Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione

Department of Information Engineering
Master Degree in Computer Engineering

Deep Learning's notes

Authors: Nicola Maritan, Gianluca Nordio
Proofreading: Francesco Boscolo Meneguolo

Accademic Year 2023-2024

Contents

1 Introduction **9**
1.1 Some definitions of intelligence . . . 9
1.2 Supervised learning . . . 9
1.3 Weak AI vs Strong AI . . . 9
1.4 Turing Test . . . 9
1.4.1 Winograd schema challenge . . . 9
1.5 Brute force in AI . . . 10

2 Pattern representation **11**
2.1 Kinds of pattern . . . 11
2.1.1 Numeric . . . 11
2.1.2 Categorical . . . 11
2.1.3 Sequences . . . 11
2.2 Pattern representation of proteins . . . 11
2.3 DNA microarray . . . 12
2.4 Local binary patterns . . . 12
2.4.1 Original LBP . . . 12
2.4.2 Extended LBP . . . 13
2.4.3 Uniform patterns . . . 13
2.4.4 LBP and images . . . 14
2.5 Two examples of dataset and their pattern representation . . . 14
2.5.1 Alzheimer dataset . . . 14
2.5.2 Autism dataset . . . 14
2.6 The dissimilarity space: bridging structural and statistical pattern recognition . . . 15
2.6.1 Dissimilarity matrix . . . 15

3 Machine Learning recall **16**
3.1 Learning tasks . . . 16
3.1.1 Classification . . . 16
3.1.2 Regression . . . 16
3.1.3 Clustering . . . 16
3.1.4 Dimensionality reduction . . . 17
3.2 Feature learning . . . 18
3.3 Basic learning approaches . . . 18
3.3.1 Supervised Learning . . . 18
3.3.2 Unsupervised Learning . . . 19
3.3.3 Semi-supervised Learning . . . 19
3.3.4 Multilabel problems . . . 19
3.3.5 Active Learning . . . 19
3.3.6 Self-supervised Learning . . . 19
3.4 Split computing . . . 19
3.5 One/few-shot learning . . . 20
3.6 Zero-shot learning . . . 21
3.7 Search and ranking . . . 21
3.7.1 PageRank . . . 22

3.7.2 Recommender systems . . . 22
3.8 Sequence learning . . . 22
3.9 Different kinds of learning . . . 22
3.10 Offline/batch learning . . . 23
3.11 Online/continual learning . . . 24
3.12 Reinforcement learning . . . 24
3.13 Computer Vision applications . . . 25
3.14 Parameters optimization . . . 25
3.15 Hyperparameters . . . 26
3.16 Performance indicators . . . 26
3.17 Overfitting . . . 27
3.18 Open Set . . . 27
3.18.1 DET and ROC curves . . . 29
3.18.2 Precision and Recall . . . 30
3.18.3 McNemar's test . . . 32
3.18.4 Average precision in object detection . . . 32
3.19 Multilabel performance indicators . . . 34
3.20 Training, Validation and Test . . . 34
3.21 Convergence . . . 35
3.22 Comparison tests . . . 35
3.22.1 Wilcoxon signed-rank test . . . 35
3.22.2 Friedman test for comparisons of multiple classfiers . . . 37
3.22.3 Post-hoc test . . . 39
3.22.4 Nemenyi test . . . 39
3.22.5 Bonferroni-Dunn test . . . 40
3.22.6 Holm test . . . 40
3.22.7 Examples of post-hoc tests . . . 40
3.23 Bias-variance Tradeoff . . . 41
3.23.1 Bias–variance decomposition of mean squared error . . . 43
3.23.2 Overfitting and underfitting . . . 44
3.23.3 Model's complexity . . . 45
3.24 Real world . . . 45
3.24.1 Automated performance evaluation . . . 45
3.24.2 More runs . . . 46
3.24.3 Ablation study . . . 46
3.24.4 Underspecification . . . 47

4 Neural Networks introduction **48**
4.1 Biological neuron . . . 48
4.2 McCulloch-Pitts neuron . . . 48
4.3 Neurons . . . 49
4.4 Feedforward neural networks . . . 49
4.5 Recurrent neural networks . . . 50
4.6 Perceptron . . . 50
4.7 Multilayer perceptron . . . 51
4.8 Forward propagation . . . 51
4.9 Training of a neural network . . . 54
4.9.1 Training of a neural network for a classification problem . . . 54
4.10 Stochastic Gradient Descent . . . 62
4.11 SoftMax . . . 62
4.12 Cross-Entropy loss . . . 63
4.13 Loss regularization . . . 65
4.14 Weight decay . . . 65

5 Learning rate **66**
5.1 Learning rate annealing . . . 66
5.2 Learning rate schedule . . . 66
5.3 Cyclic learning rate . . . 67
5.3.1 Triangular policy . . . 68
5.3.2 Effects of CLR, sharp and flat minima . . . 70
5.3.3 Stochastic Gradient Descent with Warm Restarts . . . 71
5.3.4 Visualization of CLR . . . 72
5.4 Momentum . . . 73
5.5 One-cycle learning rate . . . 73
5.6 Super-convergence . . . 74
5.7 Adaptive learning rate . . . 75
5.7.1 Adam . . . 75
5.7.2 diffGrad . . . 77
5.8 Convergence problems: check list . . . 77

6 Overfitting and dropout **78**
6.1 Overfitting revised - linear models and neural networks . . . 78
6.2 Robustness through perturbations . . . 78
6.3 Dropout . . . 79
6.3.1 Co-adaptation . . . 79
6.3.2 Implementation . . . 79

7 Deep Convolutional Neural Networks **81**
7.1 Complexity . . . 81
7.2 Topologies . . . 81
7.3 Deep Learning vs Machine Learning . . . 81
7.4 Problems of DNNs . . . 82
7.4.1 GPU computing . . . 82
7.4.2 Vanishing gradient problem . . . 82
7.4.3 Exploding gradient problem . . . 83
7.5 Parameter initialization . . . 83
7.5.1 Xavier initialization . . . 84
7.5.2 Heuristic for fully-connected layers . . . 85
7.5.3 He initialization . . . 85
7.6 Convolutional Neural Networks . . . 85
7.6.1 Images as input . . . 86
7.6.2 Convolution . . . 86
7.6.3 Convolution and cross-correlation . . . 86
7.6.4 Multiple input channels . . . 87
7.6.5 Multiple output channels . . . 87
7.6.6 1×1 convolutional layer . . . 89
7.6.7 3D volumes of neurons . . . 89
7.6.8 Feature maps . . . 89
7.6.9 Output size . . . 91
7.6.10 Invariance and constraining the network . . . 91
7.6.11 Inductive bias . . . 92
7.6.12 Activation function . . . 92
7.6.13 Pooling . . . 97
7.6.14 Final activation function and loss function for CNNs . . . 99
7.6.15 Pre-training . . . 99
7.7 Computing the gradient of a loss function . . . 99
7.7.1 Total derivative and multivariable chain rule . . . 99
7.7.2 Gradient/Jacobian of a vector function . . . 100
7.7.3 Gradient/Jacobian of the identity function . . . 100
7.7.4 Gradient/Jacobian of element-wise vector function combinations . . . 101
7.7.5 Gradient of vector sums . . . 103

7.7.6 Gradient of chain rule vector function combinations . . . 103
7.7.7 Gradient of a neuron . . . 104
7.7.8 Gradient of the loss function . . . 105
7.8 CaffeNet . . . 107
7.9 EnzyNet . . . 107
7.9.1 Voxels representation . . . 108
7.10 GoogLeNet . . . 110
7.10.1 Inception block . . . 110
7.11 ResNet . . . 111
7.11.1 Batch normalization for fully-connected layers . . . 111
7.11.2 Batch normalization for convolutional layers . . . 112
7.11.3 Why batch normalization works . . . 112
7.11.4 Function classes . . . 113
7.11.5 ResNet . . . 113
7.11.6 Residual blocks . . . 113
7.12 DenseNet . . . 114
7.13 Automatic differentiation . . . 116
7.13.1 Computational graph . . . 116
7.14 Transfer Learning . . . 119
7.14.1 Fine-Tuning . . . 120
7.14.2 Deep Features . . . 120
7.15 AutoEncoders . . . 120
7.16 Adversarial examples . . . 123
7.16.1 Generation of an adversarial example . . . 123
7.16.2 Types of attacks . . . 124
7.16.3 Physical world . . . 124
7.17 Distribution shift . . . 124
7.17.1 Covariate shift . . . 125
7.17.2 Label shift . . . 126
7.17.3 Concept shift . . . 126

8 Reinforcement Learning **128**
8.1 Setting of reinforcement learning . . . 128
8.2 Q-learning . . . 129
8.3 Robotic arm example . . . 129
8.4 Deep Q-Learning . . . 130
8.4.1 Videogame reinforcement learning . . . 130
8.4.2 Deep Q networks . . . 131
8.4.3 Experience replay . . . 131
8.4.4 ϵ-greedy approach . . . 131
8.4.5 Double DQN . . . 131

9 Sequence based neural networks **133**
9.1 Recurrent Neural Networks . . . 133
9.1.1 Unfolding in time . . . 135
9.1.2 Exploding gradient and gradient clipping . . . 135
9.1.3 Deep RNN . . . 135
9.2 Long Short-Term Memory . . . 136
9.3 Bidirectional Long Short-Term Memory . . . 137
9.4 Gated Recurrent Unit . . . 138
9.5 Temporal Convolutional Network . . . 138
9.5.1 Dilated causal convolution layer . . . 138
9.5.2 Disadvantages . . . 140
9.6 Recurrent DNNs for multilabel classification . . . 141
9.7 Time series . . . 141

10 Generative Adversarial Networks **142**
10.1 Overview . . . 142
10.2 Distribution estimation . . . 143
10.2.1 Random vector generation . . . 143
10.2.2 Comparing two probability distributions based on samples . . . 144
10.2.3 Direct training method . . . 144
10.2.4 Indirect training method . . . 144
10.2.5 The ideal case: perfect generator and discriminator . . . 145
10.2.6 The approximation: adversarial neural networks . . . 147
10.3 Mathematical details . . . 149
10.4 Manifold Matching via Metric Learning . . . 150

11 Siamese Networks **151**
11.1 Cosine similarity . . . 151
11.2 Binary Cross-Entropy loss . . . 151
11.3 Triplet loss . . . 153
11.4 Contrastive loss . . . 153
11.5 Dissimilarity space using SNNs . . . 153

12 Segmentation **155**
12.1 Image segmentation . . . 155
12.2 Instance segmentation . . . 155
12.3 Panoptic segmentation . . . 155
12.4 DeepLabV3+ . . . 156
12.4.1 Atrous convolution . . . 156
12.4.2 Depthwise separable convolution . . . 156
12.4.3 Decoder . . . 158
12.5 Region growing . . . 158
12.6 Loss functions for segmentation . . . 158
12.6.1 Dice loss . . . 158
12.6.2 Tversky loss . . . 159

13 Object detection **160**
13.1 Region-based CNN . . . 160
13.1.1 Region proposals . . . 160
13.1.2 Extracting features from the CNN . . . 162
13.1.3 Training . . . 167
13.2 Non-maximum suppression . . . 169
13.3 Fast R-CNN . . . 169
13.3.1 RoI pooling layer size . . . 173
13.3.2 CNN training . . . 173
13.4 Faster R-CNN . . . 174
13.5 You Only Look Once . . . 174
13.5.1 Execution . . . 175
13.5.2 Unified detection . . . 175
13.5.3 Training . . . 176
13.6 YOLOv2 . . . 177
13.6.1 Batch normalization . . . 177
13.6.2 High resolution classifier . . . 178
13.6.3 DarkNet-19 . . . 178
13.7 YOLOv3 . . . 178

14 Neural style transfer **179**
14.1 CNN-based style transfer . . . 179
14.1.1 Content and style layers . . . 180
14.1.2 Content loss . . . 180
14.1.3 Style loss . . . 180
14.1.4 Total variation loss . . . 181

15 Capsule networks **182**
15.1 CNNs problems . . . 182
15.2 Invariance and equivariance . . . 183
15.3 Capsule network . . . 183
15.4 Computation . . . 183
15.5 Spatial information . . . 184
15.5.1 Instantiation parameters intuition . . . 185
15.5.2 Example of capsule information . . . 185
15.6 Sample complexity . . . 185
15.7 Affine transformation . . . 185
15.8 Routing by agreement . . . 186
15.8.1 Dynamic routing . . . 186
15.9 Loss function . . . 188
15.9.1 Margin loss . . . 188
15.9.2 Reconstruction loss . . . 188
15.10 Capsule network for image recognition on MNIST . . . 189
15.10.1 PrimaryCaps layer . . . 189
15.11 Other capsule-based network . . . 190

16 Ensemble of networks **191**
16.1 Independence . . . 191
16.2 Bagging . . . 191
16.3 Majority voting . . . 192
16.3.1 Classifiers weight . . . 193
16.4 Q-statistic . . . 193
16.5 Combining continuous outputs . . . 194
16.6 Knowledge distillation . . . 196

17 Data augmentation **197**
17.1 Data augmentation techniques . . . 198
17.1.1 Geometric transformation . . . 198
17.1.2 Local filters . . . 198
17.1.3 Adding noise . . . 199
17.1.4 Color transformation . . . 200
17.1.5 Weathers . . . 200
17.1.6 Elastic deformation . . . 201
17.2 Applications . . . 201
17.3 AutoAugmented . . . 202

18 Graph Convolutional Networks **203**
18.1 Graphs . . . 203
18.1.1 Graph tasks . . . 204
18.2 Graph Convolutional Networks . . . 205
18.2.1 Graph matrix representation . . . 205
18.2.2 Graph signal . . . 207
18.2.3 Graph convolution . . . 208
18.2.4 Permutation equivariance . . . 209
18.2.5 Learning graph filters . . . 209
18.2.6 Graph perceptron . . . 209
18.2.7 Graph Convolution Networks . . . 210

18.3 GCN for node classification . . . 210
18.4 GCN for graph classification . . . 210
18.5 GCN examples . . . 211
18.5.1 Crop and weed classifier . . . 211
18.5.2 Pairwise drug combination effect on cancer cells . . . 211
18.6 GraphSage . . . 213

19 Transformer **214**
19.1 Model architecture . . . 215
19.2 Tokenization . . . 215
19.3 Embedding . . . 215
19.4 Positional encoding . . . 216
19.5 Self-attention . . . 218
19.5.1 Key-Value-Query Concept . . . 218
19.5.2 Self-attention layer . . . 219
19.5.3 Multi-head attention . . . 220
19.6 Addition and Layer Normalization . . . 222
19.7 Position-wise Feed-Forward Networks . . . 223
19.8 Encoder-decoder . . . 224
19.8.1 Masking . . . 225
19.9 Decoder block . . . 226
19.10 Decoder only . . . 226
19.11 Vision Transformer . . . 227
19.11.1 Model architecture . . . 227
19.12 Other works . . . 229
19.12.1 Comparing Transformers to CNN . . . 229
19.12.2 Transformers compared to brain . . . 230

20 Explainable AI **231**
20.1 Reasons for explainable AI . . . 231
20.2 Levels of interpretability . . . 232
20.3 LIME . . . 232
20.3.1 LIME for images . . . 233
20.4 Shapley values . . . 234
20.5 Grad-CAM . . . 234
20.6 Occlusion sensitivity . . . 235

21 Quantum Deep Learning **236**
21.1 Bra-ket notation . . . 236
21.2 Qubits . . . 236
21.3 Superposition . . . 236
21.4 Qubit representation . . . 237
21.5 Entanglement . . . 238
21.5.1 General formula for $n = 2$ qubits . . . 238
21.6 Bit vs Qubit . . . 238
21.7 Register of n qubits . . . 239
21.7.1 Quantum error correction codes . . . 239
21.8 Basic principles . . . 240
21.9 Quantum decoherence . . . 240
21.9.1 Causes of quantum decoherence . . . 240
21.9.2 Environment . . . 240
21.9.3 Hydrogen atom system . . . 241
21.10 Three main components of quantum computers . . . 241
21.11 Unitarity . . . 241
21.12 Wave-particle duality of matter . . . 241
21.13 Bloch's sphere . . . 242
21.13.1 Bloch's sphere examples . . . 243

21.14 Other basis . . . 244
21.15 Global architecture of a quantum computer . . . 245
21.16 Basic operations . . . 245
21.16.1 Identity . . . 245
21.16.2 Negation . . . 246
21.16.3 Constant-0 . . . 246
21.16.4 Constant-1 . . . 246
21.16.5 Tensor product of vectors . . . 246
21.16.6 Representing multiple bits . . . 247
21.16.7 CNOT . . . 247
21.16.8 Qubits and superposition . . . 248
21.16.9 Representing superposition . . . 249
21.16.10 Hadamard gate . . . 249
21.16.11 Bell's states generation . . . 250
21.16.12 Pauli operators . . . 250
21.16.13 Rotation operators . . . 250
21.17 Measurements . . . 251
21.18 Hybrid quantum network . . . 251
21.18.1 Encoding circuit . . . 252
21.18.2 Ladder-like PQC . . . 253

Chapter 1

Introduction

1.1 Some definitions of intelligence

The subsequent definitions are presented informally.

Intelligence is a set of complex of psychic and mental faculties that allow humans to think, understand or explain facts or actions, elaborate abstract models of reality, understand and to be understood by others, to judge, and to render it together capable of adapting to new situations

Artificial Intelligence (AI) is a partial reproduction of the intellectual properties of humans (with particular regard to processes of learning, recognition and choice) realized either through the elaboration of ideal models, or, concretely, with the development of machines that mostly use computers for this purpose. It implies thinking, understanding, elaborating, and, therefore, reasoning.

1.2 Supervised learning

Supervised learning is a paradigm in *machine learning* where input objects and a desired output value train a model. The training data is processed, building a function that maps new data on expected output values. The goal is to build a model able to *generalize* unseen data in the same application domain.

1.3 Weak AI vs Strong AI

In *weak AI*, machines acts like if they were smart. They are not able to have a mind of its own, and can only imitate physical behaviors that they can observe.

On the other hand, in *strong AI* machines think intelligently, and can apply intelligence to any problem, rather than just one specific problem.

1.4 Turing Test

The *Turing Test* is a test of a machine ability to exhibit intelligent behaviour equivalent to, or indistiguishable from, that of a human. A diagram of the test is shown in Figure 1.1, in which player C, the interrogator, is given the task of trying to determine which player - A or B - is a computer and which is human. The interrogator is limited to using the responses to written questions to make the determination. If the interrogator cannot reliably discriminate the machine from the human, the machine is said to have passed the test. The main criticism of such test is that accessing the immense amount of information available online can be used for answering in an "apparently" intelligent way.

1.4.1 Winograd schema challenge

The *Winograd schema challenge* (WSC) is a test of machine intelligence proposed in 2012 by Hector Levesque. A Winograd schema is a pair of sentences that differ in only one or two words and that contain an ambiguity. Questions of this form may be tailored to require knowledge and commonsense

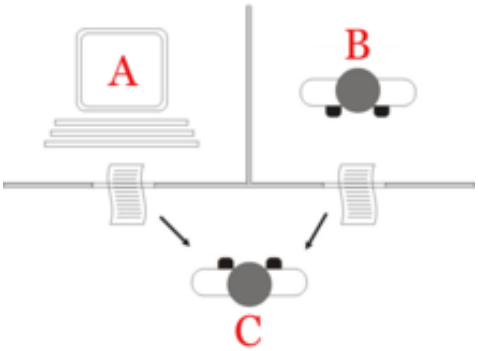


Figure 1.1: Diagram of the Turing Test. Source: [109].

reasoning in a variety of domains. The first cited example of a Winograd schema (and the reason for their name) is due to Terry Winograd:

> The city councilmen refused the demonstrators a permit because they [feared/advocated] violence.

The schema challenge question is the following: "Does the pronoun 'they' refer to the city councilmen or the demonstrators?". Switching between the two instances of the schema changes the answer. The answer is immediate for a human reader, but proves difficult to emulate in machines. Levesque argues that knowledge plays a central role in these problems: the answer to this schema has to do with our understanding of the typical relationships between and behavior of councilmen and demonstrators.

1.5 Brute force in AI

Brute force algorithms were developed for the game of chess, in which every possible placement of pieces on the board is associated with a score. Winning is associated with a $+\infty$ score and defeat with a $-\infty$ score. When the algorithm has to pick a move, it evaluates all the possible moves at its disposal, the opponent's countermoves (assuming the optimal player), his moves to the next step, and so on recursively until the end of the game. When it is possible to expand all paths to the end of the game, a simple algorithm known as a *Minimax Algorithm* allows the optimal move to always be chosen. In 1997, Deep Blue (IBM) won a game against the world champion Garry Kasparov.

Unfortunately, for non-trivial games the tree of possibilities alternatives becomes too large after a few levels of depth, therefore the evaluation of the score takes place prematurely, the choice is only "locally" excellent. Heuristic approaches for branch pruning (e.g. alpha-beta pruning) and machine learning/deep learning techniques are widely used in modern implementations. The evaluation of the score is complex and characterized by many parameters: how important a safe position is for the king in comparison to a space advantage in the center of the chess board? The optimal values for these parameters were determined by the machine learning system itself, analyzing thousands of matches of human champions.

Chapter 2

Pattern representation

Data is a fundamental ingredient of ML, as the behaviour of the algorithms are learned from the data itself. A *pattern* is a sample [22]. The term pattern will be used to refer to data. For example, a pattern can be a face, an handwritten character, a fingerprint, a sound signal, a fragment of text, the performance of a stock exchange. Pattern recognition is the discipline that studies recognition of patterns, not only with learning techniques but also with pre-programmed algorithms.

2.1 Kinds of pattern

Different kinds of patterns can be identified.

2.1.1 Numeric

Numeric patterns are associated with measurable characteristics. These values can be continuous or discrete, and they are naturally represented as vectors in $\mathbb{R}^d$. The extraction of characteristics from signals (e.g. images and sounds) produces numerical vectors, the *feature vectors*.

2.1.2 Categorical

Categorical values are associated with qualitative characteristics (e.g. gender, eye color, blood of a human being) or the presence or absence of a feature (e.g. has the driving licence or not). Sometimes they represents ordinal values (e.g. the environment temperature can be high, medium or low).

2.1.3 Sequences

Sequences are variable length sequential patterns with spatial or temporal relationships (e.g. an audio stream or sequence of sounds corresponding to the pronunciation of a word or sentence). The position in the sequence and the relationships with predecessors and successors are important.

2.2 Pattern representation of proteins

A protein is a sequence of amino acids. Proteins are usually encoded in a vector space, and there are multiple ways to encode a protein. The *amino acid composition* $AS(i)$ is the number of i-th amino acids present in a protein normalized by the length of the protein, more formally:

$$AS(i) = \frac{h(i)}{N} \quad \text{for} \quad i \in \{1, \ldots, 20\}, \tag{2.1}$$

where $h(i)$ is the number of occurrences of the i-th amino acid in a protein of length N. Therefore, $\mathbf{AS} \in \mathbb{R}^{20}$ represents a protein with 20 features.

Another way to encode a protein is *2-Gram* (2G), which represents a protein with 20^2 features, where each feature is obtained by counting the number of occurrences of a pair of amino acids:

$$2G(k) = \frac{h(i,j)}{N} \quad \text{for} \quad i,j \in \{1,\dots,20\}, k = j + 20 \times (i-1), \tag{2.2}$$

where $h(i,j)$ is the number of occurrences of a pair of amino acids (i,j) in a protein of length N.

One more way to encode a protein is the *autocovariance approach* (AC). Given the maximum distance m between two considered amino acids, $\mathbf{AC} \in \mathbb{R}^{20+m}$.

2.3 DNA microarray

Scientists know that a mutation - or alteration - in a particular gene's DNA may contribute to a certain disease. However, it can be very difficult to develop a test to detect these mutations, because most large genes have many regions where mutations can occur. The DNA microarray consists of a small glass plate encased in plastic. On the surface, each chip contains thousands of short, synthetic, single-stranded DNA sequences, which together add up to the normal gene in question, and to variants (mutations) of that gene that have been found in the human population. A typical use is to compare the gene expression profile of a sick individual with that of a healthy one to identify which genes are involved in the disease, or to choose the appropriate drug based on gene expression. Features selection techniques are used to select only some gene expressions. For example, consider a binary classification problem. The Signal-to-Noise Ratio (SNR) $S(j)$ for the j-th gene is computed as

$$S(j) = \frac{\mu_+(j) - \mu_-(j)}{\sigma_+(j) - \sigma_-(j)} \tag{2.3}$$

If $S(j) > 0$, the j-th gene is correlated with the class +1, otherwise, if $S(j) < 0$, the j-th gene is correlated with the class -1. Then, we select half "positive" and half "negative" genes.

2.4 Local binary patterns

Local binary patterns (LBP) is a type of visual descriptor used for classification in computer vision. It is a way to describe the local structure of an image in a way that is invariant to changes in illumination.

2.4.1 Original LBP

In the original LBP, given a "central" pixel, each pixel in its neighborhood is assigned a binary value based on whether its intensity is greater (1) than or less (0) than the intensity of the central pixel (threshold). Then, the binary values are combined for all neighboring pixels to obtain a binary code for the central pixel (anti-clockwise, starting from the top left corner), and convert it to a decimal value. With 8 surrounding pixels, we have a total of $2^8 = 256$ possible combinations of LBP codes. This procedure can be applied to each pixel in the image. An example of this procedure is shown in Figure 2.1. Given an image, we can create a feature vector in $\mathbb{R}^{256}$ in the following way [64]:

- For each pixel in a cell, compare the pixel to each of its 8 neighbors in anti-clockwise, starting from the top left corner.
- Where the center pixel's value is greater than the neighbor's value, write "0". Otherwise, write "1". This gives an 8-digit binary number.
- Compute the histogram, over the cell, of the frequency of each "binary number" occurring. This histogram can be seen as a 256-dimensional feature vector.
- Optionally normalize the histogram.
- Concatenate (normalized) histograms of all cells. This gives a feature vector for the entire window.

The feature vector can now be processed using the SVM or some other machine learning algorithm to classify images. Such classifiers can be used for face recognition or texture analysis.

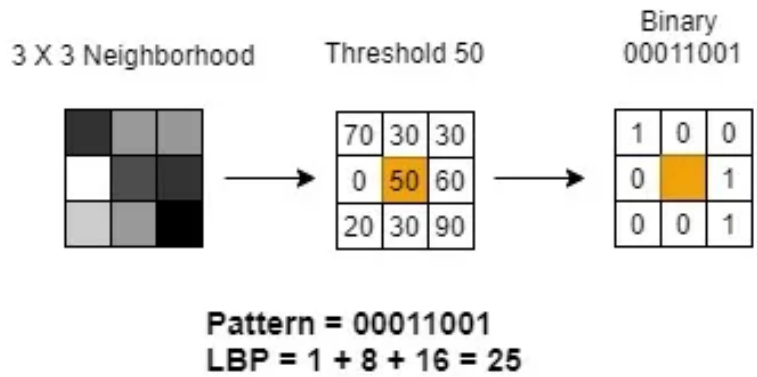


Figure 2.1: Example of how LBP works. Source: [113].

2.4.2 Extended LBP

The LBP operator was extended to use *circular* neighborhoods of different *radii*. Consider a center (x_c, y_c) and let g_c be its value. Consider the points (x_p, y_p), with $p \in \{1, \ldots, P\}$ placed at distance R from the center, i.e.

$$x_p = x_c + R\cos\left(\frac{2\pi p}{P}\right) \tag{2.4}$$

$$y_p = y_c - R\sin\left(\frac{2\pi p}{P}\right) \tag{2.5}$$

for $p \in \{1, \ldots, P\}$, and let g_p be its value. Then:

$$LBP_{P,R} = \sum_{p=0}^{P-1} s(g_p - g_c)2^p, \tag{2.6}$$

where

$$s(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.7}$$

From Equations 2.4 and 2.5, it is possible to see that the procedure "starts" from the rightmost pixel and proceeds anti-clockwise. An instance of the procedure is shown in Figure 2.2

2.4.3 Uniform patterns

A useful extension to the original operator of Section 2.4.1 is the so-called uniform pattern, which can be used to reduce the length of the feature vector and implement a simple rotation invariant descriptor. A LBP is *uniform* if it contains at most two *bitwise transitions* from 0-1 or 1-0 transitions when the bit pattern is traversed circularly. For example, the pattern 01110000 has 2 transitions, because the bit at index 1 switches from 0 to 1 and the bit at index 4 switches back from 1 to 0. The pattern does not contain other transitions, as both bit at index zero and seven are equal. Therefore, 00010000 (2 transitions) is a uniform pattern, but 01010100 (6 transitions) is not. In the computation of the LBP histogram, the histogram has a separate bin for every uniform pattern, and all non-uniform patterns are assigned to a single bin (i.e., all non-uniform patterns represents a single column in the histogram). For example, when using the original LBP, there are a total of $2^8 = 256$ patterns, 58 of which are uniform (this can be proven mathematically). Using uniform patterns, the length of the feature vector for a single cell reduces from 256 to 59 (i.e. we obtain a feature vector in $\mathbb{R}^{59}$ instead of $\mathbb{R}^{256}$ as presented in Section 2.4.1).

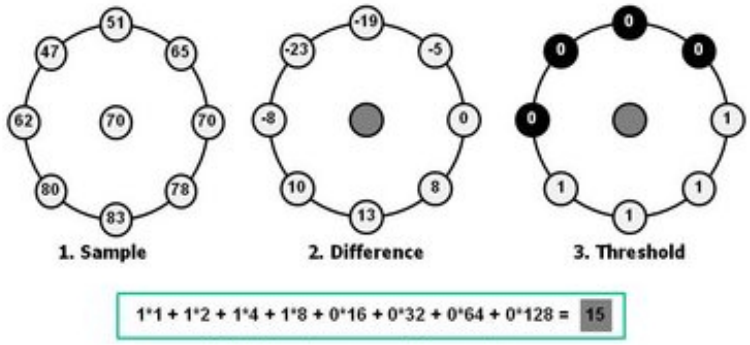


Figure 2.2: Example of how the LBP extension works. The procedure "starts" from the rightmost pixel (of original value 70) and procedes anti-clockwise. Source: [64].

2.4.4 LBP and images

The descriptors just seen, if calculated on the entire image, do not retain spatial information; to enrich the descriptor with local information it is possible to divide the image into regions calculating the descriptors for each region. For example, if an image is divided into 6 blocks then it will be described by 6x features with respect to the descriptor applied only to the entire image. In fact, by applying the procedure of Section 2.4.1, we obtain 6 different histograms (one for each block), from which we obtain a feature vector in $\mathbb{R}^{256}$. We then concatenate the 6 vectors, obtaining de facto a vector in $\mathbb{R}^{256\times 6}$.

2.5 Two examples of dataset and their pattern representation

The following datasets are briefly described. Their names are missing and were not provided from the lecturer nor found online.

2.5.1 Alzheimer dataset

The alzheimer dataset includes data from 259 individuals: 85 ranging from presymptomatic to late-stage Alzheimer Disease (AD), 11 with Other Dementia (OD), 47 having Mild Cognitive Impairment (MCI), 21 with Other Neurological Disease (OND), and 79 Non-Demented Controls (NDC). The features are the expression values of 120 known blood plasma proteins extracted from each individual.

2.5.2 Autism dataset

The dataset includes data collected from 206 children between the ages of three and ten years to assess levels of oxidative stress. The patterns collected are the result of analysis measurements of metabolite concentrations of the folate-dependent one-carbon metabolism and transulfuration pathways taken from blood samples and are composed of 25 features. The patterns belong to two classes: 83 children with Autism Spectrum Disorders (ASD) and 76 age-matched controls with no-ASD.

2.6 The dissimilarity space: bridging structural and statistical pattern recognition

Automatic systems for pattern recognition may be designed on a structural description (e.g. graphs) derived from sensor observations. Alternatively, training sets of examples can be used in statistical learning procedures. They are most powerful for vector representations. Unfortunately, structural descriptions do not match well with vectorial representations. Consequently it is difficult to combine the structural and statistical approaches to pattern recognition. Structural descriptions may be used to compare objects. This leads to a set of *pairwise dissimilarities* from which vectors can be derived for the purpose of statistical learning. The resulting dissimilarity representation bridges thereby the structural and statistical approaches. The dissimilarity space is one of the possible spaces resulting from this representation.

The *dissimilarity space* is a vector space in which the dimensions are defined by dissimilarity vectors measuring pairwise dissimilarities between examples and individual objects from the so-called representation set R (recall the graphs example). Hence, a dissimilarity representation $D(X, R)$ is addressed as a data-dependent mapping $D(\cdot, R) : X \to \mathbb{R}^n$ from an initial set of objects $X \subseteq R$ to a dissimilarity space (above it is $\mathbb{R}^n$), equipped with the traditional inner product and Euclidean metric. The representation set can be chosen as the complete training set T, a set of carefully selected or constructed prototypes.

In practice, methods based on the dissimilarity space make possible to represent complex structures as vectors even without the dissimilarity space being a *metric space*.

Definition 2.6.1 *A metric space is an ordered pair (M, d), where M is a set and d is a metric on M, i.e. a function $d : M \times M \to \mathbb{R}^+$ such that for any $x, y, z \in M$ the following holds:*

- $d(x, y) = 0 \iff x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$

For example, the dissimilarity space may be a *pseudometric space*.

Definition 2.6.2 *A pseudometric space is an ordered pair (M, d), where M is a set and d is a metric on M, i.e. a function $d : M \times M \to \mathbb{R}^+$ such that for any $x, y, z \in M$ the following holds:*

- $d(x, x) = 0$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$

Unlike a metric space, points in a pseudometric space need not to be distinguishable, that is, we may have $d(x, y) = 0$ with $x \neq y$.

2.6.1 Dissimilarity matrix

Consider n objects in "structural representation" $O = \{o_1, \ldots, o_n\} \subseteq X$. The *dissimilarity matrix* is a $n \times n$ matrix $D(O, O)$ (meaning that we consider all pairs in O) where $D_{i,j} = d(o_i, o_j)$ is the dissimilarity between o_i and o_j. We build the training set T where the i-th instance $\mathbf{t}_i \in \mathbb{R}^n$ is computed as

$$\mathbf{t}_i = D(o_i, O) = [d(o_i, o_1), d(o_i, o_2), \ldots, d(o_i, o_n)]^T. \tag{2.8}$$

In fact, $\mathbf{t}_i \in T$ represents $o_i \in O$. Similarly, we can define the k-th feature of each $\mathbf{t}_i$ as

$$D(O, o_k) = [d(o_1, o_k), d(o_2, o_k), \ldots, d(o_n, o_k)]^T. \tag{2.9}$$

Chapter 3

Machine Learning recall

This chapter focuses on providing the essential Machine Learning notions for the course.

3.1 Learning tasks

3.1.1 Classification

Classification (or *recognition*) is the supervised learning task of assigning a class to a pattern. More precisely, a function from the pattern space $\mathcal{X}$ to a class space $\mathcal{Y}$ is learnt. If $|\mathcal{Y}| = 2$, it is called *binary classification*. If $|\mathcal{Y}| > 2$, it is called *multi-class classification*.

3.1.2 Regression

Regression is the supervised learning task of assigning a continuous value to a pattern. More precisely, a function from the pattern space $\mathcal{X}$ to a class space $\mathcal{Y} = \mathbb{R}$ is learnt. A regression problem requires the prediction of a quantity. A problem can have real values or discrete input variables. A problem with multiple input variables is often called a *multivariate regression* problem. If the input variables are ordered by time is called a *time series forecasting* problem. Some examples of regression tasks are:

- Estimated sale prices for apartments in the real estate market.
- Risk estimation for insurance companies.
- Energy prediction produced by a photovoltaic system.
- Health cost prediction models.

3.1.3 Clustering

Clustering is the unsupervised learning task of identifying groups (clusters) of patterns with similar characteristics. Since it is an unsupervised learning task, the "classes" are not known and the patterns are not labeled. Often not even the number of clusters is known a priori. The clusters identified through clustering can then be used as classes in a supervised learning task (e.g. for classification). Clustering methods can be used for the following fields and scenarios:

- Marketing: definition of user groups based on consumption
- Genetics: grouping of individuals based on DNA analogies
- Bioinformatics: partitioning of genes into similar groups
- Computer vision (CV): image segmentation. It is the process of partitioning a digital image into multiple segments (sets of pixels, also known as super-pixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a

label to every pixel in an image such that pixels with the same label share certain characteristics. An example of image segmentation is reported in Figure 3.1.

Figure 3.1: Example of image segmentation. Source: [72].

3.1.4 Dimensionality reduction

Dimensionality reduction is the transformation of data from a high-dimensional space into a low-dimensional space, so that the low-dimensional representation retains some meaningful properties of the original data. The transformation is a mapping $\mathbb{R}^d \to \mathbb{R}^k$ with $k < d$. In fact, often not all d features are actually useful, since some of them can bring some "noise". The removal of the noisy features can increase the accuracy of the classification step.

Having a high dimensionality with respect to the number of patterns leads to the so-called *curse of dimensionality*. The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces (often with hundreds or thousands of dimensions) that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience. There are multiple phenomena referred to by this name in domains such as numerical analysis, sampling, combinatorics, data mining, databases and, of course, ML. The common theme of these problems is that when the dimensionality increases, the volume of the space increases so fast that the available data become *sparse*. This sparsity is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the amount of data needed to support the result often grows exponentially with the dimensionality. Organizing and searching data often relies on detecting areas where objects form groups with similar properties; in high dimensional data, however, all objects appear to be sparse and dissimilar in many ways, which prevents common data organization strategies from being efficient.

The main dimensionality reduction methods are the following:

- *Feature selection* methods for keeping the most important features:
  - Forward selection
  - Backward selection/elimination
  - Random forest
- Methods for finding a combination of new features:
  - Linear methods:
    - *Principal Component Analysis* (PCA)
    - *Factor Analysis* (FA)
    - *Linear Discriminant Analysis* (LDA)

    * Truncated *Singular Value Decomposition* (SVD)
  - Non-linear methods (manifold learning, an attempt to generalize linear frameworks like PCA to be sensitive to non-linear structure in data):
    * Kernel PCA
    * *t-Distributed Stochastic Neighbor Embedding* (t-SNE)
    * *Multidimensional Scaling* (MDS)
    * Isomap

These methods are illustrated in Figure 3.2.

