developing a more automated and adaptive image segmentation pipeline, which is a central area of research for imaging.

If the shape masks are based on a segmentation of the current image being viewed, the registration and alignment of the masks is correct, and corresponding visual genome features are computed as intended. However, for primal features based on postulated primal shapes or transfer learning of segmented shapes from another image, the alignment of the shape masks does not exist. To use such primal features in the *image pixel space* would require the masks to be stepped across the entire image for correlation, and then for strong correlations the visual genome would be computed in the shape region. We reserve future work in this area to evaluate primal feature shape mask correlation as a part of the VGM, but for now we ignore it. However, the visual genome features based on a true segmentation of the current image will provide the desired results by allowing for correlation in the *visual genome space*, rather than in the image pixel space.

Besides the 16 parvo features recorded under each mask, other features can be computed for each mask shape, such as a color histogram of the pixels under the mask, and various shape factors such as Fourier circularity descriptors or Freeman chain codes (see Chap. 6).

Note that Hubel and Weiss define primal shapes for the lowest level receptive fields as *oriented edge-like features* which VGM models in a similar manner as Genomes A, B, C, and D, see Figs. E.9 and E.10. VGM also postulates higher level primal feature shapes at the *Strand* feature level as segmented regions resembling corners, blobs, and circular regions as shown in Fig. E.13. For example, a hierarchy of shapes can be defined in the strand feature model for segmentation of the image into familiar parts based on the strand shapes, to collect the corresponding magno and parvo features in the segmented features into a strand. The primal features are recorded over time by experiential learning, see [534].

Here is a summary of the VGM feature types.

- **Magno and Parvo Tiles**—A 3×3 tile region is translated into four line segments representing micro-features as a 24-bit address (for 8-bit pixels) representing genomes A, B, C, and D. The address is the feature. The count of all detected tiles of each type is recorded at the address as a bin

count for analysis and classification. Tiles genomes are the default feature stored in memory, from which strands of tiles are built up by the proxy agent to represent bundles of higher-level concepts. Magno features are used to segment the primal shapes.

- **Strands, Primal**—A segmented primal shape is used as the basis region to assemble sequences of 3×3 tiles into a strand, analogous to a DNA chain sequence, stored in a strand memory space. Strands are defined within a magno feature shape region as a set of parvo tile features contained in the region, or perhaps defined within other preexisting primal segmented shaped regions. The strand is a genome sequence type further defined in [534].
- **Bundles**—Groups of strands, typically representing a high-level concept defined by the proxy agent, and stored in a bundle memory space. The bundle genome sequence types are defined in [534].

In simple terms, parvo features are like texture and color tile codes. Magno features are shapes. Strands are shaped sets of magno and parvo tiles. The strand memory stores the variable length strands which are collections of magno and parvo features. The strands can be created several ways, such as using preexisting feature shape masks, or masks segmented from real images. Bundles are sets of strands. The proxy agent associates and creates sets of strands into bundles.

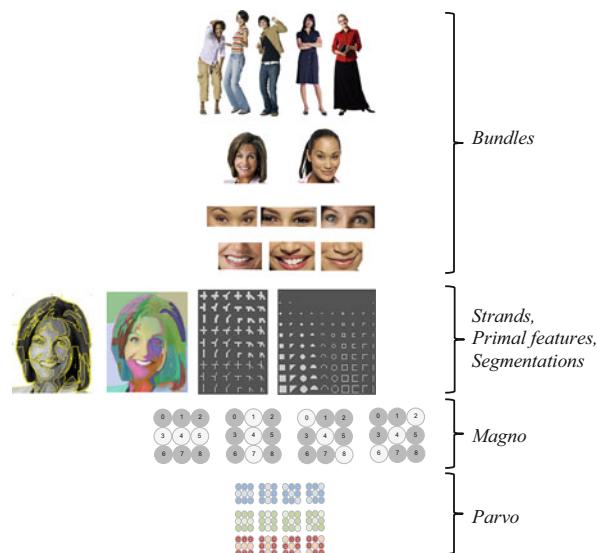


Figure E.13 This figure illustrates example primal shapes for parvo, magno, strand, and bundle features. Note that the parvo and magno features are primal shapes consisting of oriented edges, following Hubel and Weiss. Strands are also primal shapes segmenting and collecting lists of the underlying magno and parvo tile textures from the RGB and Luma regions. Bundles are high level concepts composed of the primal shape strands

VGM Proxy Agent

The proxy agent represents the intelligence of the system, and is not defined in the VGM model. The VGM represents a memory space for storing and organizing visual features, and provides controls for the proxy agent. A variety of proxy agents can be devised for an application, since the VGM does not

attempt to model the higher-level consciousness necessary. However, the following assumptions are made to devise controls within the VGM to enable the proxy agent:

- **Retinal Processing Controller**—VGM uses a very simple retinal model to provide biologically plausible image processing for the Parvo pathway (raw, sharp, blur, local contrast enhancement for high dynamic range adjustments, and global contrast normalization) and Magno pathway. This basic retinal model of processing is adequate to emulate the visual pathway, instead of resorting to a range of ad hoc processing methods as typically applied in CNN architecture feature layers. For example, the typical CNN neuron model uses a wide range of pre and post processing methods, see the taxonomy in Chap. 9 and Table 9.3. Instead the VGM assumes a strictly memory and comparator based neuron model, with no processing at the neuron except for the bit-level quantization control for *attentional level of detail*, which is biologically plausible.
- **Training Protocol Controller**—We propose a detailed learning and training model in [534] which provides for highly segmented labeled regions masked off to exclude extraneous details, rather than simply providing labeled images with multiple labels per image, or ill-prepared images with occlusions, geometric variations, lighting and color variations. Better prepared training sets should yield better results when the system is applied over real-life variations.
- **Hypothesis Controller**—A method to examine and compare objects against a range of hypotheses is part of an intelligent controller to direct domain-specific (1) creation of new memory records, and (2) to classify objects. The VGM neural model and memory hierarchy are flexible with no restriction on the controllers, and provide a quantization space to carry out progressive refinement of hypothesis evaluations.
- **Multi-Memory-Region Controller**—A proxy agent may control multiple VGM memory models which are pretrained on a range of subjects, to simulate an entire visual cortex with separate feature regions. We develop an architecture using multiple-memory regions in [534].
- **Environmental Controls In Feature Space**—It is also possible to provide some sort of environmental processing to the *pixel values* within each feature, to allow for hypothesis testing at classification time using colorimetric and environmentally accurate pixel processing to alter the features to test a given hypothesis. For example, pixel color and luminance will change at different times of the day, so the VGM model allows for what-if pixel processing of the feature space to account for seasonal lighting, cloud cover, rain, snow, fog, noise, or haze. To perform environmental hypothesis testing, the dynamic range for each RGB-L color component can be altered, treating color channels and luminance independently in each pixel in each feature, by adjusting pixel values in a color-space accurate manner. Environment-specific genomes can be recomputed, and classification can be repeated. Such environmental hypothesis testing is highly relevant for surveillance and military applications. Environmental processing of the feature sets as part of a detection and recognition process is explored in [534].

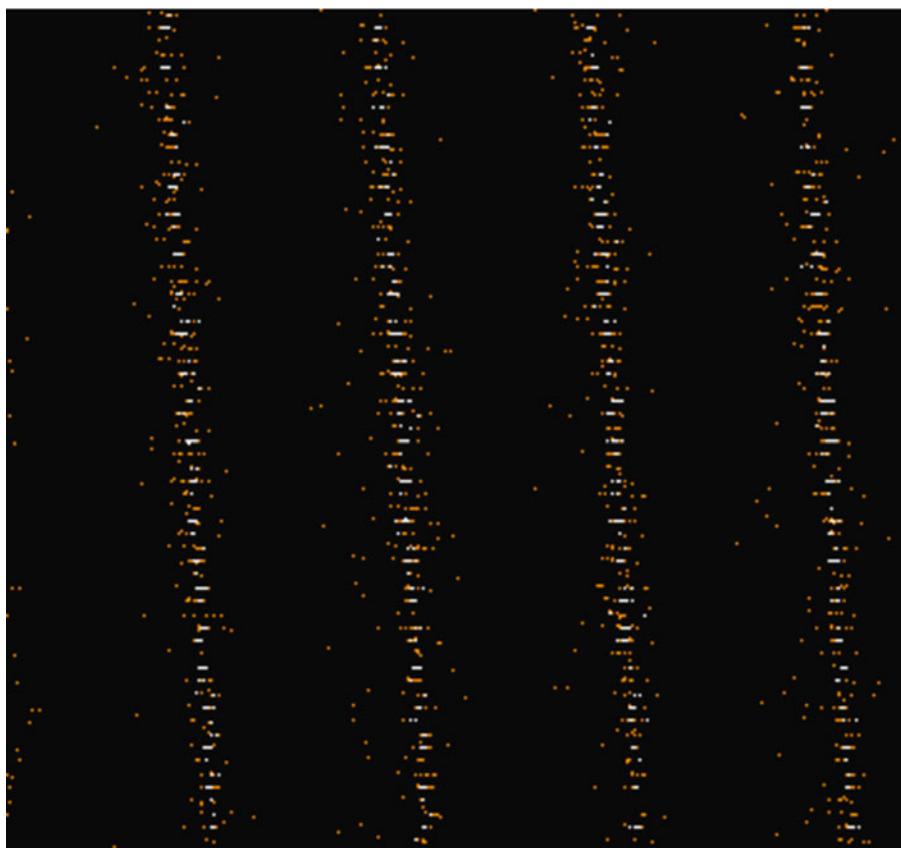
Summary

In summary, the VGM architecture is based on view-based model assumptions, a simplified retinal processing model, separate paths and purposes for magno and parvo features, and a hierarchy of primal strand and bundle features based on memory impressions, rather than using local feature descriptors, or a hierarchical feature scale pyramid created via pooling and subsampling as is typical for CNNs.

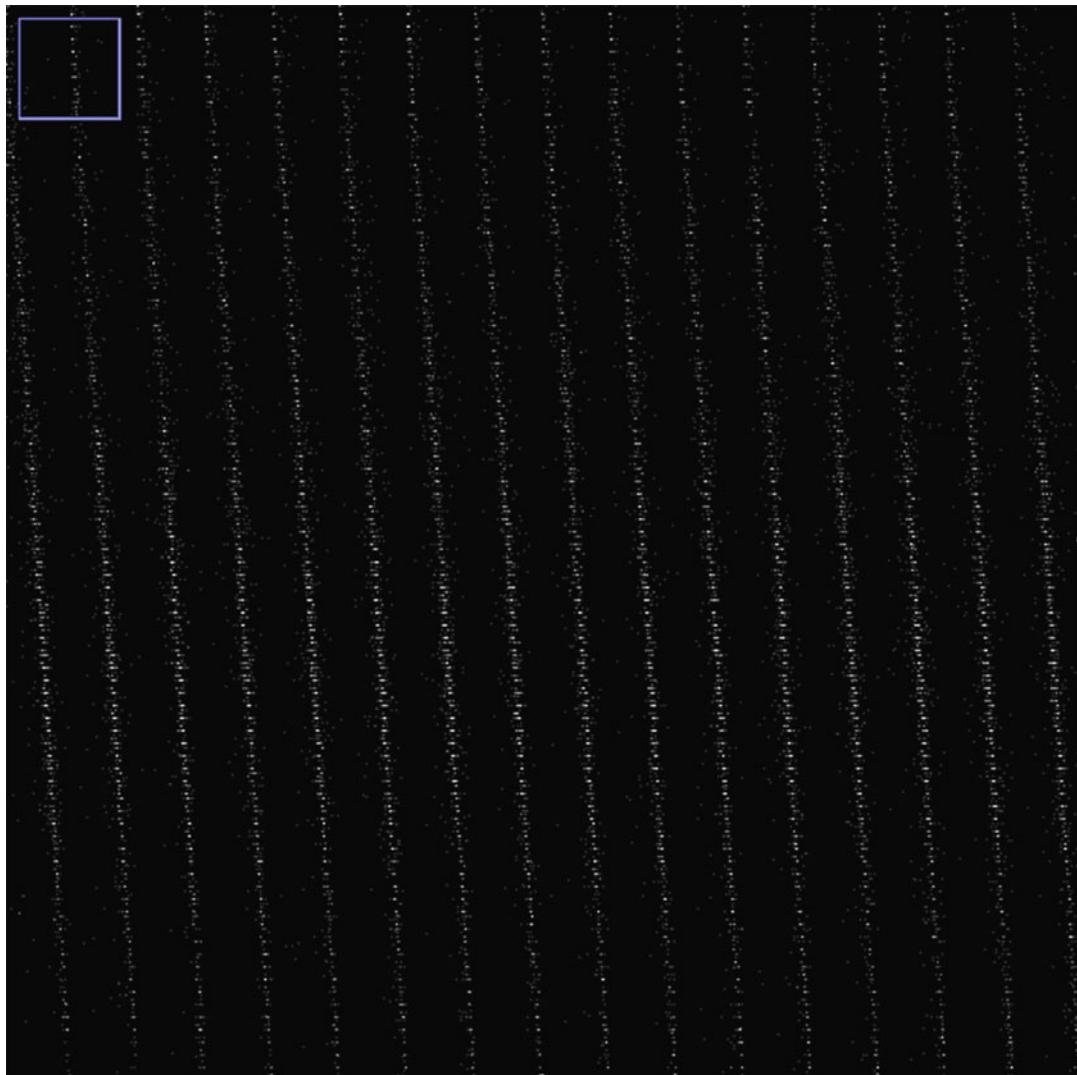
Tile Genome Renderings

For visualizing the tile genomes, each 16M feature genome is rendered into a 4096×4096 32-bit integer image. Then, some false coloring is applied to visualize the hot-spots in the address space where the most common features were detected. Image set 1 illustrates a representation of the magno A genome for an indoor scene (Bandits image), and image set 2 represents the magno A genomes for an outdoor scene (Sequoia image). Notice that the indoor scene contains many flat regions with similar texture and color for doors and walls, while the outdoor scene contains a much wider range of textures and almost no flat surfaces of similar texture or color. The magno luminance genomes clearly reflect the color and texture.

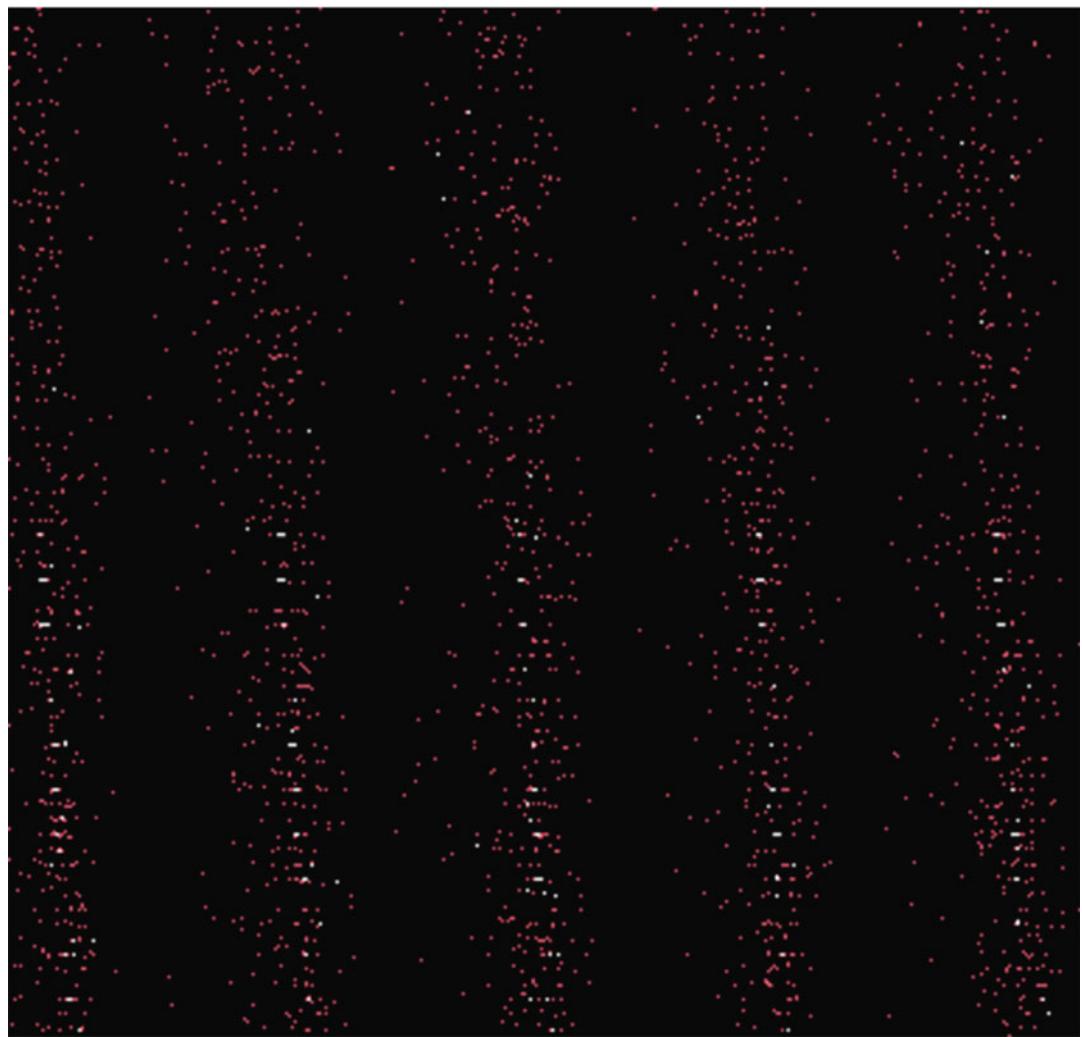
Note that classification and matching results are provided in Krig [534] for a wider range of image classes.

Image Set 1: Indoor Scene of Little Girls (Bandits), 24-Bit RGB 2448 × 3264 Image

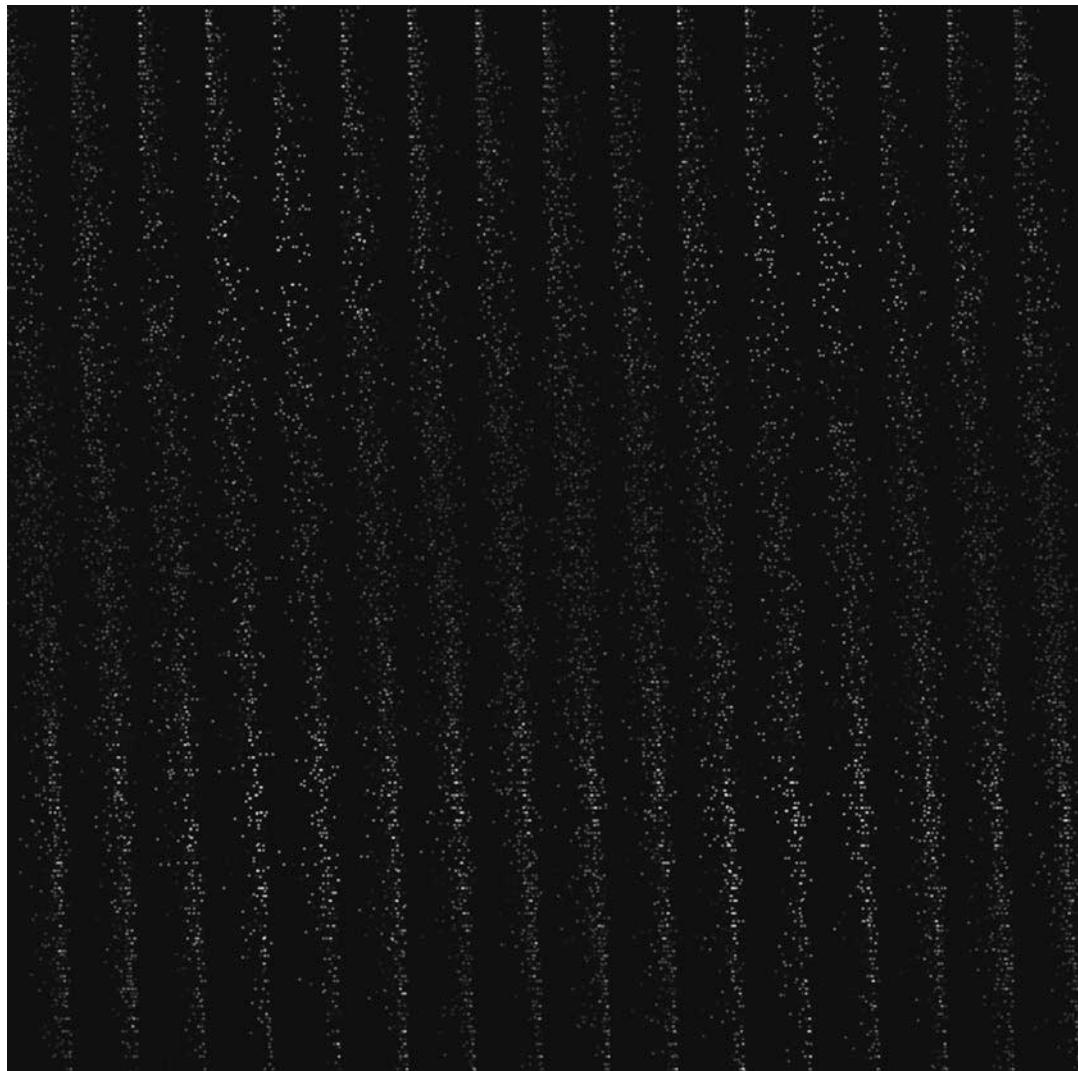
Zoom-in close on region of 4096×4096 visualized Genome A address space for indoor Bandits image, false colored to show hot spots or commonly detected Genome A features



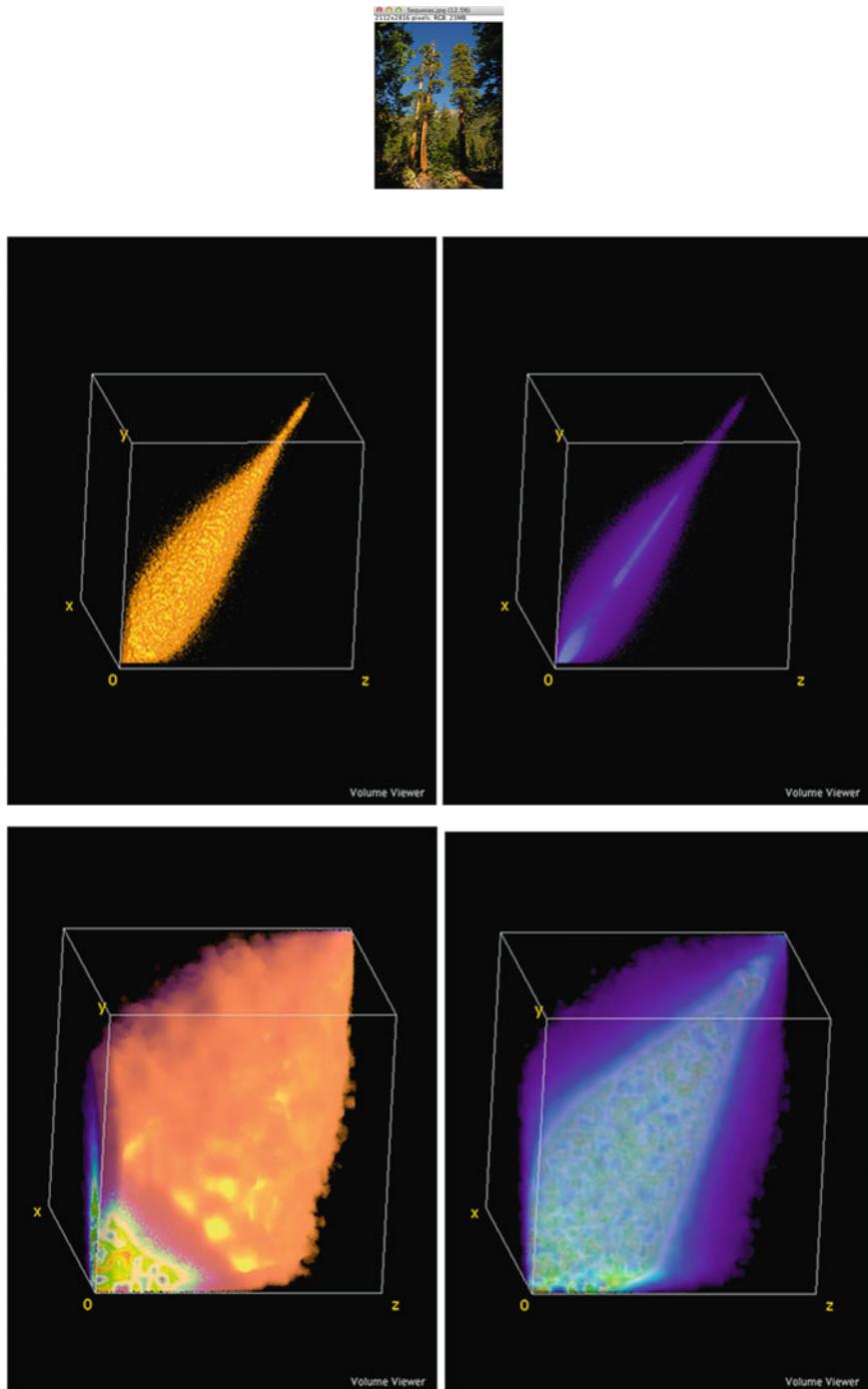
Full 4096×4096 visualized Genome A address space for indoor Bandits image, highlighted to show commonly detected features

Image Set 2: Outdoor Scene of Giant Sequoia Trees 24-Bit RGB 2112 × 2816 Image

Zoom-in close on region of 4096×4096 visualized Genome A address space for outdoor Sequoia image, false colored to show hot spots or commonly detected Genome A features

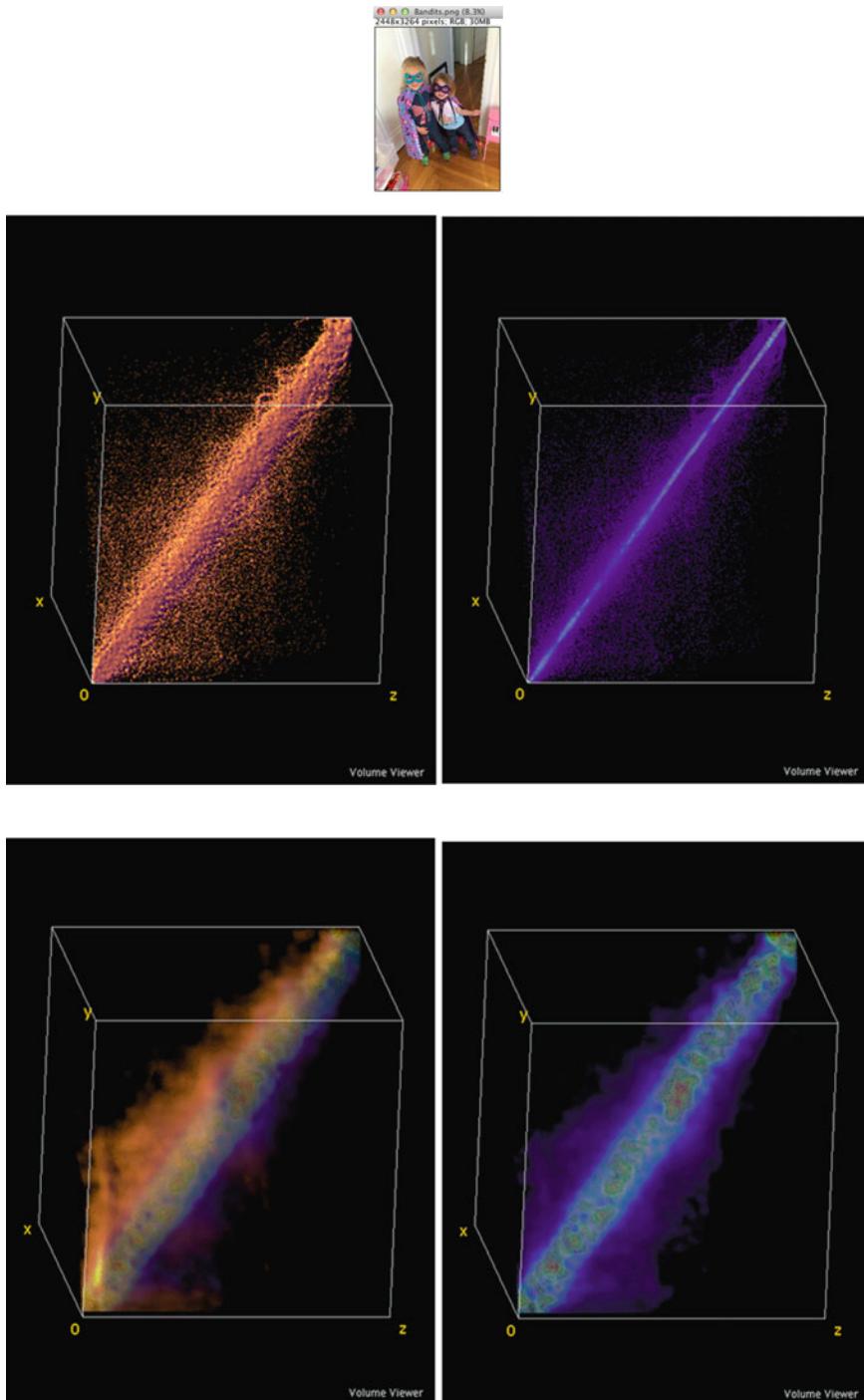


Full 4096×4096 visualized Genoome A address space for outdoor Sequoia image, highlighted to show commonly detected features

Image Set 3: Comparative Volume Renderings of Entire Genome A Feature Space for Sequoias Scene, Representing Each n-Bit Feature Component for the Volume Coordinates x , y and z 

(Top) $256 \times 256 \times 256$ 8-bit pixel value volume renderings of genome A features for the Sequoias image, (Bottom) $32 \times 32 \times 32$ 5-bit pixel value volume renderings, notice the dynamic range quantization space reduction in 5-bit pixel images vs. 8-bit pixel images. 8-bit images have a larger quantization space

Image Set 4: Comparative Volume Renderings of Entire Genome A Feature Space for Bandits Scene, Representing Each n-Bit Feature Component for the Volume Coordinates x , y and z



(Top) $256 \times 256 \times 256$ 8-bit pixel value volume renderings of genome A features for the Bandits image, (Bottom) $32 \times 32 \times 32$ 5-bit pixel value volume renderings. Notice the dynamic range quantization space reduction in 5-bit pixel images vs. 8-bit pixel images. 8-bit images have a larger quantization space

References

1. Bajcsy, R.: Computer description of textured surfaces. Int. Conf. Artif. Intell. Stat. (1973)
2. Bajcsy, R., Lieberman, L.: Texture gradient as a depth cue. Comput. Graph. Image Process. **5**(1), (1976)
3. Cross, G.R., Jain, A.K.: Markov random field texture models. PAMI **5**(1), (1983)
4. Gonzalez, R., Woods, R.: Digital Image Processing, 3rd edn. Prentice-Hall, Englewood Cliffs, NJ (2007)
5. Haralick, R.M.: Statistical and structural approaches to texture. Proc. Int. Joint Conf. Pattern Recogn. (1979)
6. Haralick, R.M., Shanmugan, R., Dinstein, I.: Textural features for image classification. IEEE Trans. Syst. Man Cybern. **3**(6), (1973)
7. Hu, M.K.: Visual pattern recognition by moment invariants. IRE Trans. Inform. Theor. **8**(2), (1962)
8. Lu, H.E., Fu, K.S.: A syntactic approach to texture analysis. Comput. Graph. Image Process. **7**(3), (1978)
9. Pratt, W.K.: Digital image processing, 3rd edn. Wiley, Hoboken, NJ (2002)
10. Rosenfeld, A., Kak, A.C.: Digital picture processing, 2nd edn. Academic Press, New York (1982)
11. Tomita, F., Shirai, Y., Tsuji, S.: Description of texture by a structural analysis. Pattern. Anal. Mach. Intell. **4**(2), (1982)
12. Wong, R.Y., Hall, E. L.: Scene matching with invariant moments. Comput. Graph. Image Process. **8** (1978)
13. Guoying, Z., Pietikäinen, M.: Dynamic texture recognition using local binary patterns with an application to facial expressions. Trans. Pattern. Anal. Mach. Intell. **29**(6), 915–928 (2007)
14. Kellokumpu, V., Guoying Z., Pietikäinen, M.: Human activity recognition using a dynamic texture based method
15. Guoying, Z., Pietikäinen, M.: Dynamic texture recognition using local binary patterns with an application to facial expressions. Pattern. Anal. Mach. Intell. **29**(6), 915–928 (2007)
16. Eichmann, G., Kasparis, T.: Topologically invariant texture descriptors. Comput. Vis. Graph. Image Process. **41** (3), (1988)
17. Lam, S.W.C., Ip, H.H.S.: Structural texture segmentation using irregular pyramid. Pattern Recogn. Lett. **15**(7), (1994)
18. Pietikäinen, M., Guoying, Z., Hadid, A.: Computer Vision Using Local Binary Patterns. Springer, New York (2011)
19. Ojala, T., Pietikäinen, M., Hardwood, D.: Performance evaluation of texture measures with classification based on kullback discrimination of distributions. Proc. Int. Conf. Pattern. Recogn. (1994)
20. Ojala, T., Pietikäinen, M., Hardwood, D.: A comparative study of texture measures with classification based on feature distributions. Pattern Recogn. **29** (1996)
21. Van Ginneken, B., Koenderink, J.J.: Texture histograms as a function of irradiation and viewing direction. Int. J. Comput. Vis. **31**(2/3), 169–184 (1999)
22. Stelu, A., Arati, K., Dong-Hui, X.: Texture analysis for computed tomography studies. Visual Computing Workshop DePaul University, (2004)
23. Krig, S.A.: Image texture analysis using spatial dependency matrices. Krig Research White Paper Series, (1994)
24. Laws, K.I.: Rapid texture identification. SPIE **238** (1980)
25. Bajcsy, R.K.: Computer identification of visual surfaces. Comput. Graph. Image Process. **2**(2), 118–130 (1973)
26. Kaizer, H.: A quantification of textures on aerial photographs. MS Thesis, Boston University, (1955)
27. Laws, K.I.: Texture energy measures. Proceedings of the Image Understanding Workshop, (1979)
28. Laws, K.I.: Rapid texture identification. SPIE **238** (1980)
29. Laws, K.I.: Textured image segmentation. PhD Thesis, University of Southern California, (1980)

30. Ade, F.: Characterization of textures by “Eigenfilters.” *Signal Process.* **5** (1983)
31. Davis, L.S.: Computing the spatial structures of cellular texture. *Comput. Graph. Image Process.* **11**(2), (1979)
32. Eichmann, G., Kaspars, T.: Topologically invariant texture descriptors. *Comput. Vis. Graph. Image Process.* **41**(3), (1988)
33. Lam, S.W.C., Ip, H.H.S.: Structural texture segmentation using irregular pyramid. *Pattern Recogn. Lett.* **15**(7), (1994)
34. Pietikäinen, M., Guoying, Z., Hadid, A.: Computer vision using local binary patterns. Springer, New York (2011)
35. Ojala, T., Pietikäinen, M., Hardwood, D.: Performance evaluation of texture measures with classification based on kullback discrimination of distributions. *Proc. Int. Conf. Pattern. Recogn.* (1994)
36. Ojala T., Pietikäinen, M., Hardwood, D.: A comparative study of texture measures with classification based on feature distributions. *Pattern Recogn.* **29** (1996)
37. Pun, C.M., Lee, M.C.: Log-polar wavelet energy signatures for rotation and scale invariant texture classification. *Trans. Pattern. Anal. Mach. Intell.* **25**(5), (2003)
38. Spence, A., Robb, M., Timmins, M., Chantler, M.: Real-time per-pixel rendering of textiles for virtual textile catalogues. *Proceedings of INTEDEC*, Edinburgh, (2003)
39. Lam, S.W.C., Horace, H.S.I.: Adaptive pyramid approach to texture segmentation. *Comput. Anal. Images Patterns Lect. Notes Comput. Sci.* **719**, 267–274 (1993)
40. Dana, K.J., van Ginneken, B., Nayar, S.K., Koenderink, J.J.: Reflectance and Texture of Real World Surfaces. Technical Report CUCS-048-96, Columbia University, (1996)
41. Dana, K.J., van Ginneken, B., Nayar, S.K., Koenderink, J.J.: Reflectance and texture of real world surfaces. *Conf. Comput. Vis. Pattern Recogn.* (1997)
42. Dana, K.J., van Ginneken, B., Nayar, S.K., Koenderink, J.J.: Reflectance and texture of real world surfaces. *ACM Trans. Graph.* (1999)
43. Suzuki, M.T., Yaginuma, Y.: A solid texture analysis based on three dimensional convolution kernels. *Proc. SPIE* **6491**, (2007)
44. Suzuki, M.T., Yaginuma, Y., Yamada, T., Shimizu, Y.: A shape feature extraction method based on 3D convolution masks. *Eighth IEEE International Symposium on Multimedia, ISM’06.* (2006)
45. Guoying, Z., Pietikainen, M.: Dynamic texture recognition using local binary patterns with an application to facial expressions. *Trans. Pattern. Anal. Mach. Intell.* **29** (2007)
46. Hadjidemetriou, E., Grossberg, M.D., Nayar, S.K.: Multiresolution histograms and their use for texture classification. *IEEE PAMI* **26**
47. Hadjidemetriou, E., Grossberg, M.D., Nayar, S.K.: Multiresolution histograms and their use for recognition. *IEEE PAMI* **26**(7), (2004)
48. Lee, K.L., Chen, L.H.: A new method for coarse classification of textures and class weight estimation for texture retrieval. *Pattern Recogn. Image Anal.* **12**(4), (2002)
49. Van Ginneken, B., Koenderink, J.J.: Texture histograms as a function of irradiation and viewing direction. *Int. J. Comput. Vis.* **31**(2/3), 169–184 (1999)
50. Shu, L., Chung, A.C.S.: Texture classification by using advanced local binary patterns and spatial distribution of dominant patterns. *ICASSP 2007. IEEE Int. Conf. Acoust. Speech Signal Process.* (2007)
51. Stelu, A., Arati, K., Dong-Hui, X.: Texture analysis for computed tomography studies. *Visual Computing Workshop DePaul University*, (2004)
52. Ade, F.: Characterization of textures by “Eigenfilters.” *Signal Process.* **5** (1983)
53. Rosin, P.L.: Measuring corner properties. *Comput. Vis. Image Understand.* **73**(2)
54. Russel, B., Jianxiong, X., Torralba, A.: Localizing 3D cuboids in single-view images. *Conf. Neural Inform. Process. Syst.* (2012)
55. Snavely, N., Seitz, S.M., Szeliski, R.: Photo tourism: exploring photo collections in 3D. *ACM Trans. Graph. (SIGGRAPH Proc.)* (2006)
56. Snavely, N., Seitz, S.M., Szeliski, R.: Modeling the world from internet photo collections. *Int. J. Comput. Vis. (TBP)*
57. Furukawa, Y., Curless, B., Seitz, S.M., Szeliski, R.: Towards internet-scale multi-view stereo. *Conf. Comput. Vis. Pattern Recogn.* (2010)
58. Yunpeng, L., Snavely, N., Huttenlocher, D., Fua, P.: Worldwide pose estimation using 3D point clouds. *Eur. Conf. Comput. Vis.* (2012)
59. Russell, B., Torralba, A., Murphy, K., Freeman, W.T.: LabelMe: A database and web-based tool for image annotation. *Int. J. Comput. Vis.* **77** (2007).
60. Oliva, A., Torralba, A.: Modeling the shape of the scene: a holistic representation of the spatial envelope. *Int. J. Comput. Vis.* **42** (2001)
61. Lai, K., Bo, L., Ren, X., Fox, D.: A large-scale hierarchical multi-view RGB-D object dataset. *Int. Conf. Robot Autom.* (2011)

62. Xiao, J., Hays, J., Ehinger, K., Oliva, A., Torralba, A.: SUN database: large-scale scene recognition from abbey to zoo. *Conf. Comput. Vis. Pattern Recogn.* (2010)
63. Fei-Fei, L., Fergus, R., Perona, P.: Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories. *Conf. Comput. Vis. Pattern Recogn.* (2004)
64. Fei-Fei, L.: ImageNet: crowdsourcing, benchmarking & other cool things. CMU VASC Semin. (2010)
65. Pirsiavash, H., Ramanan, D.: Detecting activities of daily living in first-person camera views. *Conf. Comput. Vis. Pattern Recogn.* (2012)
66. Quattoni, A., Torralba, A.: Recognizing indoor scenes. *Conf. Comput. Vis. Pattern Recogn.* (2009)
67. Lai, K., Bo, L., Ren, X., Fox, D.: A large-scale hierarchical multi-view RGB-D object dataset. *Int. Conf. Robot Autom.* (2011)
68. Silberman, N., Hoiem, D., Kohli, P., Fergus, R.: Indoor segmentation and support inference from RGBD images. *Eur. Conf. Comput. Vis.* (2012)
69. Xiaofeng R., Philipose, M.: Egocentric recognition of handled objects: benchmark and analysis. *CVPR Workshops*, (2009)
70. Xiaofeng, R., Gu, C.: Figure-ground segmentation improves handled object recognition in egocentric video. *Conf. Comput. Vis. Pattern Recogn.* (2009)
71. Fathi, A., Li, Y., Rehg, J.M.: Learning to recognize daily actions using gaze. *Eur. Conf. Comput. Vis.* (2012)
72. Dana, K.J., van Ginneken, B., Nayar, S.K., Koenderink, J. J.: Reflectance and texture of real world surfaces. *Trans. Graph.* **18**(1), (1999)
73. Ce, L., Sharan, L., Adelson, E.H., Rosenholtz, R.: Exploring features in a Bayesian framework for material recognition. *Conf. Comput. Vis. Pattern Recogn.* (2010)
74. Huang, G.B., Ramesh, M., Berg, T., Learned-Miller, E.: Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments. Technical report 07-49, University of Massachusetts, Amherst, (2007)
75. Gross, R., Matthews, I., Cohn, J.F., Kanade, T., Baker, S.: Multi-PIE. Proceedings of the Eighth IEEE International Conference on Automatic Face and Gesture Recognition, (2008)
76. Yao, B., Jiang, X., Khosla, A., Lin, A.L., Guibas, L.J., Fei-Fei, L.: Human action recognition by learning bases of action attributes and parts. *Int. Conf. Comput. Vis.* (2011)
77. LeCun, Y., Huang, F.J., Bottou, L.: Learning methods for generic object recognition with invariance to pose and lighting. *Proc. Conf. Comput. Vis. Pattern Recogn.* (2004)
78. McCane, B., Novins, K., Crannitch, D., Galvin, B.: On benchmarking optical flow. *Comput. Vis. Image Understand.* **84**(1), (2001)
79. Pirsiavash, H., Ramanan, D.: Detecting activities of daily living in first-person camera views. *Conf. Comput. Vis. Pattern Recogn.* Providence, Rhode Island. (2012)
80. Hamarneh, G., Jassi, P., Tang, L.: Simulation of ground-truth validation data via physically- and statistically-based warps. *MICCAI 2008*, the 11th International Conference on Medical Image Computing and Computer Assisted Intervention
81. Prastawa, M., Bullitt, E., Gerig, G.: Synthetic ground truth for validation of brain tumor MRI segmentation. *MICCAI 2005*, the 8th International Conference on Medical Image Computing and Computer Assisted Intervention
82. Vedaldi, A., Ling, H., Soatto, S.: Knowing a good feature when you see it: ground truth and methodology to evaluate local features for recognition. *Comput. Vis. Stud. Comput. Intell.* **285**, 27–49 (2010)
83. Dutagaci, H., Cheung, C.P., Godil, A.: Evaluation of 3D interest point detection techniques via human-generated ground truth. *The Visual Computer* **28** (2012)
84. Rosin, P.L.: Augmenting corner descriptors. *Graph. Model. Image Process.* **58**(3), (1996)
85. Rockett, P.I.: Performance assessment of feature detection algorithms: a methodology and case study on corner detectors. *Trans. Image Process.* **12**(12), (2003)
86. Shahrokni, A., Ellis, A., Ferryman, J.: Overall evaluation of the PETSc009 results. *IEEE PETSc* (2009)
87. Over, P., Awad, G., Sanders, G., Shaw, B., Martial, M., Fiscus, J., Kraaij, W., Smeaton, A.F.: TRECVID 2013: An Overview of the Goals, Tasks, Data, Evaluation Mechanisms, and Metrics, NIST USA, (2013)
88. Horn, B.K.P., Schunck, B.G.: Determining Optical Flow. AI Memo 572, Massachusetts Institute of Technology, (1980)
89. Everingham, M., Van Gool, L., Williams, C. K. I., Winn, J., Zisserman, A.: The PASCAL visual object classes (VOC) challenge. *Int. J. Comput. Vis.* **88**(2), (2010)
90. Liu, J., Luo, J., Shah, M.: Recognizing realistic actions from videos “in the Wild.” *Conf. Comput. Vis. Pattern Recogn.* (2009)
91. Arbelaez, P., Maire, M., Fowlkes, C., Malik, J.: Contour detection and hierarchical image segmentation. *Trans. Pattern. Anal. Mach. Intell.* **33**(5), (2011)
92. Fisher, R.B.: PETSc04 surveillance ground truth data set. *Proc. IEEE PETSc*. (2004)

93. Quan Y., Thangali, A., Ablavsky, V., Sclaroff, S.: Learning a family of detectors via multiplicative kernels. *Pattern. Anal. Mach. Intell.* **33**(3), (2011)
94. Ericsson, A., Karlsson, J.: Measures for benchmarking of automatic correspondence algorithms. *J. Math. Imaging Vis.* (2007)
95. Takhar, D., et al.: A new compressive imaging camera architecture using optical-domain compression. In: Proceedings of IS&T/SPIE Symposium on Electronic Imaging (2006)
96. Marco, F.D., Baraniuk, R.G.: Kronecker compressive sensing. *IEEE Trans. Image Process.* **21**(2), (2012)
97. Weinzaepfel, P., Jegou, H., Perez, P.: Reconstructing an image from its local descriptors. *Conf. Comput. Vis. Pattern Recogn.* (2011)
98. Dalal, N., Triggs, B.: Histograms of oriented gradients for human detection. *Conf. Comput. Vis. Pattern Recogn.* (2005)
99. Tuytelaars, T., Mikolajczyk, K.: Local invariant feature detectors: a survey. *Found. Trends Comput. Graph. Vis.* **3**(3), 177–280 (2007)
100. Hartigan, J.A.: Clustering Algorithms. Wiley, New York (1975)
101. Fischler, M.A., Bolles, R.C.: Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM* **24**(6), (1981)
102. Sunglok, C., Kim, T., Yu, W.: Performance evaluation of RANSAC family. *Br. Mach. Vis. Assoc.* (2009)
103. Hartigan, J.A., Wong, M.A.: Algorithm AS 136: A K-means clustering algorithm. *J. Royal Stat. Soc. Ser. C Appl. Stat.* **28**(1), 100–108 (1979)
104. Voronoi, G.: Nouvelles applications des paramètres continus à la théorie des formes quadratiques. *Journal für die Reine und Angewandte Mathematik* **133** (1908)
105. Capel, D.: Random forests and ferns. Penn. State University Computer Vision Laboratory, seminar lecture notes online:. ForestsAndFernsTalk.pdf.
106. Xiaofeng, R., Malik, J.: Learning a classification model for segmentation
107. Lai, K., Bo, L., Ren, X., Fox, D.: Sparse distance learning for object recognition combining RGB and depth information
108. Xiaofeng, R., Ramanan, D.: Histograms of sparse codes for object detection. *Conf. Comput. Vis. Pattern Recogn.* (2013)
109. Liefeng, B., Ren, X., Fox, D.: Multipath sparse coding using hierarchical matching pursuit. *Conf. Comput. Vis. Pattern Recogn.* (2013)
110. Herbst, E., Ren, X., Fox, D.: RGB-D flow: dense 3-D motion estimation using color and depth. *IEEE Int. Conf. Robot. Autom. (ICRA)* (2013)
111. Xiaofeng, R., Bo, L.: Discriminatively trained sparse code gradients for contour detection. *Conf. Neural Inform. Process. Syst.* (2012)
112. Rublee, E., Rabaud, V., Konolige, K., Bradski, G.: ORB: an efficient alternative to SIFT or SURF. *ICCV '11 Proceedings of the 2011 International Conference on Computer Vision*
113. Rosenfeld, A., Pfaltz, J.L.: Distance functions on digital images. *Pattern Recog.* **1**, 33–61 (1968)
114. Richardson, A., Olson, E.: Learning convolutional filters for interest point detection. *IEEE Int. Conf. Robot. Autom. ICRA'13 IEEE*, 631–637, (2013)
115. Moon, T.K., Stirling, W.C.: Mathematical Methods and Algorithms for Signal Processing. Prentice-Hall, Englewood Cliffs, NJ (1999)
116. Liefeng, B., Ren, X., Fox, D.: Multipath sparse coding using hierarchical matching pursuit. *Conf. Comput. Vis. Pattern Recogn.* (2013)
117. Ren, X., Ramanan, D.: Histograms of sparse codes for object detection. *Conf. Comput. Vis. Pattern Recogn.* (2013)
118. Olshausen, B., Field, D.: Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature* **381**(6583), 607–609 (1996)
119. d'Angelo, E., Alahi, A., Vanderghenst, P.: Beyond bits: reconstructing images from local binary descriptors. Swiss Federal Institute of Technology, 21st International Conference on Pattern Recognition (ICPR), (2012)
120. Dengsheng, Z., Lu, G.: Review of shape representation and description techniques. *J. Pattern Recogn. Soc.* **37**, 1–19 (2004)
121. Yang M., Kidiyo, K., Joseph, R.: A survey of shape feature extraction techniques. *Pattern Recogn.* 43–90, (2008)
122. Alahi, A., Ortiz, R., Vanderghenst, P.: Freak: fast retina keypoint. *Conf. Comput. Vis. Pattern Recogn.* (2012)
123. Leutenegger, S., Chli, M., Siegwart, R.Y.: BRISK: binary robust invariant scalable keypoints. *Int. Conf. Comput. Vis.* (2011)
124. Calonder, M., Lepetit, V., Strecha, C., Fua, P.: BRIEF: binary robust independent elementary features. *ECCV'10 Proceedings of the 11th European Conference Computer Vision: Part IV*, (2010)
125. Calonder, M., et al.: BRIEF: computing a local binary descriptor very fast. *Pattern. Anal. Mach. Intell.* **34** (2012)

126. Rublee, E., Rabaud, V., Konolige, K., Bradski, G.: ORB: an efficient alternative to SIFT or SURF. ICCV '11 Proceedings of the 2011 International Conference on Computer Vision, (2011)
127. von Hundelshausen, F., Sukthankar, R.: D-Nets: beyond patch-based image descriptors. Conf. Comput. Vis. Pattern Recogn. (2012)
128. Krig, S.: RFAN radial fan descriptors. Picture Center Imaging and Visualization System, White Paper Series (1992)
129. Krig, S.: Picture Center Imaging and Visualization System. Krig Research White Paper Series (1994)
130. Rosten, E., Drummond, T.: FAST machine learning for high-speed corner detection. Eur. Conf. Comput. Vis. (2006)
131. Rosten, E., Drummond, T.: Fusing points and lines for high performance tracking. Int. Conf. Comput. Vis. (2005)
132. Liefeng, B., Ren, X., Fox, D.: Hierarchical matching pursuit for image classification: architecture and fast algorithms. Conf. Neural Inform. Process. Syst. (2011)
133. Miksik, O., Mikolajczyk, K.: Evaluation of local detectors and descriptors for fast feature matching. Int. Conf. Pattern. Recogn. (2012)
134. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. J. Comput. Syst. **55**(1), 119–139 (1997)
135. Gleason, J.: BRISK (Presentation by Josh Gleason) at International Conference on Computer Vision, (2011)
136. Mikolajczyk, K., Schmid, C.: A performance evaluation of local descriptors. Pattern. Anal. Mach. Intell. IEEE Trans. **27**(10), (2005)
137. Gauglitz, S., Höllerer, T., Turk, M.: Evaluation of interest point detectors and feature descriptors for visual tracking. Int. J. Comput. Vis. **94**(3), (2011)
138. Viola, Jones. Robust real time face detection. Int. J. Comput. Vis. **57**(2), (2004)
139. Thevenaz, P., Ruttimann, U.E., Unser, M.: A pyramid approach to subpixel registration based on intensity. IEEE Trans. Image Process. **7**(1), (1998)
140. Qi, T., Huhns, M.N.: Algorithms for subpixel registration. Comput. Vis. Graph. Image Process. **35** (1986)
141. Zhu, J., Yang, L.: Subpixel eye gaze tracking. Autom. Face Gesture Recogn. Conf. (2002)
142. Cheezum, M.K., Walker, W.F., Guilford, W.H.: Quantitative comparison of algorithms for tracking single fluorescent particles. Biophys. J. **81**(4), 2378–2388 (2001)
143. Guizar-Sicairos, M., Thurman, S.T., Fienup, J.R.: Efficient subpixel image registration algorithms. Opt. Lett. **33** (2), 156–158 (2008)
144. Hadjidemetriou, E., Grossberg, M.D., Nayar, S.K.: Multiresolution histograms and their use for texture classification. Int. Workshop Texture Anal. Synth. **26**(7), (2003)
145. Mikolajczyk, K., et al.: A comparison of affine region detectors. Conf. Comput. Vis. Pattern Recogn. (2006)
146. Canny, A.: Computational approach to edge detection. Trans. Pattern. Anal. Mach. Intell. **8**(6), (1986)
147. Gunn, S.R.: Edge detection error in the discrete Laplacian of Gaussian. International Conference on Image Processing, ICIP 98. Proceedings. vol 2, (1998)
148. Harris, C., Stephens, M.: A combined corner and edge detector. Proceedings of the 4th Alvey Vision Conference, (1988)
149. Shi, J., Tomasi, C.: Good features to track. Conf. Comput. Vis. Pattern Recogn. (1994)
150. Turk, M., Pentland, A.: Eigenfaces for recognition. J. Cogn. Neurosci. **3**(1), 1991 © MIT Media Lab, (1991)
151. Haja, A., Jahne, B., Abraham, S.: Localization accuracy of region detectors. IEEE CVPR (2008)
152. Bay, H., Ess, A., Tuytelaars, T., Van Gool, L.: Speeded-up robust features (SURF). Comput. Vis. Image Understand. **110**(3), 346–359 (2008)
153. Lowe, D.G.: SIFT distinctive image features from scale-invariant keypoints. Int. J. Comput. Vis. **60**(2), 91–110 (2004)
154. Kadir, T., Zisserman, A., Brady, M.: An affine invariant salient region detector. Eur. Conf. Comput. Vis. (2004)
155. Kadir, T., Brady, J.M.: Scale, saliency and image description. Int. J. Comput. Vis. **45**(2), 83–105 (2001)
156. Smith, S.M., Michael Brady, J.: SUSAN—a new approach to low level image processing. Technical report TR95SMS1c (patented), Crown Copyright (1995), Defence Research Agency, UK, (1995)
157. Smith, S.M., Michael Brady, J.: SUSAN—a new approach to low level image processing. Int. J. Comput. Vis. Arch. **23**(1), 45–78 (1997)
158. Baohua, Y., Cao, H., Chu, J.: Combining local binary pattern and local phase quantization for face recognition. Int. Symp. Biometr. Secur. Technol. (2012)
159. Ojansivu, V., Heikkil, J.: Blur insensitive texture classification using local phase quantization. Proc. Image Signal Process. (2008)
160. Chan, C.H., Tahir, M.A., Kittler, J., Pietikäinen, M.: Multiscale local phase quantization for robust component-based face recognition using kernel fusion of multiple descriptors. PAMI (2012)
161. Ojala, T., Pietikäinen, M., Harwood, D.: Performance evaluation of texture measures with classification based on kullback discrimination of distributions. Proc. Int. Conf. Pattern. Recogn. (1994)