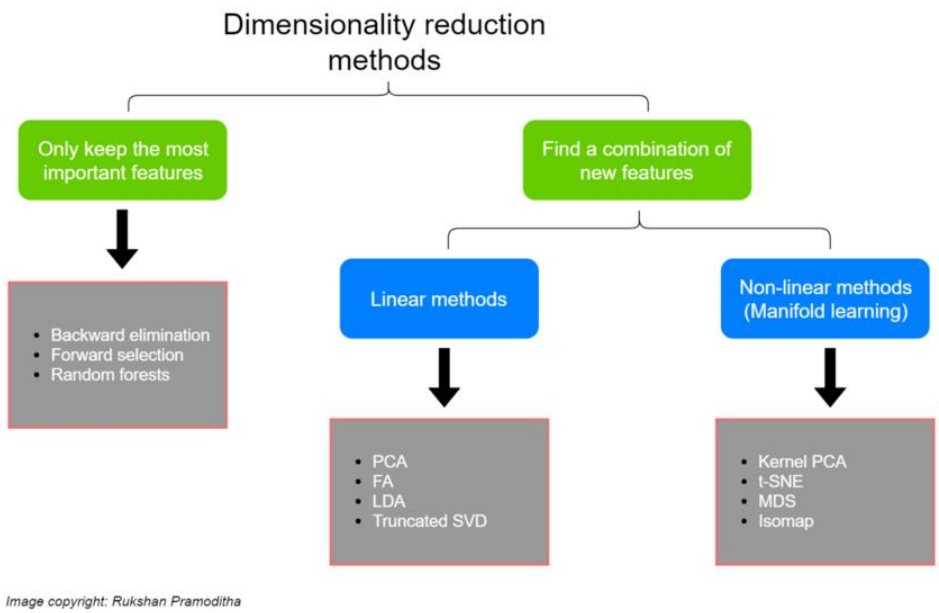


Figure 3.2: Main dimensionality reduction methods. Source: [72].

3.2 Feature learning

The success of many machine learning applications depends from the effectiveness of the representation of the patterns in terms of features. The definition of ad-hoc (hand-crafted) features for the different applications is called *feature engineering*. For example, for the recognition of objects there are numerous shape, color and texture descriptors that we can use to convert images to numeric vectors.

At this point, the question is: Can we automatically learn effective features starting from *raw data*? Or similarly, can we operate directly on raw data (e.g. intensity of the pixels of an image, amplitude of a audio signal over time) without using pre-defined features? The answer is yes: most of the deep learning techniques (e.g. convolutional neural networks) operate in this way, using as input the raw data and automatically extracting the features necessary to solve the problem of interest.

3.3 Basic learning approaches

The main learning approaches are briefly introduced.

3.3.1 Supervised Learning

In *supervised learning* the pattern classes are known, the training set is labeled. This is the case in classification, regression and some dimensionality reduction techniques (e.g. LDA).

3.3.2 Unsupervised Learning

In *unsupervised* the classes of the patterns used for training are unknown, i.e. the training set is not labeled. This is the typical situation in clustering and in most of dimensionality reduction techniques.

3.3.3 Semi-supervised Learning

In *semi-supervised* the training set is partially labeled and unlabeled patterns can help to optimize the classification rule. Not every pattern can be labeled, as the cost of manual analysis by a human expert for classification may be prohibitively high. For instance. think about classifying all the various audio signals coming from the birds that live in a given forest, we will have a few tens of thousands of audio files labeled and millions not labeled.

3.3.4 Multilabel problems

In *multilabel* problems the goal is to associates several classes to each pattern. Ad hoc algorithms were developed to handle multilabel problems. One example of multilabel problem is to assign a given newspaper article to multiple topics, where each topic is a class. Or again, the Anatomical Therapeutic Chemical (ATC) system is a multi-label classification system proposed by the World Health Organization, which categorizes drugs into classes according to their therapeutic effects and characteristics, such as alimentary tract and metabolism (class 1), blood and blood forming organs (class 2), cardiovascular system (class 3) and so on.

3.3.5 Active Learning

Active learning is a particular case of semisupervised learning where a learning algorithm can interactively query a human user (or some other information source), to label new pattern with the desired outputs. The human user must possess knowledge/expertise in the problem domain. The information source is also called teacher or oracle. In this process, the most "useful" patterns for the classification system are labeled. The number of patterns to be labeled depends on the cost of the class selection process.

3.3.6 Self-supervised Learning

Self-supervised learning (SSL) is a form of unsupervised learning where the unlabeled data provide supervision. The model learns in two steps.

- First, the training data itself is used to generate *pseudo-labels*, creating a *pretext task* that the model learns to solve. The aim of the pretext task is to guide the model to learn intermediate representations of data and in understanding the underlying structural meaning. This is typically done by designing tasks where the model predicts certain aspects of the input, such as predicting missing parts, relationships between elements, or transformations applied to the data.
- Second, the actual task is performed with supervised learning (segmentation, classification, and regression tasks), which is often called *downstream task*.

An example of pretext training for images is training a model to tell the relative position between two cropped regions of the same image. It is also possible to classify the rotation of an image [97] (as shown in Figure 3.3). The rotated images are not labeled by a human, but assigned during the rotation of the image.

Pretext tasks can also be used for text. In this case, it is possible to train a model to "fill in the blanks" by predicting a masked word using the surrounding words. Again, no manual labeling is performed.

3.4 Split computing

The *Deep Neural Network* (DNN) computational requirements preclude their device-only deployment on most of the resource-constraint systems, such as mobile phones. A possible alternative is to refer to

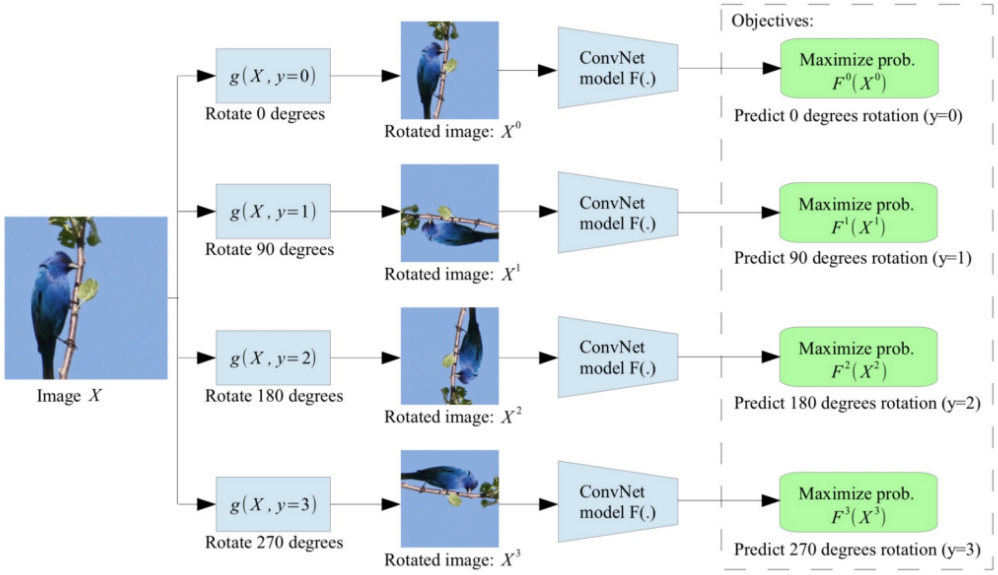


Figure 3.3: Pretext task of images, in which a model learns 4 possible rotation of an image. Source: [97].

simplified models (e.g. PyTorch Mobile) with an obvious accuracy decline which may not be desirable. The opposite cloud-only paradigm is to transfer the data captured by the device to a server through a communication network (e.g. ChatGPT). The servers, which run the DNN model, compute the inference and send it back to the device. In this case, the high data transfer time and possible server congestion are obvious downsides.

As a compromise between the *device-only* and the *cloud-only* approaches, the *split computing* (SC) frameworks propose to divide DNN models into a *head* and a *tail*, deployed at the edge device and the server, respectively, as shown in Figure 3.4. Using SC, the computational load is distributed across the two platforms, exploiting the computational power of both devices at the cost of transferring data on a network connection. Data transfer effort is mitigated by the injection of *bottlenecks*, usually carried out by autoencoders, achieving in-model compression. As formally described in [66], a bottleneck is a compression point at one layer in the model, which can be realized by reducing the number of nodes of the target layer and/or by quantizing its output. Setting up a SC system requires setting up the optimal splitting point: so far, this choice was driven primarily by architectural considerations such as memory capabilities of the edge device, desired transmission rate between head and tail, compression rate of the bottleneck, and nature of the layers.

3.5 One/few-shot learning

One/few-shot learning is an object categorization problem, found mostly in CV. Whereas most machine learning-based object categorization algorithms require training on hundreds or thousands of samples, one-shot learning aims to classify objects from one, or only a few, samples.

The ability to learn object categories from few examples, and at a rapid pace, has been demonstrated in humans. It is estimated that a child learns almost all of the 10-30 thousand object categories in the world by age six. This is due not only to the human mind's computational power, but also to its ability to synthesize and learn new object categories from existing information about different, previously learned categories. Given two examples from two object categories: one, an unknown object composed of familiar shapes, the second, an unknown, amorphous shape; it is much easier for humans to recognize the former than the latter, suggesting that humans make use of previously learned categories when learning new ones. The key motivation for solving one-shot learning is that systems, like humans, can use knowledge about object categories to classify new objects.

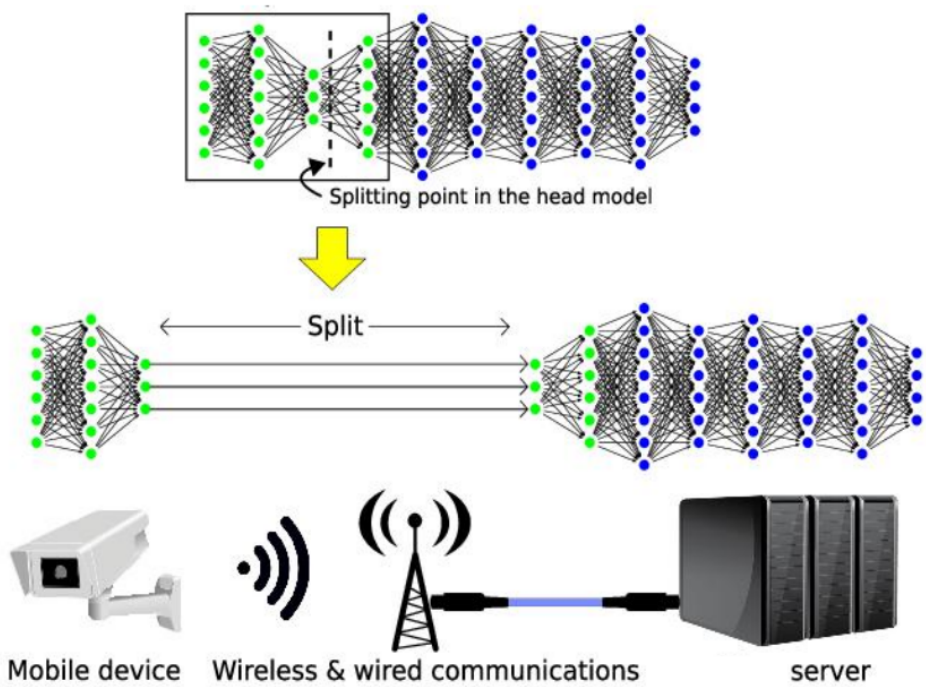


Figure 3.4: Diagram of split computing where the splitting point is reported. The 3-nodes layer is "duplicated" as conjunction of the two splits. Source: [72].

3.6 Zero-shot learning

Zero-shot learning (ZSL) is a problem setup in deep learning where, at test time, a learner observes samples from classes which were not observed during training, and needs to predict the class that they belong to. Zero-shot methods generally work by associating observed and non-observed classes through some form of auxiliary information, which encodes observable distinguishing properties of objects. This means that in ZSL data is in the following manner:

- Seen classes, classes available for training.
- Unseen classes, classes that occur only in the test set or during inference. Not present during training.
- Auxiliary information, information about both seen and unseen class labels during training time.

It is possible to think about ZSL as follows. Suppose a person knows what a horse looks like. And give that person the info that "A zebra is a horse with black and white striped skin". Then there is a high chance that a person can identify a zebra even though that person had no prior idea of what a zebra is. In a ZSL setting, the "horse" class is a seen class, the "zebra" class is an unseen class and the auxiliary information is that "a zebra is a horse with black and white striped skin".

Based on the data available during inference, ZSL can be classified into two types:

- Conventional zero-shot learning: During test time only images from unseen classes are expected.
- Generalized zero-shot learning: During testing phase images from both seen and unseen class can be present. This is the most used mode for most practical use cases.

3.7 Search and ranking

Sometimes we do not just want to assign each sample to a class or to a real value. In information retrieval and in we want to impose a *ranking* on a set of items. For instance, consider web search. The goal is to determine whether a particular page is relevant for a particular user. The focus is about the ordering of the relevant search results and the learning algorithm needs to produce *ordered* subsets of elements from a larger set.

3.7.1 PageRank

One possible solution to this problem is to first assign to every element in the set a corresponding relevance score and then to retrieve the top-rated elements. PageRank, the original algorithm behind the Google search engine, was an early example of such scoring system, even if it did not depend on the actual query. Instead, it relied on a simple relevance filter to identify the set of a relevant items and then on PageRank to order those results that contained the query term. Nowadays, search engines use machine learning and behavioural models to obtain query-dependent relevance scores.

3.7.2 Recommender systems

Recommender systems are another problem setting that is related to search and ranking. Similarly to Section 3.7.1, the goal is to display a set of relevant items to the user. The main difference is the emphasis on *personalization* to specific users in the context of recommender systems. For instance, for movie recommendations, the results page for a science fiction fan and the results page for a connoisseur of Peter Sellers comedies might differ significantly. Similar problems pop up in other recommendation settings, for example, for retail products, music and news recommendation.

In some cases, customers provide explicit feedback communicating how much they liked a particular product (e.g. the product ratings and reviews on Amazon, IMDb and Goodreads). In some other cases, they provide implicit feedback, for example, by skipping titles on a playlist, which might indicated dissatisfaction but might just indicate that the song was inappropriate in context. In the simplest formulations, these systems are trained to estimate some score, such as an estimated rating or the probability of purchase, given a user and an item.

Given such a model, for any given user, we could retrieve the set of objects with the largest scores, which could then be recommended to the user. Production systems are considerably more advanced and take detailed user activity and item characteristics into account when computing such scores.

3.8 Sequence learning

So far, we have looked at problems where we have some fixed number of inputs and produce a fixed number of outputs. For example, consider predicting house prices from a fixed set of features: square footage, number of bedrooms, number of bathrooms, walking time to downtown. Or again, think about a mapping from an image of fixed dimension to the predicted probabilities that it belongs to each of a fixed number of classes, or taking a user ID and a product ID, and predicting a star rating. In these cases, once we feed our fixed-length input into the model to generate an output, the model immediately *forgets* what it just saw.

This might be fine if our inputs truly all have the same dimensions and if successive inputs truly have nothing to do with each other. But how would we deal with video snippets? In this case, each snippet might consist of a different number of frames. And our guess of what is going on in each frame might be much stronger if we take into account the previous or succeeding frames. Same goes for language. One popular deep learning problem is machine translation: the task of ingesting sentences in some source language and predicting their translation in another language. These problems also occur in medicine. We might want a model to monitor patients in the intensive care unit and to fire off alerts if their risk of death in the next 24 hours exceeds some threshold. We definitely would not want this model to throw away everything it knows about the patient history each hour and just make its predictions based on the most recent measurements.

These problems are instances of *sequence learning*. They require a model to either ingest sequences of inputs or to emit sequences of outputs (or both). Specifically, *sequence to sequence learning* considers problems where input and output are both variable-length sequences, such as machine translation and transcribing text from the spoken speech.

3.9 Different kinds of learning

In this chapter various learning approaches were discussed (supervised, unsupervised, etc). Now two main "kinds" of learning are being introduced.

- In *batch learning* (nearly the same as *offline learning*, see Section 3.10) the training is done only once on the training set. Once the training is finished, the system switches to "working modality" and it is unable to learn further.
- In *incremental learning*, following the initial training, there are further training sessions. One type of incremental learning is *online learning*. The main risk associated with incremental learning is *catastrophic forgetting*, where the model "forgets" what it has previously learnt in favor of new knowledge.
- *Federated learning* consists in training a model on multiple local datasets contained in different nodes, without explicitly exchanging data examples. In fact, we typically train local models on local data samples and exchange the model parameters between the nodes at some frequency to generate a global model shared by all nodes. In this scenario, the datasets are typically heterogeneous and their sizes may span several orders of magnitude. Federated learning is employed when we concern about privacy issues (e.g. patient data in a hospital cannot be shared).

3.10 Offline/batch learning

So far, it was not discussed where data actually come from, or what actually happens when a machine learning model generates an output. That is because supervised learning and unsupervised learning do not address these issues in a very sophisticated way. In either case, a big pile of data is selected upfront, then the pattern recognition machines is set in motion without ever interacting with the *environment* again. Because all of the learning takes place before the algorithm is disconnected from the environment, this is sometimes called *offline learning*. It is also called *batch learning*, as the training is done only once on the training set. According to the lecturer (no written evidence) the small difference is that in offline learning we can eventually retrain a model, but only after collecting samples from the environment and disconnecting the model from it. For supervised learning, the process by considering data collection from an environment is illustrated in Figure 3.5. The upside of offline

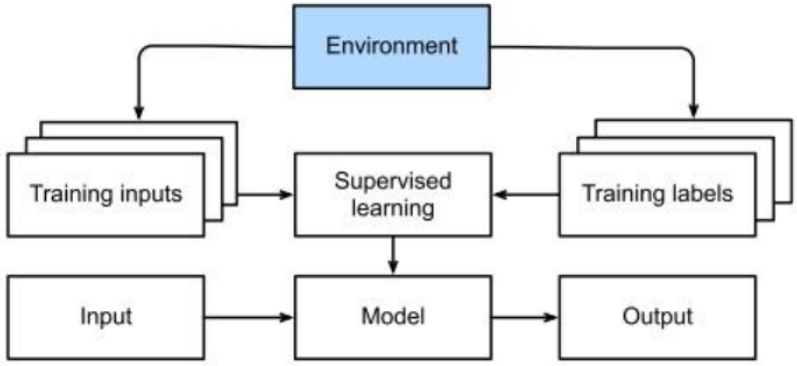


Figure 3.5: Collecting data for supervised learning. In a statistical learning setting, a training set is composed by picking training inputs $\mathcal{X}$ and their training label in $\mathcal{Y}$ from the environment, a distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. Source: [72].

learning is that it is possible to worry about pattern recognition in isolation. However, the downside is that the problem formulation is quite limiting. In fact, it is easy to imagine intelligent robots capable of not only making predictions, but also of taking actions in the world, i.e. intelligent agents which can choose actions. Moreover, unlike predictions, actions actually impact the environment. Training an intelligent agent implies taking to account how its actions might impact the future observations of the agent.

3.11 Online/continual learning

A *online/continual learning* (CL) system can be defined as an adaptive algorithm capable of learning from a continuous stream of information, with such information becoming progressively available over time and where the number of tasks to be learned (e.g. membership classes in a classification task) are not predefined. Critically, the accommodation of new information should occur without catastrophic forgetting or interference.

CL is built on the idea of learning continuously and adaptively about the external world and enabling the autonomous incremental development of ever more complex skills and knowledge. In the context of machine learning it means being able to smoothly update the prediction model to take into account different tasks and data distributions but still being able to re-use and retain useful knowledge and skills during time. Hence, CL is the only paradigm which force us to deal with an higher and realistic time-scale where data (and tasks) becomes available only during time, we have no access to previous perception data and it's imperative to build on top of previously learned knowledge.

3.12 Reinforcement learning

Reinforcement learning is a machine learning paradigm based on rewarding desired behaviors and punishing undesired ones. In general, a reinforcement learning agent, the entity being trained, is able to perceive and interpret its environment, take actions and learn through trial and error. This kind of learning has applications in robotics, to dialog systems and to develope AI for videogames. *Deep reinforcement learning* applies deep learning to reinforcement learning problems

In reinforcement learning an agent interacts with an environment over a series of time steps. At each time step, the agent receives some *observations* from the environment and must choose an *action* that is subsequently transmitted back to the environment via some mechanism (sometimes called an *actuator*). Finally, the agent receives a *reward* from the environment. After that, the agent receives a subsequent observation and chooses a subsequent action, and so on. The behaviour of a reinforcement learning agent is governed by a *policy*, a function that maps observation of the environment to actions. The goal of reinforcement learning is to produce a "good" policy. A diagram of reinforcement learning is illustrated in Figure 3.6.

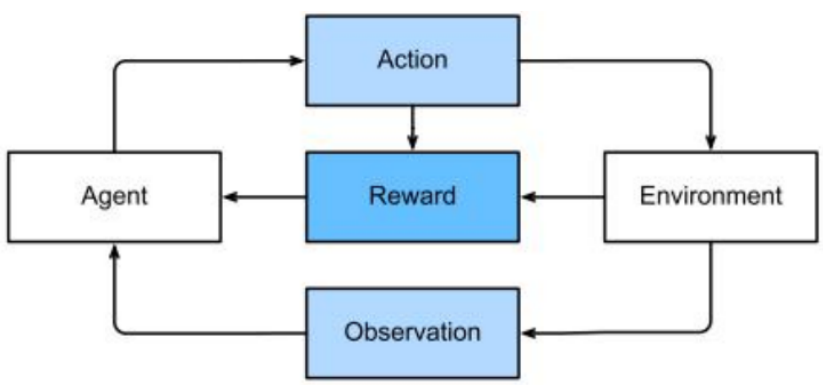


Figure 3.6: Reinforcement learning diagram. Source: [72].

More formally, there is an agent in a state s_t that execute an action a_t among the ones admissible in s_t. The agent operates in a environment E, which, in response to a_t and s_t returns the new state s_{t+1} and the reward $r_{t+1} \in \mathbb{R}$. The loop is repeated with (s_{t+1}, a_{t+1}). The goal is to find a policy $\pi(s, a) = \Pr(a_t = a \mid s_t = s)$ that maximize a function of the rewards. A typical function to maximize, at time t, is the *discounted future reward*:

$$\sum_{i=t}^{+\infty} \gamma^i r_{i+1}, \tag{3.1}$$

where r_{t+1} is the reward for transitioning from state s_t to s_{t+1}, $0 \leq \gamma < 1$ is the discount rate. γ is less than 1, so rewards in the distant future are weighted less than rewards in the immediate future.

In practice it is very difficult to obtain examples that are in the same time correct and representative of all situations in which the agent must work, therefore the classic supervised approach is not easily applicable and reinforcement learning is applied. For instance, there are numerous applications in robotics (e.g. object grasping, control, assembly, navigation). Q-learning is one of the best known and most used approaches. Its extension in deep learning called Deep Reinforcement Learning (DRL) is the basis of the successes achieved by Google DeepMind (Atari, AlphaGo).

3.13 Computer Vision applications

Deep learning has many applications in computer vision. A few of them are briefly introduced:

- Classification: Determines the class of the object.
- Classification + Localization: Determines the class of the object and its position in the image (bounding box).
- Object detection: For each object present, it determines the class and the position in the image (bounding box).
- Segmentation (already introduced in an unsupervised manner in 3.1.3): instead of a bounding box (rectangular) labels the individual pixels of the image to classes with indexes of the classes.

Figure 3.7 shows an example of each task.

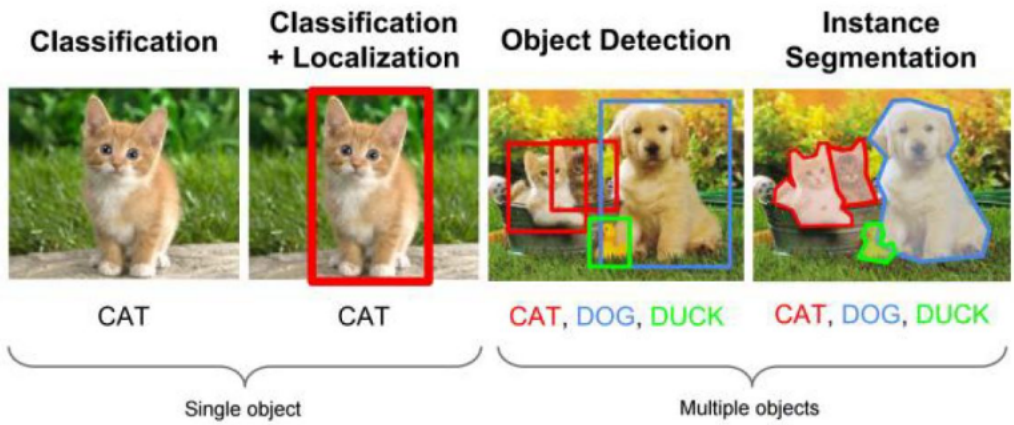


Figure 3.7: Applications of deep learning in CV. In classification and classification + localization we assume that there is only one object or in any case only one dominant object. On the other hand, in object detection and instance segmentation there may be multiple objects. Source: [72].

3.14 Parameters optimization

Typically, the behavior of a machine learning algorithm is regulated by a set of parameters (e.g. the weights of the connections in a neural network). Learning consists in determining the optimal value of these parameters. Given a training set S and a set of parameters Θ, the objective function f measures the optimality of the solution to be maximized:

$$\theta^* = \arg\max_{\theta \in \Theta} f(S, \theta), \tag{3.2}$$

or the error (loss) to be minimized:

$$\theta^* = \arg\min_{\theta \in \Theta} f(S, \theta), \tag{3.3}$$

In general, $f(S, \theta)$ can be optimized in two different ways:

- Explicitly, with methods that operates from his own mathematical definition. For example, the gradient of f is computed with respect to θ and imposed to be 0. The equation is then solved with respect to θ. A concrete example is the computation of $\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$ in ridge regression.

- Implicitly, using heuristics that modify parameters consistently with f (e.g. the genetic algorithm, heuristic based on natural selection).

3.15 Hyperparameters

Many algorithms require to define, before learning step, the value of the so-called *hyperparameters*, external configuration variables [118], which differs from the parameters of the model. Some examples of hyperparameters are:

- The number of neurons in a neural network.

- The number of neighbors k in a k-NN classifier.

- The degree of a polynomial used in a regression approach.

- The type of loss function.

Usually, we could proceed with a two-level approach in which for each "reasonable" value (e.g. learning rate $\eta > 0$) of the hyperparameters is performed a learning phase, and at the end of the procedure the hyperparameters that performed better are chosen.

A possible approach for choosing the values of hyperparameters is *grid search*. In this method, for each hyperparameter, a set of possible values is defined. The system is then evaluated on all combinations of values of all hyperparameters. This procedure can be very expensive. This approach means that we can choose each hyperparameter independently from the others.

3.16 Performance indicators

One possibility is to use directly the objective function f to quantify performance. But generally this is wrong (see *overfitting*) and it is preferred a more direct measure related to the semantic of the problem. For example, in a classification problem, we use as classification accuracy the percentage of correctly classified patterns, while the classification error is the complement. Informally:

$$\text{classification error} = \frac{\text{errors}}{\text{totals}} \tag{3.4}$$

$$\text{classification accuracy} = \frac{\text{correct}}{\text{totals}} \tag{3.5}$$

We now define the *true error*, the *empirical error* of a classifier h assuming each $(x, f(x))$ in the training set S is obtained by sampling $x \in \mathcal{X}$ is from a distribution $\mathcal{D}$ over $\mathcal{X}$ and then label x by the *oracle/target concept* f.

Definition 3.16.1 *The* true error $L_{\mathcal{D}}(h)$ *of the hypothesis h with respect to the target concept f and distribution $\mathcal{D}$ is the probability that h will missclassify an instance drawn at random according to $\mathcal{D}$:*

$$L_{\mathcal{D}}(h) = \underset{x \sim \mathcal{D}}{Pr}\left[f(x) \neq h(x)\right] \tag{3.6}$$

Definition 3.16.2 *The* empirical error $L_S(h)$ *of the hypothesis h with respect to S is the number of examples in S that h misclassifies:*

$$L_S(h) = \frac{|\{(x, f(x)) \in S \mid f(x) \neq h(x)\}|}{|S|} \tag{3.7}$$

However, in regression tasks those definition are not employable. In fact, we can measure the error of a classifier h on a set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n))\}$ (may be training, validation or test set) using the *Root Mean Square Error* (RMSE):

$$\mathrm{RMSE}(h) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}((h(\mathbf{x}_i) - y_i)^2} \tag{3.8}$$

Another important performance indicator in multiclass classification is the *confusion matrix*. It helps understanding how errors are distributed and which classes the model "confuses". Given a confusion matrix, the cell (i, j) reports the percentage of cases in which the system has predicted to be class j a pattern of true class i. Ideally, the matrix should be diagonal, meaning the model does not confuse any class. High values off the diagonal indicate concentrations of errors. Figure 3.8 shows a general diagram for a confusion matrix.

		Predicted		
		A_1	$\cdots$ A_j $\cdots$	A_n
Actual	A_1	N_{11}	N_{1j}	N_{1n}
	$\vdots$ A_i $\vdots$	N_{i1}	$\vdots$ $\cdots$ N_{ij} $\cdots$ $\vdots$	N_{in}
	A_n	N_{n1}	N_{nj}	N_{nn}

Figure 3.8: Diagram of a confusion matrix for a model, with learned classes $A_1, A_2, \ldots A_n$. The element N_{ij} represents the number of samples actually belonging to class A_i but classified by the model as class A_j. Source: [32].

3.17 Overfitting

Our goal is to maximize the accuracy on the *validation set*. By *generalization* we mean the ability to transfer the high accuracy achieved on the training set to the validation set. If the degrees of freedom of the classifier are excessive, high accuracy is achieved on training set, but not on validation set (poor generalization). In this case we speak of *overfitting*. This situation occurs very easily when the training set is small in size.

In iterative training processes, typically after a certain number of iterations, the accuracy on the validation set no longer increases (and may begin to decrease) due to overfitting. By monitoring the progress, training can be stopped at the ideal point, as shown in Figure 3.9. The degrees of freedom of the classifier must not be excessive, but adequate in relation to the complexity of the problem. It is a good idea to start with a few degrees of freedom (controllable through hyperparameters) and gradually increase them by monitoring accuracy on training and validation set.

3.18 Open Set

In many real cases, however, the patterns to be classified may belong to one of the known classes or none of these, the *open set*. *Open-set learning* is a problem of handling "unknown classes" that are not contained in the training dataset, whereas traditional classifiers assume that only known classes appear in the test environment [128]. There are two possible approaches:

- Another fictitious class "the rest of the world" is added to the classes and the so-called *negative examples* are added to the training set.

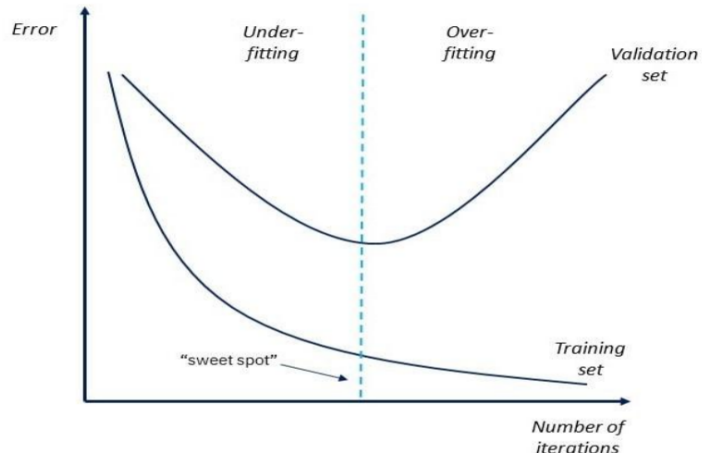


Figure 3.9: We can stop the training at the "sweet spot". Source: [72].

- We explicitly allow the system not to assign a class to a pattern. To this end, a threshold is defined and the pattern is assigned to the most probable class only when the probability that the pattern belongs to such class is higher than the threshold.

Let us consider a face detection problem with two classes, where these two classes correspond to positive (real class) and negative (rest of the world fictitious class) examples. We want to classify a portion (bounding box) of the image.

- Using the first solution, the positive examples are all portions of the image (windows) in which faces appear. Instead, the negative examples are windows (chosen at random) in which no face appear.
- If we employ the second solution, we can consider only the positive class and a system with a threshold t capable of calculating the probability p of a pattern belonging to an arbitrary class. This means that a pattern is classified as positive if $p > t$, and as negative if not.

It is possible to define performance indicators for classification with an open set. There are the following possible outcomes:

- A *true positive* is an outcome where the model correctly predicts the positive class.
- A *true negative* is an outcome where the model correctly predicts the negative class.
- A *false positive* is an outcome where the model incorrectly predicts the positive class.
- A *false negative* is an outcome where the model incorrectly predicts the negative class.

Consider a set of patterns. Let FP be the number of false positive outcomes and FN the false negative outcomes. Let N_p be the number of patterns belonging to the positive class and N_n be the number of patterns belonging to the negative class. We define the following rates:

$$\text{False Positive Rate (FPR)} = \frac{FP}{N_n}, \tag{3.9}$$

$$\text{False Negative Rate (FNR)} = \frac{FN}{N_p}, \tag{3.10}$$

Let TP be the number of true positive outcomes. Similarly, we also define:

$$\text{True Positive Rate (TPR)} = \frac{TP}{N_p}, \tag{3.11}$$

If we consider a threshold t, we find out that Equations (3.10) and (3.9) are functions of t, as described in Figure 3.10. In fact, high restrictive thresholds reduce false positives but increase the false negatives. On the other hand, tolerant thresholds reduce false negatives but increase the false positives.

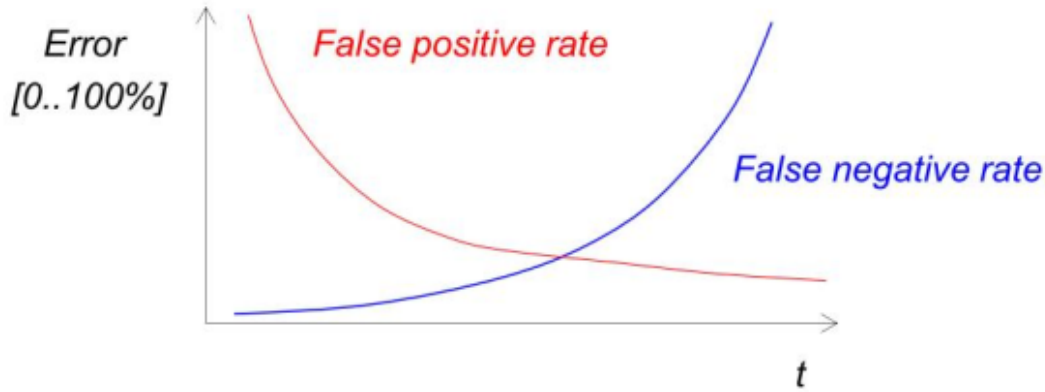


Figure 3.10: FPR and FNR as function of t. Source: [72].

We define the *accuracy* as [72]:

$$\text{Accuracy} = \frac{TP + TN}{N_n + N_p}. \tag{3.12}$$

3.18.1 DET and ROC curves

It is possible to plot a rate as a function of another by varying the threshold t. Two important such plots are *Detection Error Tradeoff* (DET) and *Receiver Operating Characteristic* (ROC). The two typical curves are reported in Figure 3.11.

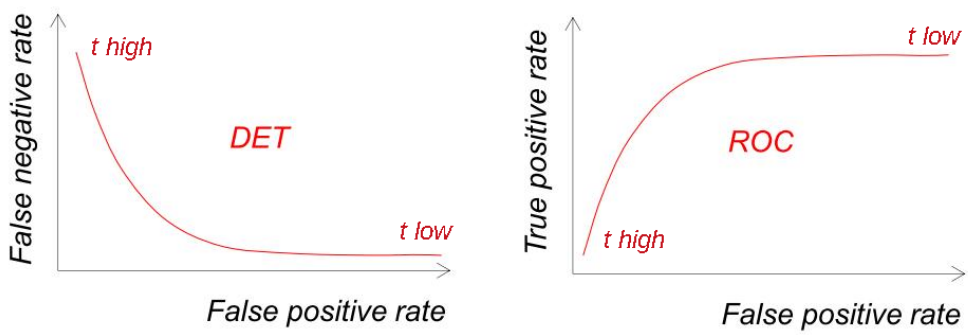


Figure 3.11: DET and ROC curves with varying t. Source: [72].

In particular, the ROC curve is often used as performance indicator. The *Area Under the Curve* (AUC) is a measure of accuracy for classification. The greater the AUC, the greater the discriminating power of the approach is. In other words, if AUC=1 the classifier has FPR=0 and TPR=1. This indicator is expressed graphically in Figure 3.12.

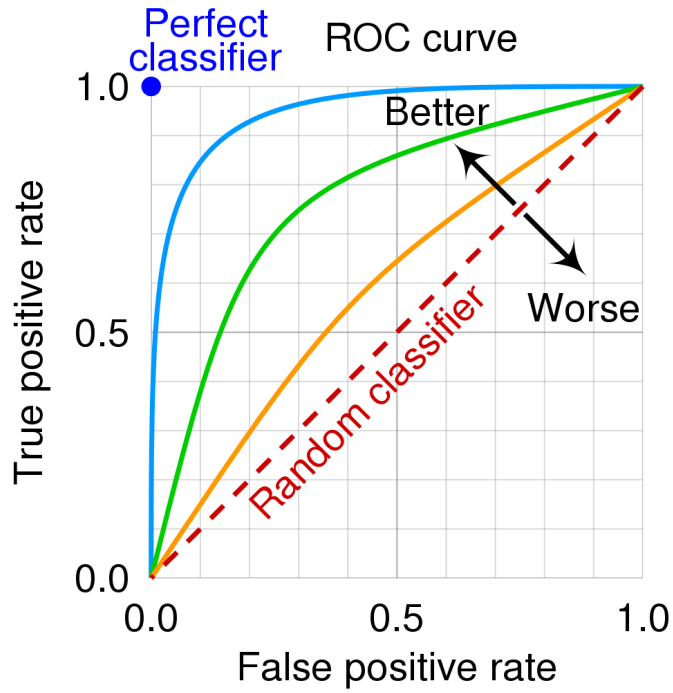


Figure 3.12: The ROC space for a "better" and "worse" classifier. Source: [92].

3.18.2 Precision and Recall

Precision and *recall* are defined as:

$$\text{Precision} = \frac{TP}{TP + FP}, \tag{3.13}$$

$$\text{Recall} = \frac{TP}{TP + FN}. \tag{3.14}$$

It is easy to see that Recall=TPR. Figure 3.13 explains graphically these two quantities.

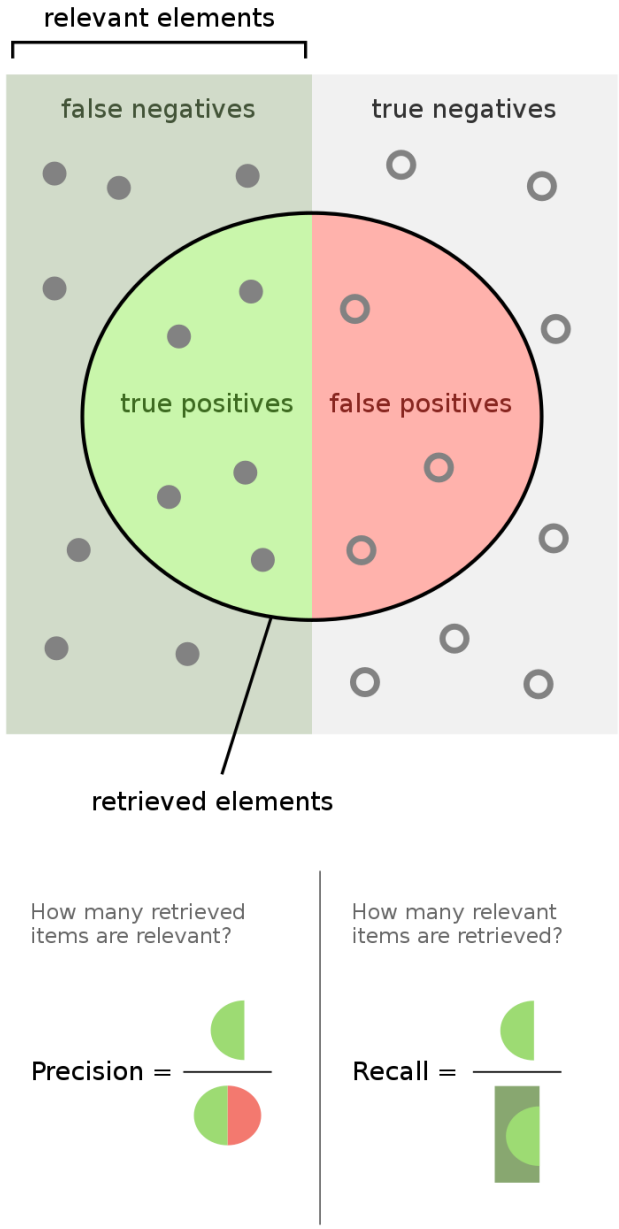


Figure 3.13: Recall and precision. The figure assumes some retrieval problem, in which the goal is to select a group of elements from a set of relevant elements. This can be naturally extended to binary classification tasks. For instance, consider a computer program for recognizing dogs (the relevant element) in a digital photograph. Upon processing a picture which contains ten cats and twelve dogs, the program identifies eight dogs. Of the eight elements identified as dogs, only five actually are dogs (true positives), while the other three are cats (false positives). Seven dogs were missed (false negatives), and seven cats were correctly excluded (true negatives). The program's precision is then 5/8 (true positives / selected elements) while its recall is 5/12 (true positives / relevant elements). Source: [86].

We define the F_1 *score* as:

$$F_1 = \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{3.15}$$

F_1 score is often used in scientific papers.

3.18.3 McNemar's test

McNemar's test is used to compare the predictive accuracy of two models. We formulate two possible hypothesis, and we have to pick one of them:

- The *null hypothesis*, stating that the two models are the same.
- The *alternative hypothesis*, stating that the two models are not the same.

Given two models, our goal is to pick one of the two hypothesis, i.e. determining if the two models are the same or not. Consider the following image.

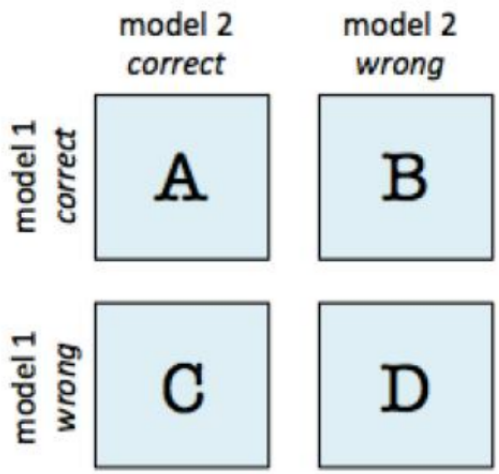


Figure 3.14: Source: [72].

The McNemar test statistic is:

$$\chi^2 = \frac{(|B - C| - 1)^2}{B + C}. \tag{3.16}$$

Given χ^2, we can compute the estimated p-value, with one degree of freedom [104]. If the $p < \alpha$, with $\alpha = 0.05$ confidence value, this provides sufficient evidence to reject the null hypothesis, in favour of the alternative hypothesis, which would mean that the marginal proportions are significantly different from each other [68, 104].

However, if we have a small sample size (e.g. $B + D < 25$), the χ^2 may not be well-approximated by the chi-squared distribution. In this case we need to compute the exact p-value instead of χ^2. The exact p-value can be computed as follows:

$$p = 2 \sum_{i=B}^{n} \binom{n}{i} 0.5^i (1 - 0.5)^{n-i}, \tag{3.17}$$

where $n = B + C$. If the $p < \alpha$, with $\alpha = 0.05$ confidence value, we reject the null hypothesis.

3.18.4 Average precision in object detection

Another formulation of precision is widely used in object detection. In fact, in this task there are several possible types of errors, including: object not found, wrong class, wrong or inaccurate bounding box. How to quantify everything with a scalar to make systems comparable? We can use the *Average Precision* (AP), a sort of AUC computed on a recall-precision curve for all objects of a single class in the whole dataset. The *medium Average Precision* (mAP) is the the average of the AP over all classes. *Intersection over Union* (IoU) measures the overlap between 2 boundaries:

$$\text{IoU} = \frac{\text{area of overlap}}{\text{area of union}}. \tag{3.18}$$

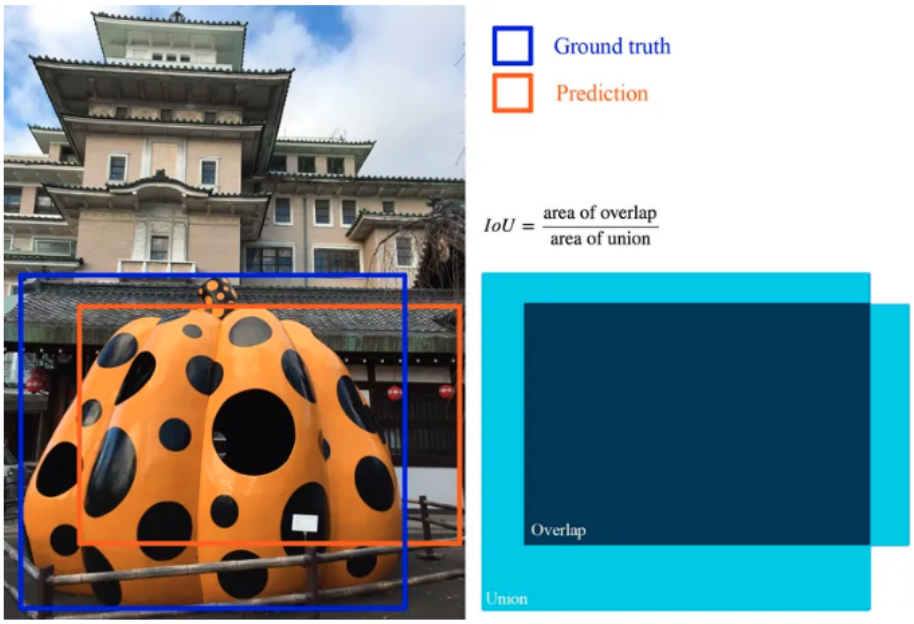


Figure 3.15: Definition of IoU. Source: [65].

We use that to measure how much our predicted boundary overlaps with the ground truth (the real object boundary). IoU is displayed in Figure 3.15. To estimate TP and FP (with a certain confidence) a correct prediction is considered when the class is correct and the IoU of the two bounding boxes is greater than a given threshold α (e.g. 0.5). Figure 3.16 shows an example of this procedure.

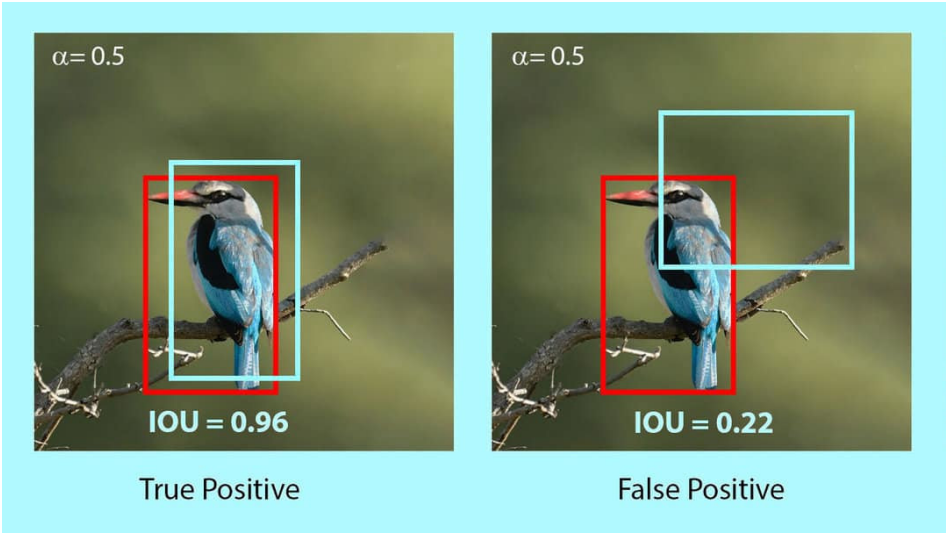


Figure 3.16: Example of false positive and false negative using IoU. Source: [54].

As already seen in Section 3.16, the confusion matrix is a common performance indicator for multiclass classification. It is possible to employ a 2×2 confusion matrix highlighting true negatives, true positives, false positives and false negatives. The binary classifier of the example in Figure 3.17 has digit 5 as a positive class and all other digits as a negative class.

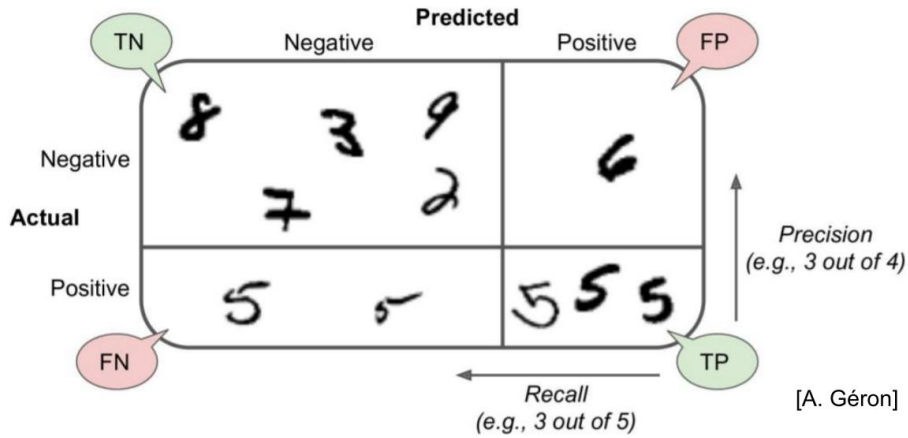


Figure 3.17: Confusion matrix of a binary classifier. Source: [72].

3.19 Multilabel performance indicators

Consider a test set $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_m, \mathbf{y}_m)\}$, where $\mathbf{x}_i \in \mathcal{X} = \mathbb{R}^d$ are the feature vectors and $\mathbf{y}_i \in \mathcal{Y} = \{-1, 1\}^q$ are the corresponding ground-truth label vector (i.e. there are q possible labels/classes). Consider a multilabel classifier $h : \mathbb{R}^d \to \mathbb{R}^q$ [126], returning a vector of scores. The predicted classes are the one with the highest scores (according to some not specified criterion). Let $\hat{\mathbf{y}}_i \in \{-1, 1\}^q$ be the predicted label vector for $\mathbf{x}_i$.

- *One error* evaluates the fraction that the label with the top-ranked predicted by the instance does not belong to its ground-truth relevant label set. Formally:

$$\text{one-error}(h) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{1}\left[y_{ij^*} = -1\right], \tag{3.19}$$

  where $j^* = \arg\max_j h(\mathbf{x}_i)$ and $\mathbb{1}[z]$ returns 1 if condition z holds, 0 otherwise. An equivalent formulation is reported by [126]:

$$\text{one-error}(h) = \frac{1}{m} \sum_{i=1}^{m} \underset{k \in \{1,\ldots,q\}}{\arg\max} \{h(\mathbf{x}_i)\} \notin \{j \mid y_{ij} = 1\}. \tag{3.20}$$

- *Hamming loss* evaluates the fraction of instance labels pairs which have been misclassified. Formally:

$$\text{hloss}(h) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{q} \sum_{j=1}^{q} \mathbb{1}[y_{ij} \neq \hat{y}_{ij}]. \tag{3.21}$$

3.20 Training, Validation and Test

The *training set* is the set of patterns used to train the system and finding the optimal value for the parameters (e.g. apply ERM). The *validation set* is the set the patterns on which to set the hyperparameters (external cycle). The *test set* is the set of patterns on which to evaluate the final system performance, i.e. to estimate the generalization error. In machine learning benchmarks the split of the patterns into training, validation and test set is often predefined to make the results comparable. It is crucial to underline that we should never use the test set if not for final evaluation of the model nor use it to make any choice relating to the model.

3.21 Convergence

The first aim to be pursued during the training is the convergence on the training set. Let's consider a classifier whose training involves an iterative process. Convergence occurs when:

- The training loss (output of the loss function) has a decreasing trend.
- The accuracy (on the test set) has an increasing trend.

We need to consider the following cases:

- If loss does not decrease (or fluctuates significantly) the system does not converge. The possible causes are:
  - The optimization method is not effective.
  - The hyperparameters are out of range.
  - The learning rate is inadequate.
  - There are implementation errors.
- If the training loss decreases but the accuracy does not increase, probably a wrong loss-function has been chosen.
- If the accuracy does not approach 100% on the training set, the possible causes are:
  - Degrees of freedom of the classifier are not sufficient to manage the complexity of the problem.
  - Features not suitable for the problem .
  - Low number of training patterns.
  - The problem is very complex.

3.22 Comparison tests

3.22.1 Wilcoxon signed-rank test

The *Wilcoxon signed-rank test* ranks the differences in performances of two classifiers for each dataset ignoring the signs, and compares the ranks for the positive and the negative differences. Consider the null and the alternative hypothesis as in Section 3.18.3. Let d_i be the difference between the performance scores of the two classifiers on the i-th dataset out of N datasets. We assign a *rank* to each dataset in the following way:

- The differences are ranked according to their absolute values. For example, if we get $d_1 = 0.003$, $d_2 = 0.0$, $d_3 = -0.001$, we have ranks 3, 1, 2, respectively.
- Average ranks are assigned in case of ties. That is, if rank 10 and 11 have the same observed differences both are assigned rank 10.5 [50].
- Ranks of $d_i = 0$ are split evenly among the sums (see second sum of Equation (3.22) and (3.23)), if there is an odd number of them, one is ignored.

We define:

$$R^+ = \sum_{d_i>0} \text{rank}(d_i) + \frac{1}{2} \sum_{d_i=0} \text{rank}(d_i), \tag{3.22}$$

$$R^- = \sum_{d_i<0} \text{rank}(d_i) + \frac{1}{2} \sum_{d_i=0} \text{rank}(d_i), \tag{3.23}$$

$$T = \min(R^+, R^-). \tag{3.24}$$

In general, we reject the null hypothesis if $T \leq \text{criticalValue}(\alpha, N)$, representing a critical value. These critical values are often put in a table as in Figure 3.18.

Two-Sided Test α	0.1	0.05	0.02	0.01
One-Sided Test α	0.05	0.025	,01	0.005
n				
5	1			
6	2	1		
7	4	2	0	
8	6	4	2	0
9	8	6	3	2
10	11	8	5	3
11	14	11	7	5
12	17	14	10	7
13	21	17	13	10
14	26	21	16	13
15	30	25	20	16
16	36	30	24	19
17	41	35	28	23
18	47	40	33	28
19	54	46	38	32
20	60	52	43	37
21	68	59	49	43
22	75	66	56	49
23	83	73	62	55
24	92	81	69	61
25	101	90	77	68

Figure 3.18: Example of table containing the critical values of T in function of the number of datasets N and the confidence α. Note that we are considering two-sided test. Source: Unavailable.

Suppose $\alpha = 0.05$. For a larger number of data, the statistic (z-score)

$$z = \frac{T - \frac{1}{4}N(N+1)}{\sqrt{\frac{1}{24}N(N+1)(2N+1)}} \tag{3.25}$$

is distributed approximately normally, and the null hypothesis can be rejected if $z < -1.96$ or $z > 1.96$.

Example

Consider the performance of the two classifiers C4.5 and C4.5+m:

	C4.5	C4.5+m	difference	rank
adult (sample)	0.763	0.768	+0.005	3.5
breast cancer	0.599	0.591	−0.008	7
breast cancer wisconsin	0.954	0.971	+0.017	9
cmc	0.628	0.661	+0.033	12
ionosphere	0.882	0.888	+0.006	5
iris	0.936	0.931	−0.005	3.5
liver disorders	0.661	0.668	+0.007	6
lung cancer	0.583	0.583	0.000	1.5
lymphography	0.775	0.838	+0.063	14
mushroom	1.000	1.000	0.000	1.5
primary tumor	0.940	0.962	+0.022	11
rheum	0.619	0.666	+0.047	13
voting	0.972	0.981	+0.009	8
wine	0.957	0.978	+0.021	10

The initial rankings, before considering the ties, are: $\text{rank}(d_{\text{lung cancer}}) = 1$, $\text{rank}(d_{\text{mushroom}}) = 2$, $\text{rank}(d_{\text{adult(sample)}}) = 3$, $\text{rank}(d_{\text{iris}}) = 4$, $\text{rank}(d_{\text{ionosphere}}) = 5$, $\text{rank}(d_{\text{liver disorders}}) = 6$, $\text{rank}(d_{\text{breast cancer}}) = 7$, $\text{rank}(d_{\text{voting}}) = 8$, $\cdots$. Since we have a tie between rank 1 and 2, we re-rank them as $\frac{1+2}{2} = 1.5$,

obtaining $\text{rank}(d_{\text{lung cancer}}) = \text{rank}(d_{\text{mushroom}}) = 1.5$. In the same way, since we have a tie between rank 3 and 4, we re-rank them as $\frac{3+4}{2}$, obtaining $\text{rank}(d_{\text{adult(sample)}}) = \text{rank}(d_{\text{iris}}) = 3.5$. The other rankings are unchanged because there are not remaining ties.

We now compute R^+, R^- and T.

$$R^+ = 3.5 + 9 + 12 + 5 + 6 + 14 + 11 + 13 + 8 + 10 + \frac{1}{2}(1.5 + 1.5) = 93, \tag{3.26}$$

$$R^- = 7 + 3.5 + 1.5 = 12. \tag{3.27}$$

Then $T = 12$. According to the table of critical values in Figure 3.18, for a confidence level of $\alpha = 0.05$ and $N = 14$ datasets, the difference between the classifiers is significant if $T \leq 21$. We therefore reject the null hypothesis.

3.22.2 Friedman test for comparisons of multiple classfiers

Consider the procedure of Section 3.22.1. If we have 7 classifiers, we should perform $\binom{7}{2} = 21$ paired tests. When so many tests are made, a certain proportion of the null hypotheses is rejected due to random chance, so listing them makes little sense.

Suppose we have N datasets and k algorithms (i.e. classifiers, models). We rank the performances of each algorithms on every dataset. Note that this is different from before, since we were ranking the differences of performances, not the actual performances. Let r_i^j be the rank of the j-th algorithm on the i-th dataset. The *Friedman test* compares the average ranks of the j-th algorithm on the i-th dataset:

$$R_j = \frac{1}{N} \sum_{i=1}^{N} r_i^j. \tag{3.28}$$

In this case, the null hypothesis states that all the k algorithms are equivalent. Under this hypothesis, we have

$$R_1 = R_2 = \cdots = R_k. \tag{3.29}$$

In addition, the Friedman statistic is distributed according to χ_F^2 with $k-1$ degrees of freedom with $N > 10$, $k > 5$ big enough, where:

$$\chi_F^2 = \frac{12N}{k(k+1)} \left(\sum_{j=1}^{k} R_j^2 - \frac{k(k+1)^2}{4} \right). \tag{3.30}$$

We compute the p-value and if $p < \alpha$, with $\alpha = 0.05$, we reject the null hypothesis.

A better statistic is the following:

$$F_F = \frac{(N-1)\chi_F^2}{N(k-1) - \chi_F^2}, \tag{3.31}$$

which is distributed according to the F-distribution with $k-1$ and $(k-1)(N-1)$ degrees of freedom. We compare F_F with the critical value and reject the null hypothesis accordingly.

Example

Consider the performance of the four following classifiers:

	C4.5	C4.5+m	C4.5+cf	C4.5+m+cf
adult (sample)	0.763 (4)	0.768 (3)	0.771 (2)	0.798 (1)
breast cancer	0.599 (1)	0.591 (2)	0.590 (3)	0.569 (4)
breast cancer wisconsin	0.954 (4)	0.971 (1)	0.968 (2)	0.967 (3)
cmc	0.628 (4)	0.661 (1)	0.654 (3)	0.657 (2)
ionosphere	0.882 (4)	0.888 (2)	0.886 (3)	0.898 (1)
iris	0.936 (1)	0.931 (2.5)	0.916 (4)	0.931 (2.5)
liver disorders	0.661 (3)	0.668 (2)	0.609 (4)	0.685 (1)
lung cancer	0.583 (2.5)	0.583 (2.5)	0.563 (4)	0.625 (1)
lymphography	0.775 (4)	0.838 (3)	0.866 (2)	0.875 (1)
mushroom	1.000 (2.5)	1.000 (2.5)	1.000 (2.5)	1.000 (2.5)
primary tumor	0.940 (4)	0.962 (2.5)	0.965 (1)	0.962 (2.5)
rheum	0.619 (3)	0.666 (2)	0.614 (4)	0.669 (1)
voting	0.972 (4)	0.981 (1)	0.975 (2)	0.975 (3)
wine	0.957 (3)	0.978 (1)	0.946 (4)	0.970 (2)
average rank	3.143	2.000	2.893	1.964

Note that the rank r_i^j is placed inside the round parenthesis after the performance. We have $k = 4$ and $N = 14$. Therefore, by looking at the average ranks:

$$\chi_F^2 = \frac{12 \cdot 14}{4 \cdot 5} \left(3.143^2 + 2.0^2 + 2.893^2 + 1.964^2 - \frac{4 \cdot 5^2}{4}) \right) = 9.28, \tag{3.32}$$

$$F_F = \frac{13 \cdot 9.28}{13 \cdot 3 - 9.28} = 3.69. \tag{3.33}$$

With $k = 4$ and $N = 14$, F_F has $4 - 1 = 3$ and $(4 - 1) \cdot (14 - 1) = 39$ degrees of freedom. The critical value for these values and $\alpha = 0.05$ is 2.85 (Figure 3.19), then we reject the null hypothesis.

	F-table of Critical Values of α = 0.05 for F(df1, df2)																		
	DF1=1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**12**	**15**	**20**	**24**	**30**	**40**	**60**	**120**	**∞**
DF2=1	161.45	199.50	215.71	224.58	230.16	233.99	236.77	238.88	240.54	241.88	243.91	245.95	248.01	249.05	250.10	251.14	252.20	253.25	254.31
2	18.51	19.00	19.16	19.25	19.30	19.33	19.35	19.37	19.38	19.40	19.41	19.43	19.45	19.45	19.46	19.47	19.48	19.49	19.50
3	10.13	9.55	9.28	9.12	9.01	8.94	8.89	8.85	8.81	8.79	8.74	8.70	8.66	8.64	8.62	8.59	8.57	8.55	8.53
4	7.71	6.94	6.59	6.39	6.26	6.16	6.09	6.04	6.00	5.96	5.91	5.86	5.80	5.77	5.75	5.72	5.69	5.66	5.63
5	6.61	5.79	5.41	5.19	5.05	4.95	4.88	4.82	4.77	4.74	4.68	4.62	4.56	4.53	4.50	4.46	4.43	4.40	4.37
6	5.99	5.14	4.76	4.53	4.39	4.28	4.21	4.15	4.10	4.06	4.00	3.94	3.87	3.84	3.81	3.77	3.74	3.70	3.67
7	5.59	4.74	4.35	4.12	3.97	3.87	3.79	3.73	3.68	3.64	3.57	3.51	3.44	3.41	3.38	3.34	3.30	3.27	3.23
8	5.32	4.46	4.07	3.84	3.69	3.58	3.50	3.44	3.39	3.35	3.28	3.22	3.15	3.12	3.08	3.04	3.01	2.97	2.93
9	5.12	4.26	3.86	3.63	3.48	3.37	3.29	3.23	3.18	3.14	3.07	3.01	2.94	2.90	2.86	2.83	2.79	2.75	2.71
10	4.96	4.10	3.71	3.48	3.33	3.22	3.14	3.07	3.02	2.98	2.91	2.85	2.77	2.74	2.70	2.66	2.62	2.58	2.54
11	4.84	3.98	3.59	3.36	3.20	3.09	3.01	2.95	2.90	2.85	2.79	2.72	2.65	2.61	2.57	2.53	2.49	2.45	2.40
12	4.75	3.89	3.49	3.26	3.11	3.00	2.91	2.85	2.80	2.75	2.69	2.62	2.54	2.51	2.47	2.43	2.38	2.34	2.30
13	4.67	3.81	3.41	3.18	3.03	2.92	2.83	2.77	2.71	2.67	2.60	2.53	2.46	2.42	2.38	2.34	2.30	2.25	2.21
14	4.60	3.74	3.34	3.11	2.96	2.85	2.76	2.70	2.65	2.60	2.53	2.46	2.39	2.35	2.31	2.27	2.22	2.18	2.13
15	4.54	3.68	3.29	3.06	2.90	2.79	2.71	2.64	2.59	2.54	2.48	2.40	2.33	2.29	2.25	2.20	2.16	2.11	2.07
16	4.49	3.63	3.24	3.01	2.85	2.74	2.66	2.59	2.54	2.49	2.42	2.35	2.28	2.24	2.19	2.15	2.11	2.06	2.01
17	4.45	3.59	3.20	2.96	2.81	2.70	2.61	2.55	2.49	2.45	2.38	2.31	2.23	2.19	2.15	2.10	2.06	2.01	1.96
18	4.41	3.55	3.16	2.93	2.77	2.66	2.58	2.51	2.46	2.41	2.34	2.27	2.19	2.15	2.11	2.06	2.02	1.97	1.92
19	4.38	3.52	3.13	2.90	2.74	2.63	2.54	2.48	2.42	2.38	2.31	2.23	2.16	2.11	2.07	2.03	1.98	1.93	1.88
20	4.35	3.49	3.10	2.87	2.71	2.60	2.51	2.45	2.39	2.35	2.28	2.20	2.12	2.08	2.04	1.99	1.95	1.90	1.84
21	4.32	3.47	3.07	2.84	2.68	2.57	2.49	2.42	2.37	2.32	2.25	2.18	2.10	2.05	2.01	1.96	1.92	1.87	1.81
22	4.30	3.44	3.05	2.82	2.66	2.55	2.46	2.40	2.34	2.30	2.23	2.15	2.07	2.03	1.98	1.94	1.89	1.84	1.78
23	4.28	3.42	3.03	2.80	2.64	2.53	2.44	2.37	2.32	2.27	2.20	2.13	2.05	2.01	1.96	1.91	1.86	1.81	1.76
24	4.26	3.40	3.01	2.78	2.62	2.51	2.42	2.36	2.30	2.25	2.18	2.11	2.03	1.98	1.94	1.89	1.84	1.79	1.73
25	4.24	3.39	2.99	2.76	2.60	2.49	2.40	2.34	2.28	2.24	2.16	2.09	2.01	1.96	1.92	1.87	1.82	1.77	1.71
26	4.23	3.37	2.98	2.74	2.59	2.47	2.39	2.32	2.27	2.22	2.15	2.07	1.99	1.95	1.90	1.85	1.80	1.75	1.69
27	4.21	3.35	2.96	2.73	2.57	2.46	2.37	2.31	2.25	2.20	2.13	2.06	1.97	1.93	1.88	1.84	1.79	1.73	1.67
28	4.20	3.34	2.95	2.71	2.56	2.45	2.36	2.29	2.24	2.19	2.12	2.04	1.96	1.91	1.87	1.82	1.77	1.71	1.65
29	4.18	3.33	2.93	2.70	2.55	2.43	2.35	2.28	2.22	2.18	2.10	2.03	1.94	1.90	1.85	1.81	1.75	1.70	1.64
30	4.17	3.32	2.92	2.69	2.53	2.42	2.33	2.27	2.21	2.16	2.09	2.01	1.93	1.89	1.84	1.79	1.74	1.68	1.62
40	4.08	3.23	2.84	2.61	2.45	2.34	2.25	2.18	2.12	2.08	2.00	1.92	1.84	1.79	1.74	1.69	1.64	1.58	1.51
60	4.00	3.15	2.76	2.53	2.37	2.25	2.17	2.10	2.04	1.99	1.92	1.84	1.75	1.70	1.65	1.59	1.53	1.47	1.39
120	3.92	3.07	2.68	2.45	2.29	2.18	2.09	2.02	1.96	1.91	1.83	1.75	1.66	1.61	1.55	1.50	1.43	1.35	1.25
∞	3.84	3.00	2.60	2.37	2.21	2.10	2.01	1.94	1.88	1.83	1.75	1.67	1.57	1.52	1.46	1.39	1.32	1.22	1.00

Figure 3.19: F-table. Source: Unavailable.

3.22.3 Post-hoc test

If the between-classifiers variability is significantly larger than the error variability, we can reject the null-hypothesis (e.g. using Friedman test) and conclude that there are some differences between the classifiers. In this case, we can proceed with a *post-hoc test* to find out which classifiers actually differ [31]. We now will see three post-hoc tests: Nemenyi, Bonferroni-Dunn and Holm tests.

3.22.4 Nemenyi test

If the null hypothesis is rejected, we can proceed with a post-hoc *Nemenyi test*. The performance of two classifiers i and j is significantly different if the corresponding average ranks differ by at least the critical difference CD (i.e. $|R_i - R_j| > CD$):

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}}, \tag{3.34}$$

where α is the confidence value, and the critical value q_α is computed through this table:

#classifiers	2	3	4	5	6	7	8	9	10
$q_{0.05}$	1.960	2.343	2.569	2.728	2.850	2.949	3.031	3.102	3.164
$q_{0.10}$	1.645	2.052	2.291	2.459	2.589	2.693	2.780	2.855	2.920

Figure 3.20: Source: [31].

3.22.5 Bonferroni-Dunn test

When all classifiers are compared with a control classifier (e.g. we introduce a new classifier R_0 and we want to compare it with the already existing ones) we can use the Bonferroni correction or similar procedures. Suppose we want to compare the i-th classifier with the remaining $k-1$ ones. In this way, we only make $k-1$ comparison. With these kinds of methods we compare the i-th and j-th classifier by computing the z-score as follows:

$$z = \frac{(R_i - R_j)}{\sqrt{\frac{k(k+1)}{6N}}}. \tag{3.35}$$

The z value is used to find the corresponding p-value from the table of normal distribution, which is then compared with an appropriate α. The tests differ in the way they adjust the value of α to compensate for multiple comparisons.

The *Bonferroni-Dunn test* applies this procedure and controls the family-wise error by dividing α by $k-1$. The alternative way to compute the same test is to calculate the CD using Equation (3.34), but using the critical values for $\alpha/(k-1)$, reported in the following table:

#classifiers	2	3	4	5	6	7	8	9	10
$q_{0.05}$	1.960	2.241	2.394	2.498	2.576	2.638	2.690	2.724	2.773
$q_{0.10}$	1.645	1.960	2.128	2.241	2.326	2.394	2.450	2.498	2.539

Figure 3.21: Source: [31].

Note that for $k = 2$ the two tables have the same entry, since we divide by $k - 1 = 1$.

3.22.6 Holm test

Suppose we perform the $k-1$ comparison using Equation (3.35) and compute the corresponding p-value. Let $p_1, p_2, \ldots$ the ordered p-values so that $p_1 \leq p_2 \leq \cdots \leq p_{k-1}$. In the *Holm test* we compare each p_i with $\alpha/(k-i)$ but in order of the tests: if $p_1 \leq \alpha/(k-1)$, the corresponding hypothesis is rejected and we are allowed to compare p_2 with $\alpha/(k-2)$. If the second hypothesis is rejected, the test proceeds with the third and so on. As soon as a certain null hypothesis cannot be rejected, all the remaining hypotheses are retained as well.

3.22.7 Examples of post-hoc tests

Nemenyi test

Consider the same performances seen before.

	C4.5	C4.5+m	C4.5+cf	C4.5+m+cf
adult (sample)	0.763 (4)	0.768 (3)	0.771 (2)	0.798 (1)
breast cancer	0.599 (1)	0.591 (2)	0.590 (3)	0.569 (4)
breast cancer wisconsin	0.954 (4)	0.971 (1)	0.968 (2)	0.967 (3)
cmc	0.628 (4)	0.661 (1)	0.654 (3)	0.657 (2)
ionosphere	0.882 (4)	0.888 (2)	0.886 (3)	0.898 (1)
iris	0.936 (1)	0.931 (2.5)	0.916 (4)	0.931 (2.5)
liver disorders	0.661 (3)	0.668 (2)	0.609 (4)	0.685 (1)
lung cancer	0.583 (2.5)	0.583 (2.5)	0.563 (4)	0.625 (1)
lymphography	0.775 (4)	0.838 (3)	0.866 (2)	0.875 (1)
mushroom	1.000 (2.5)	1.000 (2.5)	1.000 (2.5)	1.000 (2.5)
primary tumor	0.940 (4)	0.962 (2.5)	0.965 (1)	0.962 (2.5)
rheum	0.619 (3)	0.666 (2)	0.614 (4)	0.669 (1)
voting	0.972 (4)	0.981 (1)	0.975 (2)	0.975 (3)
wine	0.957 (3)	0.978 (1)	0.946 (4)	0.970 (2)
average rank	3.143	2.000	2.893	1.964

Figure 3.22: Source: [31].

We perform a Nemenyi test with $\alpha = 0.05$. From the table above the critical value is 2.569 and the corresponding CD is

$$CD = 2.569\sqrt{\frac{4 \cdot 5}{6 \cdot 14}} = 1.25. \tag{3.36}$$

Since the difference between the best (C4.5+m+cf with rank 1.964) and the worst classifier (C4.5 with rank 3.143) is smaller than that, we conclude that the post-hoc test is not powerful enough to detect any significant differences between the algorithms.

On the other hand, if we perform a Nemenyi test with $\alpha = 0.10$, we have

$$CD = 2.291\sqrt{\frac{4 \cdot 5}{6 \cdot 14}} = 1.12. \tag{3.37}$$

We can identify two groups of algorithms: the performance of pure C4.5 is significantly worse than that of C4.5+m and C4.5+m+cf. We cannot tell which group C4.5+cf belongs to, because $2.893 - 2.000 < CD$ and $2.893 - 1.964 < CD$.

Bonferroni-Dunn test

Consider now $\alpha = 0.05$. According to the table in Section 3.22.5 $q_{0.05} = 2.394$, so:

$$CD = 2.394\sqrt{\frac{4 \cdot 5}{5 \cdot 14}} = 1.16. \tag{3.38}$$

We now compare C4.5 with the other 3 models, thus obtaining the following information:

- C4.5+m+cf performs significantly better than C4.5, since $3.143 - 1.964 = 1.179 > 1.16$.
- C4.5+cf does not, since $3.143 - 2.893 = 0.25 < 1.15$.
- C4.5+m is just below the critical difference CD, but close to it, since $3.143 - 2.0 = 1.143 < 1.16$.

Holm test

By Equation (3.35) and Figure 3.25, we have:

i	classifier	$z = (R_0 - R_i)/SE$	p	α/i
1	C4.5+m+cf	$(3.143 - 1.964)/0.488 = 2.416$	0.016	0.017
2	C4.5+m	$(3.143 - 2.000)/0.488 = 2.342$	0.019	0.025
3	C4.5+cf	$(3.143 - 2.893)/0.488 = 0.512$	0.607	0.050

Figure 3.23: Source: [31].

In the table, R_0 is the algorithm compared with the other ones, in this case C4.5. The Holm procedure rejects the first ($p_1 < \alpha/(k-1)$) and then the second hypothesis ($p_2 < \alpha/(k-2)$) since the corresponding p values are smaller than the adjusted α's. The third hypothesis cannot be rejected ($p_3 > \alpha/(k-3)$). If there were any more, we would have to retain them, too.

3.23 Bias-variance Tradeoff

When we talk about model prediction, it's important to understand prediction errors: *bias* and *variance*. There is a tradeoff between a model's ability to minimize bias and variance. The understanding of these errors would help us not only to build accurate models but also to avoid the mistake of overfitting and underfitting.

In statistics and machine learning, the bias–variance tradeoff is the property of a set of predictive models whereby models with a lower bias in parameter estimation have a higher variance of the parameter estimates across samples, and vice versa. The bias–variance dilemma or bias–variance problem is the conflict in trying to simultaneously minimize these two sources of error that prevent supervised learning algorithms from generalizing beyond their training set.

Z	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
-3.9	0.00005	0.00005	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00003	0.00003
-3.8	0.00007	0.00007	0.00007	0.00006	0.00006	0.00006	0.00006	0.00005	0.00005	0.00005
-3.7	0.00011	0.00010	0.00010	0.00010	0.00009	0.00009	0.00008	0.00008	0.00008	0.00008
-3.6	0.00016	0.00015	0.00015	0.00014	0.00014	0.00013	0.00013	0.00012	0.00012	0.00011
-3.5	0.00023	0.00022	0.00022	0.00021	0.00020	0.00019	0.00019	0.00018	0.00017	0.00017
-3.4	0.00034	0.00032	0.00031	0.00030	0.00029	0.00028	0.00027	0.00026	0.00025	0.00024
-3.3	0.00048	0.00047	0.00045	0.00043	0.00042	0.00040	0.00039	0.00038	0.00036	0.00035
-3.2	0.00069	0.00066	0.00064	0.00062	0.00060	0.00058	0.00056	0.00054	0.00052	0.00050
-3.1	0.00097	0.00094	0.00090	0.00087	0.00084	0.00082	0.00079	0.00076	0.00074	0.00071
-3.0	0.00135	0.00131	0.00126	0.00122	0.00118	0.00114	0.00111	0.00107	0.00104	0.00100
-2.9	0.00187	0.00181	0.00175	0.00169	0.00164	0.00159	0.00154	0.00149	0.00144	0.00139
-2.8	0.00256	0.00248	0.00240	0.00233	0.00226	0.00219	0.00212	0.00205	0.00199	0.00193
-2.7	0.00347	0.00336	0.00326	0.00317	0.00307	0.00298	0.00289	0.00280	0.00272	0.00264
-2.6	0.00466	0.00453	0.00440	0.00427	0.00415	0.00402	0.00391	0.00379	0.00368	0.00357
-2.5	0.00621	0.00604	0.00587	0.00570	0.00554	0.00539	0.00523	0.00508	0.00494	0.00480
-2.4	0.00820	0.00798	0.00776	0.00755	0.00734	0.00714	0.00695	0.00676	0.00657	0.00639
-2.3	0.01072	0.01044	0.01017	0.00990	0.00964	0.00939	0.00914	0.00889	0.00866	0.00842
-2.2	0.01390	0.01355	0.01321	0.01287	0.01255	0.01222	0.01191	0.01160	0.01130	0.01101
-2.1	0.01786	0.01743	0.01700	0.01659	0.01618	0.01578	0.01539	0.01500	0.01463	0.01426
-2.0	0.02275	0.02222	0.02169	0.02118	0.02068	0.02018	0.01970	0.01923	0.01876	0.01831
-1.9	0.02872	0.02807	0.02743	0.02680	0.02619	0.02559	0.02500	0.02442	0.02385	0.02330
-1.8	0.03593	0.03515	0.03438	0.03362	0.03288	0.03216	0.03144	0.03074	0.03005	0.02938
-1.7	0.04457	0.04363	0.04272	0.04182	0.04093	0.04006	0.03920	0.03836	0.03754	0.03673
-1.6	0.05480	0.05370	0.05262	0.05155	0.05050	0.04947	0.04846	0.04746	0.04648	0.04551
-1.5	0.06681	0.06552	0.06426	0.06301	0.06178	0.06057	0.05938	0.05821	0.05705	0.05592
-1.4	0.08076	0.07927	0.07780	0.07636	0.07493	0.07353	0.07215	0.07078	0.06944	0.06811
-1.3	0.09680	0.09510	0.09342	0.09176	0.09012	0.08851	0.08691	0.08534	0.08379	0.08226
-1.2	0.11507	0.11314	0.11123	0.10935	0.10749	0.10565	0.10383	0.10204	0.10027	0.09853
-1.1	0.13567	0.13350	0.13136	0.12924	0.12714	0.12507	0.12302	0.12100	0.11900	0.11702
-1.0	0.15866	0.15625	0.15386	0.15151	0.14917	0.14686	0.14457	0.14231	0.14007	0.13786
-0.9	0.18406	0.18141	0.17879	0.17619	0.17361	0.17106	0.16853	0.16602	0.16354	0.16109
-0.8	0.21186	0.20897	0.20611	0.20327	0.20045	0.19766	0.19489	0.19215	0.18943	0.18673
-0.7	0.24196	0.23885	0.23576	0.23270	0.22965	0.22663	0.22363	0.22065	0.21770	0.21476
-0.6	0.27425	0.27093	0.26763	0.26435	0.26109	0.25785	0.25463	0.25143	0.24825	0.24510
-0.5	0.30854	0.30503	0.30153	0.29806	0.29460	0.29116	0.28774	0.28434	0.28096	0.27760
-0.4	0.34458	0.34090	0.33724	0.33360	0.32997	0.32636	0.32276	0.31918	0.31561	0.31207
-0.3	0.38209	0.37828	0.37448	0.37070	0.36693	0.36317	0.35942	0.35569	0.35197	0.34827
-0.2	0.42074	0.41683	0.41294	0.40905	0.40517	0.40129	0.39743	0.39358	0.38974	0.38591
-0.1	0.46017	0.45620	0.45224	0.44828	0.44433	0.44038	0.43644	0.43251	0.42858	0.42465
0.0	0.50000	0.49601	0.49202	0.48803	0.48405	0.48006	0.47608	0.47210	0.46812	0.46414

Figure 3.24: Negative z-table. To compute the p-value for a given z, find the correct cell in the table given the z-score (ignore the minus sign) and multiply its value by 2 since we are dealing with a *two-tailed test*.