162. Ojala, T., Pietikäinen, M., Hardwood, D.: A comparative study of texture measures with classification based on feature distributions. *Pattern Recogn.* **29** (1996)
163. Pietikäinen, M., Heikkilä, J.: Tutorial on image and video description with local binary pattern variants. *Conf. Comput. Vis. Pattern Recogn.* (2011)
164. Shu, L., Albert, C.S.: Chung. Texture classification by using advanced local binary patterns and spatial distribution of dominant patterns. *IEEE Int. Conf. Acoust. Speech Signal Process. ICASSP*, (2007)
165. Pietikäinen, M., Hadid, A., Zhao, G., Ahonen, T.: Computer Vision Using Binary Patterns. Computational Imaging and Vision Series, vol. 40. Springer, New York (2011)
166. Arandjelović, A., Zisserman, A.: Three things everyone should know to improve object retrieval. *Conf. Comput. Vis. Pattern Recogn.* (2011)
167. Guo Ying Z., Pietikäinen, M.: Dynamic texture recognition using local binary patterns with an application to facial expressions. *Pattern. Anal. Mach. Intell. IEEE Trans.* **29**(6), (2007)
168. Kellokumpu, V., Guo Ying Z., Pietikäinen, M.: Human activity recognition using a dynamic texture based method. *Br. Mach. Vis. Conf.* (2008)
169. Zabih, R., Woodfill, J.: Nonparametric local transforms for computing visual correspondence. *Eur. Conf. Comput. Vis.* (1994)
170. Lowe, D.G.: Object recognition from local scale-invariant features. *The Proceedings of the Seventh IEEE International Conference on Computer Vision*, (1999)
171. Abdel-Hakim, A.E., Farag, A.A.: CSIFT: a SIFT descriptor with color invariant characteristics. *Conf. Comput. Vis. Pattern Recogn.* (2006)
172. Vinukonda, P.: A study of the scale-invariant feature transform on a parallel pipeline. Thesis Project
173. Alcantarilla, P.F., Bergasa, L.M., Davison, A.: Gauge-SURF Descriptors. Elsevier, (2011)
174. Christopher, E.: Notes on the OpenSURF Library, University of Bristol Technical Paper, (2009)
175. Yan, K., Sukthankar, R.: PCA-SIFT: a more distinctive representation for local image descriptors. *Conf. Comput. Vis. Pattern Recogn.* (2004)
176. Gauglitz, S., Höllerer, T., Turka, M.: Evaluation of interest point detectors and feature descriptors for visual tracking. *Int. J. Comput. Vis.* **94** (2011)
177. Agrawal, M., Konolige, K., Blas, M.R.: CenSurE: center surround extrema for realtime feature detection and matching. *Eur. Conf. Comput. Vis.* (2008)
178. Viola, P., Jones, M.: Robust real-time object detection. *Int. J. Comput. Vis.* **57**(2), 137–154 (2002)
179. Grigorescu, S.E., Petkov, N., Kruizinga, P.: Comparison of texture features based on Gabor filters. *IEEE Trans. Image Process.* **11**(10), (2002)
180. Alcantarilla, P., Bergasa, L.M., Davison, A.: Gauge-SURF descriptors. *Image Vis. Comput.* **31**(1), 103–116 (2013). Elsevier via DOI 1302
181. Agrawal, M., Konolige, K., Blas, M.R.: CenSurE: center surround extrema for realtime feature detection and matching. *Eur. Conf. Comput. Vis.* (2008)
182. Morse, B.S.: Lecture 11: Differential Geometry. Brigham Young University, (1998/2000). <http://morse.cs.byu.edu/650/lectures/lect10/diffgeom.pdf>
183. Bosch, A., Zisserman, A., Munoz, X.: Representing shape with a spatial pyramid kernel. *CIVR '07 Proceedings of the 6th ACM International Conference on Image and Video Retrieval*
184. Rubner, Y., Tomasi, C., Guibas, L.J.: The earth mover's distance as a metric for image retrieval. *Int. J. Comput. Vis.* **40**(2), 99–121 (2000)
185. Oliva, A., Torralba, A.: Modeling the shape of the scene: a holistic representation of the spatial envelope. *Int. J. Comput. Vis.* (2001)
186. Matas, J., Chum, O., Urba, M., Pajdla, T.: Robust wide baseline stereo from maximally stable extremal regions. *Proc. Br. Mach. Vis. Conf.* (2002)
187. Scovanner, P., Ali, S., Shah, M.: A 3-dimensional SIFT descriptor and its application to action recognition. *ACM Proceedings of the 15th International Conference on Multimedia*, pp. 357–360, (2007)
188. Klaser, A., Marszalek, M., Schmid, C.: A spatio-temporal descriptor based on 3D-gradients. *Br. Mach. Vis. Conf.* (2008)
189. Laptev, I.: On space-time interest points. *Int. J. Comput. Vis.* **64** (2005)
190. Oreifej, O., Liu, Z.: HON4D: histogram of oriented 4D normals for activity recognition from depth sequences. *Conf. Comput. Vis. Pattern Recogn.* (2013)
191. Ke, Y., et al.: Efficient visual event detection using volumetric features. *Int. Conf. Comput. Vis.* (2005)
192. Zhang, L., da Fonseca, M.J., Ferreira, A.: Survey on 3D shape descriptors. União Europeia—Fundos Estruturais Governo da República Portuguesa Referência: POSC/EIA/59938/2004
193. Tangelder, J.W.H., Veltkamp, R.C.: A Survey of Content-Based 3D Shape Retrieval Methods. Springer, New York (2007)

194. Heikkila, M., Pietikäinen, M., Schmid, C.: Description of interest regions with center-symmetric local binary patterns. *Comput. Vis. Graph. Image Process. Lect. Notes Comput. Sci.* **4338**, 58–69 (2006)
195. Schmidt, A., Kraft, M., Fularz, M., Domagala, Z.: The comparison of point feature detectors and descriptors in the context of robot navigation. Workshop on Perception for Mobile Robots Autonomy, (2012)
196. Jun, B., Kim, D.: Robust face detection using local gradient patterns and evidence accumulation. *Pattern Recogn.* **45**(9), 3304–3316 (2012)
197. Froba, B., Ernst, A.: Face detection with the modified census transform. *Int. Conf. Autom. Face Gesture Recogn.* (2004)
198. Freeman, H. On the encoding of arbitrary geometric configurations. *IRE Trans. Electron. Comput.* (1961)
199. Salem, A.B.M., Sewisy, A.A., Elyan, U.A.: A vertex chain code approach for image recognition. *Int. J. Graph. Vis. Image Process. ICGST-GVIP*, (2005)
200. Kitchen, L., Rosenfeld, A.: Gray-level corner detection. *Pattern Recogn. Lett.* **1** (1992)
201. Koenderink, J., Richards, W.: Two-dimensional curvature operators. *J. Opt. Soc. Am.* **5**(7), 1136–1141 (1988)
202. Bretzner, L., Lindeberg, T.: Feature tracking with automatic selection of spatial scales. *Comput. Vis. Image Understand.* **71**(3), 385–392 (1998)
203. Lindeberg, T.: Junction detection with automatic selection of detection scales and localization scales. *Proceedings of First International Conference on Image Processing*, (1994)
204. Lindeberg, T.: Feature detection with automatic scale selection. *Int. J. Comput. Vis.* **30**(2), 79–116 (1998)
205. Wang, H., Brady, M.: Real-time corner detection algorithm for motion estimation. *Image Vis. Comput.* **13**(9), 695–703 (1995)
206. Trajkovic, M., Hedley, M.: Fast corner detection. *Image Vis. Comput.* **16**(2), 75–87 (1998)
207. Tola, E., Lepetit, V., Fua, P.: DAISY: an efficient dense descriptor applied to wide baseline stereo. *PAMI* **32**(5), (2010)
208. Arbeiter, G., et al.: Evaluation of 3D feature descriptors for classification of surface geometries in point clouds. *Int. Conf. Intell. Robots Syst.* (2012) IEEE/RSJ
209. Rupell, A., Weisshardt, F., Verl, A.: A rotation invariant feature descriptor O-DAISY and its FPGA implementation. *IROS* (2011)
210. Ambai, M., Yoshida, Y.: CARD: compact and real-time descriptors. *Int. Conf. Comput. Vis.* (2011)
211. Takacs, G., et al.: Unified real-time tracking and recognition with rotation-invariant fast features. *Conf. Comput. Vis. Pattern Recogn.* (2010)
212. Taylor, S., Rosten, E., Drummond, T.: Robust feature matching in 2.3 µs. *Conf. Comput. Vis. Pattern Recogn.* (2009)
213. Grauman, K., Darrell, T.: The pyramid Match Kernel: discriminative classification with sets of image features. *IEEE Int. Conf. Comput. Vis. Tenth* **2**, (2005)
214. Takacs, G., et al.: Unified real-time tracking and recognition with rotation-invariant fast features. *Conf. Comput. Vis. Pattern Recogn.* (2010)
215. Chandrasekhar, V., et al.: ChoG: compressed histogram of gradients, a low bitrate descriptor. *Conf. Comput. Vis. Pattern Recogn.* (2009)
216. Mainali, G.L., et al.: SIFER: scale-invariant feature detector with error resilience. *Int. J. Comput. Vis.* (2013)
217. Fowers, S.G., Lee, D.J., Ventura, D., Wilde, D.K.: A novel, efficient, tree-based descriptor and matching algorithm (BASIS). *Conf. Comput. Vis. Pattern Recogn.* (2012)
218. Fowers, S.G., Lee, D.J., Ventura, D.A., Archibald, J. K.: Nature inspired BASIS feature descriptor and its hardware implementation. *IEEE Trans. Circ. Syst. Video Technol.* (2012)
219. Bracewell, R.: *The Fourier Transform & Its Applications*, 3 ed., McGraw-Hill Science/Engineering/Math, (1999)
220. Duda, R.O., Hart, P.E.: Use of the Hough transformation to detect lines and curves in pictures. *Commun. ACM.* (1972)
221. Ballard, D.H.: Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recogn.* **13**(2), (1981)
222. Illingsworth, J., Kittler, K.: A survey of the Hough transform. *Comput. Vis. Graph. Image Process.* (1988)
223. Slaton, G., MacGill, M.J.: *Introduction to Modern Information Retrieval*. McGraw-Hill, New York (1983)
224. Niebles, J.C., Wang, H., Fei-Fei, L.: Unsupervised learning of human action categories using spatial-temporal words. *Int. J. Comput. Vis.* (2008)
225. Bosch, A., Zisserman, A., Muñoz, X.: Scene classification via pLSA. *Eur. Conf. Comput. Vis.* (2006)
226. Csurka, G., Bray, C., Dance, C., Fan, L.: Visual categorization with bags of key-points. *SLCV workshop, Eur. Conf. Comput. Vis.* (2004)
227. Dean, T., Washington, R., Corrado, G.: Sparse spatiotemporal coding for activity recognition. *Brown Univ. Tech. Rep.* (2010)
228. Le, Q.V., Zou, W.Y., Yeung, S.Y., Ng, A.Y.: Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis. *Conf. Comput. Vis. Pattern Recogn.* (2011)

229. Olshausen, B., Field, D.: Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature* **381**, 607–609 (1996)
230. Belongie, S., Malik, J., Puzicha, J.: Matching with shape context. CBAIVL '00 Proceedings of the IEEE Workshop on Content-based Access of Image and Video Libraries
231. Belongie, S., Malik, J., Puzicha, J.: Shape context: a new descriptor for shape matching and object recognition. *Conf. Neural Inform. Process. Syst.* (2000)
232. Belongie, S., Malik, J., Puzicha, J.: Shape matching and object recognition using shape contexts. *PAMI* **24**(4), (2002)
233. Belongie, S., Malik, J., Puzich, J.: Matching shapes with shape context. CBAIVL '00 Proceedings of the IEEE Workshop on Content-based Access of Image and Video Libraries
234. Liefeng, B., Ren, X., Fox, D.: Unsupervised feature learning for RGB-D based object recognition. *ISER*, vol 88 of Springer Tracts in Advanced Robotics. Springer, pp. 387–402, (2012)
235. Loy, G., Zelinsky, A.: A fast radial symmetry transform for detecting points of interest. *Eur. Conf. Comput. Vis.* (2002)
236. Wolf, L., Hassner, T., Taigman, Y.: Descriptor based methods in the wild. *Eur. Conf. Comput. Vis.* (2008)
237. Kurz, D., Ben Himane, S.: Inertial sensor-aligned visual feature descriptors. *Conf. Comput. Vis. Pattern Recogn.* (2011)
238. Kingsbury, N.: Rotation-invariant local feature matching with complex wavelets. *Proc. Eur. Conf. Signal Process. (EUSIPCO)*, (2006)
239. Dinggang, S., Ip, H.H.S.: Discriminative wavelet shape descriptors for recognition of 2-D patterns. *Pattern Recogn.* **32**(2), 151–165 (1999)
240. Edelman, S., Intrator, N., Poggio, T.: Complex cells and object recognition. *Conf. Neural Inform. Process. Syst.* (1997)
241. Hunt, R.W.G., Pointer, M.R.: *Measuring Colour*. Wiley, Hoboken, NJ (2011)
242. Hunt, R.W.G.: *The reproduction of color*, 6 ed., Wiley, (2004)
243. Berns, R.S.: *Billmeyer and Saltzman's Principles of Color Technology*. Wiley, Hoboken, NJ (2000)
244. Morovic, J.: *Color Gamut Mapping*. Wiley, Hoboken, NJ (2008)
245. Fairchild, M.: *Color appearance models*. 1st ed., Addison Wesley Longman, (1998)
246. Ito, M., Tsubai, M., Nomura, A.: Morphological operations by locally variable structuring elements and their applications to region extraction in ultrasound images. *Syst. Comput. Jpn.* **34**(3), 33–43 (2003)
247. Tsubai, M., Ito, M.: Control of variable structure elements in adaptive mathematical morphology for boundary enhancement of ultrasound images. *Electron. Commun. Jpn. Part 3 Fund. Electron. Sci.* **87**(11), 20–33
248. Mazille, J.E.: Mathematical morphology and convolutions. *J. Microsc.* **156**, 257 (1989)
249. Achanta, R., et al.: SLIC superpixels compared to state-of-the-art superpixel methods. *PAMI* **34**(11), (2012)
250. Achanta, R., et al.: SLIC superpixels. EPFL technical report no. 149300, (2010)
251. Felzenszwalb, P., Huttenlocher, D.: Efficient graph-based image segmentation. *Int. J. Comput. Vis.* (2004)
252. Levinstein, A., et al.: Turbopixels: fast superpixels using geometric flows. *PAMI* (2009)
253. Lucchi, A., et al.: A fully automated approach to segmentation of irregularly shaped cellular structures in EM images. *MICCAI* (2010)
254. Shi, J., Malik, J.: Normalized cuts and image segmentation. *PAMI* (2000)
255. Vedaldi, A., Soatto, S.: Quick shift and kernel methods for mode seeking. *Eur. Conf. Comput. Vis.* (2008)
256. Felzenszwalb, P.F., Huttenlocher, D.P.: Efficient graph-based image segmentation. *Int. J. Comput. Vis.* **59**(2), 167–181 (2004)
257. Felzenszwalb, P., Huttenlocher, D.: Efficient graph-based image segmentation. *Int. J. Comput. Vis.* **59** (2004)
258. Comaniciu, D., Meer, P.: Mean shift: a robust approach toward feature space analysis. *PAMI* **24**(5), (2002)
259. Vedaldi, A., Soatto, S.: Quick shift and kernel methods for mode seeking. *Eur. Conf. Comput. Vis.* (2008)
260. Vincent, L., Soille, P.: Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *PAMI* **13**(6), (1991)
261. Levinstein, A., et al.: Turbopixels: fast superpixels using geometric flows. *PAMI* **31**(12), (2009)
262. Scharstein, D., Pal, C.: Learning conditional random fields for stereo. *Conf. Comput. Vis. Pattern Recogn.* (2007)
263. Hirschmüller, H., Scharstein, D.: Evaluation of cost functions for stereo matching. *Conf. Comput. Vis. Pattern Recogn.* (2007)
264. Goodman, J.W.: *Introduction to Fourier optics*. McGraw-Hill, New York (1968)
265. Gaskill, J.D.: *Linear Systems, Fourier Transforms, Optics*. Wiley, Hoboken, NJ (1978)
266. Thibos, L., Applegate, R.A., Schweigerling, J.T., Webb, R.: Standards for reporting the optical aberrations of eyes. In: Lakshminarayanan, V. (ed.) *OSA Trends in Optics and Photonics, Vision Science and its Applications*. Optical Society of America, Washington, DC (2000)
267. Hwang, S.-K., Kim, W.-Y.: A novel approach to the fast computation of Zernike moments. *Pattern Recogn.* **39** (2006)

268. Khotanzad, A., Hong, Y.H.: Invariant image recognition by Zernike moments. *PAMI* **12** (1990)
269. Chao Kan, M., Srinath, D.: Invariant character recognition with Zernike and orthogonal Fourier-Mellin moments. *Pattern Recogn.* **35**, (2002)
270. Hyung, S.K., Lee, H.-K.: Invariant image watermark using Zernike moments. *IEEE Trans. Circ. Syst. Video Technol.* **13**(8), (2003)
271. Papakostas, G.A., Karras, D.A., Mertzios, B.G.: Image coding using a wavelet based Zernike moments compression technique. In: Proceeding of: Digital Signal Processing, vol 2, DSP, (2002)
272. Mukundan, R., Ramakrishnan, K.R.: Fast computation of Legendre and Zernike moments. *28*(9), 1433–1442, (1995)
273. Yongqing, X., Pawlak, M., Liao, S.: Image reconstruction with polar Zernike moments. ICAPR'05 Proceedings of the Third International Conference on Pattern Recognition and Image Analysis—Volume Part II (2005)
274. Singh, C., Upneja, R.: Fast and accurate method for high order Zernike moments computation. *Appl. Math. Comput.* **218**(15), 7759–7773 (2012)
275. Pratt, W., Chen, W.-H., Welch, L.: Slant transform image coding. *IEEE Trans. Commun.* **22**(8), (1974)
276. Enomoto, H., Shibata, K.: Orthogonal transform coding system for television signals. *IEEE Trans. Electromagn. Compatibil.* **13**(3), (1974)
277. Dutra da Silva, R., Robson, W., Pedrini Schwartz, H.: Image segmentation based on wavelet feature descriptor and dimensionality reduction applied to remote sensing. *Chilean J. Stat.* **2** (2011)
278. Arun, N., Kumar, M., Sathidevi, P.S.: Wavelet SIFT feature descriptors for robust face recognition. Springer Adv. Intell. Syst. Comput. **177** (2013)
279. Dinggang, S., Ip, H.H.S.: Discriminative wavelet shape descriptors for recognition of 2-D patterns. *Pattern Recogn.* **32** (1999)
280. Kingsbury, N.: Rotation-invariant local feature matching with complex wavelets. Proc. Eur. Conf. Signal Process. EUSIPCO (2006)
281. Wolfram Research Mathematica Wavelet Analysis Libraries
282. Strang, G.: “Wavelets.” *Am. Sci.* **82**(3), (1994)
283. Mallat, S.: A Wavelet Tour of Signal Processing: The Sparse Way, 3rd ed., Elsevier, (2008)
284. Percival, D.B., Walden, A.T.: Wavelet Methods for Time Series Analysis. Cambridge University Press, Cambridge (2006)
285. Gabor, D.: Theory of communication. *J. IEE.* **93** (1946)
286. Minor, L.G., Sklansky, J.: Detection and segmentation of blobs in infrared images. *IEEE Trans. Syst. Man Cybernetics.* **11**(3), (1981)
287. van Ginkel, M., Luengo Hendriks, C.K., van Vliet, L. J.: A short introduction to the Radon and Hough transforms and how they relate to each other. Number QI-2004-01 in the Quantitative Imageing Group Technical Report Series (2004)
288. Toft, P.A.: Using the generalized Radon transform for detection of curves in noisy images. 1996 I.E. International Conference on Acoustics, Speech, and Signal Processing, ICASSP-96. Conference Proceedings, vol 4, (1996)
289. Radon, J.: Über die Bestimmung von Funktionen durch ihre Integralwerte längs gewisser Mannigfaltigkeiten. *Berichte Sächsische Akademie der Wissenschaften, Leipzig, Mathematisch-Physikalische Klasse* **69** (1917)
290. Fung, J., Mann, S., Aimone, C.: OpenVIDIA: parallel GPU computer vision. Proc. ACM Multimed. (2005)
291. Bazin, M.J., Benoit, J.W.: Off-line global approach to pattern recognition for bubble chamber pictures. *Trans. Nuclear Sci.* **12** (1965)
292. Deans, S.R.: Hough transform from the Radon transform. *Trans. Pattern. Anal. Mach. Intell.* **3**(2), 185–188 (1981)
293. Rosenfeld, A.: Digital Picture Processing by Computer. Academic Press, New York (1982)
294. Tomasi, C., Manduchi, R.: Bilateral filtering for gray and color images. ICCV '98 Proceedings of the Sixth International Conference on Computer Vision (1998)
295. See the documentation for the ImageJ, ImageJ2 or Fiji software package for complete references to each method, [global] Auto Threshold command and Auto Local Threshold command. <http://fiji.sc/ImageJ2>
296. Garg, R., Mittal, B., Garg, S.: Histogram equalization techniques for image enhancement. *Int. J. Electron. Commun. Technol.* **2** (2011)
297. Sung, A.P., Wang, C.: Spatial-temporal antialiasing. *Trans. Visual. Comput. Graph.* **8** (2002)
298. Mikolajczyk, K., Schmid, C.: Scale & affine invariant interest point detectors. *Int. J. Comput. Vis.* **60** (2004)
299. Ozysal, M., Calonder, M., Lepetit, V., Fua, P.: Fast keypoint recognition using random ferns. *PAMI* **32** (2010)
300. Schaffalitzky, F., Zisserman, A.: Automated scene matching in movies. CIVR 2004, In: Proceedings of the Challenge of Image and Video Retrieval, London, LNCS 2383
301. Tola, E., Lepetit, V., Fua, P.: A fast local descriptor for dense matching. *Conf. Comput. Vis. Pattern Recogn.* (2008)
302. Davis, L.S.: Computing the spatial structures of cellular texture. *Comput. Graph. Image Process.* **11**(2), (1979)

303. Pun, C.M., Lee, M.C.: Log-polar wavelet energy signatures for rotation and scale invariant texture classification. *Trans. Pattern. Anal. Mach. Intell.* **25**(5), (2003)
304. Spence, A., Robb, M., Timmins, M., Chantler, M.: Real-time per-pixel rendering of textiles for virtual textile catalogues. *Proc. INTEDEC*. (2003)
305. Lam, S.W.C., Ip, H.H.S.: Adaptive pyramid approach to texture segmentation. *Comput. Anal. Images Patterns Lect. Notes Comput. Sci.* **719**, 267–274 (1993)
306. Yinping J., Fayad, L., Laine, A.: Contrast enhancement by multi-scale adaptive histogram equalization. *Proc. SPIE* **4478** (2001)
307. Jianguo, Z., Tan, T.: Brief review of invariant texture analysis methods. *Pattern Recogn.* **35** (2002)
308. Tomita, F., Shirai, Y., Tsuji, S.: Description of textures by a structural analysis. *IEEE Trans. Pattern. Anal. Mach. Intell. Arch.* **4** (1982)
309. Tomita, F., Tsuji, S.: Computer Analysis of Visual Textures. Springer, New York (1990)
310. Burt, P.J., Adelson, E.H.: The Laplacian pyramid as a compact image code. *IEEE Trans. Commun.* (1983)
311. Otsu, N.: A threshold selection method from gray-level histograms. *IEEE Trans. Syst. Man Cybern.* **9**(1), 62–66 (1979)
312. Sezgin, M., Sankur, B.: Survey over image thresholding techniques and quantitative performance evaluation. *SPIE J. Electron. Imaging* (2004)
313. Haralick, R.M., Shapiro, L.G.: Image segmentation techniques. *Comput. Vis. Graph. Image Process.* **29**, 100–132 (1985)
314. Raja, Y., Gong, S.: Sparse multiscale local binary patterns. *Br. Mach. Vis. Conf.* (2006)
315. Fleuret, F.: Fast binary feature selection with conditional mutual information. *J. Mach. Learn. Res.* **5** (2004)
316. Szelinski, R.: Computer Vision, Algorithms and Applications. Springer, New York (2011)
317. Pratt, W.K.: Digital Image Processing: PIKS Scientific Inside. 4 ed., Wiley-Interscience, (2007)
318. Russ, J.C.: The Image Processing Handbook, 5 ed., CRC Press, (2006)
319. Klein, G., Murray, D.: Parallel tracking and mapping for small AR workspaces. IMAR. (2007)
320. Newcombe, R.A., et al.: KinectFusion: real-time dense surface mapping and tracking. *ISMAR '11 Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality* (2011)
321. Izadi, S., et al.: KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera. *ACM Symp. User Interf. Software Technol.* (2011)
322. Moravec, H.: Obstacle avoidance and navigation in the real world by a seeing robot rover. Tech Report CMU-RI-TR-3, Robotics Institute, Carnegie-Mellon University, (1980)
323. Mikolajczyk, K., Schmid, C.: Indexing based on scale invariant interest points. *Int. Conf. Comput. Vis.* (2001)
324. Turcot, P., Lowe, D.G.: Better matching with fewer features: the selection of useful features in large database recognition problems. *Int. Conf. Comput. Vis.* (2009)
325. Feichtinger, H.G., Strohmer, T.: Gabor Analysis and Algorithms, 1997 ed., Birkhäuser, (1997)
326. Ricker, N.: Wavelet contraction, wavelet expansion, and the control of seismic resolution. *Geophysics* **18**, 769–792 (1953)
327. Goshtasby, A.: Description and discrimination of planar shapes using shape matrices. *PAMI* **7**(6), (1985)
328. Vapnik, V.N., Levin, E., LeCun, Y.: Measuring the dimension of a learning machine. *Neural Comput.* **6**(5), 851–876 (1994)
329. Cowan, J. D., Tesauro, G., Alspector, J.: Learning curves: asymptotic values and rate of convergence. *Adv. Neural Inform. Process.* **6** (1994)
330. Vapnik, V.N.: The Nature of Statistical Learning Theory. Springer, New York (1995)
331. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition: intelligent signal processing. *Proc. IEEE* **86**(11), 2278–2324 (1998)
332. Krizhevsky, A., Sutskever, I., Hinton, E.: ImageNet classification with deep convolutional neural networks. *Conf. Neural Inform. Process. Syst.* (2012)
333. Boser, B.E., Guyon, I.M., Vapnik, V.N.: A training algorithm for optimal margin classifiers. *COLT '92 Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, (1992)
334. Cortes, C., Vapnik, V.N.: Support-vector networks. *Mach. Learn.* **20** (1995)
335. Burges, C.J.C.: A tutorial on support vector machines for pattern recognition. *Kluwer Data Mining Discov.* **2** (1998)
336. Weinzaepfel, P., Revaud, J., Harchaoui, Z., Schmid, C.: DeepFlow: large displacement optical flow with deep matching. *Int. Conf. Comput. Vis.* (2013)
337. Keysers, T.C., Gollan, D., Ney, H.: Deformation models for image recognition. *Trans. PAMI* **20** (2007)
338. Kim, J., Liu, C., Sha, F., Grauman, K.: Deformable spatial pyramid matching for fast dense correspondences. *Conf. Comput. Vis. Pattern Recogn.* (2013)
339. Boureau, Y.-L., Ponce, J., LeCun, Y.: A theoretical analysis of feature pooling in visual recognition. *IML, 27th International Conference on Machine Learning*, Haifa, Israel, (2010)