Figure 3.25: Source: [31].

The bias error is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting). It always leads to high error on training and test data.

The variance is an error from sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting). As a result, such models perform very well on training data but has high error rates on test data.

3.23.1 Bias–variance decomposition of mean squared error

Now we describe the bias-variance decomposition of MSE taken from [11]. Suppose that we have a training set consisting of a set of points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ and real values y_i associated with each point $\mathbf{x}_i$. We assume that there is a function $f(x)$ such as $y = f(x) + \epsilon$, where the noise ϵ has zero mean and variance σ^2.

We want to find a function $\hat{f}(x)$ that approximates the true function $f(x)$ as well as possible, by means of some learning algorithm based on a training set $S = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2) \ldots, (\mathbf{x}_m, y_m)\}$ by minimizing the MSE between y and $\hat{f}(x)$. Whichever function $\hat{f}$ we select, we can decompose its expected error in an unseen sample x (i.e. conditioned to x) as follows:

$$\text{error}(x) = \mathbb{E}[(y - \hat{f}(x))^2] = \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + \sigma^2. \tag{3.39}$$

The derivation of the bias–variance decomposition for squared error proceeds as follows. For notational convenience, we abbreviate $\hat{f} = \hat{f}(x)$ and $f = f(x)$. Let us write the mean-sqaured error of our model:

$$\text{error}(x) = \mathbb{E}[(y - \hat{f})^2] = \mathbb{E}[y^2 - 2y\hat{f} + \hat{f}^2] = \mathbb{E}[y^2] - 2\,\mathbb{E}[y\hat{f}] + \mathbb{E}[\hat{f}^2] \tag{3.40}$$

First,

$$\mathbb{E}[\hat{f}^2] = \text{Var}(\hat{f}) + \mathbb{E}[\hat{f}]^2 \qquad \text{since } \text{Var}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2 \text{ for any random variable } X \tag{3.41}$$

Secondly, since we model $y = f + \epsilon$, we show that

$$\begin{aligned}
\mathbb{E}[y^2] &= \mathbb{E}[(f + \varepsilon)^2] \\
&= \mathbb{E}[f^2] + 2\,\mathbb{E}[f\varepsilon] + \mathbb{E}[\varepsilon^2] && \text{by linearity of } \mathbb{E} \\
&= f^2 + 2f\,\mathbb{E}[\varepsilon] + \mathbb{E}[\varepsilon^2] && \text{since } f \text{ does not depend on the data} \\
&= f^2 + 2f \cdot 0 + \sigma^2 && \text{since } \varepsilon \text{ has zero mean and variance } \sigma^2
\end{aligned}$$

Lastly,

$$\begin{aligned}
\mathbb{E}[y\hat{f}] &= \mathbb{E}[(f + \epsilon)\hat{f}] \\
&= \mathbb{E}[f\hat{f}] + \mathbb{E}[\epsilon\hat{f}] && \text{linearity of } \mathbb{E} \\
&= \mathbb{E}[f\hat{f}] + \mathbb{E}[\epsilon]\,\mathbb{E}[\hat{f}] && \text{independence of } \hat{f} \text{ and } \epsilon \\
&= f\,\mathbb{E}[\hat{f}] && \text{since } \mathbb{E}[\epsilon] = 0
\end{aligned}$$

Eventually, we plug these 3 formulas in Equation (3.40) and thus show that:

$$\begin{aligned}
\text{error}(x) &= f^2 + \sigma^2 - 2f\,\mathbb{E}[\hat{f}] + \text{Var}[\hat{f}] + \mathbb{E}[\hat{f}]^2 \\
&= (f - \mathbb{E}[\hat{f}])^2 + \sigma^2 + \text{Var}[\hat{f}] \\
&= \text{Bias}[\hat{f}]^2 + \text{Var}[\hat{f}] + \sigma^2
\end{aligned}$$

In the lecture notes [72] we consider $y = f(x)$ instead of $y = f(x) + \epsilon$. By repeating the same computation we obtain:

$$\text{error}(x) = \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)]. \tag{3.42}$$

In Equation (3.39), σ^2 is called *irreducible error*, since it cannot be reduced by creating good models. In fact, it is a measure of the amount of noise in our data.

3.23.2 Overfitting and underfitting

The bias-variance tradeoff is tightly connected to overfitting and underfitting. Figures 3.26 and 3.27 illustrates this connection.

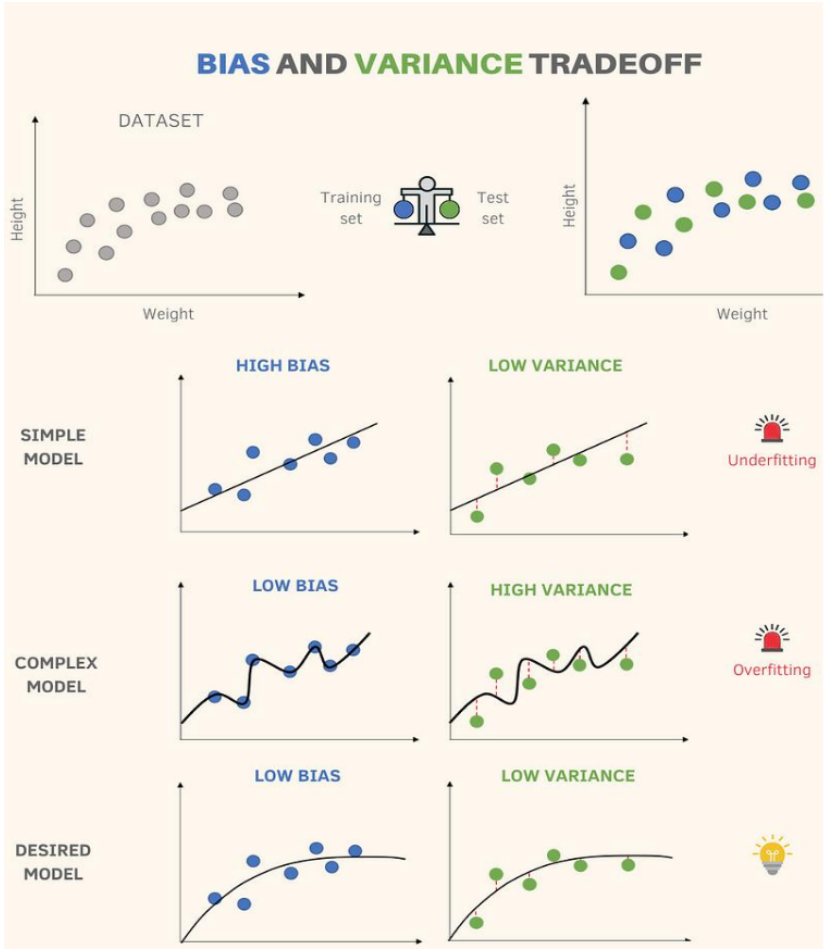


Figure 3.26: Bias-variance tradeoff in regression. In overfitting, the training patterns are all on the curve created in the training phase, but in the test set the regression is not good. Source: [72].

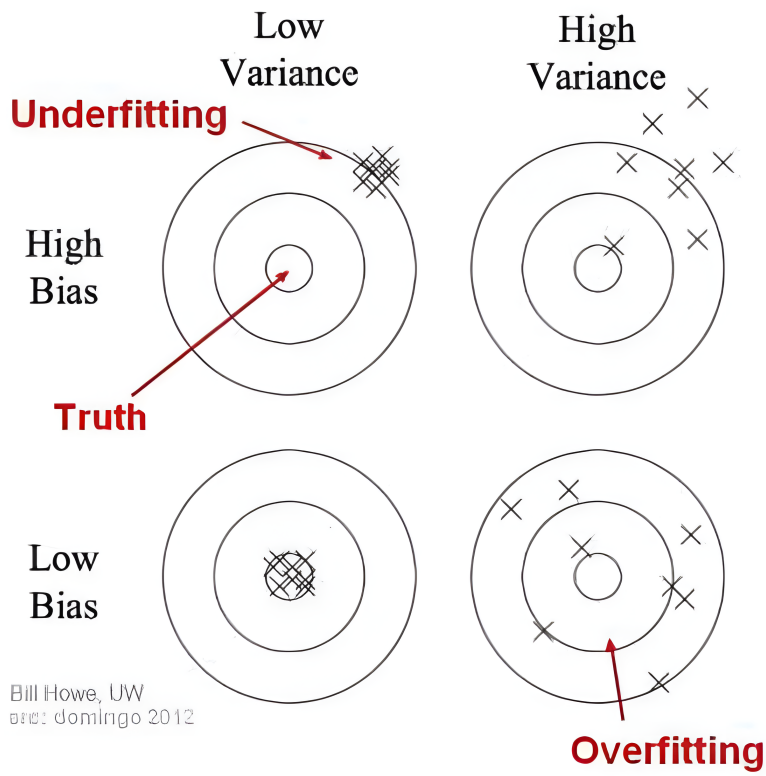


Figure 3.27: Bias-variance tradeoff using a bulls-eye diagram. The center of the target (bulls-eye) is a model that perfectly predicts correct values of the test data. If the error is always zero (in the training set) there is the risk of overfitting. As we move away from the bulls-eye our predictions become worse and worse. We can repeat our process of model building to get separate hits on the target. Source: [72].

3.23.3 Model's complexity

The *model's complexity* has a crucial role in the bias-variance tradeoff.

- If our model is too simple and has very few parameters then it may have high bias and low variance.
- On the other hand if our model has large number of parameters then it's going to have high variance and low bias.

So we need to find the right/good balance without overfitting and underfitting the data. This tradeoff in complexity is because there is a tradeoff between bias and variance. An algorithm can't be more complex and less complex at the same time. To build a good model, we need to find a good balance between bias and variance such that it minimizes the total error. An optimal balance of bias and variance would never overfit or underfit the model. Therefore understanding bias and variance is critical for understanding the behavior of prediction models. The bias-variance tradeoff and its relation with the complexity of the model is summarized in Figure 3.28.

3.24 Real world

We will briefly discuss some real word matters.

3.24.1 Automated performance evaluation

Automate your performance evaluation procedures, run them many times, and in the end you will have saved a lot of time. Compare the performance of your system only with others trained on the

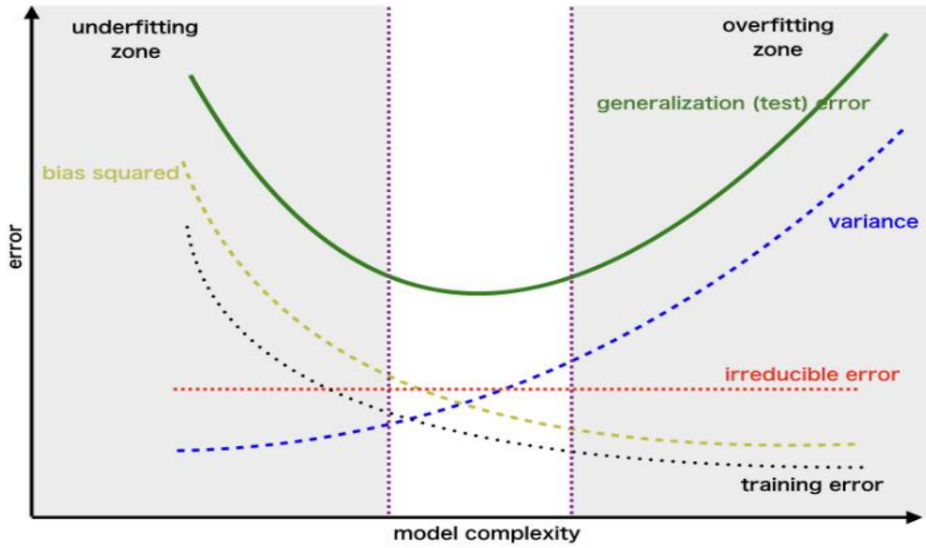


Figure 3.28: Bias-variance tradeoff with error decomposition and training error in function of the model complexity. Source: [72].

same dataset and with the same testing protocol. Pay attention to the statistical reliability of your results.

3.24.2 More runs

"On more runs" means running the simulation several times (e.g. run k-fold cross validation n times).

3.24.3 Ablation study

An *ablation study* investigates the performance of an AI system by removing certain components to understand the contribution of the component to the overall system. When we propose a new system, it is essential to report a good ablation study to motivate our choices. When creating a unique machine learning model, we frequently add numerous concepts that contribute to the overall model performance. In a study, however, it is useful to understand the influence of each of these innovations separately. To assess the impact of individual components, researchers frequently analyze their models with each of them disabled and quantify the decline of overall model performance. If we want to discover what a certain component of the organism performs, we break it and see what changes. This is known as an ablation study. Figure 3.29 shows an example of this procedure.

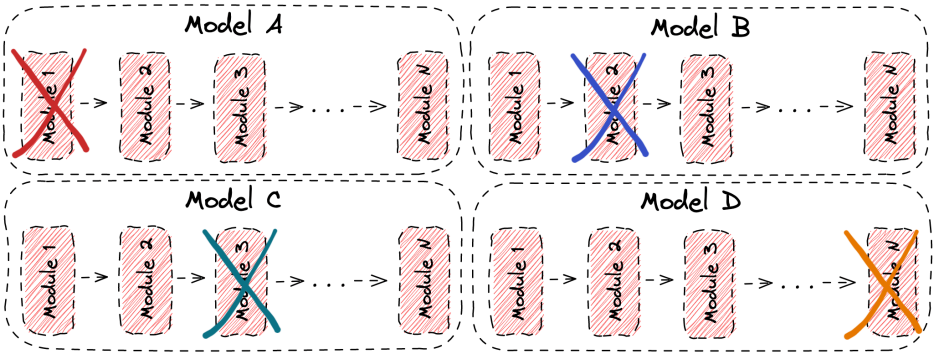


Figure 3.29: Example of an ablation study in a model with N modules. Each time we remove one of the modules and check performance on the new model to investigate the influence of the removed model. Source: [72].

3.24.4 Underspecification

Unfortunately, it is not that simple. Machine learning models often exhibit unexpectedly poor behavior when they are deployed in real-world domains. We identify *underspecification* as a key reason for these failures. A machine learning pipeline is *underspecified* when it can return many predictors with equivalently strong held-out performance in the training domain. Underspecification is common in modern machine learning pipelines, such as those based on deep learning. Predictors returned by underspecified pipelines are often treated as equivalent based on their training domain performance, but we show here that such predictors can behave very differently in deployment domains. This ambiguity can lead to instability and poor model behavior in practice, and is a distinct failure mode from previously identified issues arising from structural mismatch between training and deployment domains.

Chapter 4

Neural Networks introduction

4.1 Biological neuron

Within a nervous system, a neuron is an electrically excitable cell that fires electric signals called action potentials across a neural network. The synapses act as connecting gates for the passage of information between neurons. Dendrites are minor fibers that branch out from the cell body of the neuron, called soma. Through the synapses the dendrites collect inputs from afferent neurons and propagate them towards the soma. The axon is the main fiber that starts from the soma and moves away from it to bring the output to other neurons (even distant ones).

A connectome is a comprehensive map of neural connections in the brain, and may be thought of as its "wiring diagram". An organism's nervous system is made up of neurons which communicate through synapses. A connectome is constructed by tracing the neuron in a nervous system and mapping where neurons are connected through synapses.

4.2 McCulloch-Pitts neuron

The first computational model of a neuron was proposed by Warren MuCulloch (neuroscientist) and Walter Pitts (logician) in 1943, illustrated in Figure 4.1. It can be divided in two parts. The first part g takes an input, performs an aggregation and, based on the aggregated value, the second part f makes a decision.

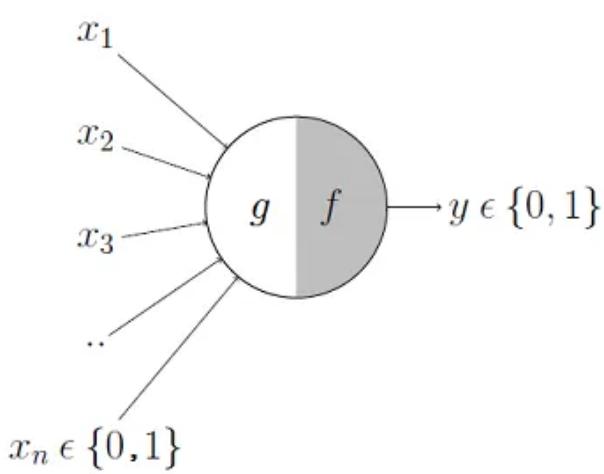


Figure 4.1: McCulloch-Pitts neuron. Source: [67].

These inputs can either be excitatory or inhibitory. Inhibitory inputs are those that have maximum effect on the decision making irrespective of other inputs. Excitatory inputs are NOT the ones that

will make the neuron fire on their own but they might fire it when combined together. Formally:

$$g(x_1, x_2, \ldots, x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i, \tag{4.1}$$

$$y = f(g(\mathbf{x})) = \begin{cases} 1 & \text{if } g(\mathbf{x}) \geq \theta, \\ 0 & \text{if } g(\mathbf{x}) < \theta. \end{cases} \tag{4.2}$$

where θ is a thresholding parameter. We can see that g is just doing a sum of the inputs, a simple aggregation.

4.3 Neurons

In biological neurons, when $g(\mathbf{x})$ exceeds a certain threshold, the neuron "fires" a spike and then returns to rest. For this reason, f is called *activation function*. In general, f is a non-linear, continuous and differentiable function. In fact, non-linearities are necessary to perform a complex mapping of the input information. Continuity and differentiability is necessary for error back-propagation. The most commonly used activation function in more recent shallow networks is the sigmoid in the variants: *standard logistics function* and *hyperbolic tangent*. The standard logistic function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{4.3}$$

and its derivative with respect to z is

$$\sigma'(z) = \frac{\partial}{\partial z}\left(\frac{1}{1 + e^{-z}}\right) = \sigma(z)(1 - \sigma(z)) \tag{4.4}$$

The hyperbolic tangent is rescaling of the logistic function. It is possible to prove that

$$\tanh(z) = 2\sigma(2z) - 1. \tag{4.5}$$

Therefore we can generalize its formulation as the following:

$$\tanh(z) = \frac{2a}{1 + e^{-2bz}} - a, \tag{4.6}$$

where $a, b \in \mathbb{R}$. Its derivative with respect to z is

$$\tanh'(z) = \frac{\partial}{\partial z}\left(\frac{2a}{1 + e^{-2bz}} - a\right) = \frac{4ab \cdot e^{-2bz}}{(1 + e^{-2bz})^2} \tag{4.7}$$

When using tanh as activation function, the recommended parameters values are $a = 1.716$ and $b = 2/3$, as $\frac{\partial \tanh}{\partial z}(0) \approx 1$. The linear range is between -1 and +1 approximately. The hyperbolic tangent is slightly more complex but preferable (often it has a faster convergence), as it is symmetrical with respect to zero.

4.4 Feedforward neural networks

A *neural network* (NN) is made of groups of artificial neurons organized in layers. Typically there is an input level, an output level, and one or more intermediate or hidden levels. Each level contains one or more neurons. In *feedforward neural networks* connections connect neurons of one level with neurons of a subsequent level. No backwards connections or connections towards the same level are allowed. It is by far the most used type of network. A diagram of a feedforward neural network is shown in Figure 4.2. Informally, we say that a neural network is *shallow* if it has only one hidden layer, while it is *deep* if it has two or more.

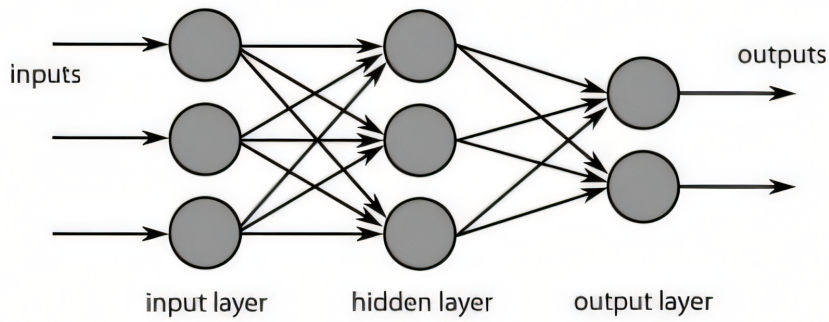


Figure 4.2: Diagram of feedforward neural network. Source: [72].

4.5 Recurrent neural networks

A *Recurrent neural network* (RNN) is opposed to feedforward neural network. In recurring networks, feedback connections (usually towards neurons of the same level, but also backwards) are allowed. This greatly complicates the flow of information and training, requiring you to consider the behavior in multiple instants of time (unfolding in time). These networks are indicated for the management of sequences (e.g. audio, video, sentences in natural language), because they are equipped with a memory effect (short term) which at time t makes the processed information available at $t-1$, $t-2$, etc. A particular type of recurring network is the *Long Short-Term Memory* (LSTM).

4.6 Perceptron

The term *perceptron* derives from the neuron model proposed by Rosenblatt in 1958, very similar to that of McCulloch and Pitts. The original perceptron, illustrated in Figure 4.3, uses a linear threshold (or step) activation function. A single perceptron, or a network of only two levels (input and output), can be trained with a simple rule called *delta rule*. The delta rule can be derived as the backpropagation algorithm for a single-layer neural network with mean-square error loss function. Consider a neuron

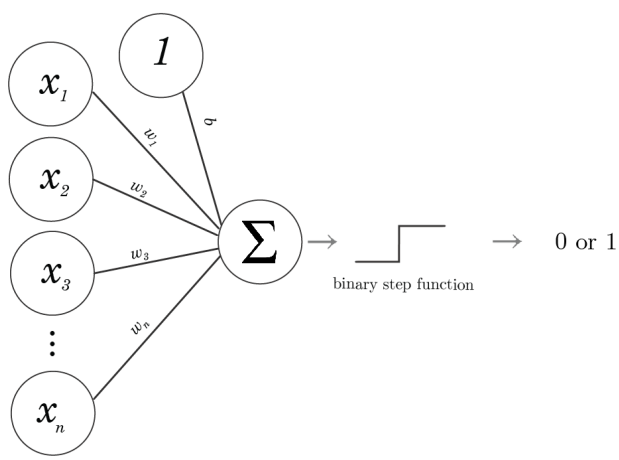


Figure 4.3: Perceptron. Source: [105].

for a neuron with a linear activation function. We now formulate the delta rule for this simplified case as done in [30]. For a neuron j with activation function $g(x)$, the simplified delta rule for neuron j's i-th weight w_{ji} is given by

$$\Delta w_{ji} = \alpha(t_j - y_j)x_i, \tag{4.8}$$

where t_j is the target output, α is the learning rate, y_j is the network output, x_i is the i-th input.

If the output layer has only one node, then j is not used and we obtain:

$$\Delta w_i = \alpha(t - y)x_i. \tag{4.9}$$

Where $\mathbf{x} = (1, x_1, x_2, \ldots, x_n)^T$ is the network input, t is the target output (i.e. the training set sample is $(\mathbf{x}, t)$), α is the learning rate, y_j is the network output. The update rule is

$$w_i \leftarrow w_i + \Delta w_i. \tag{4.10}$$

This resembles the classic perceptron update rule for halfspaces.

However, single-layer neural networks have a huge number of limitations. For example, no such network can represent the XOR function as a function $f(x_1, x_2)$. The function is reported in Figure 4.4. In particular, the XOR function cannot be represented by any linear classifier. In fact, as already

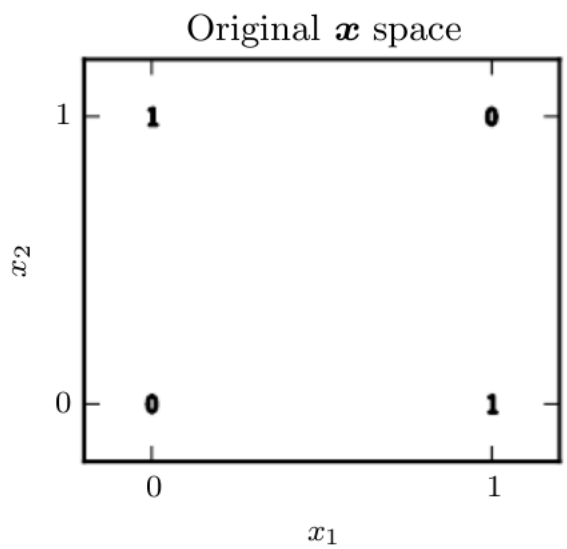


Figure 4.4: XOR in the $\mathbf{x} = (x_1, x_2)^T$ space. Source: [72].

introduced in Section 4.3, non-linearities are essential for representing complex mappings. These are introduced with more than one layer with appropriate non-linear activation functions. An example of non-linear decision surfaces for classification are shown in Figure 4.5.

4.7 Multilayer perceptron

A *Multilayer Perceptron* (MLP) is a feedforward network with at least 3 levels (one input level, one output level and at least one hidden level) and with non-linear activation functions. A theorem known as the *universal approximation theorem* asserts that any continuous function that maps ranges of real numbers to a range of real numbers can be approximated by an MLP with only one hidden layer. This is one of the reasons why for many decades (until the explosion of deep learning) there has been a focus on 3-level neural networks. On the other hand, the theoretical existence of a solution does not imply that there is an effective way to obtain it.

4.8 Forward propagation

With *forward propagation* (or *inference*) we mean the propagation of information forward, from the input level to the output level. Once trained, a neural network can simply process patterns through forward propagation. Consider the 3-level neural network in Figure 4.6 with activation function f at each level. Let d be the number of input neurons, s the number of output neurons and n_H the number

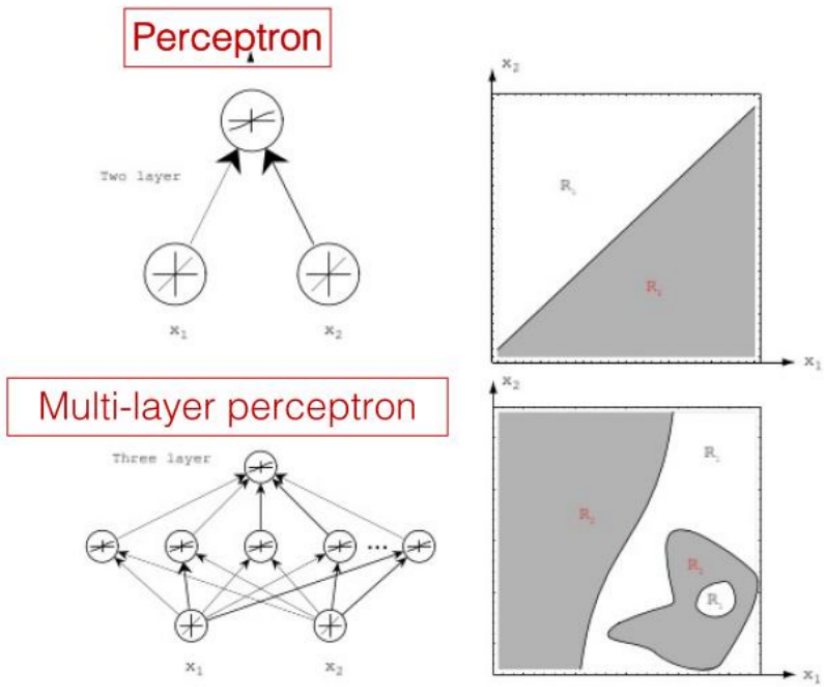


Figure 4.5: Effect of non-linearities on decision surfaces. White is region R_1, gray is region R_2. Source: [72].

of neurons in the hidden layer. The k-th output z_k can be computed as:

$$z_k = f\left(\sum_{j=1}^{n_H} w_{jk} y_j + w_{0k}\right) = f\left(\sum_{j=1}^{n_H} w_{jk} f\left(\sum_{i=1}^{d} w_{ij} x_i + w_{0j}\right) + w_{0k}\right). \tag{4.11}$$

In addition, the number of *trainable parameters* or *weights* in this network is

$$d \cdot n_H + n_H \cdot s + n_H + s, \tag{4.12}$$

where the two last terms n_H and s corresponds to the weights of the biases. A similar example with a 4-level network of layers $(2, 3, 2, 1)$ (2 hidden layers) is reported in Figure 4.7 and 4.8. Here no biases are used. In the graphics of the example the notation is a little different from the one in Figure 4.6.

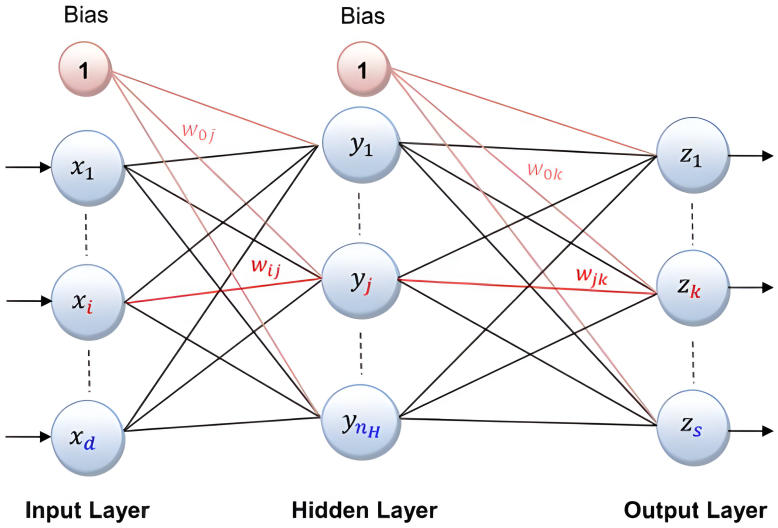


Figure 4.6: 3-level neural network. Source: [72].

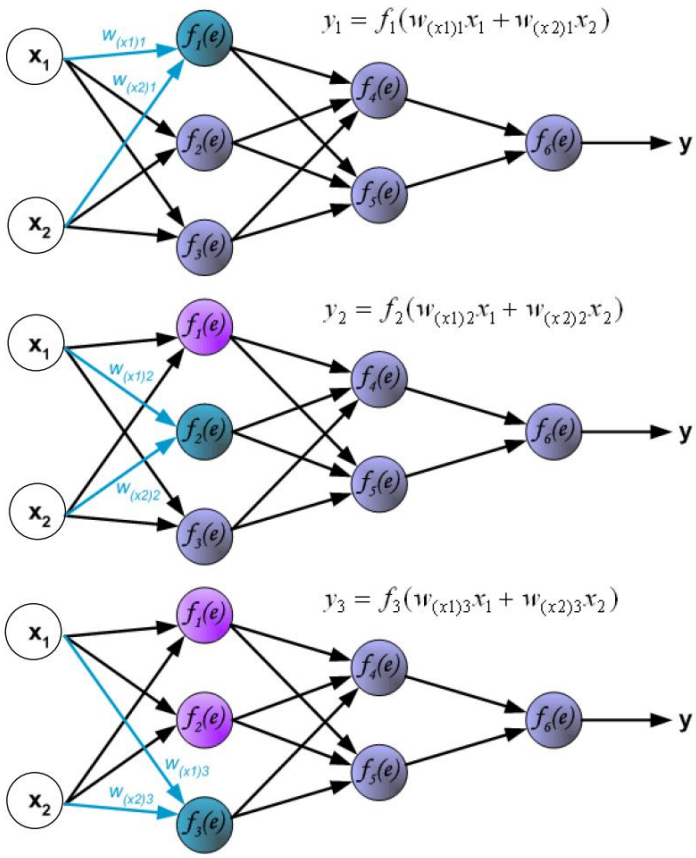


Figure 4.7: Computation of the second layer in a 4-level neural network. Source: [72].

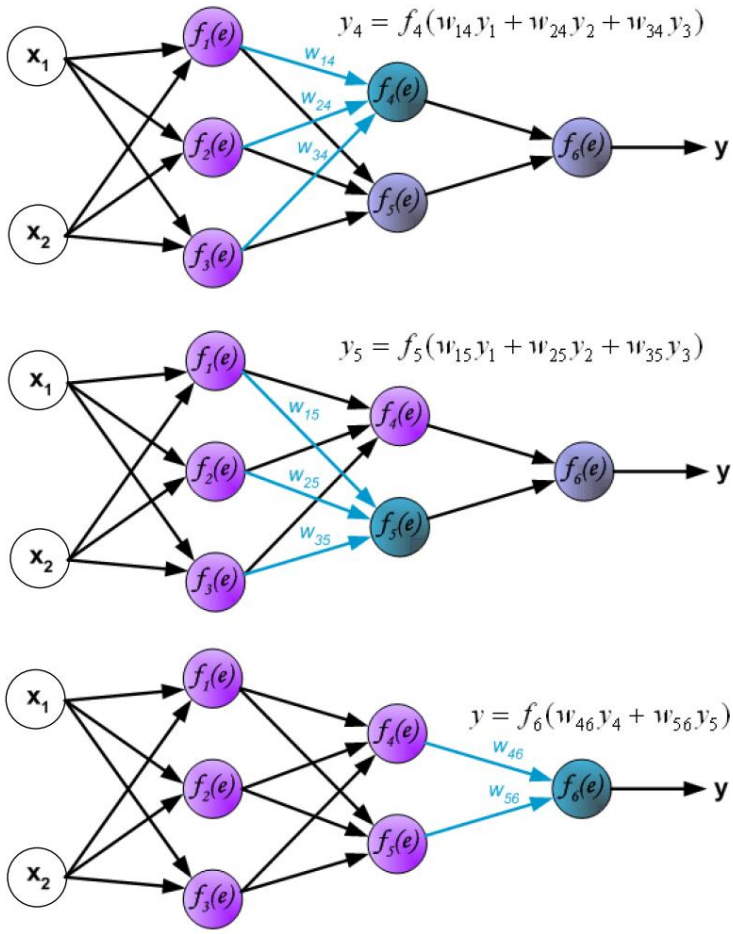


Figure 4.8: Computation of the third and fourth layers in a 4-level neural network. Source: [72].

4.9 Training of a neural network

Once the topology (number of levels and neurons) is fixed, the training of a neural network consists in finding the value of the weights that determine the desired mapping between input and output. What do we mean by desired mapping? It depends on the problem we want to solve:

- If we are interested in training a neural network to act as a classifier, the desired output is the correct class label of the input pattern.
- If the neural network has to solve a regression problem, the desired output is the correct value of the dependent variable, corresponding to the value of the independent variables supplied as input.

Although the first artificial neurons date back to the 1940s, effective training algorithms were not available until the mid-1980s. In 1986 Rumelhart, Hinton & Williams introduced the *error backpropagation algorithm*, arousing great attention in the scientific community. It is actually the application of the *chain derivation rule*, an idea that no one had successfully applied in this context for decades.

4.9.1 Training of a neural network for a classification problem

In the following we focus our attention on a supervised classification problem. To handle a classification problem with s classes and d-dimensional patterns, it is usual to use a network with d input neurons, n_H hidden neurons, and s output neurons. That is, the number of input neurons is the same of the features and the number of output neurons is equal to the number of classes. The value n_h is instead a hyperparameter. A reasonable value of n_H is $\frac{1}{10}d$.

Preprocessing

Before the training, we can apply to the training set a *preprocessing*, since it can improve the convergence during training. An example of preprocessing is the *whitening*, which is a linear transformation that transforms a dataset of samples $\mathbf{x}_1, \ldots, \mathbf{x}_n$ with a known covariance matrix Σ into a set of new samples whose covariance is the identity matrix, meaning that they are uncorrelated and each have variance 1 [123]. We first compute the mean $\boldsymbol{\mu}$:

$$\boldsymbol{\mu} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i. \tag{4.13}$$

We then perform the eigenvalue decomposition:

$$\Sigma = V \Lambda V^T, \tag{4.14}$$

where V is the matrix composed of the eigenvectors of Σ and Λ is the diagonal matrix with eigenvalues on the diagonal. The whitening transformation is:

$$\mathbf{x}_i \mapsto \Lambda^{-1/2} V^T (\mathbf{x} - \boldsymbol{\mu}). \tag{4.15}$$

Target output

Since a supervised learning task is taken into consideration, the training takes place by presenting the pattern of which the class is known to the network and propagating (forward propagation) the inputs to the outputs. Given a training pattern whose class is t_g among the possible classes $t_1, t_2, \ldots, t_s$, $g \in 1, 2, \ldots, s$, the desired output vector t (*target*) takes the form

$$\mathbf{t} = (t_1, t_2, \ldots, t_s), \tag{4.16}$$

where $t_i = 1$ for $i = g$, $t_i = -1$ otherwise. For example, if there are four classes ($s = 4$) and the target class is the third one ($g = 3$), the target vector is

$$\mathbf{t} = (-1, -1, 1, -1). \tag{4.17}$$

The squared difference between the output produced by the network and the desired one is the network error. Objective of the learning algorithm is to modify the network's weights in order to minimize the average error on training set patterns.