340. Schmid, C., Mohr, R.: Object recognition using local characterization and semi-local constraints. *PAMI* **19**(3), (1997)
341. Ferrari, V., Tuytelaars, T., Gool, L.V.: Simultaneous object recognition and segmentation from single or multiple model views. *Int. J. Comput. Vis.* **67** (2005)
342. Schaffalitzky, F., Zisserman, A.: Automated scene matching in movies. *CIVR*. (2002)
343. Estivill-Castro, V.: Why so many clustering algorithms—a position paper. *ACM SIGKDD Explor. Newslett.* **4**(1), (2002)
344. Kriegel, H.-P., Kröger, P., Sander, J., Zimek, A.: Density-based clustering. *Wiley Interdisciplinary Rev. Data Mining Knowl. Discov.* **1**(3), 231–240 (2011)
345. Hartigan, J.A.: *Clustering Algorithms*. Wiley, Hoboken, NJ (1975)
346. Hartigan, J.A., Wong, M.A.: Algorithm AS 136: A K-means clustering algorithm. *J. Roy. Stat. Soc. Ser. B* **28**(1), (1979)
347. Hastie, T., Tibshirani, R., Friedman, J.: *Hierarchical Clustering: The Elements of Statistical Learning*, 2nd edn. Springer, New York (2009)
348. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm. *J. Roy. Stat. Soc. Ser. B* **39**(1), 1–38 (1977)
349. Pearson, K.: On lines and planes of closest fit to systems of points in space. *Phil. Mag.* (1901)
350. Hotelling, H.: Relations between two sets of variates. *Biometrika* **28**(3–4), 321–377 (1936)
351. Cortes, C., Vapnik, V.N.: Support-vector networks. *Mach. Learn.* **20**(3), 273–297 (1995)
352. Haykin, S.: *Neural Networks: A Comprehensive Foundation*, 2nd edn. Prentice-Hall, Englewood Cliffs, NJ (1999)
353. Vapnik, V.: *Statistical Learning Theory*. Wiley, Hoboken, NJ (1998)
354. Hofmann, T., Scholkopf, B., Smola, A.J.: Kernel methods in machine learning. *Ann. Stat.* **36**(3), 1031 (2008)
355. Raguram, R., Frahm, J.-M., Pollefeys, M.: A comparative analysis of RANSAC techniques leading to adaptive real-time random sample consensus. *Eur. Conf. Comput. Vis.* (2008)
356. Weinberger, K.Q., Blitzer, J., Saul, L.K.: Distance metric learning for large margin nearest neighbor classification. *Conf. Neural Inform. Process. Syst.* (2004)
357. Schmid, C., Mohr, R.: Local gray value invariants for image retrieval. *PAMI* **19**(5), (1997)
358. Dork, G., Schmid, C.: Object class recognition using discriminative local features. Technical Report RR-5497, INRIA—Rhône-Alpes (2005)
359. Schlkopf, B., Smola, A.J.: *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA (2001)
360. Ferrari, V., Tuytelaars, T., Gool, L.V.: Simultaneous object recognition and segmentation from single or multiple model views. *Int. J. Comput. Vis.* **67**(2), (2006)
361. Cinbis, R.G., Verbeek, J., Schmid, C.: Segmentation driven object detection with fisher vectors. *Int. Conf. Comput. Vis.* (2013)
362. Fischler, M., Bolles, R.: Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM* **24**(6), (1981)
363. Freund, Y., Schapire, R.E.: A short introduction to boosting. *Jpn. Soc. Artif. Intell.* **14**(5), (1999)
364. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.* **55**(1), 119–139 (1997)
365. Heckerman, D.: A tutorial on learning with Bayesian networks. *Microsoft Res. Tech. Rep.* (1996)
366. Amit, Y., Geman, D.: Shape quantization and recognition with randomized trees. *Neural Comput.* **9**(7), (1997)
367. Rabiner, L.R., Juang, B.H.: An introduction to hidden Markov models. *IEEE Acoust. Speech Signal Process. Mag.* (1986)
368. Krogh, A., Larsson, B., von Heijne, G., Sonnhammer, E.L.: Predicting transmembrane protein topology with a hidden Markov model: application to complete genomes. *J. Mol. Biol.* (2001)
369. Nister, D., Stewenius, H.: Scalable recognition with a vocabulary tree. *Conf. Comput. Vis. Pattern Recogn.* (2006)
370. Freeman, W.T., Adelson, E.H.: The design and use of steerable filters. *PAMI* **13**(9), (1991)
371. Leung, T., Malik, J.: Representing and recognizing the visual appearance of materials using three-dimensional textons. *Int. J. Comput. Vis.* **43**(1) (2001)
372. Schmid, C.: Constructing models for content-based image retrieval. *Conf. Comput. Vis. Pattern Recogn.* (2001)
373. Alahi, A., Vandergheynst, P., Bierlaire, M., Kunt, M.: Cascade of descriptors to detect and track objects across any network of cameras. *Comput. Vis. Image Understand.* **114**(6), 624–640 (2010)
374. Simard, P., Bottou, L., Haffner, P., LeCun, Y.: Boxlets: a fast convolution algorithm for signal processing and neural networks. *Conf. Neural Inform. Process. Syst.* (1999)
375. Vedaldi, A., Zisserman, A.: Efficient additive kernels via explicit feature maps. *PAMI* **34**(3), (2012)
376. Brox, T., Malik, J.: Large displacement optical flow: descriptor matching in variational motion estimation. *PAMI* **33**(3), (2010)

377. Martin, E., Kriegel, H.-P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Second International Conference on Knowledge Discovery and Data Mining, pp. 226–231, (1996)
378. Mihai, A., Breunig, M.M., Kriegel, H.-P., Sander, J.: OPTICS: ordering points to identify the clustering structure. SIGMOD '99 Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data
379. Muja, M., Rusu, R.B., Bradski, G., Lowe, D.G.: REIN—a fast, robust, scalable recognition infrastructure. Int. Conf. Robot. Autom. (2011)
380. Rusu, R.B., Bradski, G., Thibaux, R., Hsu, J.: Fast 3D recognition and pose using the viewpoint feature histogram. Intell. Robots Syst. (2010)
381. Alvaro, C., Martinez, M., Siddhartha S.: Srinivasa. MOPED: a scalable and low latency object recognition and pose estimation system. Int. Conf. Robot. Autom. (2010)
382. Jacob, M., Unser, M.: Design of steerable filters for feature detection using canny-like criteria. PAMI **26**(8), (2004)
383. Moré, J.J.: The Levenberg-Marquardt algorithm implementation and theory. Numer. Anal. Lect. Notes Math. **630**, 105–116 (1978)
384. Lecun, Y.: Learning invariant feature hierarchies. Eur. Conf. Comput. Vis. (2012)
385. Ranzato, M.A., Huang, F.-J., Boreau, Y.-L., Cun, Y.L.: Unsupervised learning of invariant feature hierarchies with applications to object recognition. Conf. Comput. Vis. Pattern Recogn. (2007)
386. Boureau, Y.-L., Ponce, J., LeCun, Y.: A theoretical analysis of feature pooling in vision algorithms. Int. Conf. Mach. Learn. (2010)
387. Kingma, D., LeCun, Y.: Regularized estimation of image statistics by score matching. Conf. Neural Inform. Process. Syst. (2010)
388. Lossen, O., Macaire, L., Yang, Y.: Comparison of color demosaicing methods. Adv. Imaging Electron Phys. **162**, 173–265 (2010)
389. Xin, L., Gunturk, B., Zhang, L.: Image demosaicing: a systematic survey. Proceedings of SPIE 6822, Visual Communications and Image Processing, 68221J (2008)
390. Tanbakuchi, A.A., et al.: Adaptive pixel defect correction. Proceedings of SPIE 5017, Sensors and Camera Systems for Scientific, Industrial, and Digital Photography Applications IV, (2003)
391. Ibenthal, A.: Image sensor noise estimation and reduction. ITG Fachausschuss 3.2 Digitale Bildcodierung (2007)
392. An Objective Look at FSI and BSI, Aptina White Paper
393. Cossairt, O., Miau, D., Nayar, S.K.: Gigapixel computational imaging. IEEE Int. Conf. Comput. Photogr. (2011)
394. Eastman Kodak Company, E-58 technical data/color negative film. Kodak 160NC Technical Data Manual, (2000)
395. Kuthirummal, S., Nayar, S.K.: Multiview radial catadioptric imaging for scene capture. ACM Trans. Graph. (also Proc. of ACM SIGGRAPH), (2006)
396. Zhou, C., Nayar, S.K.: Computational cameras: convergence of optics and processing. IEEE Trans. Image Process. **20**(12), (2011)
397. Krishnan, G., Nayar, S.K.: Towards a true spherical camera. Proceedings of SPIE 7240, Human Vision and Electronic Imaging XIV, 724002 (2009)
398. Reinhard, H., Debevec, P., Ward, M., Kaufmann, M.: High Dynamic range imaging, 2nd edition acquisition, display, and image-based lighting. 2 ed., Morgan Kaufmann, (2010)
399. Gallo, O., et al.: Artifact-free high dynamic range imaging. IEEE Int. Conf. Comput. Photogr. (2009)
400. Grossberg, M.D., Nayar, S.K.: High dynamic range from multiple images: which exposures to combine? Int. Conf. Comput. Vis. (2003)
401. Nayar, S.K., Krishnan, G., Grossberg, M.D., Raskar, R.: Fast separation of direct and global components of a scene using high frequency illumination. Proc. SIGGRAPH (2006)
402. Wilson, T., Juskaiteis, R., Neil, M., Kozubek, M.: Confocal microscopy by aperture correlation. Opt. Lett. **21**(23), 1879–1881 (1996)
403. Corle, T.R., Kino, G.S.: Confocal Scanning Optical Microscopy and Related Imaging Systems. Academic Press, New York (1996)
404. Fitch, J.P.: Synthetic Aperture Radar. Springer, New York (1988)
405. Ng, R., et al.: Light field photography with a hand-held plenoptic camera. Stanford Tech Report CTSR 2005-02
406. Ragan-Kelley, J., et al.: Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans. Graph. **31**(4), (2012)
407. Levoy, M.: Experimental platforms for computational photography. Comput. Graph. Appl. **30** (2010)
408. Adams, A., et al.: The Frankencamera: an experimental platform for computational photography. Proc. SIGGRAPH. (2010)
409. Salsman, K.: 3D vision for computer based applications. Technical Report, Aptina, Inc., (2010).
410. Cossairt, O., Nayar, S.: Spectral focal sweep: extended depth of field from chromatic aberrations. IEEE Int. Conf. Comput. Photogr. (2010). (see also US Patent EP2664153A1)

411. Fife, K., El Gamal, A., Philip Wong, H.-S.: A 3D multi-aperture image sensor architecture. Proc. IEEE Custom Integr. Circ. Conf. 281–284, (2006)
412. Wang, A., Gill, P., Molnar, A.: Light field image sensors based on the Talbot effect. Appl. Optics **48**(31), 5897–5905 (2009)
413. Shankar, M., et al.: Thin infrared imaging systems through multichannel sampling. Appl. Optics **47**(10), B1–B10 (2008)
414. Flusser, B.Z.J.: Image registration methods: a survey. Image Vis. Comput. **21**(11), 977–1000 (2003)
415. Hirschmüller, H.: Accurate and efficient stereo processing by semi-global matching and mutual information. Conf. Comput. Vis. Pattern Recogn. (2005)
416. Tuytelaars, T., Van Gool, L.: Wide baseline stereo matching based on local, affinely invariant regions. Br. Mach. Vis. Conf. (2000)
417. Faugeras, O.: Three Dimensional Computer Vision. MIT Press, Cambridge, MA (1993)
418. Maybank, S.J., Faugeras O.D.: A theory of self-calibration of a moving camera. Int. J. Comput. Vis. **8**(2), (1992)
419. Hartley, R., Zisserman, A.: Multiple View Geometry in Computer Vision. Cambridge University Press, Cambridge (2004)
420. Luong, Q.-T., Faugeras, O.D.: The fundamental matrix: theory, algorithms, and stability analysis. Int. J. Comput. Vis. **17** (1995)
421. Hartley, R.I.: Theory and practice of projective rectification. Int. J. Comput. Vis. **35** (1999)
422. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. Int. J. Comput. Vis. **47** (2002)
423. Lazaros, N., Sirakoulis, G.C., Gasteratos, A.: Review of stereo vision algorithms: from software to hardware. Int. J. Optomechatron. **2**(4), 435–462 (2008)
424. Clark, D.E., Ivezkovic, S.: The Cramer-Rao lower bound for 3-D state estimation from rectified stereo cameras. IEEE Fusion (2010)
425. Nayar, S.K., Gupta, M.: Diffuse structured light. Int. Conf. Comput. Photogr. (2012)
426. Cattermole, F.: Principles of Pulse Code Modulation, 1st ed., American Elsevier Pub. Co., (1969)
427. Pagès, J., Salvi, J.: Coded light projection techniques for 3D reconstruction. J3eA, Journal sur l'enseignement des sciences et technologies de l'information et des systèmes **4**(1), (2005) (Hors-Série 3)
428. Gu, J., et al.: Compressive structured light for recovering inhomogeneous participating media. Eur. Conf. Comput. Vis. (2008)
429. Nayar, S.K.: Computational cameras: approaches, benefits and limits. Technical Report, Computer Science Department, Columbia University, (2011)
430. Lehmann, M., et al.: CCD/CMOS lock-in pixel for range imaging: challenges, limitations and state-of-the-art. CSEM, Swiss Center for Electronics and Microtechnology, (2004)
431. Andersen, J.F., Busck, J., Heiselberg, H.: Submillimeter 3-D laser radar for space shuttle tile inspection. Danisch Defense Research Establishment, Copenhagen, Denmark, (2013)
432. Grzegorzek, M., Theobalt, C., Koch, R., Kolb, A. (eds.): Time-of-Flight and Depth Imaging. Sensors, Algorithms, and Applications Lecture Notes in Computer Science, Springer (2013)
433. Levoy, M., Hanrahan, P.: Light field rendering. SIGGRAPH '96 Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (1996)
434. Curless, B., Levoy, M.: A volumetric method for building complex models from range images. SIGGRAPH '96 Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (1996)
435. Drebin, R.A.: Loren Carpenter, and Pat Hanrahan, volume rendering. SIGGRAPH (1988)
436. Levoy, M.: Display of surfaces from volume data. CG&A (1988)
437. Levoy, M.: Volume rendering using the Fourier projection slice theorem. Technical report CSL-TR-92-521, Stanford University, (1992)
438. Klein, G., Murray, D.: Parallel tracking and mapping on a camera phone. ISMAR '09 Proceedings of the 2009 8th IEEE International Symposium on Mixed and Augmented Reality (2009)
439. Klein, G., Murray, D.: Parallel tracking and mapping for small AR workspaces. In: Proceedings of International Symposium on Mixed and Augmented Reality (ISMAR'07, Nara)
440. Lucas, B.D., Kanade, T.: An image registration technique with an application to stereo vision. Proceedings of Image Understanding Workshop, (1981)
441. Beauchemin, S., Barron, J.D.: The computation of optical flow. ACM Comput. Surv. **27**(3), (1995)
442. Barron, J., Fleet, D., Beauchemin, S.: Performance of optical flow techniques. Int. J. Comput. Vis. **12**(1), 43–77 (1994)
443. Baker, S., et al.: A database and evaluation methodology for optical flow. Int. J. Comput. Vis. **92**(1), 1–31 (2009)
444. Quénot, G.M., Pakleza, J., Kowalewski, T.A.: Particle image velocimetry with optical flow. In: Experiments in Fluids, vol 25(3), pp. 177–189, (1998)

445. Trulls, E., Sanfeliu, A., Moreno-Noguer, F.: Spatiotemporal descriptor for wide-baseline stereo reconstruction of non-rigid and ambiguous scenes. *Eur. Conf. Comput. Vis.* (2012)
446. Steinman, S.B., Steinman, B.A., Garzia, R.P.: *Foundations of Binocular Vision: A Clinical Perspective*. McGraw-Hill, New York (2000)
447. Roy, S., Meunier, J., Cox, I.J.: Cylindrical rectification to minimize epipolar distortion. *Conf. Comput. Vis. Pattern Recogn.* (1997)
448. Oram, D.: Rectification for any epipolar geometry. *Br. Mach. Vis. Conf.* (2001)
449. Takita, K., et al.: High-accuracy subpixel image registration based on phase-only correlation. Institute of Electronics, Information and Communication Engineers(IEICE), (2003)
450. Huhns, T.: Algorithms for subpixel registration. *CGIP Comput. Graph. Image Process.* (1986)
451. Foroosh (Shekarforoush).: Hassan, Josiane B. Zerubia, and Marc Berthod. Extension of phase correlation to subpixel registration. *IEEE Trans. Image Process.* (2002)
452. Zitnick, L., Kanade, T.: A cooperative algorithm for stereo matching and occlusion detection. Carnegie Mellon University, Technical report CMU-RI-TR-99-35
453. Jian, S., Li, Y., Kang, S.B., Shum, H.-Y.: Symmetric stereo matching for occlusion handling. *CVPR '05 Proceedings of the 2005 I.E. Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) 2*
454. Kang, S.B., Szeliski, R., Chai, J.: Handling occlusions in dense multi-view stereo. *Conf. Comput. Vis. Pattern Recogn.* (2001)
455. Curless, B., Levoy, M.: A volumetric method for building complex models from range images. *SIGGRAPH Proc.* (1996)
456. Izadi, S., et al.: KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera. *UIST '11 Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, (2011)
457. Newcombe, R.A. et al.: KinectFusion: real-time dense surface mapping and tracking. *ISMAR '11 Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality*
458. Durrant-Whyte, H., Bailey, T.: Simultaneous localisation and mapping (SLAM): part I the essential algorithms. *IEEE Robotics Autom. Mag.* (2006)
459. Bailey, T., Durrant-Whyte, H.: Simultaneous localisation and mapping (SLAM): part II state of the art. *IEEE Robotics Autom. Mag.* (2006)
460. Seitz, S., et al.: A comparison and evaluation of multi-view stereo reconstruction algorithms. *CVPR 1*, 519–526 (2006)
461. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. J. Comput. Vis.* **47** (2002)
462. Baker, S., Matthews, I.: Lucas-Kanade 20 years on: a unifying framework. *Int. J. Comput. Vis.* **56** (2004)
463. Gallup, D., Pollefeys, M., Frahm, J.M.: 3D reconstruction using an n-layer heightmap. *Pattern Recogn. Lect. Notes Comput. Sci.* **6376** (2010)
464. Newcombe, R.A., Lovegrove, S.J., Davison, A.J.: DTAM: dense tracking and mapping in real-time. *Int Conf Comput Vis (ICCV) IEEE*, 2320–2327, (2011)
465. Hwangbo, M., Kim, J.-S., Kanade, T.: Inertial-aided KLT feature tracking for a moving camera. *Intell. Robots Syst. (IROS)—IEEE*. (2009)
466. Lovegrove, S.J., Davison, A.J.: Real-time spherical Mosaicing using whole image alignment. *Eur. Conf. Comput. Vis.* (2010)
467. Malis, E.: Improving vision-based control using efficient second-order minimization techniques. *Int. Conf. Robot Autom.* (2004)
468. Kaiming H, Sun, J., Tang, X.: Guided image filtering. *Eur. Conf. Comput. Vis.* (2010)
469. Rhemann, C., et al.: Fast cost-volume filtering for visual correspondence and beyond. *CVPR, IEEE*, 3017–3024, (2011)
470. Fattal, R.: Edge-avoiding wavelets and their applications. *SIGGRAPH* (2009)
471. Gastal, E.S.L., Oliveira, M.M.: Domain transform for edge-aware image and video processing. *ACM SIGGRAPH 2011 papers Article No. 69*
472. Wolberg, G.: *Digital Image Warping*. Wiley, Hoboken, NJ (1990)
473. Baxes, G.: *Digital Image Processing: Principles and Applications*. Wiley, Hoboken, NJ (1994)
474. Fergus, R., et al.: Removing camera shake from a single photograph. *ACM Trans. Graph.* **25**(3), (2006)
475. Rohr, K.: *Landmark-Based Image Analysis Using Geometric and Intensity Models*. Kluwer Academic Publishers, Dordrecht (2001)
476. Corbet, J., Rubini, A., Kroah-Hartman, G.: *Linux Device Drivers*, 3rd ed., O'Reilly Media, (2005)
477. Zinner, C., Kubinger, W., Isaacs, R.: PfeLib—a performance primitives library for embedded vision. *EURASIP*, (2007)
478. Houston, M.: OpenCL overview. *SIGGRAPH OpenCL BOF* (2011), also on KHRONOS website

479. Zinner, C., Kubinger, W.: ROS-DMA: a DMA double buffering method for embedded image processing with resource optimized slicing. IEEE RTAS 2006, Real-Time and Embedded Technology and Applications Symposium (2006)
480. Kreahling, W.C., et al.: Branch elimination by condition merging. Euro-Par 2003 Parallel Process. Lect. Notes Comput. Sci. **2790**, (2003)
481. Ullman, J.D., Aho, A.V.: Principles of Compiler Design. Addison-Wesley, (1977)
482. Ragan-Kelley, J., et al.: Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Trans. Graph. SIGGRAPH **31**(4), (2012)
483. Alcantarilla, P.F., Bartoli, A., Davison, A.J.: KAZE features. Eur. Conf. Comput. Vis. (2012)
484. Schneider, C.A., Rasband, W.S., Eliceiri, K.W.: NIH image to ImageJ: 25 years of image analysis. Nat. Meth. **9** (2012)
485. Muja, M.: Recognition pipeline and object detection scalability. Summer 2010 Internship Presentation, University of British Columbia
486. Viola, P.A., Jones, M.J.: Rapid object detection using a boosted cascade of simple features. Conf. Comput. Vis. Pattern Recogn. (2001)
487. Swain, M., Ballard, D.H.: Color indexing. Int. J. Comput. Vis. **7** (1991)
488. Zhang, Z.: A flexible new technique for camera calibration. EEE Trans. Pattern. Anal. Mach. Intell. **22**(11), 1330–1334 (2000)
489. Viola, P.A., Jones, M.J.: Robust real time object detection. Int. J. Comput. Vis. (2001)
490. Murase, H., Nayar, S.K.: Visual learning and recognition of 3-D objects from appearance. Int. J. Comput. Vis. **14** (1995)
491. Grosse, R., et al.: Ground-truth dataset and baseline evaluations for intrinsic image algorithms. Int. Conf. Comput. Vis. (2009)
492. Haltakov, V., Unger, C., Ilic, S.: Framework for generation of synthetic ground truth data for driver assistance applications. Pattern Recogn. Lect. Notes Comput. Sci. **8142** (2013)
493. Buades, A., Coll, B., Morel, J.-M.: A non-local algorithm for image denoising. Comput. Vis. Pattern Recogn. **2** (2005)
494. Agaian, S.S., Tourshan, K., Noonan, J.P.: Parametric Slant-Hadamard transforms. Proc. SPIE, (2003)
495. Sauvola, J., Pietaksinen, M.: Adaptive document image binarization. Pattern Recogn. **33**(2), (2000)
496. Yen, J.C., Chang, F.J., Chang, S.: A new criterion for automatic multilevel thresholding. Trans. Image Process. **4** (3), (1995)
497. Sezgin, M., Sankur, B.: Survey over image thresholding techniques and quantitative performance evaluation. Journal of Electronic Imaging **13**(1), 2004
498. Gaskill, J.D.: Linear Systems, Fourier Transforms, and Optics. Wiley, Hoboken, NJ (1978)
499. Shapiro, L.G., Stockman, G.C.: Computer Vision. Prentice-Hall, Upper Saddle River, NJ (2001)
500. Flusser, J., Suk, T., Zitova, B.: Moments and Moment Invariants in Pattern Recognition. Wiley, Hoboken, NJ (2009)
501. Mikolajcyk, K., Schmid, C.: An affine invariant interest point detector. Int. Conf. Comput. Vis. (2002)
502. Moravec, H.P.: Obstacle avoidance and navigation in the real world by a seeing robot rover. Tech. report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University & doctoral dissertation, Stanford University, (1980)
503. Sivic, J.: Efficient Visual search of videos cast as text retrieval. PAMI **31** (2009).
504. Tan, X., Triggs, B.: Enhanced local texture feature sets for face recognition under difficult lighting conditions. AMFG'07 Proceedings of the 3rd International Conference on Analysis and Modeling of Faces and Gestures (2010)
505. Scale-Space. Encyclopedia of Computer Science and Engineering. Wiley, Hoboken, NJ, (2008)
506. Lindeberg, T.: Scale-space theory: a basic tool for analysing structures at different scales. J. Appl. Stat **21**(2), 224–270 (1994)
507. Bengio, Y.: Learning Deep Architectures for AI, Foundations and Trends in Machine Learning. Now Publishers Inc USA, (2009)
508. Hinton, G.E., Osindero, S.: A fast learning algorithm for deep belief nets. Neural Comput. **18**(7), (2006)
509. Olson, E.: AprilTag: a robust and flexible visual fiducial system. Int. Conf. Robotics Autom. (2011)
510. Farabet, C., et al.: Hardware accelerated convolutional neural networks for synthetic vision systems. ISCAS IEEE 257–260, (2010)
511. Tuytelaars, T., Van Gool, L.: Matching widely separated views based on affine invariant regions. Int. J. Comput. Vis. **59** (2004)
512. Fischler, M.A., Elschlager, R.A.: The representation and matching of pictorial structures. IEE Trans. Comput. (1973)
513. Felzenszwalb, P.F., Girshick, R.B., McAllester, D., Ramanan, D.: Object detection with discriminatively trained part-based models. PAMI **32**(9), (2010)

514. Yi Y., Ramanan, D.: Articulated pose estimation with flexible mixtures-of-parts. *Conf. Comput. Vis. Pattern Recogn.* (2011)
515. Amit, Y., Trouve, A.: POP: patchwork of parts models for object recognition. *Int. J. Comput. Vis.* **75** (2007)
516. Lazebnik, S., Schmid, C., Ponce, J.: Beyond bags of features: spatial pyramid matching for recognizing natural scene categories. *Conf. Comput. Vis. Pattern Recogn.* (2006)
517. Grauman, K., Darrell, T.: The pyramid Match Kernel: discriminative classification with sets of image features. *Int. Conf. Comput. Vis.* (2005)
518. Michal, A., Elad, M., Bruckstein, A.: KSVD: an algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Trans. Signal Process.* **64** (2006)
519. Fei-Fei, L., Fergus, R., Torralba, A.: Recognizing and learning object categories. *Conf. Comput. Vis. Pattern Recogn.* (2007)
520. Johnson, A.: Spin-Images: A Representation for 3-D Surface Matching Ph.D. dissertation, technical report CMU-RI-TR-97-47, Robotics Institute, Carnegie Mellon University, (1997)
521. Zoltan-Csaba, M., Pangercic, D., Blodow, N., Beetz, M.: Combined 2D-3D categorization and classification for multimodal perception systems. *Int. J. Robotics Res. Arch.* **30**(11), (2011)
522. Kass, M., Witkin, A., Terzopoulos, D.: Snakes: active contour models. *Int. J. Comput. Vis.* (1988)
523. Tombari, F., Salti, S., Di Stefano, L.: A combined texture-shape descriptor for enhanced 3D feature matching. *Int. Conf. Image Process.* (2011)
524. Mikolajczyk, K., Schmid, C.: Indexing based on scale invariant interest points. *Int. Conf. Comput. Vis.* (2001)
525. Ragan-Kelley, J., et al.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *PLDI '13 Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (2013)
526. Kindratenko, V.V., et al.: GPU clusters for high-performance computing. In: *Proceedings of Workshop on Parallel Programming on Accelerator Clusters—PPAC'09*, (2009)
527. Munshi, A., et al.: *OpenCL Programming Guide*, 1 ed., Addison-Wesley Professional, (2011)
528. Prince, S.: *Computer Vision: Models, Learning, and Inference*. Cambridge University Press, Cambridge (2012)
529. Lindeberg, T.: *Scale Space Theory in Computer Vision*. Springer, New York (2010)
530. Pele, O.: Distance Functions: Theory, Algorithms and Applications. Ph.D. Thesis, Hebrew University, (2011)
531. Schapire, R.E., Singer, Y.: Improved boosting algorithms using confidence-rated predictions. *Mach. Learn.* (1999)
532. Bache, K., Lichman, M.: UCI Machine Learning Repository (<http://archive.ics.uci.edu/ml>), University of California, School of Information and Computer Science, Irvine, CA, (2013)
533. Zach, C.: Fast and high quality fusion of depth maps. *3DPVT Joint 3DIM/3DPVT Conference 3D Imaging, Modeling, Processing, Visualization, Transmission* (2008)
534. Visual Genomes for Synthetic Vision, Scott Krig, TBP (2016)
535. Grimes, D.B., Rao, R.P.N.: Bilinear sparse coding for invariant vision. *Neural Comput.* **17**(1), 47–73 (2005)
536. Roger, G., Raina, R., Kwong, H., Ng, A.Y.: Shift-invariant sparse coding for audio classification. In: *Proceedings of the 23rd Conference in Uncertainty in Artificial Intelligence (UAI'07)*, (2007)
537. The Statistical Inefficiency of Sparse Coding for Images (or, One Gabor to Rule them All), Technical Report, James Bergstra, Aaron Courville, and Yoshua Bengio (2011)
538. Scalable Object Detection using Deep Neural Networks Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov
539. Hinton, G., Osindero, S., Teh, Y.: A fast learning algorithm for deep belief nets. *Neural Comput.* **18**, 1527–1554 (2006)
540. Hinton, G., Salakhutdinov, R.: Reducing the dimensionality of data with neural networks. *Science* **313**(5786), 504–507 (2006)
541. Anh, N., Yosinski, J., Clune, J.: Deep neural networks are easily fooled: high confidence predictions for unrecognizable images. *CVPR* (2015)
542. He, K., Zhang, X., Ren, S., Sun, J.: Spatial pyramid pooling in deep convolutional networks for visual recognition. *ECCV* (2014)
543. Mutch, J., Lowe, D.G.: Object class recognition and localization using sparse features with limited receptive fields. *IJCV* (2008)
544. Serre, T., Wolf, L., Poggio, T.: Object recognition with features inspired by visual cortex. *CVPR* (2005)
545. Sanchez, J., Perronnin, F., Mensink, T., Verbeek, J.: Image classification with the fisher vector: theory and practice. *IJCV* (2013)
546. Min, L., Chen, Q., Yan, S.: Network in network. In: *ICLR* (2014)
547. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going Deeper with Convolutions
548. Behnke, S.: Hierarchical neural networks for image interpretation. Draft submitted to Springer Published as volume 2766 of Lecture Notes in Computer Science ISBN: 3-540-40722-7, Springer (2003)

549. Girshick, R., Iandola, F., Darrell, T., Malik, J.: Deformable part models are convolutional neural networks. CVPR (2014)
550. van de Sande, E.A., Snoek, C.G.M., Smeulders, A.W.M.: Fisher and VLAD with FLAIR. In: IEEE Conference on Computer Vision and Pattern Recognition (2014)
551. Ranzato, M., Boureau, Y., LeCun, Y.: Sparse feature learning for deep belief networks. In: Proceedings of Neural Information Processing Systems (NIPS), (2007)
552. Schmidhuber, J.: Deep learning in neural networks: an overview, Technical Report IDSIA-03-14/arXiv:1404.7828 v4
553. Li D., Yu, D.: Deep learning methods and applications, foundations and Trends® in signal processing **7**
554. Yoshua, B., Goodfellow, I.J., Courville, A.: Deep learning. MIT Press, (2016) (in preparation)
555. Anderson, J.A., Rosenfeld, E., (eds.): Neurocomputing: foundations of research. MIT Press, Cambridge MA, (1988). Also Neurocomputing vol. 2: directions for research. MIT Press, Cambridge MA, (1991)
556. Jackson, P.: Introduction to Expert Systems, 3 ed., Addison Wesley, (1998)
557. Rosenblatt, F.: The Perceptron: a probabilistic model for information storage and organization in the brain. Psychol. Rev. (1958)
558. Joseph, R.D.: Contributions to Perceptron Theory. PhD thesis, Cornell Univ. (1961)
559. Wiesel, D.H., Hubel, T.N.: Receptive fields of single neurones in the cat's striate cortex. J. Physiol. (1959)
560. Hubel, D.H., Wiesel, T.: Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. J. Physiol. **160**, 106–154 (1962)
561. McCulloch, W., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. (1943)
562. Hebb, D.O.: The Organization of Behavior. Wiley, New York (1949)
563. Rosenblatt, F.: The Perceptron—a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory (1957)
564. Ivakhnenko, A.G.: The group method of data handling—a rival of the method of stochastic approximation. Soviet Autom. Contr. (1968)
565. Ivakhnenko, A.G., Lapa, V.G.: Cybernetic predicting devices. CCM Inform. Corp. (1965)
566. Ivakhnenko, A.G., Lapa, V.G., McDonough, R.N.: Cybernetics and Forecasting Techniques. American Elsevier, NY, (1967)
567. Ivakhnenko, A.G.: Polynomial theory of complex systems. IEEE Trans. Syst. Man Cybern. **4**, 364–378 (1971)
568. Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.R.: Improving neural networks by preventing co-adaptation of feature detectors. CoRR, abs/1207.0580, (2012)
569. Ikeda, S., Ochiai, M., Sawaragi, Y.: Sequential GMDH algorithm and its application to river flow prediction. IEEE Trans. Syst. Man Cybern. **7**, 473–479 (1976)
570. Fukushima, K.: Neural network model for a mechanism of pattern recognition unaffected by shift in position—Neocognitron. Trans. IECE J. **62**(10), 658–665 (1979)
571. Fukushima, K.: Neocognitron: a self-organizing neural network for a mechanism of pattern recognition unaffected by shift in position. Biol. Cybern. **36**(4), 193–202 (1980)
572. Dreyfus, S.E.: The numerical solution of variational problems. J. Math. Anal. Appl. **5**(1), 30–45 (1962)
573. Dreyfus, S.E.: The computational solution of optimal. (1973)
574. LeCun, Y.: Une procédure d'apprentissage pour réseau ‘à seuil asymétrique. Proceedings of Cognitiva, vol 85, Paris, pp. 599–604, (1985)
575. LeCun, Y.: A theoretical framework for back-propagation. In: Touretzky, D., Hinton, G., Sejnowski, T., (eds.) Proceedings of the 1988 Connectionist Models Summer School, CMU, Morgan Kaufmann, Pittsburgh, PA, pp. 21–28, (1988)
576. LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Back-propagation applied to handwritten zip code recognition. Neural Comput. **1**(4), 541–551 (1989)
577. LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Handwritten digit recognition with a back-propagation network. In: Touretzky, D. S., (ed.) Advances in Neural Information Processing Systems, vol 2, Morgan Kaufmann, pp. 396–404, (1990a)
578. Kelley, H.J.: Gradient theory of optimal flight paths. ARS J. **30**(10), 947–954 (1960)
579. Bryson, A.E.: A gradient method for optimizing multi-stage allocation processes. In: Proc. Harvard Univ. Symposium on Digital Computers and Their Applications, (1961)
580. Bryson, Jr., A. E. and Denham, W. F.: A steepest-ascent method for solving optimum programming problems. Technical Report BR-1303, Raytheon Company, Missile and Space Division, (1961)
581. Werbos, P.J.: The roots of backpropagation: from ordered derivatives to neural networks and political forecasting. Wiley, (1994)
582. Schmidhuber, J.: Learning complex, extended sequences using the principle of history compression. Neural Comput. (1992)

583. Graves, A., Wayne, G., Danihelka, I.: Neural turing machines. (2014)
584. Hochreiter, S., Jürgen, S.: Long short-term memory, neural computation. (1997)
585. Ng, A.: Stanford CS229 Lecture notes. Support Vector Mach.
586. Shawe-Taylor, J., Cristianini, N.: Support vector machines and other kernel-based learning methods, Cambridge University Press, (2000)
587. Hinton, G.E., Sejnowski, T.J., Rumelhart, D.E., McClelland, J.L.: Learning and relearning in Boltzmann machines, PDP Research Group (1986)
588. Ackley, D.H., Hinton, G.E., Sejnowski, TJ.: A learning algorithm for Boltzmann machines. Cogn. Sci. (1985)
589. Hopfield, J.J.: Neural networks and physical systems with emergent collective computational abilities. Proc. Natl. Acad. Sci. U. S. A. (1982)
590. Smolensky, P.: Chapter 6: information processing in dynamical systems: foundations of harmony theory. In: Rumelhart, D.E., McClelland, J.L. (eds.) Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol 1, Foundations. MIT Press (1986)
591. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv (2014)
592. Also see NiN slides from ILSVRC (2014) http://www.image-net.org/challenges/LSVRC/2014/slides/ILSVRC2014_NUS_release.pdf
593. LeCun, Y.: A theoretical framework for back-propagation. In: Touretzky, D., Hinton, G., Sejnowski, T., (eds.) Proceedings of the 1988 Connectionist Models Summer School, CMU, pp. 21–28, Morgan Kaufmann, Pittsburgh, PA, (1988)
594. Vapnik, V., Lerner, A.: Pattern recognition using generalized portrait method. Autom. Remote Contr. (1963)
595. Boser, B.E., Guyon, I.M., Vapnik, V.N.: A training algorithm for optimal margin classifiers. ACM COLT '92, (1992)
596. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. (1995)
597. Vapnik, V.: Estimation of Dependences Based on Empirical Data [in Russian]. Nauka, Moscow, (1979). English translation, Springer, New York, (1982)
598. Vapnik, V.: The Nature of Statistical Learning Theory. Springer, New York (1995)
599. Vapnik, V.: Statistical Learning Theory. John Wiley and Sons, Inc., New York (1998)
600. Powell, M.J.D.: An efficient method for finding the minimum of a function of several variables without calculating derivatives. Comput. J. (1964)
601. Carreira-Perpignan, M.A., Hinton, G.E.: On contrastive divergence learning. In: Artificial Intelligence and Statistics, (2005)
602. Cireşan, D., Meier, U., Schmidhuber, J.: Multi-column Deep Neural Networks for Image Classification, cvpr (2012)
603. Coates, A., Lee, H., Ng, A.: An analysis of single-layer networks in unsupervised feature learning, AISTATS (2011)
604. Rosenblatt, F.: Principles of Neurodynamics Unclassifie—Armed Services Technical Informatm Agency. Spartan, Washington, DC (1961)
605. Baddeley, A., Eysenck, M., Anderson, M.: Memory. Psychology Press, (2009)
606. Goldman-Rakic, P.S.: Cellular basis of working memory. Neuron **14**(3), 477–485 (1995)
607. Rumelhart, D.E., McClelland, J.L., Group, P.R., et al.: Parallel distributed processing, vol 1. MIT Press, (1986)
608. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. arXiv:1409.4842, (2014)
609. Von Neumann, J.: First draft of a report on the edvac. (1945)
610. Goodfellow, I.J., Warde-Farley, D., Mirza, M., Courville, A., Bengio, Y.: Maxout networks. In: International Conference on Machine Learning (ICML), (2013)
611. Breiman, L.: Bagging predictors. Mach. Learn. **24**(2), 123–140 (1994)
612. Stollenga, M., Masci, J., Gomez, F., Schmidhuber, J.: Deep networks with internal selective attention through feedback connections. ICML (2014)
613. Rupesh Kumar, S., Masci, J., Kazerounian, S., Gomez, F., Schmidhuber, J.: Compete to compute. In: NIPS, (2013)
614. Cristian, B., Caruana, R., Niculescu-Mizil, A.: Model compression, ACM SIGKDD (2006)
615. Mansimov, E., Srivastava, N., Salakhutdinov, R.: Initialization Strategies of Spatio-Temporal Convolutional Neural Networks, Technical Report, (2014)
616. Weng, J., Ahuja, N., Huang, T.S.: Cresceptron: a self-organizing neural network which grows adaptively. In: Proceedings of Int'l Joint Conference on Neural Networks, Baltimore, MD, (1992)
617. Cadieu, CF, Hong H, Yamins DLK, Pinto N, Ardila D, Solomon EA, Majaj NJ, DiCarlo JJ.: Deep neural networks rival the representation of primate IT cortex for core visual object recognition, (2014), PLOS 2014DOI: [10.1371/journal.pcbi.1003963](https://doi.org/10.1371/journal.pcbi.1003963)
618. Coates, A., Ng, A.Y.: The importance of encoding versus training with sparse coding and vector quantization. ICML (2011)

619. Jarrett, K., Kavukcuoglu, K., Ranzato, M., Le-Cun, Y.: What is the best multi-stage architecture for object recognition?, ICCV (2009)
620. Hinton, G., Vinyals, O., Dean, J.: Distilling the Knowledge in a Neural Network. NIPS (2014)
621. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Comput. (2006)
622. Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. NIPS (2007)
623. Kandel, E.R., Schwartz, J.H., Jessel, T.M. (eds.) Principles of Neural Science, 4th ed., McGraw-Hill, (2000)
624. Rao, R.P.N., Ballard, D.H.: Predictive coding in the visual cortex: A functional interpretation of some extra-classical receptive-field effects. Nat Neurosci. (1999)
625. Rosenfeld, A., Hummel, R.A., Zucker, S.W.: Scene labeling by relaxation operations. IEEE Trans. Syst. Man Cybernetics (1976)
626. M  tin, C., Frost, D.O.: Visual responses of neurons in somatosensory cortex of hamsters with experimentally induced retinal projections to somatosensory thalamus. Proc. Natl. Acad. Sci. U. S. A. **86**(1), 357–361 (1989)
627. Roe, A.W., Pallas, S.L., Kwon, Y.H., Sur, M.: Visual projections routed to the auditory pathway in ferrets: receptive fields of visual neurons in primary auditory cortex. J. Neurosci. **12**(9), 3651–3664 (1992)
628. Bach-y-Rita, P., Kaczmarek, K.A., Tyler, M.E., Garcia-LoraVenue, J.: Form perception with a 49-point electrotactile stimulus array of the tongue: a technical note. J. Rehabil. Res. Dev. (1998)
629. Bach-y-Rita, P., Tyler, M.E., Kaczmarek, K.A.: Seeing with the brain. IJHCI (2003)
630. Laurenz, W.: How Does Our Visual System Achieve Shift and Size Invariance, Problems in Systems Neuroscience, Oxford University Press, (2002)
631. Thomas Yeo, B.T., Krienen, F.M., Sepulcre, J., Sabuncu, M.R., Lashkari, D., Hollinshead, M., Roffman, J.L., Smoller, J.W., Z  lle, L., Polimeni, J.R., Fischl, B., Liu, H., Buckner, R.L.: The organization of the human cerebral cortex estimated by intrinsic functional connectivity. J. Neurophysiol. (2011)
632. Gross, G.N., L  mo, T., Sveen, O.: Participation of inhibitory and excitatory interneurones in the control of hippocampal cortical output, Per Anderson, The Interneuron, University of California Press, Los Angeles, (1969)
633. John, C.E., Ito, M., Szent  gothai, J.: The cerebellum as a neuronal machine, Springer, New York, (1967)
634. Costas, S.: Interneuronal mechanisms in the cortex. The Interneuron, University of California Press, Los Angeles, (1969)
635. Stephen, G.: Contour enhancement, short-term memory, and constancies in reverberating neural networks, Studies in Applied Mathematics, (1973)
636. Parikh, D., Zitnick, C.L.: The role of features, algorithms and data in visual recognition. CVPR (2010)
637. Christopher, B.: Pattern Recognition and Machine Learning, Springer, (2006)
638. Eigen, D., Rolfe, J., Fergus, R., LeCun, Y.: Understanding deep architectures using a recursive convolutional network, arXiv:1312.1847 [cs.LG]
639. NIPS.: Tutorial—Deep Learning for Computer Vision (Rob Fergus) (2013)
640. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks. NIPS (2012)
641. Zeiler, M.D., Fergus, R.: Visualizing and Understanding Convolutional Networks. ECCV (2014)
642. Zeiler, M., Taylor, G., Fergus, R.: Adaptive deconvolutional networks for mid and high level feature learning. In: ICCV, (2011)
643. Olga, R., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: Large scale visual recognition challenge. ImageNet <http://arxiv.org/abs/1409.0575>, (2015)
644. Random Search for Hyper-Parameter Optimization James Bergstra JAMES.BERGSTRA@UMONTREAL.CA Yoshua Bengio, JMLR (2012)
645. Donahue, J., Jia, Y., Vinyals, O., Hoffman, J., Zhang, N., Tzeng, E., Darrell, T.: DeCAF: A deep convolutional activation feature for generic visual recognition. CVPR (2013)
646. Yamins, D.L., Hong, H., Cadieu, C., DiCarlo, J.J.: Hierarchical modular optimization of convolutional networks achieves representations similar to macaque IT and human ventral stream. NIPS (2013)
647. Haykin, S.: Neural Networks: a comprehensive foundation. Pearson Educ. (1999)
648. Pascanu, R., Mikolov, T., Bengio, Y.: On the difficulty of training recurrent neural networks. (2013)
649. Daniel L.K.Y., Honga, H., Cadieua, C.F., Solomona, E.A., Seiberta, D., DiCarloa, J.J.: Performance-optimized hierarchical models predict neural responses in higher visual cortex. Natl. Acad. Sci. (2015)
650. US Government BRAIN Initiative.: <http://www.artificialbrains.com/darpa-synapse-program>
651. European Union Human Brain Project.: <https://www.humanbrainproject.eu>
652. Canadian Government Computation & Adaptive Perception Canadian Institute For Advanced Research CIFAR. <http://www.cifar.ca/neural-computation-and-adaptive-perception-research-program>
653. Tatyana, V., Sharpee, O., Kouh M., Reynolds, J.H.: Trade-off between curvature tuning and position invariance in visual area. PNAS. (2013)
654. Neural Networks, Tricks of the Trade, 2nd ed., Springer, (2012)

655. LeCun, Y.: Convolutional networks and applications in vision, Comput. Sci. Dept., New York Univ., New York, NY, USA, Kavukcuoglu, K., Farabet, C., ISCAS. (2010)
656. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. ICLR. (2015)
657. Lyu, S., Simoncelli, E.P.: Nonlinear image representation using divisive normalization. CVPR. (2008)
658. Pinto, N., Cox, D.D., DiCarlo, J.J.: Why is real-world visual object recognition hard? PLoS Comput Biol. (2008)
659. Yang Y., Hospedales, T.M.: Deep neural networks for sketch recognition. (2015)
660. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting, JMLR. (2014)
661. Wan, L., Zeiler, M., Zhang, S., LeCun, Y., Fergus, R.: Regularization of neural network using drop connect. Int. Conf. Mach. Learn. (2013)
662. Breiman, L.: Bagging predictors. Mach. Learn. (1994)
663. Zeiler, M.D., Fergus, R.: Stochastic pooling for regularization of deep convolutional. Neural Netw.
664. Mamalet, F., Garcia, C.: Simplifying convnets for fast learning. ICANN. (2012)
665. Gens, R., Domingos, P.: Deep symmetry networks. NIPS (2014) see also slides at <http://research.microsoft.com/apps/video/default.aspx?id=219488>
666. Yosinski, J., Clune, J., Bengio, Y., Lipson, H.: How transferable are features in deep neural networks?, NIPS (2014)
667. Uijlings, J.R.R., van de Sande, K.E.A., Gevers, T., Smeulders, A.W.M.: Selective search for object recognition. IJCV (2013)
668. Hagan, M.T., Demuth, H.B., Beale, M.H.: Neural network design. PWS Publishing, (1996)
669. Dominik S., M'uller, A., Behnke, S.: Evaluation of pooling operations in convolutional architectures for object recognition. ICANN. (2010)
670. Kaiming, H., Zhang, X., Ren, S., Sun, J.: Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. CVPR (2015)
671. Field, G., Gauthier, J., Sher, A., Greschner, M., Machado, T., Jepson, L., Shlens, J., Gunning, D., Mathieson, K., Dabrowski, W., et al.: Functional connectivity in the retina at the resolution of photoreceptors. Nature. (2010)
672. Rosenblatt, F.: The Perceptron: A theory of statistical separability in cognitive systems. Cornell Aeronautical Laboratory, Buffalo, Inc. Rep. No. VG-1196-G-1, (1958)
673. Auer, P., Burgsteiner, H., Maass, W.: A learning rule for very simple universal approximators consisting of a single layer of perceptrons. Austr. Sci. Fund (2008)
674. Vapnik, V., Chervonenkis, A., Moskva, N.: Pattern Recognition Theory, Statistical Learning Problems. (1974)
675. Hearst, M.A., Berkeley, U.C.: Support vector machines. IEEE Intell. Syst. (1998)
676. John P.: How to implement SVM's, Microsoft Research. IEEE Intelligent Systems, (1998)
677. Fukushima, K.: Cognitron: a self-organizing multilayered neural network, Biological Cybernetics, Springer, (1975)
678. Fukushima, K.: Artificial vision by multi-layered neural networks: and its advances. Neural Netw. **37**, 103–119
679. Fukushima, K.: Training multi-layered neural network Neocognitron. Neural Netw. **40**, 18–31
680. Joan, B., Zaremba, W., Szlam, A., LeCun, Y.: Spectral networks and locally connected networks on graphs. arXiv:1312.6203 [cs.LG] (2014)
681. Pascanu, R., Gulcehre, C., Cho, K., Bengio, Y.: How to construct deep recurrent neural networks. ICLR. (2014)
682. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE (1998)
683. http://www.imagemagick.org/Usage/convolve/#convolve_vs_correlate
684. Springenberg, J.T., Dosovitskiy, A., Brox, T., Riedmiller, M.: Striving for simplicity: the all convolutional net. CVPR. (2015)
685. Fractional max-pooling Benjamin Graham. CVPR. (2014)
686. The Human Connectome Project is a consortium of leading neurological research labs which are mapping out the pathways in the brain. See <http://www.humanconnectomeproject.org/about/>
687. Cun, Y.L., Denker, J.S., Solla, S.A.: Optimal brain damage. NIPS. (1990)
688. Waibel, A.: Consonant recognition by modular construction of large phonemic time-delay neural networks. IEEE ASSP (1989)
689. Farabet, C., LeCun, Y., Kavukcuoglu, K., Culurciello, E., Martini, B., Akselrod, P., Talay, S.: Large-scale FPGA-based convolutional networks. (2011)
690. Clement, F., LeCun, Y., Kavukcuoglu, K., Culurciello, E., Martini, B., Akselrod, P., Talay, S.: Hardware accelerated convolutional neural networks for synthetic vision systems. ISCAS. (2010)
691. Sermanet, P., Eigen, D., Zhang X., Mathieu M., Fergus R., LeCun, Y.: OverFeat: integrated recognition, localization and detection using convolutional networks. CVPR. (2014)
692. Dong, J., Xia, W., Chen, Q., Feng, J., Huang, Z., Yan, S.: Subcategory-aware object classification. CVPR. (2013)
693. Jun, Y., Ni, B., Kassim, A.A.: Half-CNN: a general framework for whole-image regression. CVPR. (2014)