Weights initialization

Before the start of training, the weights are typically initialized with random values as following:

- For weights from the input to the hidden layer (input-hidden), uniformly in $\left(-\frac{1}{\sqrt{d}}, +\frac{1}{\sqrt{d}}\right)$.
- For weights from the hidden to the output layer (hidden-output), uniformly in $\left(-\frac{1}{\sqrt{n_H}}, +\frac{1}{\sqrt{n_H}}\right)$.

Errors

Consider the 3-level network. Let z be the output produced by the network through forward propagation in correspondence with the pattern $\mathbf{x}$ of class g supplied in input. As already mentioned, the output desired is $\mathbf{t} = (t_1, t_2 ... t_s)$, where $t_i = 1$ for $i = g$, $t_i = -1$ otherwise. By choosing the sum of the squares of the errors as the loss function (analogous to statistical learning's squared loss), the error (for pattern $\mathbf{x}$) is:

$$J(\mathbf{w}, \mathbf{x}) = \sum_{c=1}^{s} (t_c - z_c)^2, \tag{4.18}$$

which quantifies how much the output produced for pattern x differs from the desired one. The error $J(\mathbf{w})$ on the entire training set is the average of $J(\mathbf{w}, \mathbf{x})$ on all the patterns $\mathbf{x}$ belonging to the training set S, namely:

$$J(\mathbf{w}) = \frac{\sum_{\mathbf{x} \in S} J(\mathbf{w}, \mathbf{x})}{|S|}. \tag{4.19}$$

Objective function

During the training, we want to minimize an *objective function*. In deep learning, the objective function is the *cost function*, also called *error function* or *loss function* (note the difference with statistical learning).

For simplicity, consider a real value function $y = f(x)$. The derivative $f'(x)$ at point x tells us how to change x to make small improvement in y. However, if $f'(x) = 0$ we have no useful information, as x could be a local maxima, a local minima or a saddle point. This leads us to the main idea of *Gradient Descent* (GD): to minimize a function in 1 variable, we have to move in the opposite direction of the derivative. The high level formulation of the algorithm for minimizing $f(z)$ is the following:

1. Start with a random value for z.
2. Update z doing a little step in the opposite direction with respect to the gradient: $z \leftarrow z - \alpha f'(z)$.
3. Repeat until convergence (i.e. the gradient is very small).

As an example, let us minimize $f(z) = z^2$ by applying GD from the starting point $z_0 = 2$ and learning rate $\alpha = 0.25$. We have $f'(z_0) = f'(2) = 2 \cdot 2 = 4$. Therefore $z_1 = z_0 - \alpha f'(z_0) = 2 - 0.25 \cdot 4 = 1$. We now have $f'(z_1) = f'(1) = 2 \cdot 1 = 2$ and $z_2 = z_1 - \alpha f'(z_1) = 1 - 0.25 \cdot 2 = 0.5$. The procedure is applied iteratively until convergence (i.e. the gradient is very small).

By applying GD, we could arrive to a *local minimum* instead of a *global minimum*. Some local minimum may be acceptable, some others not. Figure 4.9 shows some example of local and global minimum.

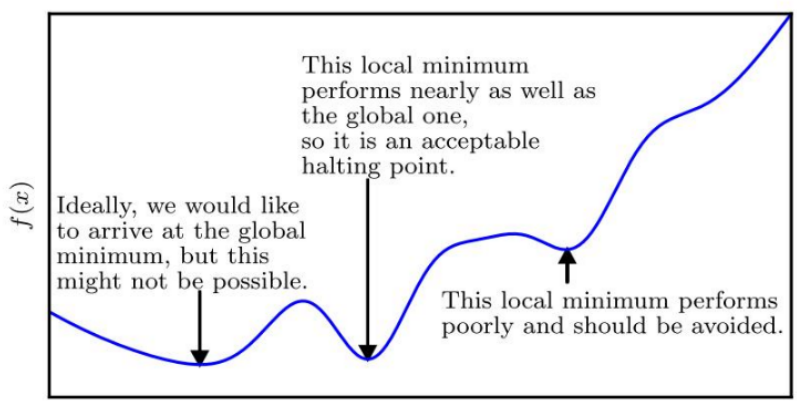


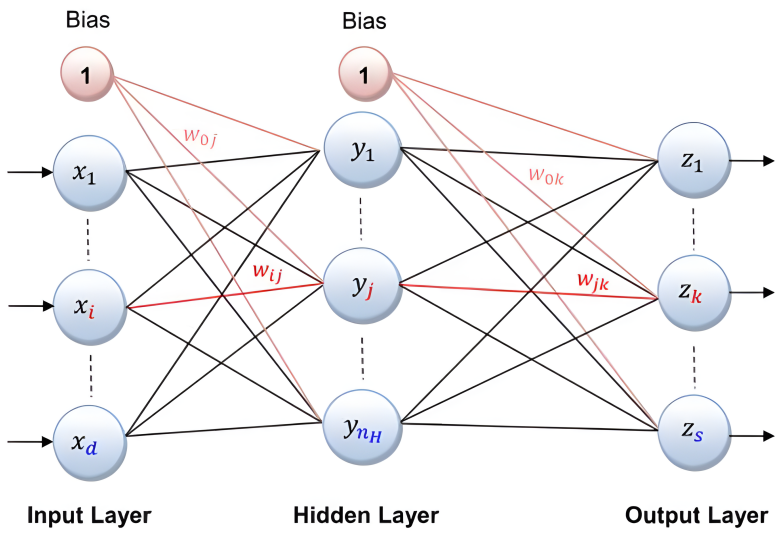
Figure 4.9: Local minimum and global minimum. Source: [72].

Minimization of the cost function: backpropagation algorithm

$J(\mathbf{w})$ can be reduced by changing the weights $\mathbf{w}$ in the opposite direction to the gradient of J. In fact, the gradient indicates the direction of greatest growth of a function (of several variables) and moving in the opposite direction we reduce (to the maximum) the error. *Backpropagation algorithm* can be used with Gradient Descent. In the following, we consider:

- First the update of the weights w_{jk} hidden-output.
- Then the update of the weights w_{ij} input-hidden.

The neural network has the following form.



For the moment, ignore the biases.

Update of the weights hidden-output

Let net_k be the input to node k in the output layer, i.e. z_k is obtained by net_k after applying the activation function f. In formulas:

$$\text{net}_k = \sum_{j=1}^{n_H} w_{jk} y_j \tag{4.20}$$

and

$$f(\text{net}_k) = f\left(\sum_{j=1}^{n_H} w_{jk} y_j\right) = z_k. \tag{4.21}$$

We consider Equation (4.18) multiplied by $\frac{1}{2}$ as loss function for ease of computation (i.e. $\frac{1}{2}$ cancels out after the first partial derivation). We compute the derivative of J with respect to the weights w_{jk} hidden-output:

$$\begin{aligned}
\frac{\partial J}{\partial w_{jk}} &= \frac{\partial}{\partial w_{jk}}\left(\frac{1}{2}\sum_{c=1}^{s}(t_c - z_c)^2\right) && \text{definition of } J \\
&= -(t_k - z_k)\frac{\partial z_k}{\partial w_{jk}} && \text{since only } z_k \text{ is influenced by } w_{jk} \\
&= -(t_k - z_k)\frac{\partial f(\text{net}_k)}{\partial w_{jk}} && \text{Equation (4.21)} \\
&= -(t_k - z_k)\frac{\partial f(\text{net}_k)}{\partial \text{net}_k}\frac{\partial \text{net}_k}{\partial w_{jk}} && \text{chain rule of derivatives} \\
&= -(t_k - z_k) f'(\text{net}_k)\frac{\partial \text{net}_k}{\partial w_{jk}} && \text{definition of derivative} \\
&= -(t_k - z_k) f'(\text{net}_k)\frac{\partial \sum_{s=1}^{n_H} w_{sk} y_s}{\partial w_{jk}} && \text{Equation (4.21)} \\
&= -(t_k - z_k) f'(\text{net}_k) y_j && \text{since only } w_{jk} y_j \text{ is influenced by } w_{jk}.
\end{aligned}$$

We then define δ_k as:

$$\delta_k = (t_k - z_k) f'(\text{net}_k). \tag{4.22}$$

Therefore:

$$\frac{\partial J}{\partial w_{jk}} = -\delta_k y_j \tag{4.23}$$

From Equation (4.23), the weight w_{jk} can be updated as:

$$w_{jk} \leftarrow w_{jk} + \eta \delta_k y_j, \tag{4.24}$$

where η is the learning rate. If η is too small, we have small convergence. On the other hand, if η is too large, the algorithm will "oscillate" around the local minimum or diverge. In general, it is advised to Start with $\eta = 0.5$ and change its value if it is not adequate, by monitoring the trend of the loss function during the iterations.

The same derivation can be used when considering the biases. In fact, the weight w_{0k} is updated by setting $y_0 = 1$.

Update of the weights input-hidden

In an analogous way we can update the weights w_{ij} input-hidden. Let net_j be the input to node j in the hidden layer, i.e. y_j is obtained by net_j after applying the activation function f. In formulas:

$$\text{net}_j = \sum_{q=1}^{d} w_{qj} x_q \tag{4.25}$$

and

$$f(\text{net}_j) = f\left(\sum_{q=1}^{d} w_{qj} x_q\right) = y_j. \tag{4.26}$$

We compute the derivative of J with respect to the weights w_{ij} input-hidden:

$$\begin{aligned}
\frac{\partial J}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}}\left(\frac{1}{2}\sum_{c=1}^{s}(t_c - z_c)^2\right) && \text{definition of } J \\
&= -\sum_{c=1}^{s}(t_c - z_c)\frac{\partial z_c}{\partial w_{ij}} && \text{since all the } z_c \text{ are influenced by } w_{ij} \\
&= -\sum_{c=1}^{s}(t_c - z_c)\frac{\partial z_c}{\partial \text{net}_c}\frac{\text{net}_c}{\partial w_{ij}} && \text{chain rule of derivatives} \\
&= -\sum_{c=1}^{s}(t_c - z_c)\frac{\partial f(\text{net}_c)}{\partial \text{net}_c}\frac{\partial \text{net}_c}{\partial w_{ij}} && \text{Equation (4.21)} \\
&= -\sum_{c=1}^{s}(t_c - z_c) f'(\text{net}_c)\frac{\partial \text{net}_c}{\partial w_{ij}} && \text{Equation (4.21)} \\
&= -\sum_{c=1}^{s}\delta_c \frac{\partial \text{net}_c}{\partial w_{ij}} && \text{Equation (4.22)}
\end{aligned}$$

Consider now $\frac{\partial \operatorname{net}_c}{\partial w_{ij}}$.

$$
\begin{aligned}
\frac{\partial \operatorname{net}_c}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \sum_{r=1}^{n_H} w_{rc} y_r && \text{Equation (4.21)} \\
&= \frac{\partial}{\partial w_{ij}} \sum_{r=1}^{n_H} w_{rc} f(\operatorname{net}_r) && \text{Equation (4.26)} \\
&= \frac{\partial}{\partial w_{ij}} \left(w_{jc} f(\operatorname{net}_j) \right) && \text{since only } \operatorname{net}_j \text{ is influenced by } w_{ij} \\
&= w_{jc} \frac{\partial f(\operatorname{net}_j)}{\partial w_{ij}} && w_{jc} \text{ is a constant} \\
&= w_{jc} \frac{\partial f(\operatorname{net}_j)}{\partial \operatorname{net}_j} \frac{\partial \operatorname{net}_j}{\partial w_{ij}} && \text{chain rule of derivatives} \\
&= w_{jc} f'(\operatorname{net}_j) \frac{\partial \operatorname{net}_j}{\partial w_{ij}} && \text{definition of derivative} \\
&= w_{jc} f'(\operatorname{net}_j) \frac{\partial}{\partial w_{ij}} \sum_{q=1}^{d} w_{qj} x_q && \text{Equation (4.26)} \\
&= w_{jc} f'(\operatorname{net}_j) x_i && \text{since only } w_{ij} x_i \text{ is influenced by } w_{ij}
\end{aligned}
$$

By plugging in $\frac{\partial \operatorname{net}_c}{\partial w_{ij}}$ into $\frac{\partial J}{\partial w_{ij}}$ we have:

$$
\frac{\partial J}{\partial w_{ij}} = -\sum_{c=1}^{s} \delta_c w_{jc} f'(\operatorname{net}_j) x_i = -x_i f'(\operatorname{net}_j) \sum_{c=1}^{s} \delta_c w_{jc} \tag{4.27}
$$

We then define δ_j as:

$$
\delta_j = f'(\operatorname{net}_j) \sum_{c=1}^{s} w_{jc} \delta_c. \tag{4.28}
$$

Therefore:

$$
\frac{\partial J}{\partial w_{ij}} = -\delta_j x_i \tag{4.29}
$$

From Equation (4.29), the weight w_{ij} can be updated as:

$$
w_{ij} \leftarrow w_{ij} + \eta \delta_j x_i, \tag{4.30}
$$

where η is the learning rate.

The same derivation can be used when considering the biases. In fact, the weight w_{0j} is updated by setting $x_0 = 1$.

The term *backpropagation* is due to Equation (4.28), as it resembles the inference procedure, but starting from the nodes of the output layer. An example of computation of δ is reported in Figure 4.10 and 4.11. Figure 4.12 shows the update of the weights after the computation of the $\delta_6, \delta_5, \ldots, \delta_1$.

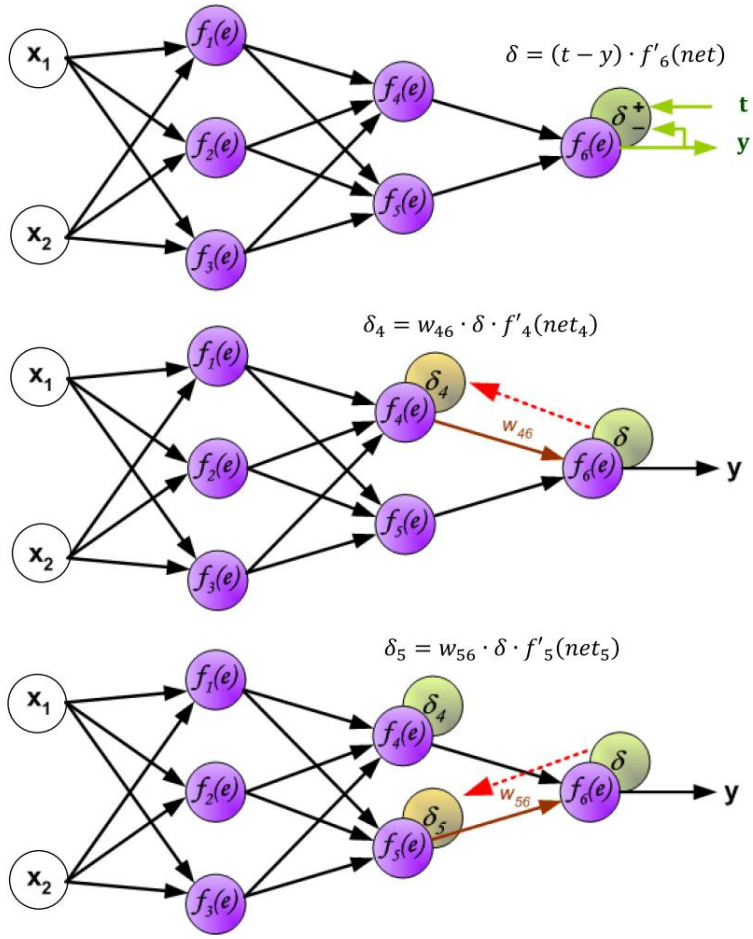


Figure 4.10: Computation of δ_4 and δ_5. In this notation, $\delta = \delta_6$. Source: [72].

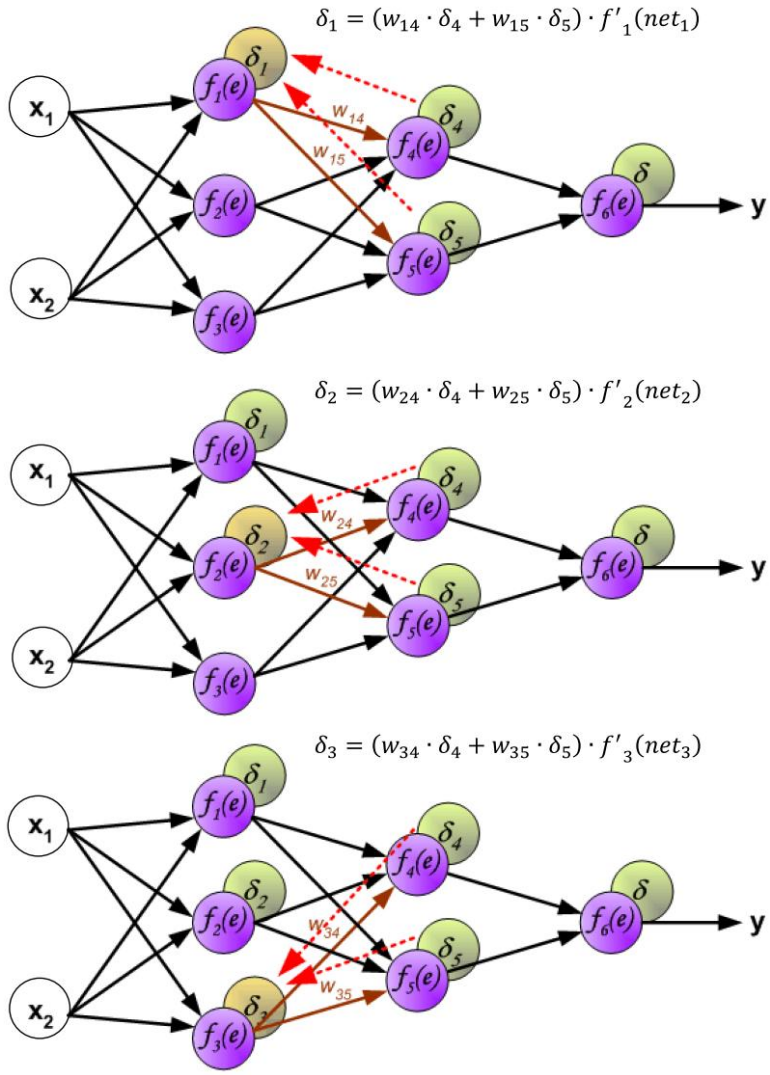


Figure 4.11: Computation of δ_1, δ_2 and δ_3. Source: [72].

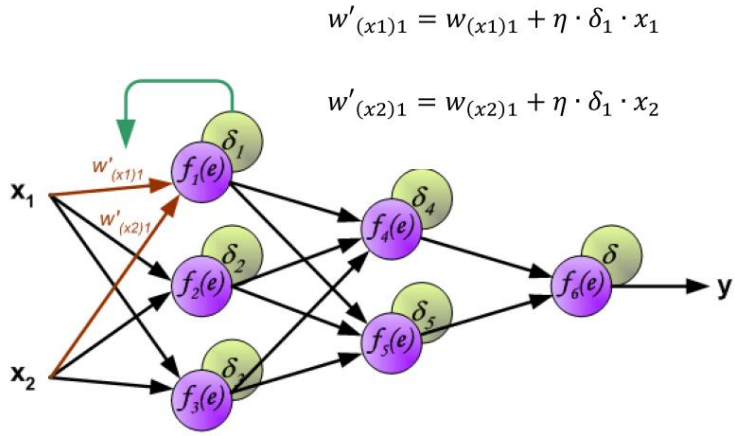


Figure 4.12: Update of the weights from the first to the second layer. Source: [72].

4.10 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is the most used approach to implement backpropagation. At each *epoch*, the n patterns of the training set are randomly ordered and then divided, considering them sequentially, into m groups called *mini-batches* of equal size:

$$\text{size}_{\text{mb}} = \frac{n}{m}. \tag{4.31}$$

The gradient values are algebraically accumulated in temporary variables, one for each weight. In other words, the gradient values of each pattern in the training set is summed to obtain the total gradient. The update of the weights occurs only when all the patterns of a mini-batch have been processed. The algorithm is described in Algorithm 4.10.1.

Algorithm 4.10.1: Backpropagation Algorithm

```
Data: Training set S of n patterns , n_H, m, weights w, η, maxEpoch
epoch = 0;
do
    epoch = epoch + 1;
    Random sort S;
    foreach mini-batch B of size n/m do
        grad_ik = 0;
        grad_ij = 0;
        foreach x ∈ B do
            Compute z_k for k = 1, 2, ..., s through the forward propagation of x;
            grad_jk = grad_jk + δ_k y_j, for k = 1, 2, ..., s, j = 1, 2, ..., n_H;
            grad_ij = grad_ij + δ_j x_i, for j = 1, 2, ..., n_H, i = 1, 2, ..., d;
        end
        w_jk = w_jk + η · grad_jk, for k = 1, 2, ..., s, j = 1, 2, ..., n_H;
        w_ij = w_ij + η · grad_ij, for j = 1, 2, ..., n_H, i = 1, 2, ..., d;
    end
while epoch < maxEpoch;
```

With *epoch* we mean the presentation (one time) of all the n patterns of the training set to the network. With *iteration* we mean the presentation (one time) of the patterns making up a mini-batch B and the consequent update of the weights. $n/\text{size}_{\text{mb}}$ is the number of iterations per epoch. If it is not an integer, at the last iteration, fewer patterns are processed.

4.11 SoftMax

The output depends on the activation function applied in the output layer. For example, if we apply tanh as activation function, the output values are in the range $[-1, 1]$, and therefore they cannot be interpreted as probabilities. If we use the sigmoid (standard logistic function) the output values are included in the interval $[0, 1]$, but we have no guarantee that the sum on the output neurons is 1 (a fundamental requirement so that they can be interpreted as a probabilistic distribution). In fact, in general we will have

$$\sum_{c=1}^{s} z_c \neq 1. \tag{4.32}$$

When a neural network is used as a multi-class classifier, the use of the *softmax activation function* allows us to transform the net values produced by the last level of the network into *class probabilities*. The net_k activation level of the neuron k of the last level is calculated in the usual way, but z_k is computed as follows:

$$z_k = f(\text{net}_k) = \frac{e^{\text{net}_k}}{\sum_{c=1}^{s} e^{\text{net}_c}}. \tag{4.33}$$

The values z_k can be interpreted as probabilities of the classes as $z_k \in [0, 1]$ and

$$\sum_{c=1}^{s} z_c = 1. \tag{4.34}$$

The choice of tanh activation function is not optimal, as the output values do not represent probabilities, and the non-imposition of the sum constraint to 1 makes learning (generally) less effective.

4.12 Cross-Entropy loss

In Section 4.9 we used the sum of squared error loss function. For a multi-class classification problem, it is recommended to use *Cross-Entropy* (also called Multinomial Logistic Loss) as a loss function.

Definition 4.12.1 *The* cross-entropy *between two discrete probability distributions p and q (which give the measure of how much q differs from p) with the same support $\mathcal{X}$ is*

$$H(p, q) = -\sum_{x \in \mathcal{X}} p(x) \cdot \log(q(x)). \tag{4.35}$$

The minus sign makes the entropy into a non-negative number (for any base $b > 1$ logarithms of numbers between 0 and 1 are negative numbers). In general, when used as a loss function, p (fixed) is the target vector, while q is the output vector of the network. The minimum value of $H = 0$ (note that $H \geq 0$) occurs when the target vector coincides with the output. The logarithm is not defined in 0. Luckly, this is not a problem, as the outputs provided by softmax and sigmoid activation functions are never 0 (even if they can tend towards it asymptotically).

Let the target vector take the form (*one-hot form*)

$$\mathbf{t} = (t_1, t_2, \ldots, t_s), \tag{4.36}$$

where $t_i = 1$ for $i = g$, $t_i = 0$ otherwise (note the strong similarity with Equation (4.16)). Let $\mathbf{z}$ be the vector of the network's outputs:

$$\mathbf{z} = (z_1, z_2, \ldots, z_s). \tag{4.37}$$

Then the cross-entropy loss function is:

$$J(\mathbf{w}, \mathbf{x}) = H(\mathbf{t}, \mathbf{z}) = -\sum_{c=1}^{s} t_c \log(z_c) = -\log(z_g). \tag{4.38}$$

If z_g is obtained by the softmax activation function as in Equation (4.33), then:

$$J(\mathbf{w}, \mathbf{x}) = -\log(z_g) = -\log\left(\frac{e^{\mathrm{net}_g}}{\sum_{c=1}^{s} e^{\mathrm{net}_c}}\right) = -\mathrm{net}_g + \log\left(\sum_{c=1}^{s} e^{\mathrm{net}_c}\right). \tag{4.39}$$

An example of this procedure is illustrated in Figure 4.13.

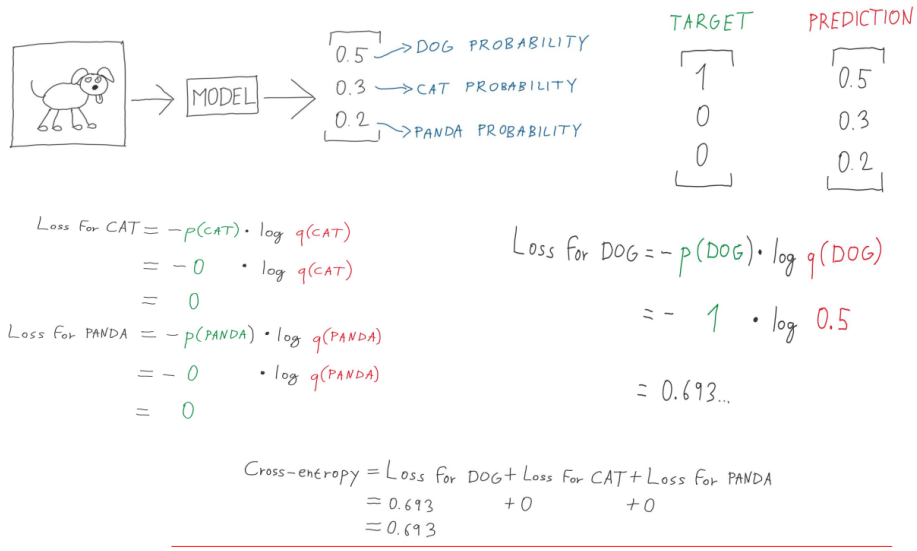


Figure 4.13: Example of cross-entropy loss. Source: [72].

In multi-class classification we use softmax activation function in the final level and cross-entropy as loss function,. If the target vector $\mathbf{t}$ takes the one-hot form the simplification of Equation (4.39) is used. In case of uncertainty about the correct class, it is possible to use *soft target*, with single values in $[0, 1]$ that sum up to 1. In this case we use a complete cross-entropy formula of Definition 4.12.1.

In binary classification it is possible to fall back to the previous case and treat the problem as a multi-class with 2 classes. However, it is preferable for having a better convergence to use 1 output neuron that encodes the probability of the first class only, while the second probability is the complement of 1. To this end, the sigmoid activation function is used to obtain value in $[0, 1]$ for the first neuron. The normalization to 1 of the probability of the two classes is implicit. Note that with 1 neuron t, z are scalar values in $[0, 1]$ (if there is no uncertainty about the correct class, they are in $\{0, 1\}$). In this case we have:

$$H(t, z) = -(t \log(z) + (1 - t) \log(1 - z)) \tag{4.40}$$

A special case is the multi-label classification where a pattern can belong to several classes. For example, an image that contains both a dog and a cat could be classified as belonging to 2 classes. In this case each output is in $[0, 1]$ but the sum constraint to 1 must be relaxed for the outputs. This can be obtained using the *Binary Cross-Entropy loss* [111]. It is a sigmoid activation plus a Cross-Entropy loss. Each output node represents a binary classification problem, for which we can compute the Cross-Entropy loss, i.e. for the neuron c (corresponding to the c-th class) we compute $H(t_c, z_c)$. Then we sum up each $H(t_c, z_c)$ to compute the loss for the pattern. Formally:

$$J(\mathbf{w}, \mathbf{x}) = \sum_{c=1}^{s} H(t_c, z_c) = \sum_{c=1}^{s} H\left(t_c, \frac{1}{1 + e^{-\mathrm{net}_c}}\right). \tag{4.41}$$

In practice, the most seen form of Equation (4.41) is obtained combining each term with Equation (4.40) for the c-th term of the sum. In formulas:

$$J(\mathbf{w}, \mathbf{x}) = -\sum_{c=1}^{s} t_c \log(z_c) + (1 - t_c) \log(1 - z_c). \tag{4.42}$$

For example, consider a pattern $\mathbf{x}$ and let $\mathbf{t} = (1, 1, 0, 0)$ and for the output layer $\mathbf{net} = (2.197, 1.386, -2.197, -1.386)$. We have that $\mathbf{z} = (0.9, 0.8, 0.1, 0.2)$. Then (considering the logarithm with base 2) $H(t_1, z_1) = H(1, 0.9) = 0.152$. Similarly, we have $H(t_2, z_2) = 0.322$, $H(t_3, z_3) = 3.322$ and $H(t_4, z_4) = 2.322$. Then, by Equation (4.41), we have $J(\mathbf{w}, \mathbf{x}) = 0.152 + 0.322 + 3.322 + 2.322 = 6.118$.

4.13 Loss regularization

To reduce the risk of overfitting of the training set of a neural network with many parameters, *regularization* techniques can be used. Regularization is very important when the training set is not large compared to the model complexity. Networks whose weights, or part of them, assume small values (close to zero) produce more regular and stable output, often leading to better generalization. This is a form of regression, that constrains (regularizes, shrinks) the coefficient estimates towards zero. In other words, this technique discourages learning a more complex or flexible model, so as to avoid the risk of overfitting. To push the network to adopt weights of small value, a regularization term can be added to the loss. For example in the case of Cross-Entropy Loss:

$$J(\mathbf{w}) = J_{\text{Cross-Entropy}}(\mathbf{w}) + J_{\text{Reg}}(\mathbf{w}). \tag{4.43}$$

Let $\lambda \in \mathbb{R}^+$. In the l_2 regularization, the added term corresponds to the sum of the squares of all the weights of the network (notice that the aim is to minimize this term):

$$J_{\text{Reg}}(\mathbf{w}) = \frac{1}{2}\lambda \sum_i w_i^2. \tag{4.44}$$

In the l_1 regularization the absolute value is used:

$$J_{\text{Reg}}(\mathbf{w}) = \lambda \sum_i |w_i|. \tag{4.45}$$

l_1 can have a sparsifying effect (i.e. bring numerous weights to 0) greater than l_2. In fact, when the weights assume values close to zero, the calculation of the square in l_2 has the effect of excessively reducing the corrective measures to the weights, making it difficult to zero them.

4.14 Weight decay

Weight decay is l_2 regularization applied to a neural network. Consider Equation (4.43). By linearity of partial derivatives, it is easy to see that:

$$\frac{\partial J_{\text{Tot}}}{\partial w_k} = \frac{\partial J_{\text{Cross-Entropy}}}{\partial w_k} + \frac{\partial J_{\text{Reg}}}{\partial w_k}. \tag{4.46}$$

We compute the second term as:

$$\frac{\partial J_{\text{Reg}}}{\partial w_k} = \frac{\partial}{\partial w_k}\left(\frac{1}{2}\lambda \sum_i w_i^2\right) = \lambda w_k. \tag{4.47}$$

Therefore, updating of weights following backpropagation includes an additional term called weight decay which has the effect of pulling them towards 0. For example, considering the SGD weights update with no regularization term

$$w_k \leftarrow w_k - \eta \frac{\partial J_{\text{Cross-Entropy}}}{\partial w_k}, \tag{4.48}$$

it becomes

$$w_k \leftarrow w_k - \eta \left(\frac{\partial J_{\text{Cross-Entropy}}}{\partial w_k} + \underbrace{\lambda w_k}_{\text{weight decay}}\right). \tag{4.49}$$

Analogous reasoning for the l_1 regularization.

Chapter 5

Learning rate

Calibrating the learning rate in an optimal way is very important (Figures 5.1 and 5.2):

- If it is too small, SGD will have slow convergence.
- If it is too large, SGD will swing and/or diverge.

The optimal value changes according to the network architecture, to the loss function, etc. If we start from a known architecture already applied by others to similar problems, we should start from the recommended values of the learning rate and try to increase or decrease it by monitoring convergence and generalization. In any case, the fact that the loss decreases without oscillating does not mean that the value is optimal (especially in the tuning of pre-trained networks): we should monitor accuracy on the validation set. Adaptive learning rate techniques can help.

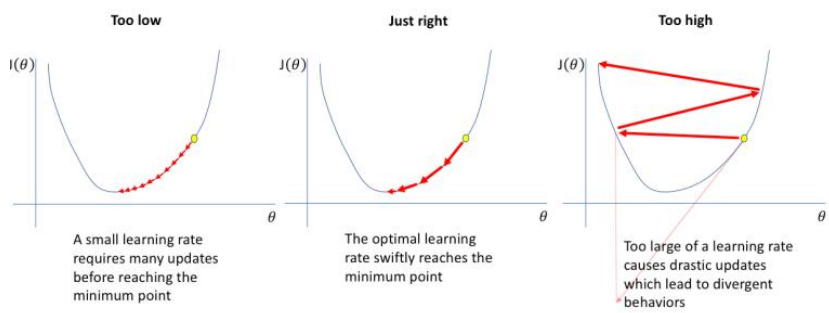


Figure 5.1: Three main cases of learning rate. Source: [72].

5.1 Learning rate annealing

Learning rate annealing consists in starting with a relatively high learning rate and then gradually lowering it during training. The intuition behind this approach is that we would like to traverse quickly from the initial parameters to a range of "good" parameter values. After that, we need a learning rate small enough that we can explore the "deeper, but narrower parts of the loss function". The most popular form of learning rate annealing is a *step decay* reported in Figure 5.3, where the learning rate is reduced by some percentage after a set number of training epochs.

5.2 Learning rate schedule

More generally, we can establish that it is useful to define a *learning rate schedule* in which the learning rate is updated during training according to some specific rule.

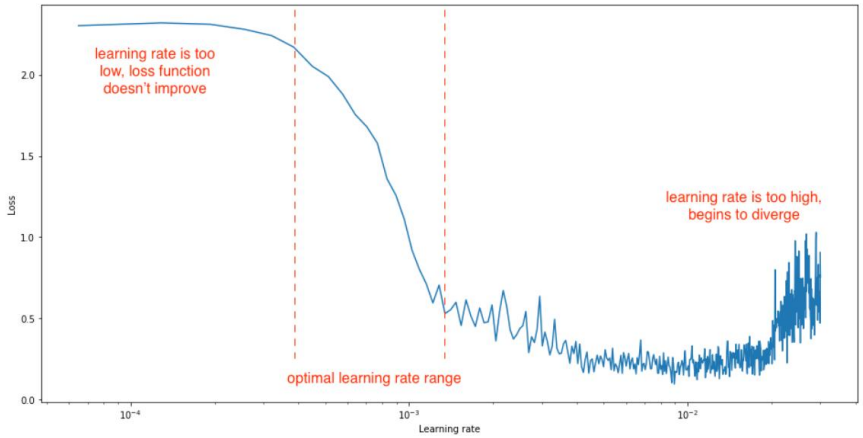


Figure 5.2: Loss oscillation in function of the learning rate. Source: [72].

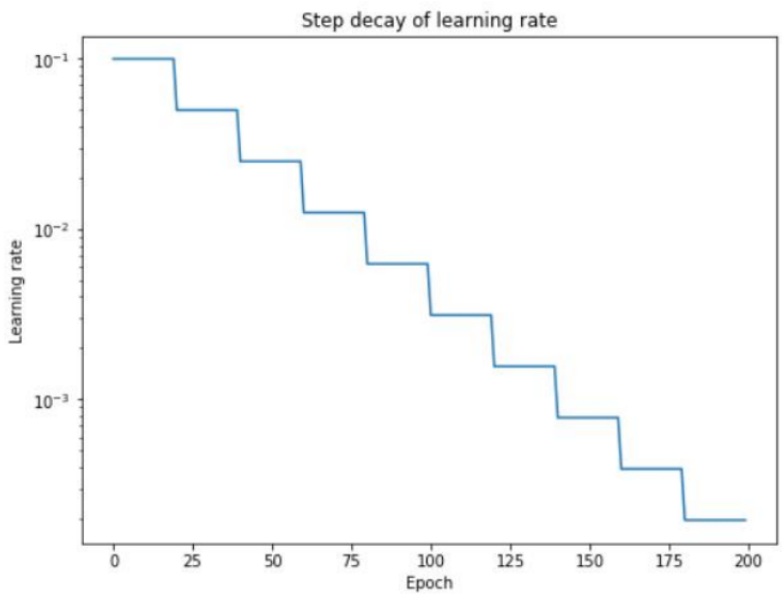


Figure 5.3: Step decay of the learning rate. Source: [72].

5.3 Cyclic learning rate

In *Cyclic learning rate* (CLR) the learning rate fluctuates between the range of learning rate. We need to define:

1. The bounds between which the learning rate will vary, *base learning rate* (base_lr) and *max learning rate* (max_lr).

2. The step_size.

In CLR, we vary the learning rate between a lower and higher bound. The logic is that periodic higher learning rates within each epoch helps to come out of any saddle points or local minima. If the saddle point happens to be an elaborated plateau, lower learning rates will probably never generate enough gradient to come out of it, resulting in difficulty in minimizing the loss.

Some examples of CLR are reported in Figures 5.4, 5.5 and 5.6. The input parameter step_size is the number of iterations in half a cycle [99].

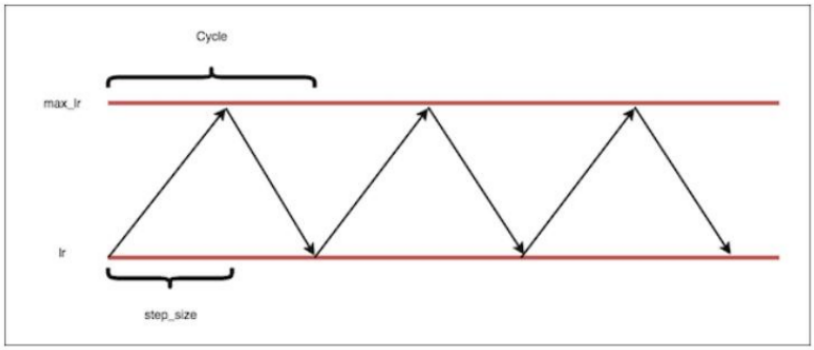


Figure 5.4: Triangular CLR. Source: [72].

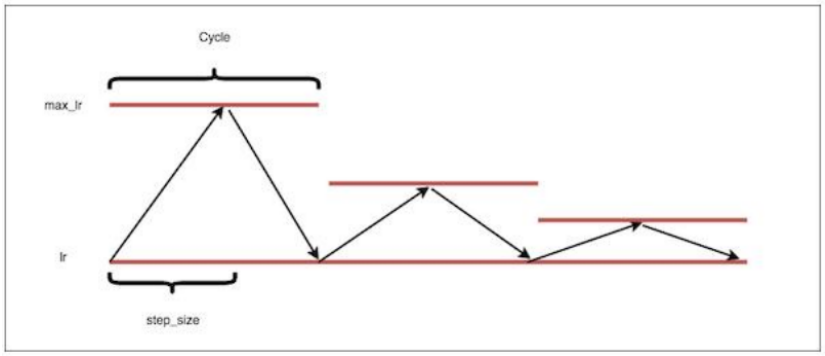


Figure 5.5: Triangular 2, triangular CLR where max_lr is halved after every cycle. Source: [72].