694. Hugo, L., Bengio, Y., Louradour, J., Lamblin, P.: Exploring strategies for training deep neural networks. *JMLR*. (2009)
695. Yu, C., Yu, F.X., Feris, R.S., Kumar, S., Choudhary, A., Chang, S.-F.: Fast neural networks with circulant projections. (2015)
696. Jochem, T., Dean Pomerleau, AI.: Life in the fast lane the evolution of an adaptive vehicle control system. *Magazine* (1996)
697. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. *JMLR*. (2010)
698. Hastie, T., Friedman.: *The Elements of Statistical Learning*. 2nd ed., Springer, (2009)
699. Boureau, Y.-L., Le Roux, N., Bach, F., Ponce, J., Lecun, Y.: Ask the locals: multi-way local pooling for image recognition *ICCV'11*
700. Ren, W., Yan, S., Shan, Y., Dang, Q., Sun, G.: Deep image: scaling up image recognition. *CVPR*. (2015)
701. Karen, S., Simonyan, K.: <http://imagenet.org/tutorials/cvpr2015/recent.pdf>, ILSVRC Submission Essentials in the light of recent developments. *ImageNet*, Tutorial (2015)
702. Jon Shlens Google Research.: Directions in convolutional neural networks at Google, (2015), http://vision.stanford.edu/teaching/cs231n/slides/jon_talk.pdf
703. Sergey, I., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. *CVPR*. (2015)
704. Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. *CVPR*. (2014)
705. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. *Int. Conf. Artif. Intell. Stat.* (2010)
706. Chunhui, G., Lim, J.J., Arbelaez, P., Malik, J.: Recognition using regions. *CVPR*. (2009)
707. Ross G.: Fast R-CNN. *CVPR*. (2015)
708. Volodymyr, M., Heess, N., Graves, A., Kavukcuoglu, K.: Recurrent models of visual attention. *NIPS*. (2014)
709. Oriol, V., Toshev, A., Bengio, S., Erhan, D.: Show and tell: a neural image caption generator. (2015)
710. Ren, M., Kiros, R., Zemel, R.: Exploring models and data for image question answering. *ICML* (2015)
711. Subhashini, V., Rohrbach, M., Donahue, J., Mooney, R., Darrell, T., Saenko, K.: Sequence to sequence—video to text. (2015)
712. Graves, A.: Generating sequences with recurrent neural networks. (2014)
713. Schmidhuber, J., Wierstra, D., Gagliolo, M., Gomez, F.: Training recurrent networks by evolino. *Neural Comput.* (2007)
714. Weston, J., Chopra, S., Bordes, A.: Memory networks. *ICLR*. (2015)
715. LaRue, J.P.: A Bi-directional Neural Network Based on a Convolutional Neural Network and Associative Memory Matrices That Meets the Universal Approximation Theorem, Jadco Signals, Charleston, SC, USA, 1 315 717 9009 james@jadcosignals.com
716. Zhou, R.W., Quek, C.: DCBAM: A discrete chainable bidirectional associative memory. *Pattern Recogn. Lett.* (1991)
717. Kosko, B.: Bidirectional associative memories. *IEEE Trans. Syst. Man Cybern.* **7**, 49–60 (1988)
718. Kohonen, T.: Correlation matrix memories. *IEEE Trans. Comput.* 353–359, (1972)
719. Hopfield, J.J.: Neural networks and physical systems with emergent collective computational abilities. *Proc. Natl. Acad. Sci. U. S. A.* **79**(8), 2554–2558 (1982)
720. Schmidhuber, J.: Long Short-Term Memory: Tutorial on LSTM Recurrent Networks, <http://people.idsia.ch/~juergen/lstm/>
721. Hochreiter, S., Steven, Y.A., Conwell, P.R.: Learning to learn using gradient descent. *ICANN*. (2001)
722. Schmidhuber, J.: Learning to control fast-weight memories: an alternative to recurrent nets. *Neural Comput.* (1992)
723. Jeff, D., Hendricks, L.A., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., Darrell, T.: Long-term recurrent convolutional networks for visual recognition and description. *CVPR*. (2015)
724. Mengye, R., Kiros, R., Zemel, R.: Exploring models and data for image question answering. *ICML*. (2015)
725. Alex, G., Doktors der Naturwissenschaften.: Supervised Sequence Labelling with Recurrent Neural Networks
726. Graves, A., Fernandez, S., Schmidhuber, J.: Multi-dimensional recurrent neural networks. *ICANN*. (2007)
727. Baldi, P., Pollastri, G.: The principled design of large-scale recursive neural network architectures—DAG-RNN's and the protein structure prediction problem. *JMLR*. (2003)
728. Karol, G., Danihelka, I., Graves, A., Rezende, D., Wierstra, D.: DRAW: a recurrent neural network for image generation. *ICML*. (2015)
729. Richard, S., Huval, B., Bhat, B., Manning, C.D., Ng, A.Y.: Convolutional-recursive deep learning for 3D object classification. *NIPS*. (2012)
730. B., Shuai, Z., Gang, W.: Quaddirectional 2D-recurrent neural networks for image labeling. *IEEE SPL*. (2015)

731. Zuo, Z., Shuai, B., Wang, G., Liu, X., Wang, X., Wang, B., Chen, Y.: Convolutional recurrent neural networks: learning spatial dependencies for image representation. *CVPR*. (2015)
732. Alex, G., Schmidhuber, J.: Offline handwriting recognition with multidimensional recurrent neural networks. *NIPS*. (2008)
733. Graves, A., Fernandez, S., Gomez, F., Schmidhuber, J.: Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. *ICML*. (2012)
734. Kyunghyun, C., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. *EMNLP*. (2014)
735. Kyunghyun, C., van Merriënboer, B., Bahdanau, D., Bengio, Y.: On the properties of neural machine translation: encoder-decoder approaches. *SSST-8*. (2014)
736. Peter, T., Horne, B.G., Lee Giles, C.: Collingwood, P.C.: Finite state machines and recurrent neural networks—automata and dynamical systems approaches. *Neural Networks Pattern Recogn.* Chapter 6, (1998)
737. Arai, K., Nakano, R.: Stable behavior in a recurrent neural network for a finite state machine. *Neural Netw.* **13**(6), (2000)
738. Wojciech, Z., Sutskever, I.: Learning to execute
739. Rumelhart, D.E., McClelland, J.L.: Parallel Distributed processing: explorations in the microstructure of cognition. (1986)
740. Elman, J.L.: Finding structure in time. *Cogn. Sci.* (1990)
741. Elman, J.L.: Distributed representations, simple recurrent networks, and grammatical structure. *Mach. Learn.* (1991)
742. Elman, J.L.: Learning and development in neural networks: the importance of starting small. *Cognition* (1993)
743. Williams, R.J., Zipser, D.: Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity. *Back-propagation: Theory, Architectures and Applications*, Lawrence Erlbaum Publishers, (1995)
744. Robinson, A.J., Fallside, F.: The Utility Driven Dynamic Error Propagation Network. Technical Report CUED/F-INFENG/TR.1, Cambridge, (1987)
745. Werbos, P.: Backpropagation through time: what it does and how to do it. *Proc. IEEE* (1990)
746. Boden, M.: A guide to recurrent neural networks and backpropagation. (2014)
747. Ders, F.: Long Short-Term Memory in Recurrent Neural Networks, PhD Dissertation, (2001)
748. Qi, L., Zhu, J.: Revisit long short-term memory: an optimization perspective. *NIPS*. (2015)
749. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. *NIPS*. (2014)
750. Kyunghyun, C., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation. (2014)
751. Liang, M., Hu, X.: Recurrent convolutional neural network for object recognition. *CVPR*. (2015)
752. Socher, R., Lin, C.C., Manning, C., Ng, A.Y.: Parsing natural scenes and natural language with recursive neural networks. In: *Proceedings of the 28th International Conference on Machine Learning (ICML)*, (2011)
753. Socher, R., Manning, C.D., Ng, A.Y.: Learning continuous phrase representations and syntactic parsing with recursive neural networks. In: *Advances in Neural Information Processing Systems*, NIPS. (2010)
754. Volodymyr, M., Heess, N., Graves, A., Kavukcuoglu, K.: Recurrent Models of Visual Attention
755. Steve, B., Wah, C., Schroff, F., Babenko, B., Welinder, P., Perona, P., Belongie, S.: Visual recognition with humans in the loop. In *Computer Vision–ECCV*, Springer, (2010)
756. Tom, S., Glasmachers, T., Schmidhuber, J.: High dimensions and heavy tails for natural evolution strategies. *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. ACM. (2011)
757. Zaremba, W., Sutskever, I.: Reinforcement Learning Neural Turing Machines. (2015)
758. Hebb, D.: *The Organization of Behaviour*. Wiley, New York (1949)
759. Liefeng, B., Lai, K., Ren, X., Fox, D.: Object recognition with hierarchical kernel descriptors. *CVPR*. (2011)
760. Ivakhnenko, G.A., Cerdà R.: Inductive Self-Organizing GMDH Algorithms for Complex Systems Modeling and Forecasting. <http://www.gmdh.net/articles/index.html>, see the general GMDH website for several other resources, <http://www.gmdh.net>
761. The review of problems solvable by algorithms of the group method of data handling. *Pattern Recogn. Image Anal.* (1995), www.gmdh.net/articles/
762. Ladislav, Z.: Learning simple dependencies by polynomial neural network. *J. Inform. Contr. Manag. Syst.* **8**(3), (2010)
763. Liefeng, B., Sminchisescu, C.: Efficient match kernel between sets of features for visual recognition. *NIPS*. (2009)
764. Julesz, B.: Textons, the elements of texture perception and their interactions. *Nature* **290**, 91–97 (1981)
765. Zhang, J., Marszałek, M., Lazebnik, S., Schmid, C.: Local features and kernels for classification of texture and object categories: a comprehensive study. *IJCV*. (2007)
766. Lazebnik, S., Schmid, C., Ponce, J.: A maximum entropy framework for part-based texture and object recognition. *IEEE CV*. (2005)
767. Lampert, C.H.: Kernel methods in computer vision. *Found. Trends Comput. Graph. Vis.* **4**(3), 193–285 (2009)

768. Jurie, F., Triggs, B.: Creating efficient codebooks for visual recognition. ICCV. (2005)
769. Youngmin, C., Saul, L.K.: Kernel methods for deep learning. NIPS. (2009)
770. Vedaldi, A., Gulshan, V., Varma, M., Zisserman, A.: Multiple kernels for object detection. (2009)
771. Varma, M., Ray, D.: Learning the discriminative power-invariance trade-off. Int. Conf. Comput. Vis. (2007)
772. Klaus-Robert, M., Mika, S., Rätsch, G., Tsuda, K., Schölkopf, B.: An introduction to kernel-based learning algorithms. IEEE TNN. (2001)
773. Nilsson, M.-E., Zisserman, A.: A visual vocabulary for flower classification. In: CVPR. (2006)
774. Liefeng, B., Ren, X., Fox, D.: Kernel descriptors for visual recognition. NIPS. (2010)
775. Boswell, D.: Introduction to Support Vector Machines. (2002)
776. Radu Tudor, I., Popescu, M., Grozea, C.: Local learning to improve bag of visual words model for facial expression recognition. ICML. (2013)
777. Haussler, D.: Convolution kernels on discrete structures. Tech. Rep. (1999)
778. Pati, Y.C., Rezaifar, R., Krishnaprasad, P.S.: Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition. Asilomar Conf. Signals Syst. Comput. (1993)
779. Aharon, M., Elad, M., Bruckstein, A.: K-SVD: an algorithm for designing overcomplete dictionaries for sparse representation. IEEE Trans. Signal Process. **54**(11), 4311–4322 (2006)
780. Bruna, J., Mallat, S.: Invariant Scattering Convolution Networks. (2012)
781. Wonmin, B., Breuel, T.M., Raue, F., Liwicki, M.: Scene labeling with LSTM recurrent neural networks. CVPR. (2015)
782. Du, Y., Wei, W., Liang, W.: Hierarchical recurrent neural network for skeleton based action recognition. CVPR. (2015)
783. Jianchao, Y., Yu, K., Lv, F., Huang, Yihong Gong, T.: Locality-constrained Linear Coding for image classification. CVPR (2001) Jinjun Wang Akiira Media Syst., Palo Alto, CA, USA
784. Reubold, J.: Kernel descriptors in comparison with hierarchical matching pursuit. Seminar Thesis, Proceedings of the Robot Learning Seminar, (2010)
785. John, S.-T., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, (2004)
786. Hofmann, T., Scholkopf, B., Smola, A.J.: Kernel methods in machine learning. Ann. Stat.
787. Rojas, R.: Neural Networks—A Systematic Introduction, Springer, (1996)
788. Teknomo, K.: Support Vector Machines Tutorial
789. Vladimir, C., Mulier, F.M.: Learning from Data: Concepts, Theory, and Methods, 2nd ed., Wiley, (2007)
790. Dan, C., Meier, U., Schmidhuber, J.: Multi-column Deep Neural Networks for Image Classification. CVPR. (2012)
791. Amnon, S., Hazan, T.: Algebraic set kernels with application to inference over local image representations. (2005)
792. Gehler, P., Nowozin, S.: On feature combination for multiclass object classification. CVPR. (2009)
793. Lanckriet, G.R.G., Cristianini, N., Bartlett, P., El Ghaoui, L., Jordan, M.I.: Learning the kernel matrix with semidefinite programming. JMLR. (2004)
794. Mairal, J., Koniusz, P., Harchaoui, Z., Schmid, C.: Convolutional kernel networks. NIPS. (2009)
795. Candes, E., Romberg, J.: Sparsity and incoherence in compressive sampling. Inverse Probl. **23**, 969 (2007)
796. Kai, Y., Lin, Y., Lafferty, J.: Learning image representations from the pixel level via hierarchical sparse coding. CVPR. (2011)
797. Jian, Z.F., Song, L., Yang X.K., Zhang, W.: Sub clustering K-SVD: size variable dictionary learning for sparse representations. ICIP. (2009)
798. Olshausen, B., Field, D.: Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature. (1996)
799. Mallat, S.G., Zhang, Z.: Matching pursuits with time-frequency dictionaries. IEEE Trans. Signal Process. 3397–3415, (1993)
800. Kwon, S., Wang, J., Shim, B.: Multipath matching pursuit. IEEE Trans. Inform. Theor. (2014)
801. Lloyd, S.P.: Least square quantization in PCM. Bell Telephone Laboratories Paper. Published in journal much later: Lloyd, S.P.: Least squares quantization in PCM, IEEE Trans. Inform. Theor. (1957/1982)
802. Voronoi, G.: Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Journal für die Reine und Angewandte Mathematik **133**(133), 97–178 (1908)
803. Mairal, J.: Sparse Coding for Machine Learning, Image Processing and Computer Vision. PhD thesis. Ecole Normale Supérieure de Cachan. (2010)
804. Mairal, J., Sapiro, G., Elad, M.: Multiscale sparse image representation with learned dictionaries. In: IEEE International Conference on Image Processing, San Antonio, Texas, USA, (2007), Oral Presentation
805. Mairal, J., Sapiro, G., Elad, M.: Learning multiscale sparse representations for image and video restoration. SIAM Multiscale Model. Simul. **7**(1), 214–241 (2008)
806. Mairal, J., Jenatton, R., Obozinski, G., Bach, F.: Learning hierarchical and topographic dictionaries with structured sparsity. In: Proceeding of the SPIE Conference on Wavelets and Sparsity XIV. (2011)
807. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification, 2nd edn. Wiley-Interscience, New York (2000)

808. Ethem, A.: *Introduction to Machine Learning*, MIT Press, (2004)
809. Tom, M.: *Machine Learning*, McGraw Hill, (1997)
810. LeCun, Y., Chopra, S., Hadsell, R., Huang, F.-J., Ranzato, M.-A.: A Tutorial on Energy-Based Learning, in *Predicting Structured Outputs*, MIT Press, (2006)
811. Pursuit, R.R., Zibulevsky, M., Elad, M.: Efficient Implementation of the K-SVD algorithm using Batch Orthogonal Matching. Technical Report—CS Technion, (2008)
812. Riesenhuber, M., Poggio, T.: Hierarchical models of object recognition in cortex. *Nature*. (1999)
813. Logothetis, N.K., Pauls, J., Poggio, T.: Shape representation in the inferior temporal cortex of monkeys. *Curr. Biol.* **5**(5), 552–563 (1995)
814. Tarr, M.: News on views: pandemonium revisited. *Nat. Neurosci.* (1999)
815. Selfridge, O.G.: Pandemonium: a paradigm for learning. *Proceedings of the Symposium on Mechanisation of Thought Processes* (1959)
816. Bülthoff, H., Edelman, S.: Psychophysical support for a two-dimensional view interpolation theory of object recognition. *Proc. Natl. Acad. Sci. U. S. A.* **89**, 60–64 (1992)
817. Logothetis, N., Pauls, J., Bülthoff, H., Poggio, T.: Shape representation in the inferior temporal cortex of monkeys. *Curr. Biol.* **4**, 401–414 (1994)
818. Tarr, M.: Rotating objects to recognize them: a case study on the role of viewpoint dependency in the recognition of three-dimensional objects. *Psychonom Bull. Rev.* **2**, 55–82 (1995)
819. Booth, M., Rolls, E.: View-invariant representations of familiar objects by neurons in the inferior temporal visual cortex. *Cereb. Cortex* **8**, 510–523 (1998)
820. Kobatake, E., Wang, G., Tanaka, K.: Effects of shape-discrimination training on the selectivity of inferotemporal cells in adult monkeys. *J. Neurophysiol.* **80**, 324–330 (1998)
821. Perrett, D., et al.: Viewer-centred and object-centred coding of heads in the macaque temporal cortex. *Exp. Brain Res.* **86**, 159–173 (1991)
822. Perrott, D.I., Rolls, E.T., Caan, W.: Visual neurons responsive to faces in the monkey temporal cortex. *Exp. Brain Res.* **47**, 329–342 (1982)
823. Tanaka, K., Saito, H.-A., Fukada, Y. & Moriya, M.: Coding visual images of objects in the inferotemporal cortex of the macaque monkey. *J. Neurophysiol.* **66**, 170–189
824. Parental olfactory experience influences behavior and neural structure in subsequent generations. *Nat. Neurosci.* **17**, 89–96, (2014)
825. Gjoneska, E., Pfenning, A., Mathys, H., Quon, G., Kundage, A., Tsai, L.H., Kellis, M.: Conserved epigenomic signals in mice and humans reveal immune basis of Alzheimer's disease. *Nature* (2015), doi: [10.1038/nature14252](https://doi.org/10.1038/nature14252)
826. Tanaka, K.: Inferotemporal cortex and object vision. *Annu. Rev. Neurosci.* **19**, 109–139 (1996)
827. Logothetis, N.K., Sheinberg, D.L.: Visual object recognition. *Annu. Rev. Neurosci.* **19**, 577–621 (1996)
828. Mutch, J., Lowe, D.: Multiclass object recognition with sparse, localized features. *CVPR*. (2006)
829. Serre, R.: Realistic modeling of simple and complex cell tuning in the HMAX model, and implications for invariant object recognition in cortex. *CBL Memo*. **239** (2004)
830. Hu, X.-L., Zhang, J.-W., Li, J.-M., Zhang, B.: Sparsity-regularized HMAX for visual recognition. *PLOS One*. **9**(1), (2014)
831. Charles, C., Kouh, M., Riesenhuber, M., & Poggio, T.: Shape Representation in V4: Investigating Position-Specific Tuning for Boundary Conformation with the Standard Model of Object Recognition. *AI Memo 2004-024* (2004)
832. Christian, T., Thome, N., Cord, M.: HMAX-S: deep scale representation for biologically inspired image categorization. *ICIP*. (2011)
833. Riesenhuber, M., Poggio, T.: Neural mechanisms of object recognition. *Curr. Opin. Neurobiol.* **12**, 162–168 (2002)
834. Ungerleider, L.G., Haxby, J.V.: “What” and “Where” in the human brain. *Curr. Opin. Neurobiol.* **4**, 157–165a, (1994), National Institute of Mental Health, Bethesda, USA
835. Serre, T., Wolf, L., Bileschi, S., Riesenhuber, M., Poggio, T.: Robust object recognition with cortex-like mechanisms. *PAMI*. (2007)
836. Mutch, J.: HMAX architecture models slide presentation. (2010)
837. <http://maxlab.neuro.georgetown.edu/hmax/>
838. Perronnin, F., Dance, C.: Fisher kernels on visual vocabularies for image categorization. In: *Proceedings of CVPR*, (2006)
839. Florent, P., Sánchez, J., Mensink, T.: Improving the fisher kernel for large-scale image classification. *ECCV*. (2010)
840. Giorgos, T., Avrithis, Y., Jégou, H.: To aggregate or not to aggregate: selective match kernels for image search. *ICCV*. (2013)

841. Jaakkola, T., Haussler, D.: Exploiting generative models in discriminative classifiers. In: NIPS, (1999)
842. Jegou, H., Douze, M., Schmid, C., Perez, P.: Aggregating local descriptors into a compact image representation. INRIA Rennes, Rennes, France, CVPR. (2010)
843. Relja, A., Zisserman, A.: All about VLAD. CVPR. (2013)
844. Chatfield, K., Lempitsky, V., Vedaldi, A., Zisserman, A.: The devil is in the details: an evaluation of recent feature encoding methods. Br. Mach. Vis. Conf. (2011)
845. Zhou, X., Yu, K., Zhang, T., Huang, T.S.: Image classification using super-vector coding of local image descriptors. In: Proceedings of ECCV, (2010)
846. van Gemert, J.C., Geusebroek, J.M., Veenman, C.J., Smeulders, A.W.M.: Kernel codebooks for scene categorization. In: Proceedings of ECCV, (2008)
847. Perronnin, F., Liu, Y., Sánchez, J., Poirier, H.: Large-scale image retrieval with compressed fisher vectors. CVPR. (2010)
848. Perronnin, F., Sánchez, J., Mensink, T.: Improving the fisher kernel for large-scale image classification. In: Proceedings of ECCV, (2010)
849. Jégou, H., Douze, M., Schmid, C.: Improving bag-of-features for large scale image search. Int. J. Comput. Vis. **87** (3), 316–336 (2010)
850. Farabet, C., Couprie, C., Najman, L., LeCun, Y.: Learning hierarchical features for scene labeling. IEEE PAMI. (2012)
851. Hong Lau, K., Tay, Y.H., Lo, F.L.: A HMAX with LLC for visual recognition. CVPR. (2015)
852. Smith, K.: Brain decoding: reading minds. Nature **502**(7472), (2013)
853. Smith, K.: Mind-reading with a brain scan. Nature (2008)
854. Bartholomew-Biggs, M., Brown, S., Christianson, B., Dixon, L.: “Automatic differentiation of algorithms” (PDF). J. Comput. Appl. Math. **124**(1-2), 171–190 (2000)
855. Plaut, D., Nowlan, S., Hinton, G.: Experiments on Learning by Back Propagation, Carnegie Mellon University, (1986)
856. Cayley, A.: On the theory of groups, as depending on the symbolic equation $\theta^n = 1$. Phil. Mag. **7**, (1854)
857. Cayley, A.: On the theory of groups. Am. J. Math. **11** (1889)
858. Voytek, B.: Brain metrics. Nature (2013)
859. Langleben Daniel, D., Dattilio Frank, M.: Commentary: the future of forensic functional brain imaging. J. Am. Acad. Psychiatry Law **36**(4), 502–504 (2008)
860. Finn, E.S., Shen, X., Scheinost, D., Rosenberg, M.D., Huang, J., Chun, M.M., Papademetris, X., Todd Constable, R.: Functional connectome fingerprinting: identifying individuals using patterns of brain connectivity. Nature (2015)
861. Bergami, M., Masserdotti, G., Temprana, S.G., Motori, E., Eriksson, T.M., Göbel, J., Yang, S.M., Conzelmann, K.-K., Schinder, A.F., Götz, M., Berninger, B.: A critical period for experience-dependent remodeling of adult-born neuron connectivity. Neuron (2015)
862. Allen Lee, W.-C., Huang, H., Feng, G., Sanes, J.R., Brown, E.N., So, P.T., Nedivi, E.: Dynamic remodeling of dendritic arbors in gabaergic interneurons of adult visual cortex. PLoS **4**(2), e29 (2006)
863. Wu, Z., Shuran, S., Aditya, K., Fisher, Y., Lingguang, Z., Xiaouou, T., Jianxiong, X.: 3D ShapeNets: a deep representation for volumetric shapes. CVPR. (2015)
864. Xiang, Y., Wongun, C., Yuanqing, L., Silvio, S.: Data-driven 3D voxel patterns for object category recognition. CVPR. (2015)
865. Papazov, C., Marks, T.K., Jones, M.: Real-time 3D head pose and facial landmark estimation from depth images using triangular surface patch features. CVPR. (2015)
866. Martinovic, A., Jan, K., Riemenschneider, H., Van Gool, L.: 3D All the way: semantic segmentation of urban scenes from start to end in 3D. CVPR. (2015)
867. Rock, J., Tanmay, G., Justin, T., JunYoung, G., Daeyun, S., Derek, H.: Completing 3D object shape from one depth image. CVPR. (2015)
868. Yub, J., Lee, H., Seok Heo, S., Dong Yun, Y., II: Random tree walk toward instantaneous 3D human pose estimation. CVPR. (2015)
869. Shape Priors Karimi Mahabadi, R., Hane, C., Pollefeys, M.: Segment based 3D object shape priors. CVPR (2015)
870. Xiaowei, Z., Spyridon, L., Xiaoyan, H., Kostas, D.: D shape estimation from 2D landmarks: a convex relaxation approach. CVPR (2015)
871. Levi, G., Hassner, T.: LATCh: learned arrangements of three patch codes, arXiv preprint arXiv:1501.03719 (2015)
872. He, K., Zhang, X., Ren, S., Sun, J.: Deep Residual Learning for Image Recognition. (2015)
873. Hinton, G.E., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531 (2015)

874. Romero, A., Nicolas, B., Samira Ebrahimi, K., Antoine, C., Carlo, G., Yoshua, B.: FitNets: hints for thin deep nets. arXiv:1412.6550 [cs], (2014)
875. Bucila, C., Caruana, R., Niculescu-Mizil, A.: Model compression. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06, ACM (2006)
876. Bengio, Y.: Learning deep architectures for AI. Found. Trends Mach. Learn. (2009)
877. Niklaus, M., Eddy, I., Philip H., Philipp F., Daniel C., Alexey D., Thomas B.: A Large Dataset to Train Convolutional Networks for Disparity, Optical Flow, and Scene Flow Estimation. CVPR, (2016)
878. Horn, B.K.P.: Shape from Shading: A Method for Obtaining the Shape of a Smooth Opaque Object from One View, MIT DARPA report, (1970)
879. Mutto, C.D., Zanuttigh, P., Cortelazzo, G.M.: Microsoft Kinect™ Range Camera. Springer, (2014)
880. Mojsilovic, A.: A method for color naming and description of color composition in images, ICIP, (2002)
881. van de Weijer, J., Schmid, C., Verbeek, J.: Learning color names from real world images. CVPR, (2007)
882. Khan, R., Van de Weijer, J., Shahbaz Khan, F., Muselet, D., Ducottet, C., Barat, C.: Discriminative Color Descriptors. CVPR, (2013)
883. van de Weijer, J., Schmid, C.: Coloring Local Feature Extraction. ECCV, (2006)
884. Sung-Hyauk Cha.: Comprehensive Survey on Distance/Similarity Measures between Probability Density Functions, IJMMMAS, (see also Duda [826])
885. Deza, E., Deza, M.M.: Dictionary of Distances, Elsevier, (2006)
886. Glasner, D., Bagon, S., Irani, M.: Super-Resolution From a Single Image. ICCV, (2009)
887. Vedaldi, V., Varma, G.M., Zisserman, A.: Multiple Kernels for Object Detection A. (2009)
888. Vondrick, C., Khosla, A., Malisiewicz, T., Torralba, A.: HOGgles: Visualizing Object Detection Features. ICCV, (2013)
889. Huang, Y., Nat. Lab. of Pattern Recognition (NLPR); Inst. of Autom.; Beijing, China; Wu, Z., Wang, L., Tan, T., PAMI.: Feature Coding in Image Classification: A Comprehensive Study, (2014)
890. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Networks 2(5), 359–366 (1989)
891. Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., Li, F.-F.: Imagenet: a large-scale hierarchical image database. In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, pp. 248–255. IEEE, 2009
892. Targ, S., Almeida, D., Lyman K.: Resnet in Resnet: generalizing residual architectures, arXiv: 1603.08029. (2016)
893. Szegedy, C., Ioffe, S., Vanhoucke, V.: Inception-v4, Inception-ResNet and the impact of residual connections on learning. arXiv: 1602.07261, (2016)