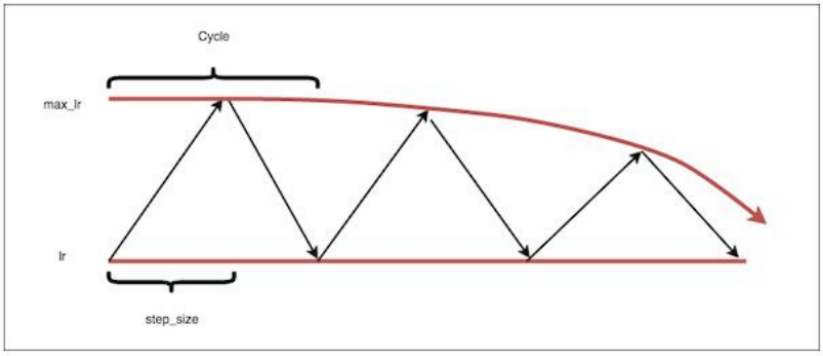


Figure 5.6: Triangular CLR where max_lr is reduced exponentially with each iteration. Source: [72].

5.3.1 Triangular policy

The triangular policy in Figure 5.4 can be described as [72]:

$$\eta_t = \eta_{\min} + (\eta_{\max} - \eta_{\min}) \max(0, 1 - x), \tag{5.1}$$

where

$$x = \left| \frac{\text{iterations}}{\text{stepsize}} - 2 \cdot \text{cycle} + 1 \right|, \tag{5.2}$$

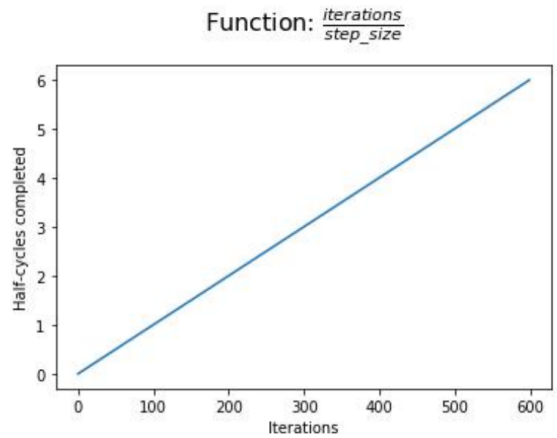
and

$$\text{cycle} = \left\lfloor 1 + \frac{\text{iterations}}{2 \cdot \text{stepsize}} \right\rfloor, \tag{5.3}$$

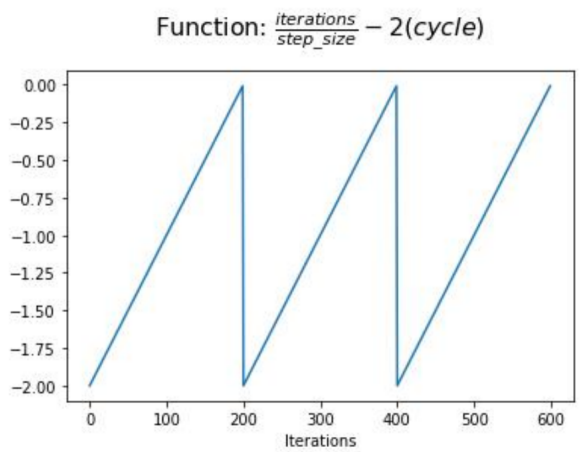
where $\eta_{\min}$ and $\eta_{\max}$ define the bounds of the learning rate, iterations represents the number of completed mini-batches, stepsize is the number of iterations in half a cycle.

Now we will progressively build the equation with visualizations. For the visual below, the triangular update for 3 full cycles are shown with a step size of 100 iterations.

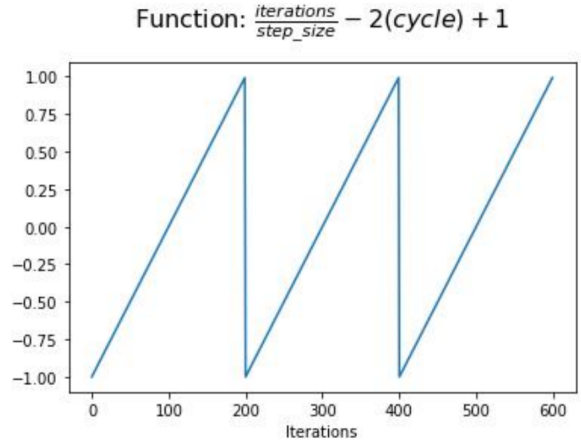
Foremost, we can establish our "progress" during training in terms of half-cycles we have completed. We measure out progress in terms of half-cycles and not full cycles so that we can achieve symmetry within a cycle.



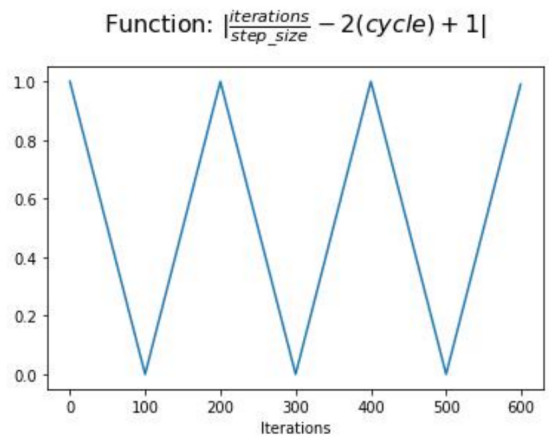
Next, we compare our half-cycle progress to the number of half-cycles which will be completed at the end of the current cycle. At the beginning of a cycle, we have two half-cycles yet to be completed. At the end of a cycle, this value reaches zero.



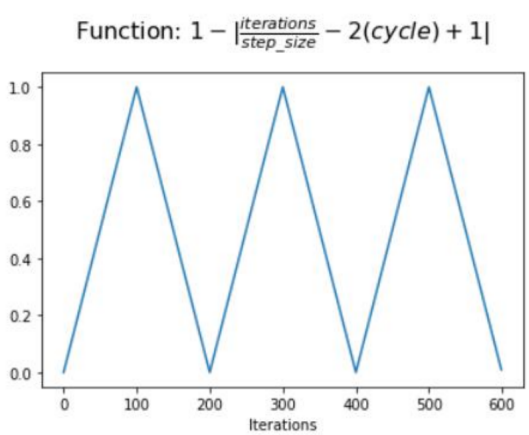
Next, we add 1 to this value in order to shift the function to be centered on the y-axis. Now we are showing our progress within a cycle with reference to the half-cycle point.



At this point, we can take the absolute value to achieve a triangular shape within each cycle. This is the value that we assign to x.



However, we would like our learning rate schedule to start at the minimum value, increasing to the maximum value at the middle of a cycle, and then decrease back to the minimum value. We can accomplish this by simply computing $1 - x$.



5.3.2 Effects of CLR, sharp and flat minima

The main assumption behind the rationale for CLR (as opposed to one which only decreases) is that increasing the learning rate might have a short term negative effect and yet achieve a longer term

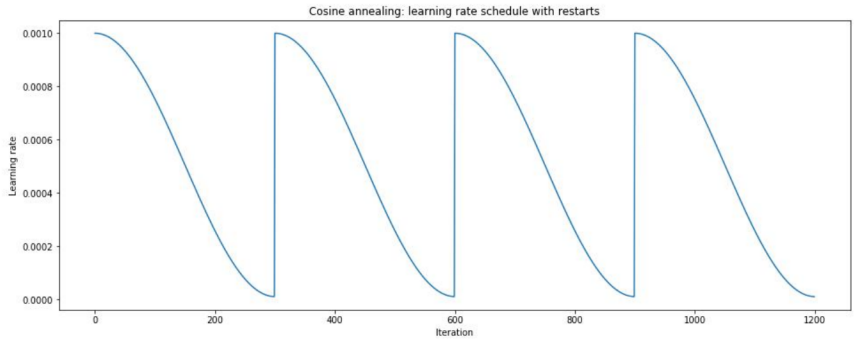


Figure 5.7: Cosine annealing: learning rate schedule with restarts.

beneficial effect. Smith in [99] includes several examples of a loss function evolution which temporarily deviates to higher losses while ultimately converging to a lower loss when compared with a benchmark fixed learning rate.

To gain intuition on why this short-term effect would yield a long-term positive effect, it is important to understand the desirable characteristics of our converged minimum. Ultimately, we would like our network to learn from the data in a manner which generalizes to unseen data. Further, a network with good generalization properties should be robust in the sense that small changes to the network's parameters do not cause drastic changes to performance. With this in mind, it makes sense that *sharp minima* lead to poor generalization, as small changes to the parameter values may result in a drastically higher loss. By allowing for our learning rate to increase at times, we can "jump out" of sharp minima which would temporarily increase out loss but may ultimately lead to convergence on a more desirable minima.

5.3.3 Stochastic Gradient Descent with Warm Restarts

In *Stochastic Gradient Descent with Warm Restarts* (SGDR) an aggressive annealing schedule is combined with periodic *restarts* to the original starting learning rate. Figure 5.7 shows a *cosine annealing* with restarts. The schedule can be rewritten as:

$$\eta_t = \eta^i_{\text{min}} + \frac{1}{2}\left(\eta^i_{\text{max}} - \eta^i_{\text{min}}\right)\left(1 + \cos\left(\frac{T_{\text{current}}}{T_i}\pi\right)\right) \tag{5.4}$$

where η_t is the learning rate at timestep t (incremented each mini batch), η^i_{max} and η^i_{min} define the range of desired learning rates, T_{current} represents the number of epochs since the last restart (this value is calculated at every iteration and thus can take on fractional values), and T_i defines the number of epochs in a cycle. We now break this equation down.

This annealing schedule relies on the cosine function, which varies between -1 and 1. $\frac{T_{\text{current}}}{T_{\text{i}}}$ is capable of taking on values between 0 and 1, which, when multiplied by π, is the input of our cosine function. The corresponding region $[0, \pi]$ of the cosine function is highlighted below in green.

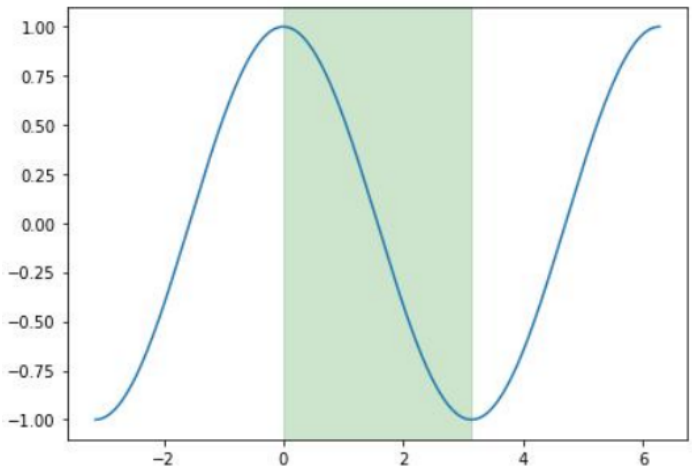


Figure 5.8: Source: [72].

By adding 1, our function varies between 0 and 2, which is then scaled by $\frac{1}{2}$ to now vary between 0 and 1 . Thus, we are simply taking the minimum learning rate and adding some fraction of the specified learning rate range $(\eta^i_{\max} - \eta^i_{\min})$. Because this function starts at 1 and decreases to 0, the result is a learning rate which starts at the maximum of the specified range and decays to the minimum value. Once we reach the end of a cycle, T_{current} resets to 0 and we start back at the maximum learning rate.

The authors note that this learning rate schedule can further be adapted to:

1. Lengthen the cycle as training progresses.
2. Decay $\eta^i_{\max}$ and $\eta^i_{\min}$ after each cycle.

5.3.4 Visualization of CLR

By drastically increasing the learning rate at each restart, we can essentially exit a local minima and continue exploring the "loss landscape". In fact, by snapshotting the weights at the end of each cycle, researchers were able to build an ensemble of models at the cost of training a single model. This is because the network "settles" on various local optima from cycle to cycle, as shown in Figure 5.9.

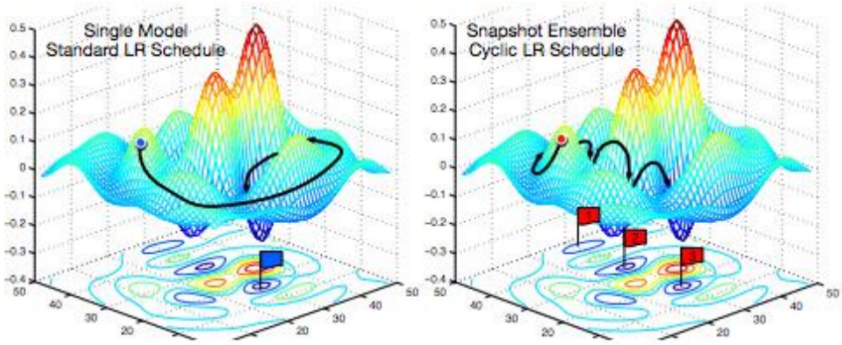


Figure 5.9: Left: Illustration of SGD optimization with a typical learning rate schedule. The model converges to a minimum at the end of training. Right: Illustration of Snapshot Ensembling. The model undergoes several learning rate annealing cycles, converging to and escaping from multiple local minima. We take a snapshot at each minimum for test-time ensembling. Source: [72].

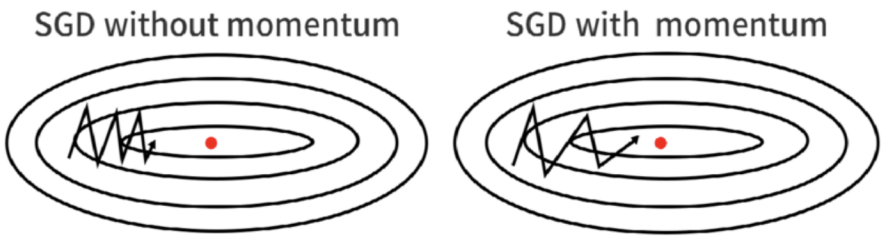


Figure 5.10: Effect of the momentum on SGD. Source: [72].

5.4 Momentum

Momentum is an extension to the GD optimization algorithm that allows the search to build inertia in a direction in the search space and overcome the oscillations and coast across flat spots of the search space.

SGD with mini-batches can cause a "zigzag gradient" which slows down convergence and makes it less stable. We can think of SGD as a ball rolling down on a valley. We want it to sit in the deepest place of the mountains. However, it is easy to see that things can go wrong. Attracted by the force of gravity, the ball tries to move to the lowest point. In physics, the moment gives a motion without sudden oscillations and changes of direction.

In SGD, to avoid oscillations, the physical behavior can be emulated: the last update of each parameter is saved, and the new update is calculated as a linear combination of the previous update (which gives stability) and the current gradient (which corrects the direction).

Without momentum, the *weight increment* of w_{ij} at iteration t is:

$$\Delta w_{ij}^t = -\eta \frac{\partial J}{\partial w_{ij}}. \tag{5.5}$$

If we add momentum, it becomes:

$$\Delta w_{ij}^t = -\eta \frac{\partial J}{\partial w_{ij}} + \gamma \Delta w_{ij}^{t-1}, \tag{5.6}$$

where γ is the *momentum factor* (or *decay rate*) and Δw_{ij}^{t-1} is the weight increment at the previous iteration. Typically, $\gamma = 0.9$. Figure 5.10 shows the effect of momentum on SGD.

During training the update direction tends to resist change when momentum is added to the update scheme. When the neural network approaches a shallow local minimum it is like applying brakes, but not sufficiently to instantly affect the update direction and magnitude. Hence the neural networks trained this way will overshoot past smaller local minima points and only stop in a deeper global minimum.

Thus momentum in neural networks helps them get out of local minima points so that a more important global minimum is found. However, too much momentum may create issues, as well as systems that are not stable may create oscillations that grow in magnitude. In such cases one needs to add decay terms and so on. It is physics applied to neural network training or numerical optimizations.

5.5 One-cycle learning rate

The notorious AI company Fast.ai no longer recommends cosine annealing discussed in Section 5.3.3 because it is no longer the most performant general-purpouse learning rate scheduler. Instead, the *one-cycle rate scheduler* [100] should be preferred. It has three phases:

1. Warm up, in which the learning rate increases.
2. Cool down, in which the learning rate decreases.
3. Fine-tune, where we annihilate the learning rate way below the initial learning rate value.

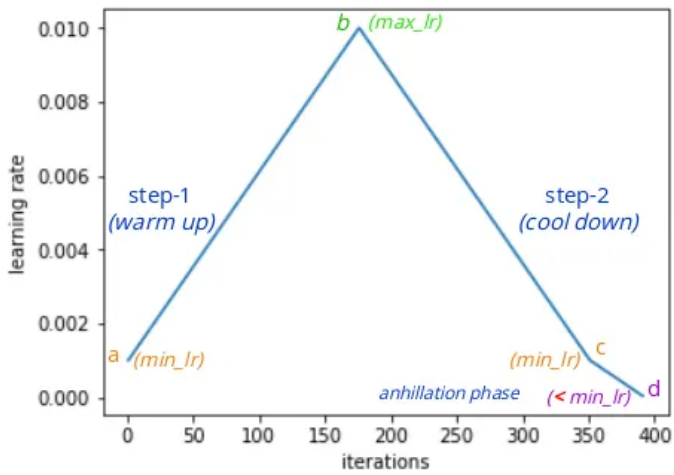


Figure 5.11: One-cycle policy for the learning rate. Source: [80]

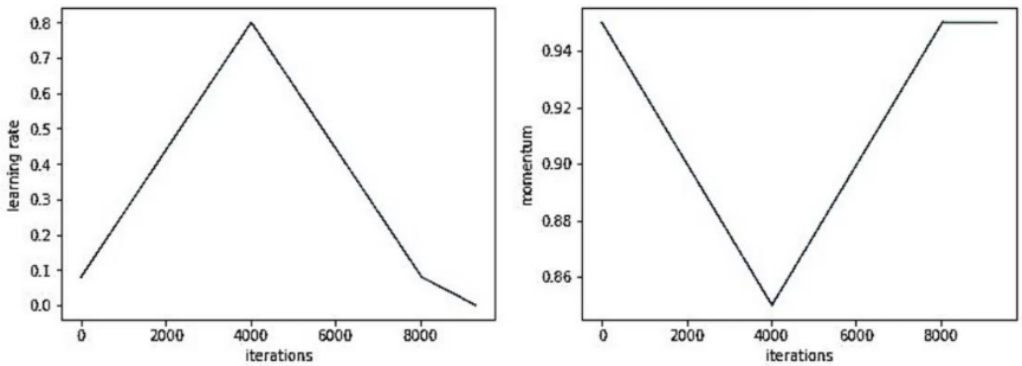


Figure 5.12: Left: One-cycle policy for learning rate. Right: One-cycle policy for momentum. Source: [72].

Let the cycle length be slightly lesser than the total number of epochs to be trained. This is to allow the small fine-tune phase after the first two. During that period, the learning rate drops to 1/10 of its initial value. The one-cycle policy is illustrated in Figure 5.11.

The one-cycle policy can be applied to both learning rate and momentum, as shown in Figure 5.12. Momentum is counterproductive when the learning rate is very high, which is why momentum is annealed in the opposite of the way in which the learning rate is annealed in the optimizer.

5.6 Super-convergence

The authors of [100] refers to *super-convergence*, a phenomenon where neural networks can be trained an order of magnitude faster than with standard training methods. The existence of super-convergence is relevant to understanding why deep networks generalize well. One of the key elements of super-convergence is training with one learning rate cycle (such as the one described in Section 5.5) and a large maximum learning rate. Figure 5.13 shows two examples of super-convergence. Though we should not count on results this compelling in practice, super-convergence has indeed been demonstrated on a broad range of datasets and problem domains.

There are several other papers in the literature which state that wide, flat local minima produce solutions that generalize better than sharp minima [100]. Super-convergence results align with these results in the middle of training, yet a small learning rate is necessary at the end of training, implying that the local minimum is narrow. In fact, [35] showed that also sharp minima can generalize. This work states the following: "It has been observed empirically that minima found by standard deep

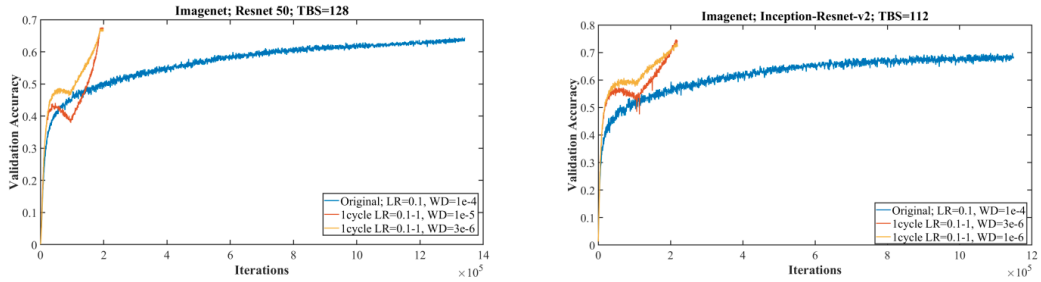


Figure 5.13: Training ResNet50 (Left) and Inception-resnet-v2 (Right) architectures on the ImageNet dataset with the standard learning rate policy (blue curve) versus a one-cycle policy that displays super-convergence. Illustrates that deep neural networks can be trained much faster (20 versus 100 epochs) than by using the standard training methods. Source: [100].

learning algorithms that generalize well tend to be flatter than found minima that did not generalize well. However, when following several definitions of flatness, we have shown that the conclusion that flat minima should generalize better than sharp ones cannot be applied as is without further context. Previously used definitions fail to account for the complex geometry of some commonly used deep architectures. In particular, the non-identifiability (i.e. some parameters can be compensated by other parameters) of the model induced by symmetries, allows one to alter the flatness of a minimum without affecting the function it represents. Additionally the whole geometry of the error surface with respect to the parameters can be changed arbitrarily under different parametrizations. Our work indicates that more care is needed to define flatness to avoid degeneracies of the geometry of the model under study. Also such a concept can not be divorced from the particular parametrization of the model or input space."

5.7 Adaptive learning rate

Adaptive learning rate methods provide heuristic approach without requiring expensive work in tuning hyperparameters for the learning rate schedule manually [62]. In fact, in addition to momentum, there are adaptive learning approaches for updating the weights that can accelerate the convergence in the gradient descent:

- Nesterov Accelerate Gradient (NAG)
- Adaptive Gradient (Adagrad)
- Adadelta
- RMSProp
- Adam

Most of the above approches adapt the global learning rate to each specific weight. Adam is considered to be the state of the art, but doesn't always deliver better results than the more classic SGD with momentum.

5.7.1 Adam

Adaptive Moment Estimation, in short *Adam*, is an optimizer introduced in [58] that computes adaptive learning rates for each parameter combining the ideas of momentum and adaptive gradient. Its update rule is based on the value of the gradient at the current step and on the exponential moving averages of the gradient and its square. More precisely, Adam defines the *moving averages* at time t, $\mathbf{m}_t$ (the first moment) and $\mathbf{u}_t$ (the second moment) as:

$$\mathbf{m}_t = \rho_1 \mathbf{m}_{t-1} + (1 - \rho_1)\mathbf{g}_t, \tag{5.7}$$

$$\mathbf{u}_t = \rho_2 \mathbf{u}_{t-1} + (1 - \rho_2)\mathbf{g}_t^2, \tag{5.8}$$

where $\mathbf{g}_t = \nabla J_t$ the gradient at time t, the square on $\mathbf{g}_t$ stands for the component-wise square (e.g. $(1, -2, 3)^2 = (1, 4, 9)$), $\rho_1, \rho_2 \in \mathbb{R}$ are hyperparameters representing the exponential decay rate for the first moment and the second moment estimates (usually set 0.9 and 0.999 respectively). The moments are initialized to a vector of 0, $\mathbf{m}_0 = \mathbf{u}_0 = 0$. Note that, if there are n weights in the model, $\mathbf{m}_t, \mathbf{u}_t, \mathbf{g}_t \in \mathbb{R}^n$. In order to take into account the fact that the value of moving averages will be very small due to their initialization to zero (especially in the first step), the authors of [58] define a bias-corrected version of the moving averages:

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \rho_1^t}, \tag{5.9}$$

$$\hat{\mathbf{u}}_t = \frac{\mathbf{u}_t}{1 - \rho_2^t}. \tag{5.10}$$

The final update for the weights $\mathbf{w}_t$ at time t of the network is:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{u}}_t} + \epsilon}, \tag{5.11}$$

where η is the learning rate, $\epsilon > 0$ is a very small positive number to prevent any division by zero, usually set to 10^{-8} and all operations are meant to be component-wise. Notice that, while $\mathbf{g}_t$ might have positive or negative components, $\mathbf{g}_t^2$ has only positive components. Hence, if the gradient changes sign often, the value of $\hat{\mathbf{m}}_t$ might be much lower than $\sqrt{\hat{\mathbf{u}}_t}$. This means that in this case the step size is very small. In fact, if the gradient changes sign often, it means that sometimes we increase the value of $\hat{\mathbf{m}}_t$, and other times we decrease it. Therefore, the value of the first moment does not change across several time steps t. Instead, in the case of the second moment, the value always increases, thanks to the term $\mathbf{g}_t^2$. So, sooner or later, in this situation the second moment value will be much larger than the first moment value.

Intuition

The intuition behind the Adam is that we don't want to roll so fast just because we can jump over the minimum, we want to decrease the velocity a little bit for a careful search. Adam tends to converge faster, while SGD often converges to more optimal solutions. SGD's high variance disadvantages gets rectified by Adam. To summarize:

- Adam converges rapidly to a sharp minimum.
- SGD is computationally heavy, converges to a flat minimum and performs well on the test data.

Adam is useful when the computation time of SGD is unfeasible or very expensive. Moreover, it is usually applied in transformers.

One-cycle SGD vs Adam

Fast.ai recommends one-cycle SGD over Adam because, subject to some tuning (getting max_lr correct is particularly important), it trains models with roughly equal or marginally worse performance in a fraction of the time, thanks to super-convergence.

However, despite Fast.ai advocacy, at present, most practitioners still use the Adam optimizer as their default. For example, in Kaggle competitions the use of Adam predominates. In large part, it is because Adam is pretty much parameter-free and it is much more robust to model changes than one-cycle SGD. This makes it much easier to develop with, as it is one fewer set of hyperparameters that you have to optimize. These benefits seem to have proven to be more important than the computational benefit of one-cycle SGD super-convergence in practice.

However, once we are in the later optimization stage of a medium-sized model training project, experimenting with moving off Adam and onto one-cycle SGD is well worth doing.

5.7.2 diffGrad

diffGrad is Adam, but with an adaptive "friction clamp" built in that helps the optimizer lock down into global optima better than Adam. It takes into account the difference of the gradient in order to set the learning rate. According to the observation that when gradient changes begin to reduce over training, this is often indicative of the presence of a global minima, diffGrad applies an adaptive adjusting driven by the difference between the present and the immediate past to lock parameters into global minima. Therefore, the step size is larger for faster gradient changing and lower for lower gradient changing parameters. Let

$$\Delta \mathbf{g}_t = \left| \mathbf{g}_t - \mathbf{g}_{t-1} \right|, \tag{5.12}$$

$$\boldsymbol{\xi}_t = \sigma(\Delta \mathbf{g}_t), \tag{5.13}$$

where $\sigma(\mathbf{x})$ performs component-wise sigmoid operation (e.g. $\sigma((-1,0,1)) = (0.269, 0.5, 0.731)$). Note that $\Delta g_{t,i} \geq 0$, therefore $\xi_{t,i} \in [0.5, 1]$. The final update for the weights $\mathbf{w}_t$ at time t is:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \cdot \boldsymbol{\xi}_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{u}}_t} + \epsilon}, \tag{5.14}$$

where η is the learning rate, $\epsilon > 0$ is a very small positive number to prevent any division by zero, usually set to 10^{-8} and all operations are meant to be component-wise.

If we excplitly consider each component of $\mathbf{w}_t$, as done in [38], we have:

$$\Delta g_{t,i} = \left| g_{t,i} - g_{t-1,i} \right|, \tag{5.15}$$

$$\xi_{t,i} = \sigma(\Delta g_{t,i}), \tag{5.16}$$

and the update for each $w_{t,i}$ at time t is:

$$w_{t,i} = w_{t-1,i} - \eta \cdot \xi_{t,i} \frac{\hat{m}_{t,i}}{\sqrt{\hat{u}_{t,i}} + \epsilon}. \tag{5.17}$$

5.8 Convergence problems: check list

Here are some things to check when the convergence on the training set does not start (the network seems to be blocked):

- Have the inputs been normalized (min-max, standard scaling)?
- In the case of logistic regression, has the output been normalized to $[0, 1]$?
- Have the weights been randomized into reasonable ranges?

If all of these points are checked, we could be in the saturation zone from which it is not possible to escape.

- Are the patterns of the different classes randomly mixed within the mini-batches?
- Are the labels in the right format (scattered integers, one hot vectors, float by regression, etc.)?
- Is the loss function correct and adequate for the type of provided labels? Pay attention to the difference between sparse labels and one-hot vectors: sparse labels represent the categories directly by their indices, rather than using a one-hot encoding. For example, if we have three categories cat, dog, bird, and a sample is labeled as "cat", the sparse label might simply be 0, indicating the index of "cat" in the categorical list [ChatGPT 3.5]?
- Is the learning rate adequate?

Chapter 6

Overfitting and dropout

6.1 Overfitting revised - linear models and neural networks

Faced with more features than examples, linear models tend to overfit. But, given more examples than features, we can generally count on linear models not to overfit. Unfortunately, the reliability with which linear models generalize comes at a cost. Naively applied, linear models do not take into account interactions among features. For every feature, a linear model must assign either a positive or a negative weight, ignoring context.

In traditional texts, this fundamental tension between generalizability and flexibility is described as the *bias-variance tradeoff* already presented in Section 3.23. Linear models have high bias: they can only represent a small class of functions. However, these models have low variance: they give similar results across different random samples of the data.

DNNs inhabit the opposite end of the bias-variance spectrum. Unlike linear models, neural networks are not confined to looking at each feature individually. They can learn interactions among groups of features. For example, they might infer that "Nigeria" and "Western Union" appearing together in an email indicates spam but that separately they do not.

Even when we have far more examples than features, DNNs are capable of overfitting. In 2017, a group of researchers demonstrated the extreme flexibility of neural networks by training deep nets on randomly labeled images. Despite the absence of any true pattern linking the inputs to the outputs, they found that the neural network optimized by SGD could label every image in the training set perfectly. This means that, if the labels are assigned uniformly at random and there are 10 classes, then no classifier can do better than 10% accuracy on holdout data and the generalization gap here is a whopping 90%. In fact, by overfitting over the randomly labeled images, we manage to obtain nearly a 100% accuracy on the training set (by definition of overfitting). If our models are so expressive that they can overfit this badly, then when should we expect them not to overfit?

6.2 Robustness through perturbations

A good predictive model should perform well on unseen data. Classical generalization theory suggest that to close the gap between train and test performance, we should aim for a simple model. Simplicity can come in the form of small number of dimension.

Another useful notion of simplicity is *smoothness*, i.e., that the function should not be sensitive to small changes to its inputs. For instance, when we classify images, we would expect that adding some random noise to the pixels should be mostly harmless. In 1995, Bishop formalized this idea when he proved that training with input noise is equivalent to Tikhonov regularization. This work drew a clear mathematical connection between the requirement for a function to be smooth (and thus simple) and its resilience to perturbations in the input.

6.3 Dropout

In 2014 Srivastava et al. developed a clever idea for how to apply Bishop's idea to the internal layers of a network too. Namely, they proposed to inject noise into each layer of the network before calculating the subsequent layer during training. They realized that when training a DNN with many layers, injecting noise enforces smoothness just on the input-output mapping.

Their idea, called *dropout*, involves injecting noise while computing each internal layer during forward propagation, and it has become a standard technique for training neural networks. The method is called dropout because we literally drop out some neurons during training. Throughout training, on each iteration, standard dropout consists of *zeroing* out some fraction of the nodes in each layer before calculating the subsequent layer.

6.3.1 Co-adaptation

The authors argue that neural network overfitting is characterized by a state in which each layer relies on a specific pattern of activations in the previous layer, calling this condition *co-adaptation*. In neural networks, co-adaptation refers to when different hidden units in a neural networks have highly correlated behavior. In fact, as a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features providing some specialization. Neighboring neurons become to rely on this specialization, which if taken too far can result in a fragile model too specialized (overfit) to the training data. It is better for computational efficiency and the the model's ability to learn a general representation if hidden units can detect features independently of each other [15].

The authors claimed that dropout is useful to reduce co-adaptation. We can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network. The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data.

6.3.2 Implementation

The key challenge then is how to inject this noise. One idea is to inject the noise in an *unbiased* manner so that the expected value of each layer, while fixing the others—equals to the value it would have taken absent noise.

Input noise

In Bishop work, he added Gaussian noise to the inputs to a linear model. At each training iteration, he added noise ϵ sampled from a distribution $\mathcal{N}(0, \sigma^2)$ to the input $\mathbf{x}$, yielding a perturbed point $\mathbf{x}' = \mathbf{x} + \epsilon$. In expectation, $\mathbb{E}[\mathbf{x}'] = \mathbf{x}$.

Dropout

In standard dropout regularization, one debiases each layer by normalizing by the fraction of nodes that were retained (not dropped out). In other words, with *dropout probability* p, each intermediate activation h, $h = f(\text{net}_i)$ for some hidden layer's node i, is replaced by a random variable h' as follows:

$$h' = \begin{cases} 0 & \text{with probability } p, \\ \frac{h}{1-p} & \text{with probability } 1-p. \end{cases} \tag{6.1}$$

Trivially, $\mathbb{E}[h'] = h$. Figure 6.1 shows an example of dropout on a hidden layer. Note that, when we apply dropout to a hidden layer, zeroing out each hidden unit with probability p, the result can be viewed as a network containing only a subset of the original neurons. In Figure 6.1, the hidden nodes h_2 and h_5 are removed. Consequently, the calculation of the outputs no longer depends on h_2 or h_5 and their respective gradient also vanishes when performing backpropagation. In this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \ldots, h_5$. Typically, we disable dropout at test time. Given a trained model and a new example, we do not dropout any nodes and

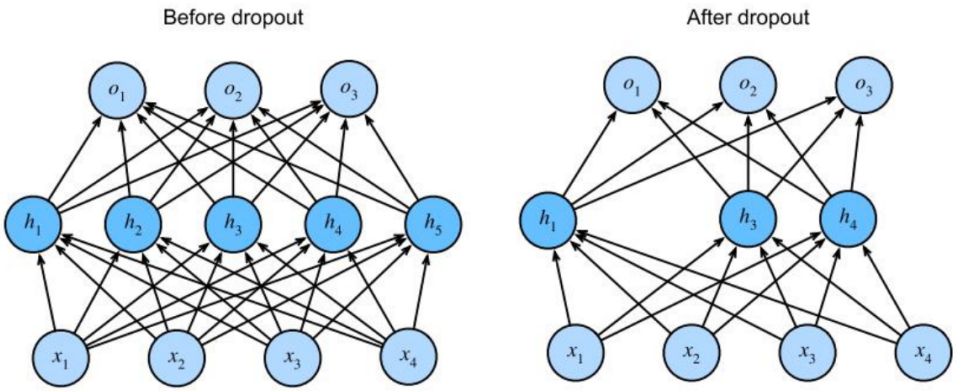


Figure 6.1: MLP before (left) and after (right) dropout. Source: [72].

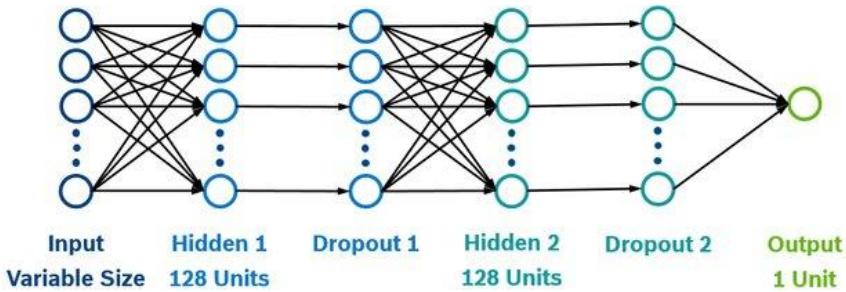


Figure 6.2: Example of dropout layers in a neural network. Source: [72].

thus do not need to normalize. However, there are some exceptions. In fact, some researchers use dropout at test time as a heuristic for estimating the *uncertainty* of neural network predictions: if the predictions agree across many different dropout masks, then we might say that the network is more confident.

Many frameworks, such as Keras, implements as a special layer [37]. Figure 6.2 displays two dropout layers.

Chapter 7

Deep Convolutional Neural Networks

The term *Deep Neural Network* (DNN) denotes networks made up of at least two hidden layers (hence the term "deep"). The implications of universal approximation theorem and the difficulty of training networks with many layers have long led to focus on networks with only one layer.

DNN are inspired by the human brain. In fact, our visual system operates on a hierarchy of levels, similarly to the layers of a network. The most commonly used DNNs today consist of a number of levels between 7 and 150. There are "only" about ten levels between the retina and the actuator muscles (otherwise we would be too slow to react to stimuli).

7.1 Complexity

The *depth* of a neural network is the number of its layers. Depth is only one of the complexity factors. Number of neurons, connections and weights also characterize the complexity of a DNN. The greater the number of weights (i.e. parameters to be learned), the greater the complexity of the training. At the same time, a large number of neurons (and connections) makes forward and back propagation more expensive, as the number of operations required increases.

7.2 Topologies

The most well known feedforward models for classification and regression with (mainly) supervised training are:

- *Convolutional Neural Network* (CNN or ConvNet).
- Fully Connected DNN (FC DNN), a MLP with at least two hidden layers.

Moreover, we could have:

- Unsupervised training for "generative" models trained to reconstruct the input, e.g. for pre-training other models and for producing salient features.
- Recurring networks, used for sequences, speech recognition, sentiment analysis natural language processing, etc.
- Reinforcement learning (to learn behaviors).

7.3 Deep Learning vs Machine Learning

CNNs already achieved good performance (but not state of the art) in 1998 in some small datasets (e.g. character recognition, low resolution object recognition), but we have to wait until 2012 (AlexNet) for a radical change of pace. AlexNet does not introduce significant innovations compared to the 1998 LeCun CNNs, but some conditions have changed in the meantime:

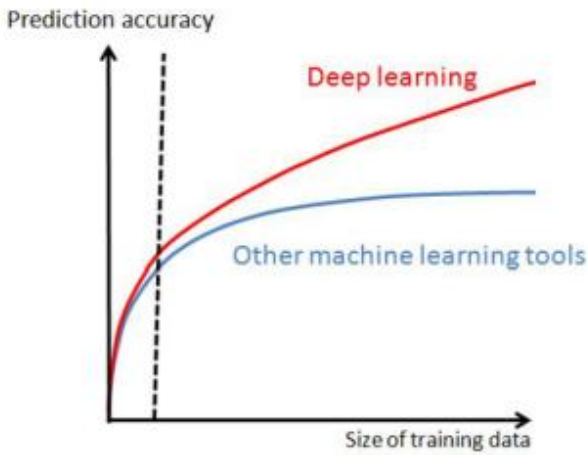


Figure 7.1: Deep Learning vs "classical" Machine Learning. Source: [72].