Index

A

Acceleration methods, 277, 304, 313
Accuracy and trackability, 143–144
Accuracy optimizations, 145
Activation function, 323–325, 343, 349, 353, 363, 367, 368, 386, 388, 390, 392, 395, 399–402, 405, 408, 412, 415, 420, 425, 426, 429, 430, 434–436, 487, 504
Adaptive detector tuning method, 130
AdjusterAdapter class, 519
ADL activity recognition dataset, 549
Adopt-And-Modify dataset, 252
Adopt data set, 252
Advanced instruction set items, 311
Affine invariant, 80, 150, 189, 193, 256, 438–440, 497, 543
AlexNet, 411, 419–421, 423, 463, 508
Algorithm fitting, 254
Algorithm performance, 250
ALU, 59, 178, 277–278, 280–282, 306, 307, 309
Analytics, 196, 249, 251, 284, 321, 333, 345–346, 402, 407, 511, 512
Anchor point, 116, 146, 187
ANN *See* Artificial Neural Network (ANN)
Approximated radial gradient transform (ARGT), 229, 230
Arbitrary periodic functions, 59
Area operations, 44, 71, 274, 288, 295, 299, 305–309, 313, 314
Array camera, 9, 11, 14, 18–19
Artifact filtering, 54–55
Artificial intelligence, 322, 329, 333, 334, 345, 346, 377, 560
Artificial Neural Network (ANN), 164, 320–323, 325–327, 331, 332, 338–339, 346, 348–351, 353–360, 362, 366, 367, 371, 372, 375, 377–379, 408, 413, 419, 421, 424, 426, 431, 434, 437, 438, 442, 469–471, 512, 561
Artificial neuron, 323, 325, 327, 331, 332, 354, 361, 375, 388, 394, 397–399, 414, 415, 448, 469, 489
Augmented reality, 28, 168, 229, 273, 282, 299–300, 302, 303, 315
Auto-correlation, 79, 83–84, 218
Autoencoder (AE), 326, 336

Automobile identification application, 283

Automobile recognition, 161, 176, 260, 282–284, 315

B

Back propagation, 321, 323, 325–327, 335–337, 342, 355, 360, 364, 367, 371, 375, 379, 383, 387, 388, 390–392, 396, 400, 402–408, 412, 414, 415, 417, 420, 429, 436, 440, 441, 443–447, 450, 452, 453, 471, 510
Back-side-illuminated (BSI) sensor, 4
Bagging, 356, 365
Bag of visual words, 325
Bag of words, 161, 242, 470, 472, 473
Bag of words methods, 161, 472–474
Baidu Deep Image, 422–424, 437
Basis features, 58, 109, 155, 161, 163, 168, 325, 326, 328, 329, 339, 343, 344, 366, 389, 469–472, 474, 476, 477, 479, 486, 491, 499
Basis function(s), 8, 58, 59, 96, 104, 109–111, 139, 163, 168, 174, 176, 219, 220, 233–235, 325, 329, 337, 339, 344, 356–359, 366, 368, 372, 376, 429, 468–471, 479, 496, 499, 580
Basis Function Network (BFN), 339, 356, 358, 359, 366, 372, 376, 377, 438, 468–471, 486
Basis set, 8, 52, 108–110, 163, 205, 219, 233, 235, 325, 357, 366, 469, 471, 472, 476, 477, 479, 485, 492, 581
Basis space
 descriptors, 36, 72, 167, 168, 174, 175, 185, 186, 197, 233
 metrics, 104–113
BFN *See* Basis Function Network (BFN)
Bias input neuon, 164
Big data, 319, 345–346
Binary
 mask, 269
 morphology, 62–63, 196
Binary/Boolean distance, 125, 182
Binary histogram intersection minimization (BHIM), 160
Bin mean density, 86, 91, 93, 565, 568, 573
Binning method, 212, 214, 215
Binocular stereo, 12–13
Black point value, 47
Blur invariance, 232, 233, 266

- Boosting, 66, 77, 138, 156, 159–160, 197, 198, 203, 408, 463–466, 469, 486, 505, 506, 508
- Bounding box, 96, 174, 182, 185, 234, 239, 262, 266, 285, 286, 288–290, 292–293, 295, 296, 549, 551
- Boxlet method, 313
- Bray Curtis distance, 125
- BRIEF, 39, 126, 131, 134, 137, 141, 142, 144, 155, 180, 181, 183, 206, 208, 279
- BRISK
- detector, 522, 523, 528, 537
 - method, 39, 207
 - patterns, 136–137
- C**
- Cache, 138, 181, 276, 278–281, 304–307, 311, 347, 409
- Camera calibration, 12, 13, 250, 286
- Canberra distance, 125, 174, 182
- Candidate edge interest point filters, 189
- Canny method, 57, 294
- Capturing depth information, 11
- CARD *See* Compact and Realtime Descriptor (CARD)
- Cartesian
- coordinates, 20, 22, 65, 85, 101, 122, 127, 152, 212, 214, 227, 233, 240, 293, 344, 459
 - models, 22
 - vs. polar coordinates, 24
- CCCP *See* Cross channel parametric pooling (CCCP)
- CCH *See* Chain code histogram (CCH)
- CCP *See* Cross channel pooling (CCP)
- Cell placement aspect ratio, 4
- CenSurE and STAR, 216–217, 219
- Census transform, 39, 181, 205
- Center Surround Extrema (CenSurE) method, 179, 216
- Center symmetric LBP (CS-LBP), 203, 214–215
- Centroid, 26, 36, 64, 65, 72, 79, 86, 88, 96, 103, 132, 135, 158, 168, 176, 177, 184, 185, 194, 206, 234, 237–239, 285, 286, 288, 297, 474, 475, 478, 479, 565, 570
- Chain code histogram (CCH), 104, 177, 230–231
- Charge-coupled device (CCD) cells, 1
- Chebyshev distance, 124
- Classified invariant local feature approach, 77
- Classifier, 68, 77, 138, 170, 197, 248, 287, 325, 548, 577,
- Clustering methods, 151, 157, 470, 472–474, 479
- CNN *See* Convolutional Neural Network (CNN)
- Coarse-grain parallelism, 307
- Codebook, 37, 77, 155, 160–162, 168, 173, 174, 176, 235, 291, 325, 328, 469, 470, 472–482, 485, 490–494
- Codebook learning, 469, 475, 490, 492
- Coding methods/feature descriptors, 105
- Color accuracy and precision, 50
- Color corrections, 6, 7, 38, 49, 223, 308
- Color criteria, 171
- Color enhancements, 49–50
- Colorimetry, 5, 38, 45–50, 177, 204, 258, 284, 286, 289, 595
- Color management systems, 46, 47
- Color morphology, 63, 72
- Color segmentation, 61, 66, 297, 299, 300
- Color spaces and perception, 45–48, 63, 128, 171, 223, 279, 423, 558
- Columbia-Utrecht Reflectance and Texture Database, 550
- Commercial products, 330, 342, 346, 350, 555
- Common data-parallel language, 295, 309, 365
- Compact and Realtime Descriptor (CARD), 176, 226–228
- method, 226
- Compressed HOG (CHOG) method, 229
- Computational imaging methods, 7, 9
- Compute-centric vs. data-centric approaches, 307
- Compute complexity SIFT, 174, 213
- Computer system elements, 276
- Computer vision, 1, 35, 75, 120, 168, 187, 247, 273, 319, 377, 549, 555, 582
- Connectome image, 319, 351
- Constraints, 120, 125, 145, 151–160, 165, 301, 302, 455, 475, 495
- Containment measure, 91
- Contour/edge histograms, 104
- Conventional camera system, 7
- Convnet (Convolutional Neural Network), 325, 387–390
- Convolutional filtering and detection, 50–52
- Convolutional Kernel Network (CKN), 489
- Convolutional Neural Network (CNN), 163, 164, 323, 325–329, 331–344, 351, 353–355, 358–363, 366–368, 370, 377, 386–392, 394–402, 404, 407, 408, 412–414, 417–422, 424–426, 429–435, 437–445, 447, 450, 451, 453, 457, 460–466, 469–473, 485, 486, 489, 490, 492, 494, 495, 499, 501, 505–510, 561, 577, 578, 595
- Convolutional-RNN (C-RNN), 458, 460–461, 507
- Co-occurrence matrices *See* Extended SDM (Krig Metrics)
- Coordinate and complex spaces, 126
- Corner, 28, 50, 76, 116, 169, 188, 247, 279, 400, 516, 593
- Corner detector, 28, 163, 169, 181, 191–193, 227, 251, 264, 268, 516
- Corrected coverage, 89, 93, 565, 568, 571
- Correlation
- distance, 123, 174, 182, 587
 - templates, 25, 30, 121, 131, 138, 140, 155, 161, 172, 217–218, 291, 294, 325, 328, 329, 332, 333, 336, 338–340, 342–344, 354, 356, 360, 363, 366, 388, 399, 413, 461, 511
- Correspondence, 14, 22, 25–26, 99–101, 120, 138, 146, 150, 154, 169, 181, 182, 206, 215, 216, 224, 232, 236, 237, 240, 245, 248, 250, 259, 261, 286, 289, 294–296, 298, 300–302, 317, 371, 377, 386, 426, 430, 474
- Cosine distance/similarity, 123
- Cosine modulated Gaussian (CMG), 214
- Cost function, 326
- Covariant, 169, 189, 250
- CPUs, 159, 178, 273–278, 280, 282, 283, 287–289, 295, 296, 299, 300, 303–312, 315, 337, 342, 345, 347, 348, 441, 456, 475, 553, 580, 590
- CPU threads, 275, 288, 289, 295, 296, 299, 300, 303, 304
- CRCV, 553

- C-RNN *See* Convolutional-RNN (C-RNN)
Cross channel parametric pooling (CCCP), 324, 343, 366, 427–429
Cross channel pooling (CCP), 324, 427, 428, 430
Cross-correlation, 79, 81, 83–84, 98, 143, 145, 218
CSIFT, 209
CS-LBP retrofit, 214–215
Curless and Levoy method, 27
Cylindrical distortion, 23
- D**
DAG-RNN, 448, 449, 457, 458
Daisy, 179, 181, 225–226
dasNet *See* Deep Attention Selective Network (dasNet)
Data
 composition, 248, 256
 construction, 154
 conversions, 44, 45
 fitting, 254
 labeling, 249
 structures, 26, 31, 68, 95, 152, 157, 178, 179, 185, 281, 307, 484
 tiling, 299, 304, 306, 310
 transfers, 273, 305
 types, 173, 174, 178, 184, 185, 197, 262, 274–276, 288, 296, 300, 303, 308, 309, 311, 314
Data access pattern taxonomy, 312
Dataset creation, 170, 248, 473
DBN *See* Deep Belief Network (DBN)
DCT *See* Discrete cosine transform (DCT)
Dead pixel correction, 6, 38
Decision Trees, 486
Deep Attention Selective Network (dasNet), 408, 463–465, 507, 508
Deep Belief Network (DBN), 161, 326, 338
Deep Learning (DL), 25, 131, 155, 160, 165, 217, 320–322, 325, 327, 329–334, 336, 338, 339, 341–342, 344–347, 371, 372, 375–512, 561
Deep learning approach, 25, 320, 322
Deep Neural Network (DNN), 273, 325, 326, 330–333, 338, 341–344, 355, 356, 360, 364, 392, 395, 397, 402, 416, 417, 422, 423, 426, 429, 437, 438, 469, 486, 488, 508–512, 581, 582
Deep Residual LEarning (DRL), 397, 422, 432, 509
Deformable patches, 131
Degrees-of-freedom (DOF), 27, 237
De-mosaicking, 6
Dense
 search, 146, 181
 tracking, 29
Dense depth methods, 22
Dense depth-sensing methods, 21
Depth
 fusion, 23
 granularity, 24, 25
 segmentation, 61, 65–66, 72, 286, 297–299
 sensing, 5, 7, 11, 12, 14, 18, 21, 22, 24, 25, 27–31, 66, 77, 127, 146, 150, 173, 241, 296, 297
Depth/Z limited search, 151
- Descriptor
 concatenation, 201
 density, 129
 memory, 173, 174, 179, 184, 185, 262
 representation, 126–128, 479
 shape topology, 130–134
Descriptor-NETS (D-NETS), 131, 133, 135, 137, 146, 180, 231, 232, 328
Detector, 8, 50, 76, 115, 168, 187, 247, 279, 328, 388, 515
Determinant of Hessian (DoH) method, 191, 193, 257
Device color models, 47–48
Dictionary, 42, 161, 174, 176, 317, 325, 327, 329, 450, 472–475, 479, 482, 489–491, 499, 503–505
Dictionary learning, 327, 329, 491–494
Difference of Gaussians (DoG), 103, 132, 143, 148, 163, 174, 177, 183, 191, 193, 209–211, 213, 214, 216, 262, 279, 546
Difference of Hexagons (DoH), 216
Diffraction gratings, 8, 11, 16–17, 109
Digital mirror device (DMD), 8
Digital signal processor (DSP), 259, 274–277, 280, 281, 288, 296, 299, 300, 303, 356, 419
Digital terrain mapping, 12
Directionality, 79, 199, 200, 353
Direct memory access (DMA), 275–277, 306, 311, 312, 316
Discrete cosine transform (DCT), 60, 163, 176, 328, 477, 492, 494
Discrete sine transform (DST), 60
Discrimination and uniqueness, 170, 173, 186
Discriminative query expansion (DQE) method, 216
Distance function, 26, 27, 65, 98, 103, 115, 118, 120–126, 157, 158, 160, 162, 164, 169, 170, 174, 182, 184, 185, 201, 205–208, 213, 217, 218, 220, 222, 224–226, 228–230, 232, 233, 262, 313, 317, 326, 371, 447, 451, 472, 473, 475, 478, 479, 505
Distinctive *vs.* indistinctive feature, 120
DMA *See* Direct memory access (DMA)
DMA unit, 277, 306, 311
D-NETS *See* Descriptor-NETS (D-NETS)
DNN *See* Deep Neural Network (DNN)
DoG *See* Difference of Gaussians (DoG)
Drawkeypoints () function, 521, 523
DRL *See* Deep Residual LEarning (DRL)
Dropout, 327, 338, 356, 361, 363–365, 368, 370, 392, 393, 396, 415, 419, 421, 424, 426, 428–431, 434, 437, 438, 442, 487
DSP *See* Digital signal processor (DSP)
D-token, 133, 231
DynamicAdaptedFeatureDetector class, 188, 519
Dynamic range and noise, 2, 5, 7, 9, 75, 78, 171, 349, 583–585, 595, 601, 602
Dynamic textures, 81, 95, 132, 172, 179, 201, 553
- E**
Early stopping, 326
Earth movers distance (EMD), 125, 174, 182
Earth movers distance (EMD)/Wasserstein metrics, 125

- Edge
 contrast, 82
 density, 82
 detectors, 56–57, 93, 94, 168, 178, 225, 294, 328, 344, 366, 413, 417
 directivity, 83
 entropy, 82
 linearity, 83
 metrics, 81–83
 periodicity, 83
 size, 83
- Edge-based region methods (EBR), 189, 244
- Edge primitive length measures, 83
- EDM *See* Euclidean distance maps (EDM)
- Efficiency variables, 173, 183
- Efficient match kernels (EMK), 488
- Efficient Second Order Minimization (ESM) method, 29, 30
- Eigen property, 191
- EMD *See* Earth movers distance (EMD)
- EMK *See* Efficient match kernels (EMK)
- Ensemble(s), 343, 359, 371, 375, 433, 469, 506–508, 510, 511
 methods, 343, 359, 469, 506–508
 networks, 359, 507, 508, 511
- Error minimization, 156, 157, 326, 331, 335, 337, 403
- Euclidean/Cartesian distance, 122, 182
- Euclidean distance maps (EDM), 63–64
- Expert systems, 330–331, 333, 371
- Exploding gradients, 337
- Extended SDM (Krig Metrics), 81, 86–88, 93, 563–575, 592
- F**
- Face emotion and age recognition, 289–290
- Faces in the wilds, 317, 551
- Facial feature ratios, 291
- Facial landmarks location, 293
- FAST
 detector, 163, 195, 207, 529, 538
 method, 109, 194, 228
- Fast Fourier transform (FFT), 20, 36, 37, 44, 45, 58–61, 72, 76, 78, 84, 85, 94, 95, 106, 107, 145, 185, 198, 234, 238, 239, 263, 411
- Fast Hartley transform, 60
- Fast wavelet transform (FWT), 111
- FC layer *See* Fully Connected Layer (FC layer)
- Feature
 density, 168, 173, 174, 180, 184, 185, 201, 205–208, 213, 217, 218, 220, 222, 224–226, 228–230, 232, 233, 262
 description, 14, 25, 26, 32, 35–39, 41, 45, 58, 61, 75, 77, 81, 85, 94, 97, 99, 102, 105–107, 109–111, 114, 116–120, 126, 130, 138, 140, 145, 149, 163, 167–185, 187, 208, 213, 217, 228, 232, 234, 241, 244, 273, 279, 282, 286, 294, 298, 299, 302, 313, 315, 319, 322, 339, 371, 482, 518, 556
 detection, 35, 81, 101, 110, 129, 163, 171, 191, 199, 219, 220, 224, 227, 255, 262, 265, 294, 336, 390, 519, 521, 524–532, 534–541, 543, 580, 587, 590
 generation, 321–322, 438, 441
 initialization, 357, 366, 401, 419, 421, 424, 426, 431, 434, 437, 438, 442
 map, 324, 325, 333, 340, 344, 353, 355, 356, 360, 366, 368, 370, 389, 390, 395, 399, 401, 413–418, 421, 424–430, 432, 433, 435, 440, 461–464, 498, 499, 501–505
 pattern, 155, 173, 174, 179–180, 184, 185, 201, 205–208, 213, 217, 218, 220, 222, 224–226, 228–230, 232, 233, 262, 385
 pyramids, 126, 129, 149–150
 search methods, 173, 174, 180, 262
 set, 105, 118, 120, 152, 155, 162, 163, 220, 244, 254, 280, 291, 293, 301, 323, 325, 328, 332, 333, 335, 338–340, 343, 344, 357, 358, 365, 366, 388, 419–421, 424, 426, 431, 434, 437–439, 442, 460, 472, 473, 476, 481, 485, 489, 494, 495, 505, 507, 580, 595
 shapes, 37, 62, 121, 131, 139, 173, 174, 179, 184, 185, 201, 205–208, 213, 217, 218, 220, 222, 224–226, 228–230, 232, 233, 239, 259, 262, 388, 593, 594
 space, 121, 128, 149, 151–153, 156, 157, 159, 163, 165, 172, 203–205, 219, 281, 282, 370, 407, 431, 438–440, 471, 476, 477, 481–483, 485, 492, 595, 600, 602
- Feature descriptors, 14, 35, 76, 115, 167, 187, 247, 279, 320, 375, 515, 549, 578
- Feature learning (FL), 77, 115, 155, 160, 161, 164, 165, 176, 235, 319–372, 375–512
- Feature learning terminology, 319–372
- Feature metric evaluation (FME), 174, 182–185, 197
- Feed-Forward Neural Network (FNN), 323, 326, 331, 332, 335, 336, 338, 339, 352, 356, 358, 372, 376–378, 383, 385, 443–450, 462, 463, 465, 466, 469, 509
- FERNS, 143, 158, 160, 231, 486
- FERNS training, 160
- FFT *See* Fast Fourier transform (FFT)
- FFT spectrum, 84, 85, 94, 95, 106, 185, 198
- Filter, 2, 37, 76, 116, 172, 188, 257, 279, 322, 378, 518
- Fine-grain data parallelism, 308
- Finer-grain and metric composition approaches, 77
- Fisher Vectors, 474, 475
- FLAIR, 475
- FME *See* Feature metric evaluation (FME)
- FNN *See* Feed-Forward Neural Network (FNN)
- Fourier
 description, 107
 descriptors, 37, 107, 135, 157, 180, 234, 237–239, 282, 285, 286, 288, 289
 methods, 25, 41, 50, 81, 85, 110, 112, 145, 176, 234
 transforms, 37, 58–60, 75, 78, 83, 84, 96, 104, 107–109, 111, 177, 178, 219, 231–234, 383, 411
- Foveon method, 2, 4
- Fractal methods, 80
- FREAK, 36, 39, 72, 78, 117, 126, 130, 134–136, 141, 142, 144, 145, 155, 160, 164, 168, 176, 177, 179–181, 183, 208, 261, 262, 279, 281, 282, 290, 294–296, 313, 320, 327–329, 338–342, 354, 359, 361, 363, 379, 380, 473, 481, 490, 494

- FREAK retinal pattern, 135–136
Fully Connected Layer (FC layer), 326, 327, 354, 361, 365, 366, 370, 371, 390–396, 399, 416, 418, 421, 424–427, 429, 430, 433, 435, 437, 438, 441, 442, 461, 469, 471–473, 485, 486, 506, 510, 511
- FWT *See* Fast wavelet transform (FWT)
- G**
- Gabor functions, 112, 219, 325, 328, 329, 358, 359, 469, 471, 472, 487, 496, 497, 504
Gamma-curve correction method, 5
Gamut mapping, 46, 48, 49
GAP *See* Global average pooling (GAP)
Gauge coordinates, 128, 222
Gaussian filter, 37, 56, 79, 103, 109, 110, 145, 147, 148, 189, 193, 204, 210, 211, 214, 223, 225, 364, 499, 521, 522, 532, 533
Gaussian weighting method, 145, 212, 221, 222
General vision taxonomy, 115, 197, 247, 261, 262
Geometric
 calibration, 13
 corrections, 6, 7, 23, 38, 274, 277, 285, 288
 discrimination factors, 140
 distortion, 2, 22, 128, 170, 172, 186, 212, 259, 260, 342, 433
 modeling, 22
GFFT detector, 526, 536
Global
 auto thresholding, 69
 histograms, 40, 68, 98–99
 thresholding, 67–69, 196
Global Average Pooling (GAP), 357, 370, 392, 427, 429–431, 463, 465
Global uniform texture metrics, 79
GLOH *See* Gradient Location and Orientation Histogram (GLOH)
GMDH *See* Group Method for Data Handling (GMDH)
Good feature descriptors, 118
GoogLeNet, 429, 431
GPGPU, 29, 159, 178, 274, 277, 304, 305, 310, 311, 314, 337, 347, 441, 556, 558
GPUs, 7, 29, 31, 38, 112, 138, 159, 178, 273–278, 280, 282, 283, 287, 288, 295, 296, 299, 300, 303, 304, 307–311, 314, 315, 337, 342, 347, 348, 356, 357, 369, 370, 420, 421, 437, 441, 475, 553, 561
Gradient
 descent, 323, 326, 327, 335–337, 344, 355, 364, 367, 375, 400, 402, 403, 407, 408, 420, 440, 441, 445, 452, 453, 455, 467, 471, 510
 direction, 41, 56, 57, 82, 83, 99, 174–176, 190, 204, 209, 212, 221, 227, 229, 241, 242, 262
 magnitude, 41, 56, 57, 76, 82–84, 99, 113, 150, 174–176, 183, 190, 195, 209, 212, 213, 223, 225, 226, 228, 232, 234, 241–244, 262, 292, 370, 488, 489
Gradient-ascent-based super-pixel methods, 65
Gradient Location and Orientation Histogram (GLOH), 127, 143, 176, 209, 212, 214, 225–227, 233
Graph-based super-pixel methods, 64–65
- Gray level co-occurrence matrices (GLCM) *See* Extended SDM (Krig Metrics)
Gray scale morphology, 63, 239
GridAdaptedFeatureDetector class, 519
Grid distance
 family, 182
 metrics, 124
Grid search method, 146
Ground truth
 correspondence, 250
 data, 13, 32, 40, 81, 119, 143, 152, 155, 156, 161, 163, 167, 170, 171, 176, 181, 185, 214, 247–270, 286, 290, 291, 297–301, 317, 323, 328, 336, 339, 340, 425, 516, 519, 547–554
Group Method for Data Handling (GMDH), 335, 364, 486, 487
- H**
- HAAR-like features, 37, 40, 172, 177, 216, 219–221
HAAR transforms, 108
Half-CNN, 392, 425, 426, 435
Hamming distance, 16, 78, 101, 121, 126, 157, 168, 174, 182, 184, 197, 201, 205–208, 227, 229, 262, 313, 327, 392, 475
Hamming Encoding, 475, 486
Haralick texture metrics, 87
HARRIS detector, 76, 129, 189, 530, 539
Harris–Hessian–affine method, 191, 193
Harris–Hessian–Laplace method, 191, 193
Harris–Laplace method, 191
Harris–Stephens corner detector, 192
Hellinger distance, 124, 182, 213, 215
Hessian-affine corner detector, 193
Hessian–Laplace method, 193
Hessian matrix method, 193
Hidden layers, 325, 326, 332, 333, 353, 354, 360, 379, 383, 395, 418, 444, 451, 460, 511
Hidden unit, 325, 326, 332, 415, 418, 419, 448, 458
Hierarchical feature learning, 340, 468
Hierarchical Kernel Descriptors (HKD), 469, 485, 488–490
Hierarchical Learning (HL), 323, 325, 341
Hierarchical Matching Pursuit (HMP), 339, 357, 370, 434, 469, 477, 481, 485, 489–495
 method, 485, 490
Hierarchical Model Optimization (HMO), 320, 331, 506, 508
High dynamic range (HDR) cameras, 7, 9
High frame rate (HF) cameras, 9
Histogram Encoding, 474, 475
Histogram equalization, 38, 40, 41, 49, 68, 70, 71, 119, 196, 365, 368, 401, 421, 584
Histogram of Gradients (HoG), 209, 211, 221, 223–224, 328
Histograms of Sparse Codes (HSC), 162, 235, 328, 490
Histogram specification, 69
Historical survey of features, 75–81
HKD *See* Hierarchical Kernel Descriptors (HKD)