- Big data, i.e. the availability of large labeled datasets (ImageNet has millions of images and tens of thousands of classes).
- High computational power.

In fact, the superiority of deep learning techniques over other approaches is more clear when large amounts of training data are available (Figure 7.1). It is not a law, but an empirical result shown in many works. CNN may not always outperform a handcrafted feature approach combined with a traditional method like SVM for a specific application.

7.4 Problems of DNNs

We briefly describe some issues regarding DNNs.

7.4.1 GPU computing

The training of complex models requires high computational power. Luckily, the classification time after training is very low, even hundreds of images per second on powerful GPU. The availability of GPUs with thousands of cores and GB of internal memory has allowed to drastically reduce training times: from months to days. Another problem is huge RAM occupation, as hundreds of GB are needed with large datasets.

7.4.2 Vanishing gradient problem

As the backpropagation algorithm advances backward from the output layer towards the input layer, the gradients often get smaller and smaller while approaching zero, which eventually leaves the weights of the initial layers nearly unchanged. As a result, the GD never converges to the optimum. This is known as the *vanishing gradient problem*. An example of vanishing gradient can be seen in Figure 7.2. In fact, one frequent culprit causing the vanishing gradient problem is the choice of the activation function.

Consider a DNN with L layers, input $\mathbf{x}$ and output $\mathbf{z}$. With each layer l defined by a trasformation f_l parametrized by weights $\mathbf{W}^{(l)}$, whose hidden variables is $\mathbf{h}^{(l)}$ (i.e., f_l is the composition of the activation function with the linear operation). Let $\mathbf{h}^{(0)} = \mathbf{x}$. Our network can be expressed as:

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}). \tag{7.1}$$

Therefore:

$$\mathbf{z} = f_L \circ f_{L-1} \circ \cdots \circ f_1(\mathbf{x}). \tag{7.2}$$

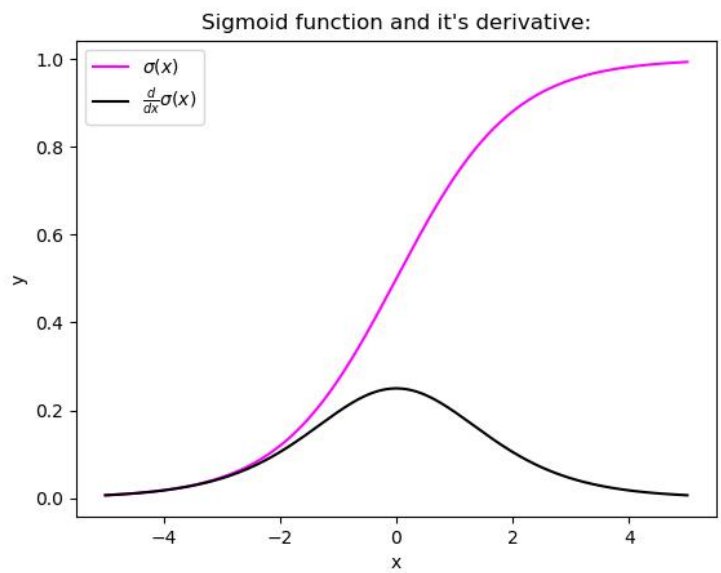


Figure 7.2: Observing the graph of the sigmoid function, we can see that for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero. Thus, when the backpropagation algorithm chips in, it virtually has no gradients to propagate backward in the network, and whatever little residual gradients exist keeps on diluting as the algorithm progresses down through the last layers. So, this leaves nothing for the first layers. Source: [72].

We can write the gradient of $\mathbf{z}$ with respect to $\mathbf{W}^{(l)}$ as:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{h}^{(L-1)}} \cdot \frac{\partial \mathbf{h}^{(L-1)}}{\partial \mathbf{h}^{(L-2)}} \cdot \cdots \cdot \frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \cdot \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{W}^{(l)}}, \tag{7.3}$$

where

$$\frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{h}^{(l-1)}} = \mathbf{M}^{(l)}. \tag{7.4}$$

Then we can write Equation 7.3 as:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{W}^{(l)}} = \mathbf{M}^{(L)} \cdot \mathbf{M}^{(L-1)} \cdot \cdots \cdot \mathbf{M}^{(l+1)} \cdot \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{W}^{(l)}}. \tag{7.5}$$

In other words, this gradient is the product of $L-1$ matrices $\mathbf{M}^{(L)}, \cdots, \mathbf{M}^{(l+1)}$ and the tri-dimensional tensor $\frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{W}^{(l)}}$. Thus, we are susceptible to *arithmetic underflow*, a condition in a computer program where the result of a calculation is a number of more precise absolute value than the computer can actually represent. This condition often appears when multiplying together too many probabilities.

7.4.3 Exploding gradient problem

Contrary to the vanishing gradient problem, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the *exploding gradients problem*. In fact, if the gradient values become very large, this leads to too much variation in the updated weights.

7.5 Parameter initialization

One way of addressing, or at least mitigating, the issues raised above is through careful initialization of the weights. Additional care during optimization and suitable regularization can further enhance stability.

7.5.1 Xavier initialization

Let o_j the output of a neuron j for some fully connected layer l *without non-linearities*. With n_{in} inputs x_i (neurons in layer $l-1$) and their associated weights w_{ij} for this layer, an output is given by

$$o_j = \sum_{i=1}^{n_{\text{in}}} w_{ij} x_i. \tag{7.6}$$

The weights w_{ij} are all drawn independently from the same distribution, which we assume to have zero mean and variance σ^2. For now, let us assume that the input to the layer l x_i also have zero mean and variance γ^2 and that they are independent from w_{ij} and independent of each other. Therefore we can compute the mean and the variance of o_j as follows:

$$\begin{aligned}
\mathbb{E}[o_j] &= \sum_{i=1}^{n_{\text{in}}} \mathbb{E}[w_{ij} x_i] && \text{linearity of expectation} \\
&= \sum_{i=1}^{n_{\text{in}}} \mathbb{E}[w_{ij}]\,\mathbb{E}[x_i] && \text{independence of } w_{ij} \text{ and } x_i \\
&= 0 && \mathbb{E}[w_{ij}] = \mathbb{E}[x_i] = 0 \\
\text{Var}[o_j] &= \mathbb{E}[o_j^2] - \mathbb{E}[o_j]^2 \\
&= \sum_{i=1}^{n_{\text{in}}} \mathbb{E}[w_{ij}^2 x_i^2] - 0 \\
&= \sum_{i=1}^{n_{\text{in}}} \mathbb{E}[w_{ij}^2]\,\mathbb{E}[x_i^2] && \text{independence of } w_{ij} \text{ and } x_i \\
&= n_{\text{in}} \sigma^2 \gamma^2.
\end{aligned}$$

We need to keep the variance fixed by imposing $n_{\text{in}}\sigma^2 = 1$. One way is to set $\sigma = \sqrt{\frac{1}{n_{\text{in}}}}$. Now consider backpropagation. There we face a similar problem, albeit with the gradients being propagated from the layers closer to the output. Using the same reasoning as for forward propagation (as we did in Equation (7.6)), we see that the gradients' variance can blow up unless $n_{\text{out}}\sigma^2 = 1$, where n_{out} is the number of outputs of this layer. This leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously. Instead, we try to satisfy

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1, \tag{7.7}$$

or equivalently

$$\sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}. \tag{7.8}$$

This is the reasoning underlying the now-standard and practically beneficial *Xavier initialization*. This initialization samples the weights from the Gaussian distribution

$$\mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right), \tag{7.9}$$

with zero mean and variance $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$. However, we can also adapt Xavier's intuition to choose the variance when sampling weights from a uniform distribution. Note that the uniform $U(-a, a)$ has variance $\frac{a^2}{3}$. Plugging $\frac{a^2}{3}$ into our condition on σ^2 we initialize the weights according to

$$U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right). \tag{7.10}$$

Though the assumption for nonexistence of non-linearities can be easily violated in neural networks, the Xavier initialization method turns out to work well in practice. As stated in [72, slide 206], biases can be set to pre-defined constant, usually 0 or 0.1 for ReLU activation function.

7.5.2 Heuristic for fully-connected layers

For fully-connected layers we could use the following heuristic for sampling w_{ij}:

$$U\left(-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\right). \tag{7.11}$$

7.5.3 He initialization

He initialization proposes different initialization for each activation function.

Sigmoid

We use a uniform distribution

$$U\left(-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right), \tag{7.12}$$

or a normal distribution with 0 mean and

$$\sigma = \sqrt{\frac{2}{n_{\text{in}}+n_{\text{out}}}}. \tag{7.13}$$

Hyperbolic tangent

We use a uniform distribution

$$U\left(-4\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, 4\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right), \tag{7.14}$$

or a normal distribution with 0 mean and

$$\sigma = 4\sqrt{\frac{2}{n_{\text{in}}+n_{\text{out}}}}. \tag{7.15}$$

ReLU and its variants

We use a uniform distribution

$$U\left(-\sqrt{2}\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{2}\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right), \tag{7.16}$$

or a normal distribution with 0 mean and

$$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{in}}+n_{\text{out}}}}. \tag{7.17}$$

7.6 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) were introduced by LeCun in 1998. The main difference compared to MLP is the local processing: neurons are only locally connected to neurons of the previous level. In this way, each neuron performs local processing. In addition, there is a significant reduction in the number of connections, as weights are shared in groups (*weight sharing*). In this way, different neurons of the same level perform the same type of processing on different portions of the input.

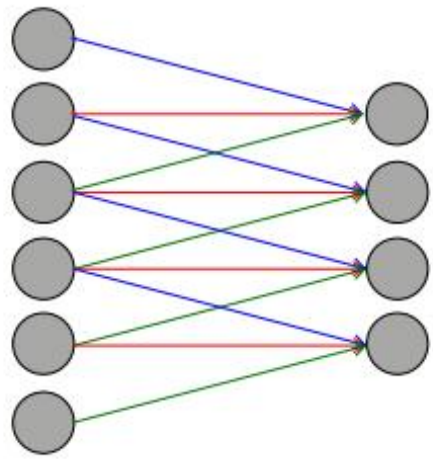


Figure 7.3: Example of weight sharing. Each of the 4 neurons on the right is connected only to 3 neurons of the previous level. Weights are shared (same color corresponds to the same weight). In total, there are 12 connections and 3 weights against the 24 connections and 24 weights of an equivalent portion of MLP. Source: [72].

7.6.1 Images as input

From the architectural point of view, a CNN is made up of a hierarchy of layers. The input layer is directly connected to the image pixels, the last layers are generally fully-connected and operate as an MLP classifier, while in the intermediate layers local connections and shared weights are used.

The receptive field of neurons increases as they move up the hierarchy. Local and shared connections mean that neurons process different portions of the image in the same way. This is a desired behavior, as different regions of the visual field contain the same type of information (edges, portions of objects, etc.).

7.6.2 Convolution

Convolution is one of the most important image processing operations through which digital *filters* (or *kernels*) are applied. A digital filter (a small 2D mask of weights) is scrolled over the different input positions. An output value is generated for each position by running the scalar product between the mask and the portion of the input covered (both treated as vectors). De facto, convolution implements a similar scenario of Figure 7.3, where each shared weight is the a weight of a filter. Figure 7.4 displays an example of convolution without considering the bias. We consider the pixels of an image as neurons and the input and output images as layers of a network. Given a 3×3 filter, if we connect a neuron to the 9 neurons it "covers" in the previous level, and use the filter weights $w_1, w_2, \ldots, w_9$, as weights of the connections, we notice that a classic neuron actually performs a convolution. In formulas:

$$\text{net} = \sum_{j=1}^{9} w_j \cdot in_j. \tag{7.18}$$

7.6.3 Convolution and cross-correlation

Convolution and cross-correlation actually refers to two different concepts [21]. Consider a two-dimensional image $\mathbf{I}$ and a two-dimensional kernel $\mathbf{K}$. Then convolution $\mathbf{S}$ is a two-dimensional image where:

$$S_{i,j} = \sum_m \sum_n I_{i-m,j-n} K_{m,n}. \tag{7.19}$$

On the other hand, the cross-correlation $\mathbf{S}$ is a two-dimensional image where:

$$S_{i,j} = \sum_m \sum_n I_{i+m,j+n} K_{m,n}. \tag{7.20}$$

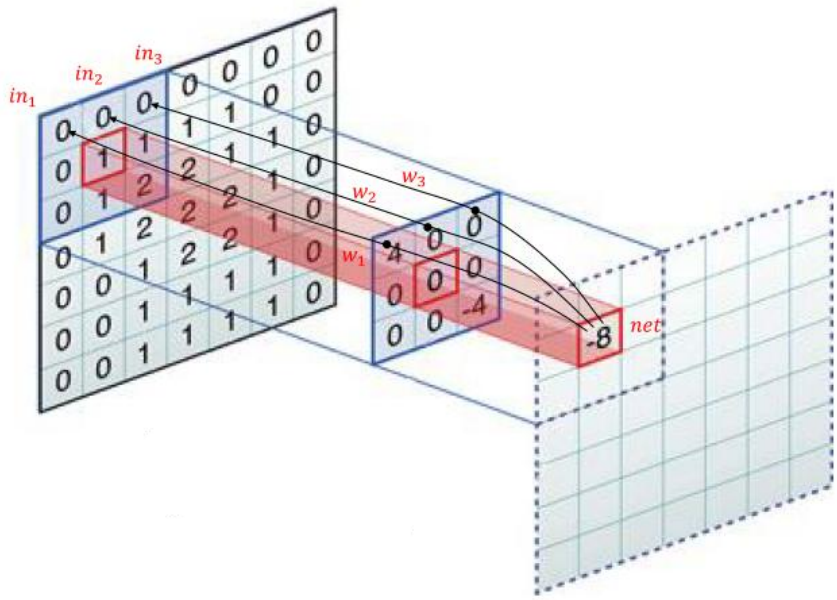


Figure 7.4: Example of convolution, without considering the bias. Only 3 of the 9 connections are diplayed for graphic clarity. Source: [72].

The only difference is that we don't have to "flip" the kernel relative to the input when applying cross-correlation. In general, in CNNs cross-correlation and convolution both refers to Equation (7.20).

7.6.4 Multiple input channels

When we add channels into the mix, our inputs and hidden representation both becomes three dimensional tensors. For example, each RGB input image has shape $3 \times h \times w$. We refer to this axis, in this example with a size of 3, as the *channel dimension*.

When the input data contain multiple channels, we need to construct a kernel with the same number of input channels as the input data, so that it can perform convolutions. Assuming that the number of channels for the input data is c_i, then the number of channels of the kernel also needs to be c_i. Note that we are considering *one* kernel. The fact that it is a three-dimensional kernel, it does not mean that we are considering different kernels. If the kernel's shape is $k_h \times k_w$ and $c_i = 1$, we can think of the kernel as a matrix (two-dimensional tensor) of shape

$$k_h \times k_w. \tag{7.21}$$

However, when $c_i > 0$, we need a kernel that contains a tensor of shape $k_h \times k_w$ for every input channel. Concatenating these c_i tensors together yields a kernel of shape

$$c_i \times k_h \times k_w. \tag{7.22}$$

Since the input and the kernel each have c_i channels, we can perform convolution operation on the two-dimensional tensor of the input and the two-dimensional tensor of the kernel for each channel, adding the c_i results together to yield a two dimensional tensor. This is the convolution of a two-dimensional convolution between a multi-channel input and a multi-input-channel kernel.

7.6.5 Multiple output channels

Many deep learning frameworks, such as PyTorch, let us define the number of *output channels* [19]. The number of output channels is equivalent to the *number of kernels* [51]. In fact, as we saw in Section 7.5, each kernel produces a two-dimensional tensor as output, as the convolution between the input and the corresponding two-dimensional tensor, for each channel, is summed up. This means that, if we have multiple kernels, we will obtain multiple output channels. This concept is visualized in Figure 7.6.

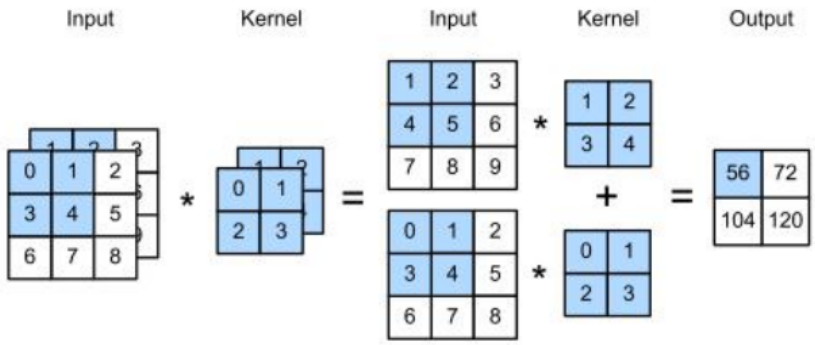


Figure 7.5: Convolution (cross-correlation) computation with 2 input channels. Note that we are considering *one* kernel. In particular, it is a $2 \times 2 \times 2$ kernel with $c_i = 2$. Source: [72].

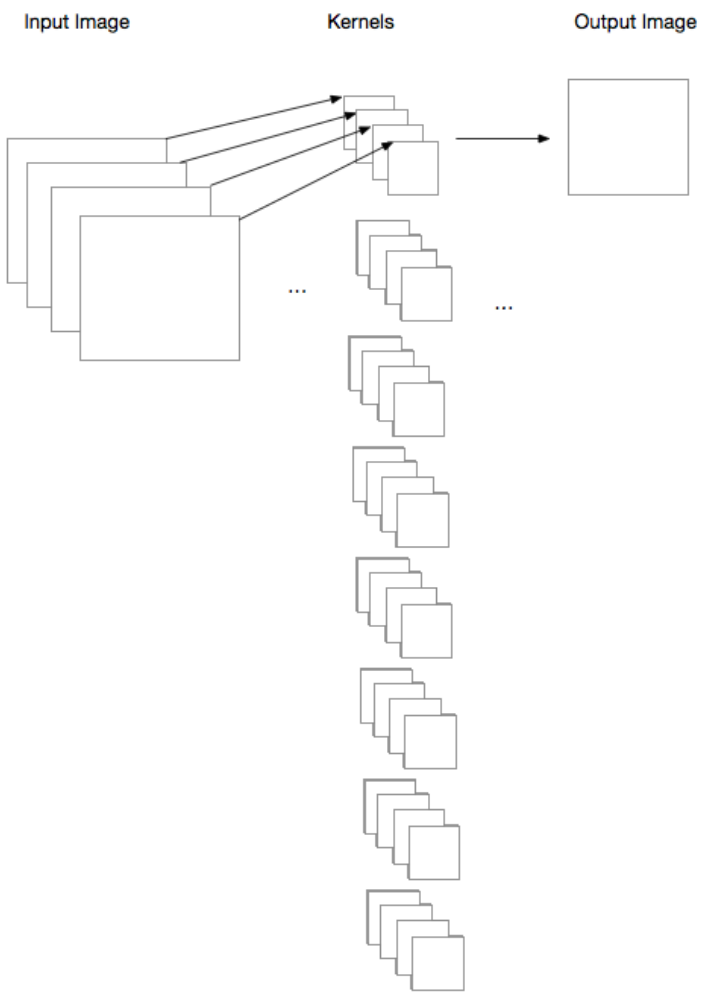


Figure 7.6: Convolutional layer with 8 kernel tensors with 4 channels each. Source [20].

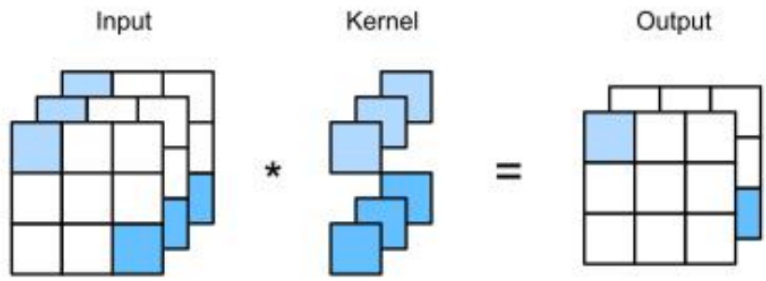


Figure 7.7: 1×1 convolution with 3 channels and 2 kernels (output channels). The input and the output have the same height and width. Source: [72].

7.6.6 1×1 convolutional layer

The role of convolution is to correlate (from cross-correlation) adjacent pixels. To this end, at first, a 1×1 convolution, with $k_h = k_w = 1$, does not seem to make much sense, since it does not correlate adjacent pixels. Nonetheless, they are popular operations that are sometimes included in the designs of complex DNNs.

Because the minimum window is used, 1×1 convolution loses the ability of larger convolutional layers to recognize patterns consisting of interactions among adjacent elements in the height and width dimensions. The only computation of the 1×1 convolution occurs on the channel dimension, which is the sum of the different channels' convolutions.

Figure 7.7 shows the convolution computation using the 1×1 convolution kernel with 3 input channels and 2 output channels. Note that the inputs and outputs have the same height and width, since the 1×1 convolution. Each element in the output is derived from a linear combination of elements *at the same position* in the input image. We could think of the 1×1 convolutional layer as constituting a fully-connected layer applied at every single pixel location to transform the c_i corresponding input values into c_o, output values. Because this is still a convolutional layer, the weights are tied across pixel location. Thus the 1×1 convolutional layer requires $c_o \times c_i$ weights plus the biases, one for each filter [49].

7.6.7 3D volumes of neurons

CNNs take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular neural network, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth (Figure 7.8). Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a neural network, which can refer to the total number of layers in a network. For example, a 32×32 RGB image has width and height 32, while the depth is 3, one for each channel. Moreover, if the CNN is used for classification with 10 labels, the final output layer for this specific image have dimensions $1 \times 1 \times 10$, because by the end of the CNN architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension [1]. Figure 7.9 shows an example of the application of 6 kernels using this graphical notation, and how it can be useful for understanding local connections.

7.6.8 Feature maps

Each "slice" of neurons in the same depth denotes a *feature map*. In the previous example (Figure 7.9) we find:

- 3 feature maps (size 32×32) in the input volume.
- 6 feature maps (size 28×28) in the output volume.

Weights are shared at the feature map level. Neurons in the same feature map process *different portions* of the input volume *in the same way*. Each feature map can be seen as the result of a specific fixed kernel. In the example (Figure 7.9) the number of connections between the two levels is

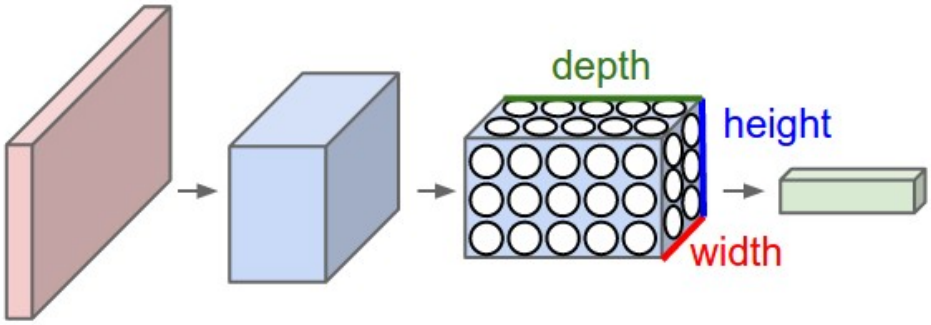


Figure 7.8: A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Each volume is represents an image or, equivalently, a set of neurons, obtained after a convolution operation. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (red, green, blue channels). In the third volume, depth is the number of *feature maps* (output channels, one for each kernel [122]) and the image (ignoring the maps) is 2×3, as the width is 2 and the height is 3. Source: [1].

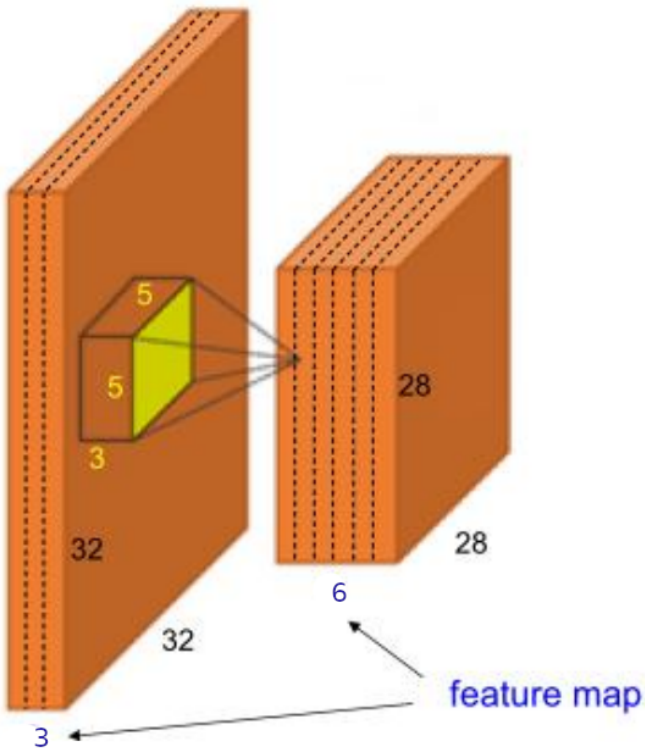


Figure 7.9: Example of the application of 6 kernels. Each feature map is associated to a kernel. The input image has shape $32 \times 32 \times 3$ ($c_i = 3$). Each one of the 6 kernels has shape $5 \times 5 \times 3$ (the depth was constrained to be 3 from the input size). The output image has shape $28 \times 28 \times 6$ ($c_o = 6$, number of kernels/feature maps, while the height and the width can be computed with a formula). Each neuron of the output volume is connected to $5 \times 5 \times 3 = 75$ neurons of the previous level. Source: [72].

$(28 \times 28 \times 6) \times (5 \times 5 \times 3) = 352800$, but the total number of weights is $6 \times (5 \times 5 \times 3 + 1) = 456$, thanks to weight sharing. The +1 in the previous formula is given by the bias.

7.6.9 Output size

When a filter is scrolled on the input volume, instead of moving in unit steps (of 1 neuron) a larger step, called *stride*, can be used. This operation reduces the size of the feature maps in the output volume and consequently the number of connections. The *Stride* parameter denotes the step size (in pixels). On the initial levels of the network applying small strides (e.g. 2, 4), it is possible to obtain a high gain in efficiency at the expense of a slight accuracy reduction. Another possibility (to adjust the size of the feature maps) is to add a *zero padding* (zero values) to the input volume. The *Padding* parameter denotes the thickness (in pixels) of the border. Let W_{out} be the (horizontal) dimension of the output feature map and W_{in} the corresponding dimension in the input. Furthermore, let F be the dimension of the filter. The following relationship applies:

$$W_{out} = \left\lfloor \frac{W_{in} - F + 2 \cdot Padding}{Stride} + 1 \right\rfloor . \tag{7.23}$$

7.6.10 Invariance and constraining the network

Imagine that you want to detect an object in an image. It seems reasonable that whatever method we use to recognize objects should not be overly concerned with the precise location of the object in the image. Ideally, our system should exploit this knowledge. Pigs usually do not fly and planes usually do not swim. Nonetheless, we should still recognize a pig were one to appear at the top of the image. We can draw some inspiration here from the children's game "Where's Waldo". The game consists of a number of chaotic scenes bursting with activities. Waldo shows up somewhere in each game, typically lurking in some unlikely location. The readers goal is to locate him. Despite his characteristic outfit, this can be surprisingly difficult, due to the large number of distractions. However, what Waldo looks like does not depend upon where Waldo is located. We could sweep the image with a Waldo detector that could assign a score to each patch, indicating the likelihood that the patch contains Waldo. CNNs systematize this idea of *spatial invariance*, exploiting it to learn useful representations with fewer parameters.

We can now make these intuitions more concrete by enumerating a few desiderata to guide our design of a neural network architecture suitable for computer vision:

1. The earliest layers of the network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called *translation invariance*.

2. The earliest layers of the network should focus on local regions, without regard for the contents of the image in distant regions. This is the *locality principle*. Eventually, these local representations can be aggregated to make predictions at the whole image level.

Let us see how this translates into mathematics.

Constraining the MLP

To start of, we can consider an MLP with two-dimensional images $\mathbf{X}$ as inputs and their immediate hidden representations $\mathbf{H}$ similarly represented as matrices in mathematics and as two-dimensional tensors in code, where both $\mathbf{X}$ and $\mathbf{H}$ have the same shape. We now conceive of not only the inputs but also the hidden representations as possessing spatial structure.

Let $X_{i,j}$ and $H_{i,j}$ denote the pixel at location (i, j) in the input image and hidden representation, respectively. Consequently, to have each of the hidden units receive input from each of the input pixels, we would switch from using weight matrices to representing our parameters as a four-dimensional weight tensors $\mathbf{W}$. Suppose that $\mathbf{U}$ contains biases, we could formally express the fully-connected layer as

$$H_{i,j} = U_{i,j} + \sum_k \sum_l W_{i,j,k,l} X_{k,l} = U_{i,j} + \sum_a \sum_b V_{i,j,a,b} X_{i+a,j+b}, \tag{7.24}$$

where the switch from $\mathbf{W}$ to $\mathbf{V}$ is entirely cosmetic for now, since there is a one-to-one correspondence between coefficients in both four-dimensional tensors. We simply re-index the subscripts (k, l) such

that $k = i + a$ and $l = j + b$. In other words, $V_{i,j,a,b} = W_{i,j,i+a,j+b}$. The indices a and b run over both positive and negative offsets, covering the entire image. For any given location (i, j) in the hidden representation $H_{i,j}$, we compute its value by summing over pixels in $\mathbf{X}$, centered around (i, j) and weighted by $V_{i,j,a,b}$.

Translation invariance

Translation invariance implies that a shift in the input $\mathbf{X}$ should simply lead to a shift in the hidden representation $\mathbf{H}$. This is only possible if $\mathbf{V}$ and $\mathbf{U}$ do not actually depend on (i, j), i.e. we have

$$V_{i,j,a,b} = V_{a,b}, \tag{7.25}$$

$$U_{i,j} = U. \tag{7.26}$$

As a result, we can simplify the definition of $\mathbf{H}$:

$$H_{i,j} = U + \sum_a \sum_b V_{a,b} X_{i+a,j+b}. \tag{7.27}$$

This is equivalent to cross-correlation and convolution in Equation (7.20) with the addition of a bias. We are effectively weighting pixels at $(i + a, j + b)$ in the vicinity of location (i, j) with coefficients $V_{a,b}$ to obtain $H_{i,j}$. Note that $V_{a,b}$ needs many fewer coefficients than $V_{a,b,i,j}$, since it no longer depends on the location within the image. In this way, $\mathbf{V}$ and U represents the kernel's parameters.

Locality

Locality implies that we should not have to look very far away from location (i, j) in order to obtain relevant information about $H_{i,j}$. In fact, in Equation (7.27), we are still iterating all over the full image. Therefore, outside some range $|a| > \Delta$ or $|b| > \Delta$ we should set $V_{a,b} = 0$. Equivalently:

$$H_{i,j} = U + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} V_{a,b} X_{i+a,j+b}. \tag{7.28}$$

This is the operation done in a *convolutional layer* (most trivial case, in which $c_{in} = 1$ and $c_{out} = 1$).

Returning to our Waldo detector, the convolutional layer picks windows of a given size and weights intensities according to the filter $\mathbf{V}$. We aim to learn a model so that wherever the "waldoness" is highest, we should find a peak in the hidden layer representations.

7.6.11 Inductive bias

All learning depends on the imposing *inductive bias*. Every machine learning model requires some type of architecture design and possibly some initial assumptions about the data we want to analyze. Generally, every building block and every belief that we make about the data is a form of inductive bias [119]. In statistical learning theory, it is analogous to the choice of the hypothesis class $\mathcal{H}$.

When the bias agrees with reality, we get models that generalize well to unseen data. But of course, if those biases do not agree with reality, e.g. images turned out not to be translation invariant, our models might struggle even to fit our training data.

7.6.12 Activation function

Historically, in MLP networks, most used activation function is the sigmoid. However, in DNNs, the use of the sigmoid is problematic for the backpropagation of the gradient because of the vanishing gradient problem. As already discussed in Figure 7.2, the derivative of the sigmoid is less than 1 and the application of the chain derivation rule leads to multiplying many terms less than 1 with the consequence of greatly reducing the gradient values in the levels far from the output. In addition, the sigmoid has a saturating behavior with large arguments in absolute value. In the saturation regions the derivative is 0 and therefore the gradient vanishes.

Now we present different activation functions used in practice. Nonetheless, the choice of the activation function is still a research area and many variants have been proposed and others are under study. In general, the goal of any activation function is to avoid:

- Vanishing gradient.
- Exploding gradient.
- Overfitting.
- Underfitting.

In the literature many properties of the activation functions are considered useful for avoiding the above cited problems.

ReLU

Rectified Linear Unit (ReLU) (Figure 7.10) is defined as:

$$f(x) = \max(0, x) = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{if } x \geq 0. \end{cases} \tag{7.29}$$

Trivially:

$$f'(x) = \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{if } x \geq 0. \end{cases} \tag{7.30}$$

It leads to *sparse activations*, meaning that part of the neurons are switched off, which can confer greater robustness.

Leaky ReLU

Leaky ReLU, sometimes abbreviated as LReLU, (Figure 7.10) is defined as:

$$f(x) = \begin{cases} ax & \text{if } x < 0, \\ x & \text{if } x \geq 0, \end{cases} \tag{7.31}$$

where $a \geq 0$ is a small number. Leaky ReLU solves the *dying ReLU problem* i.e if the input is $x \leq 0$ the gradient of ReLU becomes 0 and thus the network cannot make backpropagation for that neuron. The gradient is

$$f'(x) = \begin{cases} a & \text{if } x < 0, \\ 1 & \text{if } x \geq 0. \end{cases} \tag{7.32}$$

ELU

Exponential Linear Unit (ELU) (Figure 7.10) is defined as:

$$f(x) = \begin{cases} a(e^x - 1) & \text{if } x < 0, \\ x & \text{if } x \geq 0, \end{cases} \tag{7.33}$$

where $a \geq 0$. ELU is bounded from below by $-a$. If $a = 1$, ELU is differentiable in 0. Moreover, its gradient is always positive. The gradient is

$$f'(x) = \begin{cases} ae^x & \text{if } x < 0, \\ 1 & \text{if } x \geq 0. \end{cases} \tag{7.34}$$

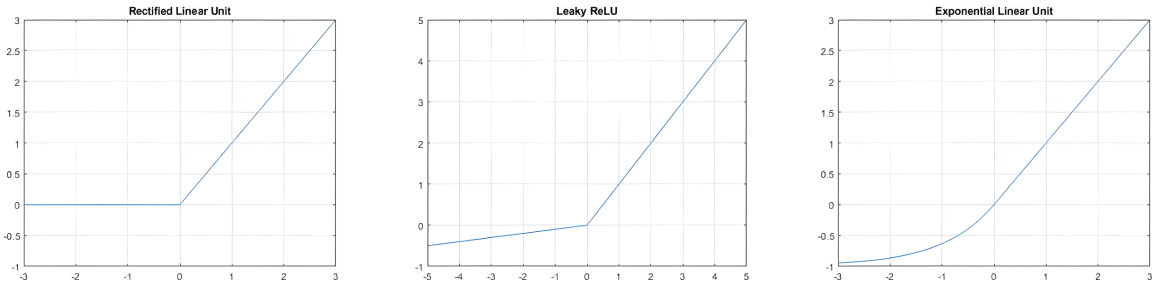


Figure 7.10: ReLU (left), Leaky ReLU (center) and ELU(right). Source: [72].

PReLU

Parametric ReLU (PReLU) is defined as:

$$f(x) = \begin{cases} a_c x & \text{if } x < 0, \\ x & \text{if } x \geq 0, \end{cases} \tag{7.35}$$

where $a_c \geq 0$ is different for every channel c of the input. PReLU is the same as Leaky ReLU, except that each parameter a_c is learnable. Therefore, we have to define the two partial derivatives:

$$\frac{\partial f(x)}{\partial x} = \begin{cases} a_c & \text{if } x < 0, \\ 1 & \text{if } x \geq 0, \end{cases} \tag{7.36}$$

$$\frac{\partial f(x)}{\partial a_c} = \begin{cases} x & \text{if } x < 0, \\ 0 & \text{if } x \geq 0. \end{cases} \tag{7.37}$$

In general, each a_c is initialized to 0.

SReLU

S-Shaped ReLU (SReLU) is composed of three piecewise linear functions that are expressed by four learnable parameters:

$$f(x) = \begin{cases} t_c^l + a_c^l(x - t_c^l) & \text{if } x \leq t_c^l, \\ x & \text{if } t_c^l < x < t_c^r, \\ t_c^r + a_c^r(x - t_c^r) & \text{if } x \geq t_c^r, \end{cases} \tag{7.38}$$

where t_c^l, a_c^l (left), t_c^r, a_c^r (right) are four real number learnable parameters, initialized as [1]:

- $a^l = t^l = 0$.
- $a^r = 1$ [77].
- $t^r = \text{max_input}$, an hyperparameter, .

The subscript c indicates that we allow SReLU to vary in different channels [57]. SReLU's large number of parameters results in high representation power. SReLU learns both convex and non-convex functions.

The gradients of SReLU with respect to its four learning parameters are:

$$\frac{\partial f(x)}{\partial t_c^r} = \mathbb{1}[x \geq t_c^r] \cdot (1 - a_c^r), \tag{7.39}$$

$$\frac{\partial f(x)}{\partial a_c^r} = \mathbb{1}[x \geq t_c^r] \cdot (x - t_c^r), \tag{7.40}$$

[1] As far as I know, the original paper [57] never states this.