- HMAX, 320, 322, 331, 332, 337, 351, 357, 359, 366, 370, 389, 408, 438, 467, 469, 471, 495, 496, 498–506, 508, 580, 582
- HMO *See* Hierarchical Model Optimization (HMO)
- HMP *See* Hierarchical Matching Pursuit (HMP)
- HoG *See* Histogram of Gradients (HoG)
- HOG descriptor, 141, 162, 183, 212, 223, 229, 241, 242, 254, 489
- Hole filling, 14
- Holes and occlusion, 26
- HON4D, 177
- Horopter region, 23, 582
- Hotelling transforms, 110
- Hough and Radon transforms, 112
- Human expectations *vs.* machine vision, 264
- Hybrid networks, 469
- Hybrid synthetic interest points, 264
- Hyperparameters, 326, 409, 414, 416, 417, 419, 421, 433
- Hypothetical assignment, 274, 275
- Hysteresis thresholds, 57, 67
- I**
- IBR *See* Intensity-based region methods (IBR)
- I-LIDS, 552
- Illuminants, 45–47, 558
- Illumination attributes, 171
- Image
- classification, 163, 223–225, 273, 282, 296–298, 315, 443, 475
 - coding, 61, 105–109, 111, 235
 - moments, 69, 76, 96, 133, 168, 176, 185, 196, 235, 237
 - processing, 2, 8, 31, 35–36, 38, 43–45, 49, 58, 60, 67, 69, 72, 75, 76, 80, 97, 105, 106, 111, 112, 114, 192, 197, 198, 234, 236, 239, 275, 312, 314, 320, 322, 356, 365, 399, 412, 413, 441, 546, 555–559, 584, 595
 - processing operator, 80, 197, 198, 412, 546
 - pyramids, 28, 29, 37, 101–103, 107, 110, 117, 118, 129, 130, 146–149, 172, 174, 180, 181, 184, 206, 210, 214, 219, 227–230, 241, 242, 262, 293, 340, 401, 519, 521–523
 - reconstruction, 140–143, 164, 183, 248, 255, 468, 474, 476, 481
 - sensor technology, 1–7, 31
- ImageMultiply, 269
- Inception, 356, 366, 371, 392, 396, 397, 422, 429, 431–433, 440, 507, 509
- InceptionNet, 422, 428, 432–434, 437
- Input unit, 325, 326
- Integral image(s)
- and box filters, 55, 56
 - contrast filters, 70
- Integral volume method, 242
- Intensity-based region methods (IBR), 189
- Interest point(s), 22, 35, 76, 115, 168, 187, 247, 279, 338, 440, 516, 553
- Interest point detector, 76, 115, 117, 118, 120, 130, 163, 164, 169, 187–244, 247, 250, 252, 256, 261, 263, 264, 266, 279, 281, 294, 295, 521–523, 542
- Invariance, 10, 37, 77, 115, 167, 187, 254, 293, 324, 383, 516, 579
- Invariance attributes, 80, 107, 116, 117, 119, 138, 139, 143, 154, 167, 173, 179, 183, 187, 197, 255, 258, 261, 340, 343, 344
- Invariant texture metrics, 80
- I/O performance, 282
- J**
- Jaccard similarity and dissimilarity, 126
- K**
- Karhunen–Loeve transform (KLT), 41, 109, 110, 229
- Kernel
- descriptors, 159, 469, 485, 488, 489
 - machines, 121, 156–159, 165, 371, 382, 469, 471, 482–484, 488
 - sets, 56
 - trick, 485
- Kernel Codebook Encoding, 474
- Kernel Connected Layers, 327, 354, 392
- Kernel filtering and shape selection, 52–53
- Keypoints, 116, 119, 145, 149, 160, 169, 170, 181, 187–189, 206, 207, 209–211, 213, 227, 228, 231, 242, 244, 269, 516, 518, 519, 521–523, 534
- Kinect Fusion approach, 27
- KITTI benchmark suite, 548
- KLT *See* Karhunen–Loeve transform (KLT)
- K-Means, 157, 158, 160, 163, 215, 225, 244, 286, 317, 474–479, 489, 490, 492
- K-means method, 158, 160, 475
- Krig Metrics (Extended SDM), 81, 86–88, 93, 563–575, 592
- K-SVD, 328, 475, 477, 479, 490–492, 494
- L**
- Labeled data, 254, 326, 347, 366, 421, 438
- Landmark detection steps, 294
- Laplacian derivative, 189
- Laplacian of Gaussian (LoG), 191–193, 210
- Laser-stripe pattern methods, 15
- Latent variables, 326
- Laws metrics, 93
- Layers, 30, 161, 280, 323, 375, 517, 577
- LBL,
- LBP *See* Local binary pattern (LBP)
- Learned
- convolution masks, 163
 - features, 155, 264, 321, 325, 329, 332, 339, 343, 413, 419, 425
- Learning
- hyperparameters, 414, 416, 419
 - parameters, 330, 342, 391, 403, 407, 408, 414, 416, 507
- LeNet, 336, 338, 383, 388, 390, 397, 417–419
- Lens and sensor configurations, 12
- Levenberg–Marquardt method, 28, 158, 403
- LGP *See* Local gradient pattern (LGP)
- Linear binary pattern, 121, 176

- Linearity
strength, 86, 92, 93, 565, 568, 573, 574
variation, 91, 92
- Linear ramp function, 68
- Line operations, 44
- L2 norm, 123, 174, 182, 225
- Local auto threshold methods, 71
- Local binary descriptors, 36, 39, 53, 72, 121, 126, 130, 131, 134–146, 164, 167, 168, 174, 175, 177, 181–183, 185, 194, 197, 206, 208, 209, 222, 262, 280, 313, 361, 380
- Local binary pattern (LBP)
histogram, 37, 95, 176, 198, 200, 234
kernels, 198
variants, 197, 203, 241
- Local curvature methods, 195–196
- Local feature approaches, 77
- Local gradient pattern (LGP), 232
- Local histogram equalization, 38, 41, 68, 70, 401, 421
- Localized guided-filter method, 29
- Local phase quantization (LPQ), 41, 232–233, 235
- Local region histograms, 99, 230, 485
- Local texture, 38, 65, 86, 94, 108, 134, 176, 197, 435
- Local thresholding, 39, 69–71
- Local winner-take all (LWTA), 357, 369, 370, 387, 388
- Locus length measures, 90
- Locus mean density, 86, 88, 90, 93, 565, 568, 572
- LoG *See* Laplacian of Gaussian (LoG)
- Long-Term Short-Term Memory (LSTM), 337, 362, 403, 443, 448, 449, 451–458
- Lookup tables (LUTs), 67, 68, 97, 99, 227
- Low-frequency coverage, 86, 89
- Low-pass filter shape, 60
- LPB-TOP, 202
- LPQ *See* Local phase quantization (LPQ)
- LSTM *See* Long-Term Short-Term Memory (LSTM)
- Lucas Kanade (LK) method, 29, 30, 440
- LUTs *See* Lookup tables (LUTs)
- LWTA *See* Local winner-take all (LWTA)
- LWTA pooling, 357, 369, 370
- M**
- Machine learning (ML), 77, 115, 151–154, 156–159, 163, 248, 249, 282, 320–323, 329–339, 342, 347, 355, 371, 407, 450, 451, 469, 482, 486, 548, 557, 561
- Mahalanobis distance, 125, 174, 182
- Major initiatives in neural computing, 346
- Manhattan distance, 123–125, 174, 182, 478
- Manual and compiler methods, 312
- Manually designed feature detectors, 154
- Mapping (DTAM), 29
- Markov random field methods, 80
- Matching cost and correspondence, 120
- Maximally Stable Extremal Regions (MSER)
detector, 236, 527, 542
method, 236–237
- Maxout, 357, 368, 426–431, 463, 464
- MD-LSTM, 457
- MD-RNN *See* Multidimensional RNN (MD-RNN)
- Measure (GCM), 250
- Medical imaging, 12, 20, 319, 559
- Memory
bandwidth, 59, 75, 274, 277, 280, 281, 487
issues, 279, 281, 304
optimizations, 287, 304–305, 311
size, 118, 173, 179, 280, 281, 347, 455
system, 274, 275, 278, 279, 306, 307, 335, 447, 450, 451
- Middlebury dataset, 249, 255, 257
- Mikolajczyk and Schmidt methodology, 249
- MINWA, 437–438, 508
- MIT Flickr material surface category dataset, 551
- MIT indoor scenes scene classification, 550
- MLP *See* Multilayer Perceptron (MLP)
- Model compression, 508, 510, 511
- Modified census transform (MCT), 205
- Modified discrete cosine transform (MDCT), 60
- Modified upright SURF (MU-SURF), 217, 222
- Monocular depth processing, 27–28
- Monocular depth sensing, 21, 24, 27–31, 33, 146, 150
- MOPED, 157, 159
- Moravec corner detector, 191, 192
- Morphological method, 63, 179, 196, 237
- Morphology optimizations and refinements, 63
- Mosaic method, 4
- MP-LSH *See* Multi-probe locally sensitive hashing (MP-LSH)
- MSRA, 397, 422, 425, 434–437
- MSSNN, 434–437
- Multidimensional descriptor, 117
- Multidimensional RNN (MD-RNN), 443, 448, 449, 454, 457–461
- Multigeometry descriptor, 117
- Multi Layer Perceptron (MLP), 335, 366, 368, 379, 383, 387, 391, 426–432, 434, 449, 450, 462, 463, 510
- Multilayer pooling, 324
- Multi-modal feature metrics fusion, 77–78
- Multimodal sensor data, 128
- Multi-patch sets, 132
- Multi-probe locally sensitive hashing (MP-LSH), 206
- Multi-resolution histograms, 102, 103, 150, 175, 224
- Multi-scale pyramid search, 147
- Multivariate descriptor, 45, 78, 117, 121, 128, 138, 152, 196–198, 304, 482, 486, 489
- Multivariate spaces, 128
- Multi-view stereo (MVS), 1, 11, 12, 14, 18, 21–26, 28, 259, 260, 296, 297, 299, 300, 549
- MVS *See* Multi-view stereo (MVS)
- MVS depth map, 299, 300
- N**
- NAP *See* Neural Abstraction Pyramid (NAP)
- NCC *See* Normalized cross-correlation (NCC)
- Neighborhood comparison, 200
- Neocognitron, 336, 383, 385–388, 413, 466
- Network-in-network (NiN), 357, 358, 366, 368, 389, 392, 397, 424, 426–433, 440, 462, 463
- Neural Abstraction Pyramid (NAP), 408, 465–468

- Neural computing (NC), 323, 345, 346
 Neural function, 158, 164, 323, 326, 367, 398, 453, 486, 487
 Neural model, 164, 321, 323–326, 334, 378, 379, 393, 399, 401, 408, 426, 444, 449, 451, 452, 468–470, 487, 577, 578, 580, 582, 595
 Neural network (NN), 138, 320, 375, 561
 Neuron, 164, 323, 375, 561, 577
 Neutral axis, 47
NiN See Network-in-network (NiN)
 NIST, 80, 552, 559, 560
 Noise, 2, 37, 79, 118, 172, 189, 250, 284, 343, 403, 521, 591
 Noise-filtering methods, 141
 Non-Linearity, 5, 25, 323, 324, 340, 353, 363, 365, 368, 390, 396, 399–403, 411, 423, 429, 430, 436, 440, 441, 457, 546
 NORB D object recognition from shape, 551
 Normalized cross-correlation (NCC), 218
 NTM, 451, 454–456
 Numeric conditioning, 338, 342, 343, 354, 357, 359, 360, 363, 365, 368, 383, 395, 396, 400, 415, 419, 424, 426, 429, 431, 437, 438, 442
 Nyquist frequency, 1, 8, 476
- O**
 Object models, 77, 151–154, 156, 165, 253
 Object polygon shapes, 133
 Object shape metrics, 96, 237–239, 282
 Octave, 38, 147–149, 180, 206, 210, 211, 214, 216, 217, 224, 225, 227, 238, 242, 313, 517, 522, 533
 O-Daisy, 225, 226
 OpenCV test methodology, 516
 Open rating systems, 255–257, 270
 Operations and compute, 275, 287, 295
 Optical flow, 21, 22, 28, 30, 131, 548, 549, 551, 552
 ORB detector, 527, 541
 Oriented BRIEF (ORB), 36, 37, 39, 72, 78, 116, 126, 134, 135, 137, 144, 149, 155, 164, 168, 176, 177, 180, 181, 206–208, 222, 231, 232, 263, 264, 279, 308, 313, 328, 339, 341, 361, 363, 366, 380, 473, 490, 515–517, 519, 521–523, 527, 532, 533, 542, 543, 545
 Oscillating gradients, 337, 453
 Output unit, 325, 326
 Overlays and tracking, 302
 Oversampling, 1, 133
- P**
 Pairing metrics, 261
 Panum’s area, 23
 Parallel FFT line transforms, 59
 Parallelization, 59, 299, 327, 420
 Parallel Tracking and Mapping (PTAM), 28, 29, 146, 150, 181
 Parametric ReLu (PreLu), 422, 434–437
 Pattern recognition, 76, 96, 99, 122, 156, 235, 322, 333, 335, 486, 559, 560
 Pattern region size, 174, 181, 184, 185, 262
- Peaks and valleys histogram, 67–68
 Perceptron, 220, 334, 335, 366–368, 377–384, 387, 388, 391, 392, 408, 426, 427, 513
 Performance optimizations, 215, 226, 328
 PETs crowd sensing dataset challenge, 552
 PHOG See Pyramid Histogram of Oriented Gradients (PHOG)
 Photo mosaicking, 12
 Photonic mixer device (PMD), 18
 Pipeline operations and compute resources, 287, 295
 Pipeline stages and operations, 286, 294, 298, 302
 Plenoptic methods, 19
 PMD See Photonic mixer device (PMD)
 PNN See Polynomial Neural Network (PNN)
 Point
 filtering, 44, 53, 280
 metrics, 61, 96–98
 operation, 38, 44, 51, 54, 308
 Point-pair
 patterns, 39, 134–137, 176, 197, 206
 sampling method, 134
 Polar and log polar coordinates, 127
 Polygon shape descriptors, 36, 37, 72, 96, 117, 154, 167, 168, 174–176, 185–187, 191, 196, 197, 235–240, 250, 282
 Polynomial Neural Network (PNN), 335, 389, 469, 471, 486–487
 Pooling, 145, 322, 382, 580
 Pooling and subsampling, 324, 343, 389, 390, 395, 396, 418, 595
 Positive and negative training, 216, 248
 Post-processing, 19, 21, 23, 51, 223, 357, 361, 368, 388, 390, 395, 396, 415, 419, 421, 424, 426, 431, 434, 437, 438, 442, 595
 Power usage, 90, 278
 PreLu See Parametric ReLu (PreLu)
 Principal Curvature Descriptors (PCD), 241
 Probability density function (PDF), 193
 PTAM See Parallel Tracking and Mapping (PTAM)
 PyramidAdaptedFeatureDetector class, 188, 519
 Pyramid Histogram of Oriented Gradients (PHOG), 102, 117, 180, 224–225
- Q**
 Quad-Directional RNN (QDRNN), 460, 461, 507
 Quality, 1, 14, 35, 63, 116, 138, 183, 189, 247, 249, 250, 255, 256, 333, 343, 344, 430, 463, 517
- R**
 Radial
 cameras, 19
 coordinates system, 127
 histograms, 103, 104
 Radial gradient transform (RGT), 127, 229, 230
 Radius-based surface descriptors (RSD), 241
 Randomized trees method, 143, 160
 RBM See Restricted Boltzman Machine (RBM)
 RCL-RNN, 442
 R-CNN, 434–437, 509

- Real applications robustness criteria, 259–261
Receiver operating characteristic (ROC), 251
Receptive field, 326, 341, 342, 350, 353, 354, 375, 378, 379, 381–383, 385, 392, 401, 402, 410, 411, 422, 488, 495, 496, 502, 504, 578, 582, 584, 593
Rectangle patterns, 219
Rectification, 14, 26, 296, 323, 324, 357, 368, 411, 430 process, 14, 324
Recurrent Neural Network (RNN), 326, 329, 331, 332, 335–337, 339, 356, 358–360, 366, 372, 379, 381, 442–452, 454–456, 458–462, 469, 470, 507
Region-limited search, 150
Region segmentation, 61, 81, 117, 198, 286, 297
Regularization, 326, 356, 363–365, 396, 411, 430
Reinforcement learning, 326, 381, 443, 455, 463, 464
REIN method, 158
Relative and absolute position, 120
Relative power, 86, 89, 90, 93, 565, 568, 572
Repeatability, 118, 120, 264, 266, 518, 546
Residual Learning, 422, 509–510
Resolution and accuracy, 172
Resource(s), 12, 13, 29, 35, 45, 80, 96, 111, 159, 169, 173, 273–282, 287, 288, 295, 296, 299, 300, 303, 304, 308, 314, 315, 330, 339, 342, 402, 437, 443, 486, 495, 553, 555–561
Resource assignments, 282, 287, 288, 295, 299, 304, 315
Restricted Boltzman Machine (RBM), 326, 336, 338, 477
RGB-D object recognition dataset, 550
RGT *See* Radial gradient transform (RGT)
RL-NTM, 454–456
RNN *See* Recurrent Neural Network (RNN)
RNN-NTM, 454–456
Robust fast feature matching method, 228–229
Robustness attributes, 85, 117, 169, 171, 183–185, 197, 343 criteria, 6, 96, 118, 138, 143, 167, 170, 172, 185, 247, 258–261, 269, 284, 290, 296, 300, 582
RootSift method, 215
Rosin corner metrics, 251
Rosin’s method, 137, 206
Rotational invariance, 25, 37, 84, 94, 95, 109, 118, 127, 137, 139, 140, 144, 145, 175, 176, 192, 201, 204, 206, 216, 217, 221, 222, 226, 244, 261, 265–266, 409, 501, 516, 532–534, 543, 545
Rotation invariant fast features (RIFF) method, 175, 227, 229–230
- S**
- SAD *See* Sum of absolute differences (SAD)
Salient regions, 191, 193
Scale invariance, 80, 118, 129, 139, 170, 189, 192, 193, 195, 203, 207, 254, 255, 261, 264, 266, 293, 383, 423, 434, 435, 497, 501
Scale invariant detectors, 189 interest points, 210
Scale invariant feature detector with error resilience (SIFER) method, 214
Scale invariant feature transform (SIFT) descriptor, 128, 132, 142, 143, 155, 176, 177, 209, 213, 215, 225, 244, 298, 301, 303, 308, 325, 474, 475
detector, 531, 540 features, 25, 142, 154, 163, 224, 243, 248, 262, 297–302, 339, 340, 477, 483, 488, 495 vertex descriptor, 301, 302
Scale selection method, 193
Scale space and image pyramids, 147–149 pyramid creation, 209
Scatter diagram, 99–102
Scene and object modeling approaches, 77
SDM extended metrics, 563
Segmentation limited search, 150 method, 43, 61, 64, 79, 80, 134, 150, 181, 189, 198, 258, 292, 297, 298
Sensor materials, 2 processing, 1, 2, 4–6, 23, 85, 274, 277, 280, 317 stacking method, 4
Serial/algorithmic code, 45
SFM *See* Structure from motion (SFM)
Shader kernel languages and GPGPU, 310
Shape(s) context method, 239, 240 factors, 133, 139, 157, 160, 172, 176–178, 184–185, 285, 286, 593 features, 42, 96, 168, 224, 235, 285, 287, 297, 543, 581 and patterns discrimination, 139
Shape-based regions (SBR), 189
Shape selection/forming Kernels, 52–53
Shi-Tomasi and Kanade corner detector, 192
Shi-Tomasi method, 192–193, 264
SHOT *See* Signatures of Histogram Orientations (SHOT)
SIFT *See* Scale invariant feature transform (SIFT)
SIFT-PCA method, 213
Signatures of Histogram Orientations (SHOT), 241
Silicon-based image sensors, 2
SIMD *See* Single instruction multiple data (SIMD)
Similarity measure, 240, 321, 482
SIMPLE BLOB detector, 534
SIMT *See* Single instruction multiple threading (SIMT)
Simultaneous localization and mapping (SLAM), 21, 22, 27, 28, 30
Sine and cosine transform, 60 waves, 58, 107
Single and sub-patches, 131
Single instruction multiple data (SIMD), 5, 27, 29, 138, 218, 274, 275, 277, 278, 281, 283, 287–289, 296, 299, 300, 303, 304, 308–311, 314, 337, 347, 409, 411, 437, 441, 466, 556, 558
Single instruction multiple threading (SIMT), 29, 138, 275, 277, 278, 287, 288, 296, 299, 300, 303, 304, 308–310, 314, 337, 347, 409, 437, 466, 556, 558
Single-pixel cameras, 8

- Single program multiple data (SPMD), 275, 308, 309
SLAM *See* Simultaneous localization and mapping (SLAM)
 Slant transform, 41, 108–109
 SNAKES method, 239
 SNN *See* Spiking Neural Network (SNN)
 SOC *See* System-on-a-chip (SOC)
 SOC components, 276
 Softmax, 327, 357, 368, 370, 371, 392, 416, 423, 424, 430, 433, 438, 441–443, 457, 461, 463, 464, 471, 472, 510
 Sparse coding, 41, 155, 161–162, 235, 244, 325, 327, 328, 339, 375, 469, 472, 473, 475–477, 479, 481, 482, 486, 490, 493, 494, 505
 Sparse connected layers, 327, 361
 Sparse depth-sensing methods, 21, 28
 Sparse local pattern methods, 117
 Sparse predictive search pipeline, 150
 Spatial filtering method, 50–56
 Spatial pyramid pooling (SPP), 422, 425, 434, 435, 437, 492, 509
 Spatial single-pattern methods, 15
 Spatio-temporal applications, 80, 197
 SP-CNN, 377
 Spectra
 descriptor family, 174, 175
 dimensions, 173–175, 183–185, 262
 discrimination, 138–139
 Spectral type, 175
 Speeded-up robust features method (SURF), 1, 36, 77, 119, 174, 190, 255, 277, 339, 490, 515, 551, 582
 Spherical
 camera, 9
 coordinates system, 128
 mosaicking method, 23, 29
 Spiking Neural Network (SNN), 442, 508
 SPMD *See* Single program multiple data (SPMD)
 SPP *See* Spatial pyramid pooling (SPP)
 SPP-NET, 434–437
 Squared Euclidean distance, 123, 174, 182
 SSD *See* Sum of squared differences (SSD)
 SSD/L2 norm, 123, 174, 182
 Stacking method, 4
 Stages operation and resources, 274–275, 282
 Stanford actions, 551
 Stanford 3D scanning repository, 548
 STAR detector, 525, 535
 Statistical difference metrics, 124
 Statistical distance family, 182
 Statistical methods, 67, 75, 76, 79–81, 94, 155, 156, 163, 298, 321, 322, 331, 361, 371, 376, 396, 469, 471
 Statistical region metrics, 96–104
 Steerable filters, 52, 61, 109–110, 143, 291, 293, 294
 Stereo calibration, 12, 29
 Storage formats, 31, 173, 174, 178, 183–185, 262
 Strip and radial fan shapes, 132
 Structural and model-based approaches, 79–80
 Structured light method, 11, 16, 26
 Structure from motion (SFM), 21, 28, 30
 Sub-pixel accuracy, 145–146, 172, 181, 218, 294
 Sub-region overlapping, 145
 Subsampling, 38, 148, 251, 265, 324, 325, 336, 338, 343, 354, 357, 369–370, 383, 386–390, 395, 396, 402, 413, 418, 419, 421, 424, 426, 431, 434, 437, 438, 442, 492, 499, 584, 586, 595
 Summary bin counts, 534, 543, 545
 Summary count, 518, 521, 524–532, 534–542
 Sum of absolute differences (SAD), 106, 121, 123, 154, 160, 174, 182, 217, 218, 230, 474
 Sum of squared differences (SSD), 28, 98, 106, 121, 123, 174, 182, 192, 218, 230, 404, 412, 474
 SUN, 548
 Super Vectors, 474, 475
 Supervised learning, 326, 334, 337
 Support Vector Machine (SVM), 121, 156–159, 163, 216, 326, 331, 335, 337, 343, 358, 361, 371, 382, 392, 396, 425, 450, 469–472, 474, 482–486, 488, 505, 506
 SURF *See* Speeded-up robust features method (SURF)
 Surface fusion, 21, 26
 Surface reconstruction, 1, 21, 26, 27, 30, 31
 Surface reconstruction and fusion, 26
 SURF detector, 521, 523, 532, 541
 Survey data, 252
 SUSAN method, 194
 SVM *See* Support Vector Machine (SVM)
 SYMNET, 357, 370, 438–442, 497
 Synthetic
 alphabets overlays, 268, 520, 521, 527–532
 apertures, 9
 feature alphabets, 262–269, 515
 interest point, 189, 251, 262–268, 516, 520–522, 532, 542, 545
 neuron, 323
 overlay images, 542, 546
 vision, 319, 320, 337, 371, 495, 511, 512
 Synthetic corner
 alphabet, 267
 point alphabet, 263, 520, 522–527, 533
 System compute optimization, 296
 System-on-a-chip (SOC), 273, 276, 278, 281, 282, 295, 308, 315
 System requirements, 289, 300

T

- Texture
 analysis, 75, 76, 78, 80, 85, 86, 93–96, 102, 114, 172, 184, 551
 region metrics, 81–95
 Threads and multiple cores, 308
 Threshold, 37, 80, 121, 177, 195, 261, 292, 324, 382, 517, 587
 Timed multiplexing multi-pattern methods, 15
 Time-of-flight (TOF) sensor, 17–18, 276
 Total coverage, 86, 89, 93, 565, 568, 571
 Total power, 86, 89, 90, 93, 565, 568, 572
 Training and statistical learning, 156
 Training DNNs, 342

- Training protocol DNNs, 342, 435, 439
Training system, 155–156
Trajkovic and Hedly method, 194
Transfer function, 323, 324, 332, 342, 353, 356, 367–369,
 399, 400, 403, 419, 421, 424, 426, 431, 434, 437,
 438, 442, 466
TRECVID, 552
Triangulation, 14, 25, 150, 158
Tuning controls, 312, 313
Tuning parameters, 364, 445, 503, 506, 507, 516–519,
 546
- U**
UCB contour detection and image segmentation, 553
Unfolding an RNN, 445–447
Univalue Segment Assimilating Nucleus (USAN), 194
Unlabeled data, 326, 366, 460, 490
Unsupervised learning, 153, 326, 334, 336, 337, 385,
 387, 468
USAN *See* Univalue Segment Assimilating Nucleus
 (USAN)
Use-case fitting, 254
- V**
Vanishing gradients, 337, 406, 407, 452, 453
Vector and Locally Applied Descriptors (VLAD), 474,
 475, 486
VFH *See* Viewpoint Feature Histogram (VFH)
VGG, 397, 408, 410, 424, 437, 438
Viewpoint Feature Histogram (VFH), 241
Vignette correction, 6
Viola Jones method, 40, 77, 116, 140, 146, 159, 177, 220,
 241, 486
Vision optimization resources, 314
- Vision pipelines, 7, 31, 35–37, 44, 66, 72, 87, 150, 151,
 156, 168, 173, 176, 177, 180, 182, 258, 273–315,
 347, 558
Visual
 dictionaries, 473
 encodings, 349, 355
 genomes, 322, 355, 389, 408, 429, 451, 577–602
 metric taxonomy, 262
 vocabulary, 41, 160–163, 168, 176, 242, 317, 325,
 328, 329, 375, 470, 472–474
Visual pathway standard visual pathway model, 496
VLAD *See* Vector and Locally Applied Descriptors
 (VLAD)
VLBP *See* Volume LBP (VLBP)
Volume LBP (VLBP), 95, 132, 179, 201–203
- W**
Wafer-scale cameras, 1
Walsh–Hadamard transform, 41, 108
Wang and Brady method, 76, 196
Wavelets, 27, 36, 40, 41, 61, 72, 76, 84–85, 108, 110–112,
 140, 159, 176, 214, 219–223, 231, 441, 468, 479
Weight(s)
 sharing, 336, 354, 388, 418, 443, 446, 447, 466
 tuning, 364, 378, 379, 381, 387, 391, 401, 404, 407
Weighting values, 199
White point color value, 47
Whole object approaches, 76
- Z**
Zernike polynomials, 31, 76, 109, 470
Zero-mean normalized cross-correlation (ZNCC), 218
ZFNeT, 419