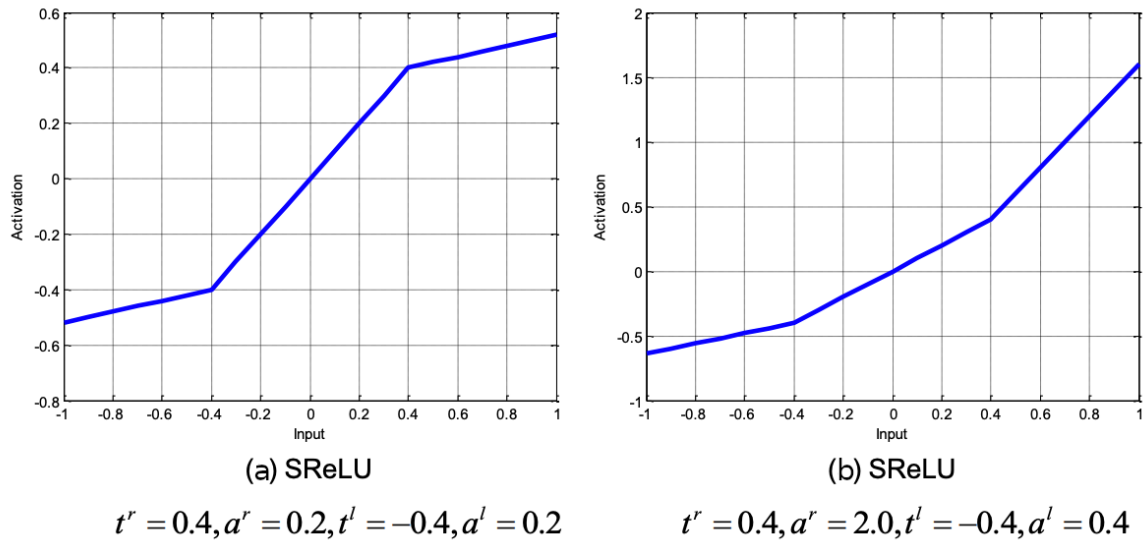


Figure 7.11: Examples of SReLU. Source [72].

$$\frac{\partial f(x)}{\partial t_c^l} = \mathbb{1}[x \leq t_c^l] \cdot (1 - a_c^l), \tag{7.41}$$

$$\frac{\partial f(x)}{\partial a_c^l} = \mathbb{1}[x \leq t_c^l] \cdot (x - t_c^l). \tag{7.42}$$

On the other hand, the gradient of SReLU with respect to the input x is:

$$\frac{\partial f(x)}{\partial x} = \begin{cases} a^l & \text{if } x \leq t_c^l, \\ 1 & \text{if } t_c^l < x < t_c^r 0, \\ a^r & \text{if } x \geq t_c^r, \end{cases} \tag{7.43}$$

Two examples of SReLU with different parameters are shown in Figure 7.11.

APLU

Adaptive Piecewise Linear Unit (APLU) is defined as:

$$f(x) = \text{ReLU}(x) + \sum_{c=1}^{n} a_c \max(0, -x + b_c), \tag{7.44}$$

where a_c and b_c are real number learnable parameters [4], different on each channel, n is the number of hinges. APLU is a linear piecewise function able to approximate any continuous function on a compact set. We have:

$$\nabla f(x) = \nabla \text{ReLU}(x) + \sum_{c=1}^{n} \nabla(a_c \max(0, -x + b_c)), \tag{7.45}$$

and:

$$\frac{\partial f(x)}{\partial a_c} = \begin{cases} -x + b_c & \text{if } x < b_c, \\ 0 & \text{if } x \geq b_c, \end{cases} \tag{7.46}$$

$$\frac{\partial f(x)}{\partial b_c} = \begin{cases} a_c & \text{if } x < b_c, \\ 0 & \text{if } x \geq b_c. \end{cases} \tag{7.47}$$

The parameters a_c were initialized to 0, while b_c where randomly initialized. It is possible to apply l_2 regularization to $a_1, \ldots, a_n$ by adding to the loss function the term

$$J_{reg} = \sum_{c=1}^{n} |a_c|^2. \tag{7.48}$$

j	1	2	3	4	5	6	7
a_j	512	256	768	128	384	640	896
λ_j	512	256	256	128	128	128	128

Figure 7.12: Value of the first seven mexican hat functions with $maxInput = 256$. Source: [72].

Without l_2 regularization, the optimizer is free to choose very large values of a_c and b_c balanced by very small weights, which would lead to numerical instability and exploding gradient. In addition, high values means a large update of the parameters, and possible jumps over minimums.

Mexican ReLU

Let $\phi^{a,\lambda}(x)$ be a "mexican hat" like function

$$\phi^{a,\lambda}(x) = \max(0, \lambda - |x - a|), \tag{7.49}$$

where $a, \lambda \in \mathbb{R}$. When $|x - a| > \lambda$, the function is null. However, when $|x - a| \leq \lambda$, the function increases with a derivative of 1 in $[1, a - \lambda]$ and decreases with a derivative of -1 in $[a, a + \lambda]$.

The *Mexican ReLU* (MeLU) is defined as:

$$f(x) = \text{PReLU}^{c_0}(x) + \sum_{j=1}^{k-1} c_j \phi^{a_j,\lambda_j}(x), \tag{7.50}$$

where $c_0, c_1, c_2, \ldots, c_{k-1}$ are the k learnable parameters for each channel. In particular, c_0 is the learnable parameter of PReLU, while $c_1, c_2, \ldots, c_{k-1}$ are specific to MeLU. Since the gradient can be zero, we sum Mexican Hat with PReLU for avoiding a vanishing gradient.

a_j and λ_j are fixed recursively as we are going to the describe now. Initially, the parameter $maxInput$ is set and we proceed as follows.

- The first mexican hat (corresponding to $j = 1$) has its maximum in $a_j = 2 \cdot maxInput$, and it is zero outside $[0, 4 \cdot maxInput]$, then $\lambda_j = 2 \cdot maxInput$.
- The second mexican hat (corresponding to $j = 2$) has its maximum in $a_j = maxInput$, and it is zero outside $[0, 2 \cdot maxInput]$, then $\lambda_j = maxInput$.
- The second mexican hat (corresponding to $j = 3$) has its maximum in $a_j = 3 \cdot maxInput$, and it is zero outside $[2, 4 \cdot maxInput]$, then $\lambda_j = maxInput$.

The next four mexican hat functions are defined by iteratively dividing each interval into two parts. In particular:

$$\lambda_j = \frac{1}{2} maxInput, \tag{7.51}$$

$$a_j = \frac{1}{2} maxInput \cdot (1 + 2(j - 4)). \tag{7.52}$$

Figure 7.12 shows an example of the parameters' values.

The Mexican hat functions are continuous and piecewise differentiable, hence MeLU inherits these properties. If all the c_j are initialized at zero, MeLU coincides with ReLU. This helps transfer learning when we substitute MeLU in a network pretrained with ReLU. The same holds for networks trained with Leaky ReLU or PReLU.

MeLU is similar to SReLU and APLU in having multiple learnable parameters, but this number is larger in APLU. MeLU is also similar to APLU since it can approximate the same set of piecewise linear functions equal to identity when x is large enough, but how this is done is different for each function. Given the right choice of the parameters, APLU can be equal to any piecewise linear function because the points of non-differentiability are learnable, but MeLU is more efficient because it can represent

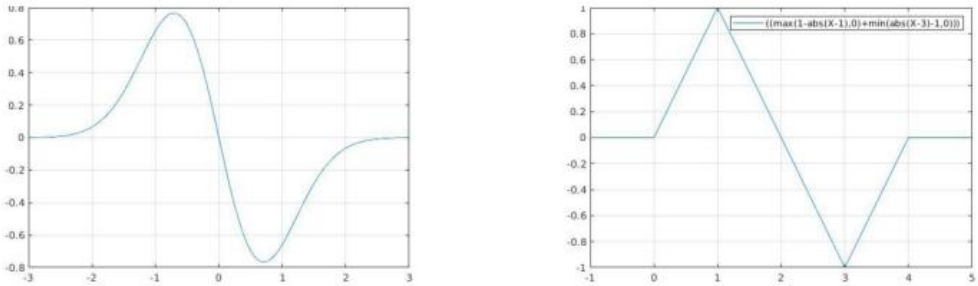


Figure 7.13: Gaussian wavelet (left) and $\phi^{1,1}(x)$ (right). Source: [72].

	j	1	2	3	4	5	6	7
MeLU	a_j	2	1	3	0.5	1.5	2.5	3.5
	λ_j	2	1	1	0.5	0.5	0.5	0.5
GaLU	a_j	1	0.5	2.5	0.25	1.25	2.25	3.25
	λ_j	1	0.5	0.5	0.25	0.25	0.25	0.25

Figure 7.14: Value of the first seven parameters with $maxInput = 1$. Source: [72].

every piecewise linear function by exploiting only the joint optimization of the weight matrix and the biases. Moreover, the gradients of the two activations with respect to the learnable parameters are different. The optimization of a neural network depends in large part on two factors: the output of every hidden layer and the gradient of that output with respect to the parameters. The learnable parameters in MeLU are initialized to zero, so the activation starts off as ReLU, thereby exploiting all the nice properties of ReLU at the beginning.

GaLU

Gaussian ReLU is based on MeLU and possesses the same properties. To define GaLU, let

$$\phi^{a,\lambda}(x) = \max(0, \lambda - |x - a|) + \min(0, |x - a - 2\lambda| - \lambda), \tag{7.53}$$

a Gaussian type function, where $a, \lambda \in \mathbb{R}$. In Figure 7.13 a standard gaussian wavelet function is compared with a Gaussian type function.

GaLU is defined, similarly to MeLU, as:

$$f(x) = \mathrm{PReLU}^{c_0}(x) + \sum_{j=1}^{k-1} c_j \phi^{a_j,\lambda_j}(x). \tag{7.54}$$

In fact, piecewise linear functions constructed of many linear pieces approximate functions better than ReLU does. The rationale behind GaLU is to add more linear pieces than MeLU. Figure 7.14 shows an example of the parameters' value.

7.6.13 Pooling

A *pooling* layer aggregates information in the input volume, resulting in smaller feature maps. The goal is to confer invariance with respect to simple transformations of the input while maintaining significant information for pattern discrimination. In general, aggregation operates within each feature map, so that the number of feature maps in the input and output volume is the same. The most used aggregation operators are the *average* (for *average pooling*) and the *maximum* (for *max pooling*): both "rather" invariant under small translations. This type of aggregation has no weights to learn. As convolution, pooling has a *stride* and a *filter size* (or *pool size*) (note that these are totally different from the learnable kernels). Two examples are reported in Figure 7.15. Equation (7.23) for computing W_0 is still valid for pooling.

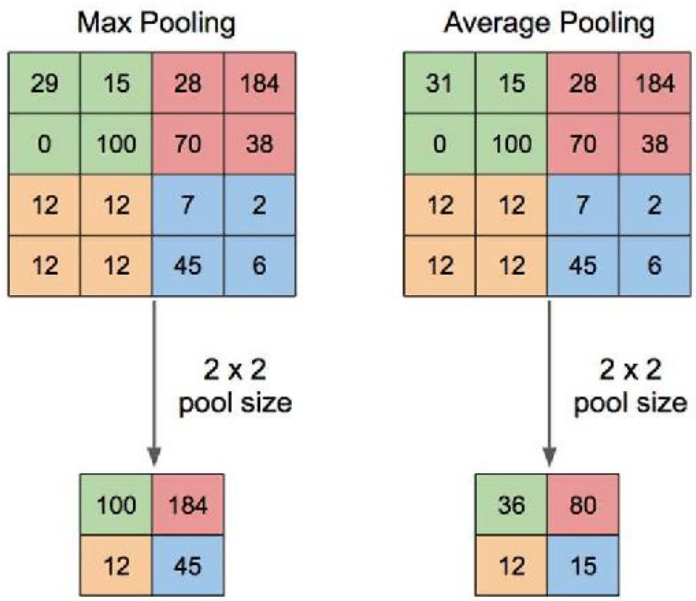


Figure 7.15: Example of max pooling (left) and average pooling (right) with a stride of 2 and a 2×2 pool size. Source: [72].

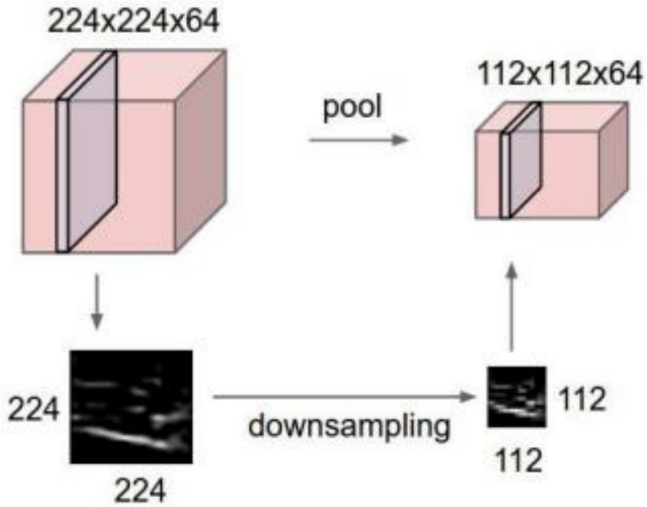


Figure 7.16: Pooling operation with volumes. The figure represents pooling operation on one channel (image), but the operation is applied to all 64 of them. In this example, pooling downsamples a channel from size 224×224 to 112×112, since $Stride = 2$, $F = 2$ and $Padding = 0$, from Equation (7.23). Source: [72].

7.6.14 Final activation function and loss function for CNNs

In classification tasks, common choices in modern CNNs are:

- The use of a final *softmax activation function*, already discussed in Section 4.11.
- The use of the *Cross-Entropy loss* as loss function, already discussed in 4.12.

7.6.15 Pre-training

Pre-training refers to the process of initializing a model with pre-existing knowledge before training it on specific tasks or datasets [85]. Two examples of pre-training are:

- Unsupervised pre-training: train an unsupervised model on the training data and use the resulting model as initialization (in some way).
- Supervised pre-training, such as *transfer-learning*, in which we initialize the weights with a model trained on a related task. It is especially effective with CNNs.

7.7 Computing the gradient of a loss function

We now show how to compute the gradient of

$$C(\mathbf{y}, \mathbf{w}, \mathbf{X}, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \max(0, \langle \mathbf{w}, \mathbf{x}_i\rangle + b))^2, \tag{7.55}$$

where $(\mathbf{x}_i, y_i)$ is the i-th sample of the training set. We are using the MSE loss function with ReLU activation function.

7.7.1 Total derivative and multivariable chain rule

Suppose we want to compute the derivative of

$$y = x + x^2$$

using the chain rule. We introduce the following intermediate variables:

$$u_1(x) = x^2,$$

$$u_2(x, u_1) = x + u_1(x).$$

We cannot apply the single-variable chain rule, as:

$$\frac{dy}{dx} = \frac{du_2}{dx} = \frac{du_2}{du_1}\frac{du_1}{dx} = 1 \cdot 2x = 2x \neq \frac{dy}{dx}.$$

To compute $\frac{dy}{dx}$ we need to sum up all possible contributions from changes in x to change in y. The *total derivative* of y is given by:

$$\frac{dy}{dx} = \frac{du_2}{dx} = \frac{du_2}{dx}\frac{dx}{dx} + \frac{du_2}{du_1}\frac{du_1}{dx} = \frac{du_2}{dx} + \frac{du_2}{du_1}\frac{du_1}{dx}. \tag{7.56}$$

In simpler terms, we add up the effect of a change in x directly to u_2 and the effect of a change in x through u_1 to u_2.

In general, it holds the *multivariable chain rule*:

$$\frac{\partial f(x, u_1, \ldots, u_n)}{\partial x} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial u_1}\frac{\partial u_1}{\partial x} + \frac{\partial f}{\partial u_2}\frac{\partial u_2}{\partial x} + \cdots + \frac{\partial f}{\partial u_n}\frac{\partial u_n}{\partial x} = \frac{\partial f}{\partial x} + \sum_{i=1}^{n}\frac{\partial f}{\partial u_i}\frac{\partial u_i}{\partial x} \tag{7.57}$$

If we introduce an alias $x = u_{n+1}$, then:

$$\frac{\partial f(u_1, \ldots, u_n, u_{n+1})}{\partial x} = \sum_{i=1}^{n+1}\frac{\partial f}{\partial u_i}\frac{\partial u_i}{\partial x} \tag{7.58}$$

As an example, consider $f(x) = \sin(x + x^2)$. Let (without the arguments for readability reasons)

$$u_1 = x^2,$$

$$u_2 = x + u_1,$$

$$u_3 = \sin(u_2).$$

Equation (7.57) is equivalent to build a *computational graph* and, for each path from x to $f(x)$, we sum up the product of partial derivatives, as shown in Figure 7.17. In the figure, each yellow block represents an operation. For example, before the *square* operation, we are computing

$$\frac{\partial u_1}{\partial x} = \frac{\partial x^2}{\partial x}.$$

While, before the + operation, we are computing, on the right path:

$$\frac{\partial u_2}{\partial u_1} = \frac{\partial (x + u_1)}{\partial u_1} = 1,$$

while on the left path:

$$\frac{\partial u_2}{\partial x} = \frac{\partial (x + u_1)}{\partial x} = 1.$$

In few words, in each arc, we are computing the partial derivative of the "entering block" with respect to the "leaving block". Finally, on the last arc, we are computing:

$$\frac{\partial u_3}{\partial u_2} = \frac{\partial \sin(u_2)}{\partial u_2} = \cos(u_2).$$

In the end, we sum up all the possible path's partial derivative concatenation. Right-side path:

$$\frac{\partial u_3}{\partial u_2}\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial x} = \cos(u_2) \cdot 1 \cdot 2x = 2x\cos(x + x^2).$$

Left-side path:

$$\frac{\partial u_3}{\partial u_2}\frac{\partial u_2}{\partial x} = \cos(u_2) \cdot 1 = \cos(x + x^2).$$

Therefore:

$$\frac{\partial f}{\partial x} = 2x\cos(x + x^2) + \cos(x + x^2).$$

7.7.2 Gradient/Jacobian of a vector function

Let $[f(x, y), g(x, y)]^T$ be a vector function. Its *Jacobian* is defined as:

$$J = \begin{bmatrix} \nabla f(x, y)^T \\ \nabla g(x, y)^T \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix} \tag{7.59}$$

7.7.3 Gradient/Jacobian of the identity function

Let $\mathbf{f}(\mathbf{x}) = \mathbf{x}$ be a vector function, i.e.:

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}), & \ldots, & f_n(\mathbf{x}) \end{bmatrix}^T = \begin{bmatrix} x_1, & \ldots, & x_n \end{bmatrix}^T, \tag{7.60}$$

i.e.

$$f_i(\mathbf{x}) = x_i. \tag{7.61}$$

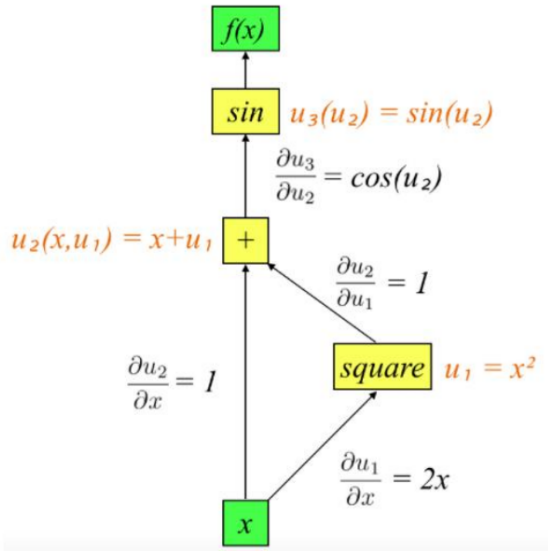


Figure 7.17: Example of computational graph and the partial derivatives. Source: [72].

Then:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = J = \begin{bmatrix} \nabla f_1(\mathbf{x})^T \\ \vdots \\ \nabla f_n(\mathbf{x})^T \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_2}{\partial x_n} \\ & \vdots & \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}. \tag{7.62}$$

Trivially:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = J = \begin{bmatrix} \frac{\partial x_1}{\partial x_1} & \cdots & \frac{\partial x_1}{\partial x_n} \\ \frac{\partial x_2}{\partial x_1} & \cdots & \frac{\partial x_2}{\partial x_n} \\ & \vdots & \\ \frac{\partial x_n}{\partial x_1} & \cdots & \frac{\partial x_n}{\partial x_n} \end{bmatrix} = I, \tag{7.63}$$

where I is the identity matrix of size $n \times n$.

7.7.4 Gradient/Jacobian of element-wise vector function combinations

Element-wise binary operators are operations that applies an operator consecutively, from the first of both vectors to get the first item of output, then the second item of both vectors to get the second item of output, and so forth. Example of such operations are the addition $\mathbf{x} + \mathbf{y}$ or the boolean operator $\mathbf{x} > \mathbf{y}$, which returns a vector of ones and zeroes.

We denote as $\circ$ any element-wise operator, and not composition of functions. Consider the two vector functions $\mathbf{f}(\mathbf{x})$ and $\mathbf{g}(\mathbf{w})$ and let

$$\mathbf{y} = \mathbf{f}(\mathbf{w}) \circ \mathbf{g}(\mathbf{x}). \tag{7.64}$$

Since we have two sets of functions, we need two Jacobians, one representing the gradient with respect to $\mathbf{x}$ and one with respect to $\mathbf{w}$:

$$J_{\mathbf{w}} = \begin{bmatrix} \frac{\partial(f_1 \circ g_1)}{\partial w_1} & \cdots & \frac{\partial(f_1 \circ g_1)}{\partial w_n} \\ \frac{\partial(f_2 \circ g_2)}{\partial w_1} & \cdots & \frac{\partial(f_2 \circ g_2)}{\partial w_n} \\ & \vdots & \\ \frac{\partial(f_n \circ g_n)}{\partial w_1} & \cdots & \frac{\partial(f_n \circ g_n)}{\partial w_n} \end{bmatrix}. \tag{7.65}$$

$$J_{\mathbf{x}} = \begin{bmatrix} \frac{\partial(f_1 \circ g_1)}{\partial x_1} & \dots & \frac{\partial(f_1 \circ g_1)}{\partial x_n} \\ \frac{\partial(f_2 \circ g_2)}{\partial x_1} & \dots & \frac{\partial(f_2 \circ g_2)}{\partial x_n} \\ & \vdots & \\ \frac{\partial(f_n \circ g_n)}{\partial x_1} & \dots & \frac{\partial(f_n \circ g_n)}{\partial x_n} \end{bmatrix}. \tag{7.66}$$

Most arithmetic operations we will need are simple ones, so $f(\mathbf{w}) = \mathbf{w}$. Suppose also $g(\mathbf{x}) = \mathbf{x}$. Then:

$$\frac{\partial}{\partial w_j}(f_i(\mathbf{w}) \circ g_i(\mathbf{x})) = \frac{\partial}{\partial w_j}(w_i \circ x_i), \tag{7.67}$$

$$\frac{\partial}{\partial x_j}(f_i(\mathbf{w}) \circ g_i(\mathbf{x})) = \frac{\partial}{\partial x_j}(w_i \circ x_i). \tag{7.68}$$

Off the diagonal $i \neq j$, so the partial derivative become zero:

$$J_{\mathbf{w}} = \begin{bmatrix} \frac{\partial(f_1 \circ g_1)}{\partial w_1} & \dots & 0 \\ 0 & \dots & 0 \\ & \vdots & \\ 0 & \dots & \frac{\partial(f_n \circ g_n)}{\partial w_n} \end{bmatrix} = \operatorname{diag}\left(\frac{\partial(f_1 \circ g_1)}{\partial w_1}, \dots, \frac{\partial(f_n \circ g_n)}{\partial w_n}\right). \tag{7.69}$$

$$J_{\mathbf{x}} = \begin{bmatrix} \frac{\partial(f_1 \circ g_1)}{\partial x_1} & \dots & 0 \\ 0 & \dots & 0 \\ & \vdots & \\ 0 & \dots & \frac{\partial(f_n \circ g_n)}{\partial x_n} \end{bmatrix} = \operatorname{diag}\left(\frac{\partial(f_1 \circ g_1)}{\partial x_1}, \dots, \frac{\partial(f_n \circ g_n)}{\partial x_n}\right). \tag{7.70}$$

Sum

If $\circ = +$, then

$$\frac{\partial(\mathbf{w} + \mathbf{x})}{\partial \mathbf{w}} = J_{\mathbf{w}} = J_{\mathbf{x}} = \frac{\partial(\mathbf{w} + \mathbf{x})}{\partial \mathbf{x}} = I \tag{7.71}$$

In fact:

$$\frac{\partial}{\partial w_i}(f_i(\mathbf{w}) + g_i(\mathbf{x})) = \frac{\partial}{\partial w_i}(w_i + x_i) = 1 + 0 = 1, \tag{7.72}$$

$$\frac{\partial}{\partial x_i}(f_i(\mathbf{w}) + g_i(\mathbf{x})) = \frac{\partial}{\partial x_i}(w_i + x_i) = 0 + 1 = 1. \tag{7.73}$$

Multiplication

If $\circ = \cdot$, then

$$\frac{\partial(\mathbf{w} \cdot \mathbf{x})}{\partial \mathbf{w}} = J_{\mathbf{w}} = \operatorname{diag}(\mathbf{x}), \tag{7.74}$$

$$\frac{\partial(\mathbf{w} \cdot \mathbf{x})}{\partial \mathbf{x}} = J_{\mathbf{x}} = \operatorname{diag}(\mathbf{w}), \tag{7.75}$$

In fact:

$$\frac{\partial}{\partial w_i}(f_i(\mathbf{w}) \cdot g_i(\mathbf{x})) = \frac{\partial}{\partial w_i}(w_i \cdot x_i) = x_i, \tag{7.76}$$

$$\frac{\partial}{\partial x_i}(f_i(\mathbf{w}) \cdot g_i(\mathbf{x})) = \frac{\partial}{\partial x_i}(w_i \cdot x_i) = w_i. \tag{7.77}$$

Subtraction

If $\circ = -$, then

$$\frac{\partial(\mathbf{w}-\mathbf{x})}{\partial\mathbf{w}} = J_{\mathbf{w}} = I, \tag{7.78}$$

$$\frac{\partial(\mathbf{w}-\mathbf{x})}{\partial\mathbf{x}} = J_{\mathbf{x}} = -I, \tag{7.79}$$

Division

If $\circ = \div$, then

$$\frac{\partial(\mathbf{w}\div\mathbf{x})}{\partial\mathbf{w}} = J_{\mathbf{w}} = \mathrm{diag}(\mathbf{1}\div\mathbf{x}), \tag{7.80}$$

where $\mathbf{1}$ is a vector of ones, and

$$\frac{\partial(\mathbf{w}\div\mathbf{x})}{\partial\mathbf{x}} = J_{\mathbf{x}} = \mathrm{diag}(-\mathbf{w}\div\mathbf{x}^2), \tag{7.81}$$

where both $\div$ and $\cdot^2$ are component-wise operators.

7.7.5 Gradient of vector sums

Let $\mathbf{x} = [x_1, \ldots, x_n]^T$ be a vector and

$$y = \mathrm{sum}(\mathbf{x}) = \sum_{i=1}^{n} x_i. \tag{7.82}$$

Then

$$\frac{\partial y}{\partial\mathbf{x}} = \left[\sum_{i=1}^{n}\frac{\partial x_i}{\partial x_1}, \quad \sum_{i=1}^{n}\frac{\partial x_i}{\partial x_2}, \quad \ldots, \quad \sum_{i=1}^{n}\frac{\partial x_i}{\partial x_n}\right]^T \tag{7.83}$$

And since the partial derivative of a function with respect to a variable that is not in the function is zero, the equation can be further simplified as:

$$\frac{\partial y}{\partial\mathbf{x}} = \left[\frac{\partial x_1}{\partial x_1}, \quad \frac{\partial x_2}{\partial x_2}, \quad \ldots, \quad \frac{\partial x_n}{\partial x_n}\right]^T = \begin{bmatrix}1, & 1, & \ldots, & 1\end{bmatrix}^T = \mathbf{1}^T. \tag{7.84}$$

Consider now the gradient of

$$y = \mathrm{sum}(z\mathbf{x}). \tag{7.85}$$

Then

$$\frac{\partial y}{\partial\mathbf{x}} = \left[\sum_{i=1}^{n}\frac{\partial x_i z}{\partial x_1}, \quad \sum_{i=1}^{n}\frac{\partial x_i z}{\partial x_2}, \quad \ldots, \quad \sum_{i=1}^{n}\frac{\partial x_i z}{\partial x_n}\right] = \left[\frac{\partial x_1 z}{\partial x_1}, \quad \frac{\partial x_2 z}{\partial x_2}, \quad \ldots, \quad \frac{\partial x_n z}{\partial x_n}\right]^T = \begin{bmatrix}z, & z, & \ldots, & z\end{bmatrix}^T, \tag{7.86}$$

and

$$\frac{\partial y}{\partial z} = \frac{\partial}{\partial z}\sum_{i=1}^{n} x_i z = \sum_{i=1}^{n}\frac{\partial}{\partial z} x_i z = \sum_{i=1}^{n} x_i = \mathrm{sum}(\mathbf{x}). \tag{7.87}$$

7.7.6 Gradient of chain rule vector function combinations

Let

$$\mathbf{f}(x) = \begin{bmatrix} f_1(\mathbf{g}(x)) \\ f_2(\mathbf{g}(x)) \end{bmatrix}, \tag{7.88}$$

where

$$\mathbf{g}(x) = \begin{bmatrix} g_1(x) \\ g_2(x) \end{bmatrix} \tag{7.89}$$

We can use Equation (7.57) to compute the derivative of $\mathbf{f}$. Simply compute the derivative of $f_1(x)$ and $f_2(x)$ and place them one above the other:

$$\frac{\partial \mathbf{f}}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial g_1}\frac{\partial g_1}{\partial x} + \frac{\partial f_1}{\partial g_2}\frac{\partial g_2}{\partial x} \\ \frac{\partial f_2}{\partial g_1}\frac{\partial g_1}{\partial x} + \frac{\partial f_2}{\partial g_2}\frac{\partial g_2}{\partial x}. \end{bmatrix} \tag{7.90}$$

We can represent this as:

$$\frac{\partial \mathbf{f}}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial g_1} & \frac{\partial f_1}{\partial g_2} \\ \frac{\partial f_2}{\partial g_1} & \frac{\partial f_2}{\partial g_2} \end{bmatrix} \begin{bmatrix} \frac{\partial g_1}{\partial x} \\ \frac{\partial g_2}{\partial x} \end{bmatrix} = \frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\partial \mathbf{g}}{\partial x} \tag{7.91}$$

As an example, let:

$$\mathbf{f}(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \end{bmatrix} = \begin{bmatrix} \ln(x^2) \\ \sin(3x) \end{bmatrix},$$

and

$$\mathbf{g}(x) = \begin{bmatrix} g_1(x) \\ g_2(x) \end{bmatrix} = \begin{bmatrix} x^2 \\ 3x \end{bmatrix}.$$

Then, by Equation (7.91):

$$\frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\partial \mathbf{g}}{\partial x} = \begin{bmatrix} \frac{1}{g_1} & 0 \\ 0 & \cos(g_2) \end{bmatrix} \begin{bmatrix} 2x \\ 3 \end{bmatrix} = \begin{bmatrix} \frac{1}{g_1}2x \\ 3\cos(g_2) \end{bmatrix} = \begin{bmatrix} \frac{2}{x} \\ 3\cos(3x) \end{bmatrix}.$$

If instead of a scalar x we have a vector $\mathbf{x}$, we have:

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{g}(\mathbf{x})) = \frac{\partial \mathbf{f}}{\partial \mathbf{g}}\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial g_1} & \frac{\partial f_1}{\partial g_2} & \cdots & \frac{\partial f_1}{\partial g_k} \\ \frac{\partial f_2}{\partial g_1} & \frac{\partial f_2}{\partial g_2} & \cdots & \frac{\partial f_2}{\partial g_k} \\ & & \vdots & \\ \frac{\partial f_m}{\partial g_1} & \frac{\partial f_m}{\partial g_2} & \cdots & \frac{\partial f_m}{\partial g_k} \end{bmatrix} \begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_n} \\ & & \vdots & \\ \frac{\partial g_k}{\partial x_1} & \frac{\partial g_k}{\partial x_2} & \cdots & \frac{\partial g_k}{\partial x_n} \end{bmatrix}. \tag{7.92}$$

If f_i is a function of only g_i, then every term off the diagonal becomes zero and:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{g}} = \operatorname{diag}\left(\frac{\partial f_1}{g_1}, \ldots, \frac{\partial f_n}{g_n}\right). \tag{7.93}$$

If g_i is a function of only x_i analogously:

$$\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \operatorname{diag}\left(\frac{\partial g_1}{x_1}, \ldots, \frac{\partial g_n}{x_n}\right). \tag{7.94}$$

7.7.7 Gradient of a neuron

A neuron is the function

$$\text{neuron}(\mathbf{x}) = \max(0, \langle \mathbf{w}, \mathbf{x} \rangle + b). \tag{7.95}$$

We have

$$\frac{\partial \,\text{neuron}}{\partial \mathbf{w}} = \frac{\partial \,\text{neuron}}{\partial z}\frac{\partial z}{\partial \mathbf{w}}, \tag{7.96}$$

where $z = \langle \mathbf{w}, \mathbf{x} \rangle + b$.

Computing $\frac{\partial z}{\partial \mathbf{w}}$

Trivially:

$$\langle \mathbf{w}, \mathbf{x} \rangle = \operatorname{sum}(\mathbf{w} \cdot \mathbf{x}). \tag{7.97}$$

Let

$$\mathbf{v} = \mathbf{w} \cdot \mathbf{x}, \tag{7.98}$$

$$u = \operatorname{sum}(\mathbf{v}). \tag{7.99}$$

Then (Equation (7.74))

$$\frac{\partial \mathbf{v}}{\partial \mathbf{w}} = \frac{\partial (\mathbf{w} \cdot \mathbf{x})}{\partial \mathbf{w}} = \operatorname{diag}(\mathbf{x}), \tag{7.100}$$

and (Equation (7.84))

$$\frac{\partial u}{\partial \mathbf{v}} = \frac{\partial}{\partial \mathbf{v}} \operatorname{sum}(\mathbf{v}) = \mathbf{1}^T. \tag{7.101}$$

Therefore

$$\frac{\partial u}{\partial \mathbf{w}} = \frac{\partial u}{\partial \mathbf{v}} \frac{\partial \mathbf{v}}{\partial \mathbf{w}} = \mathbf{1}^T \operatorname{diag}(\mathbf{x}) = \mathbf{x}^T. \tag{7.102}$$

Finally:

$$\frac{\partial z}{\partial \mathbf{w}} = \frac{\partial u}{\partial \mathbf{w}} + \frac{\partial b}{\partial \mathbf{w}} = \mathbf{x}^T + \mathbf{0}^T = \mathbf{x}^T, \tag{7.103}$$

$$\frac{\partial z}{\partial b} = \frac{\partial u}{\partial b} + \frac{\partial b}{\partial b} = 0 + 1 = 1. \tag{7.104}$$

Computing $\frac{\partial \operatorname{neuron}}{\partial z}$

Trivially:

$$\frac{\partial}{\partial z} \max(0, z) = \begin{cases} 0 & \text{if } z \leq 0, \\ 1 & \text{if } z > 0. \end{cases} \tag{7.105}$$

Computing $\frac{\partial \operatorname{neuron}}{\partial \mathbf{w}}$ and $\frac{\partial \operatorname{neuron}}{\partial b}$

From Equation (7.96):

$$\frac{\partial \operatorname{neuron}}{\partial \mathbf{w}} = \frac{\partial \operatorname{neuron}}{\partial z} \frac{\partial z}{\partial \mathbf{w}} = \begin{cases} \mathbf{0}^T & \text{if } z \leq 0, \\ \mathbf{x}^T & \text{if } z > 0. \end{cases} = \begin{cases} \mathbf{0}^T & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b \leq 0 \\ \mathbf{x}^T & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b > 0. \end{cases} \tag{7.106}$$

Similarly:

$$\frac{\partial \operatorname{neuron}}{\partial b} = \begin{cases} 0 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b \leq 0 \\ 1 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b > 0. \end{cases} \tag{7.107}$$

7.7.8 Gradient of the loss function

We want to compute the gradients of Equation (7.55). Let:

$$u(\mathbf{w}, b, \mathbf{x}_i) = \max(0, \langle \mathbf{w}, \mathbf{x}_i \rangle + b), \tag{7.108}$$

$$v(y, u) = y - u, \tag{7.109}$$

$$C(v) = \frac{1}{n} \sum_{i=1}^{n} v^2. \tag{7.110}$$

Computing $\frac{\partial C}{\partial \mathbf{w}}$

u is simply our neuron function, therefore (from Equation (7.106)):

$$\frac{\partial u}{\partial \mathbf{w}} = \frac{\partial \,\text{neuron}}{\partial z}\frac{\partial z}{\partial \mathbf{w}} = \begin{cases} \mathbf{0}^T & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq 0 \\ \mathbf{x}_i^T & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b > 0. \end{cases} \tag{7.111}$$

v is simply $y - u$, therefore:

$$\frac{\partial v}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}}(y - u) = \mathbf{0}^T - \frac{\partial u}{\partial \mathbf{w}} = -\frac{\partial u}{\partial \mathbf{w}} = \begin{cases} \mathbf{0}^T & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq 0 \\ -\mathbf{x}_i^T & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b > 0. \end{cases} \tag{7.112}$$

Finally, we need to find the derivative of the whole loss function with respect to $\mathbf{w}$. Using the chain rule, we know that:

$$\frac{\partial C(v)}{\partial \mathbf{w}} = \frac{\partial C}{\partial v}\frac{\partial v}{\partial \mathbf{w}}. \tag{7.113}$$

Trivially:

$$\frac{\partial C}{\partial v} = \frac{\partial}{\partial v}\frac{1}{n}\sum_{i=1}^{n} v^2 = \frac{1}{n}\sum_{i=1}^{n} 2v. \tag{7.114}$$

From Equation (7.118), multiplying the two derivatives (Equation (7.119)):

$$\frac{\partial C}{\partial \mathbf{w}} = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} 2v\mathbf{0}^T & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq 0 \\ -2v\mathbf{x}_i^T & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b > 0 \end{cases} \tag{7.115}$$

When substituting:

$$\frac{\partial C}{\partial \mathbf{w}} = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} \mathbf{0}^T & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq 0 \\ -2(y_i - \max(0, \langle \mathbf{w}, \mathbf{x}_i \rangle + b))\mathbf{x}_i^T & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b > 0 \end{cases} \tag{7.116}$$

Computing $\frac{\partial C}{\partial b}$

u is simply our neuron function, therefore (from Equation (7.106)):

$$\frac{\partial u}{\partial b} = \frac{\partial \,\text{neuron}}{\partial z}\frac{\partial z}{\partial b} = \begin{cases} 0 & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq 0 \\ 1 & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b > 0. \end{cases} \tag{7.117}$$

v is simply $y - u$, therefore:

$$\frac{\partial v}{\partial b} = \frac{\partial}{\partial b}(y - u) = \mathbf{0}^T - \frac{\partial u}{\partial b} = -\frac{\partial u}{\partial b} = \begin{cases} 0 & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq 0 \\ -1 & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b > 0. \end{cases} \tag{7.118}$$

Finally, we need to find the derivative of the whole loss function with respect to $\mathbf{w}$. Using the chain rule, we know that:

$$\frac{\partial C(v)}{\partial b} = \frac{\partial C}{\partial v}\frac{\partial v}{\partial b}. \tag{7.119}$$

Trivially:

$$\frac{\partial C}{\partial v} = \frac{\partial}{\partial v}\frac{1}{n}\sum_{i=1}^{n} v^2 = \frac{1}{n}\sum_{i=1}^{n} 2v. \tag{7.120}$$

From Equation (7.118), multiplying the two derivatives (Equation (7.119)):

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} 2v \cdot 0 & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq 0 \\ -2v & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b > 0 \end{cases} \tag{7.121}$$

When substituting:

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} 0 & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq 0 \\ -2(y_i - \max(0, \langle \mathbf{w}, \mathbf{x}_i \rangle + b)) & \text{if } \langle \mathbf{w}, \mathbf{x}_i \rangle + b > 0 \end{cases} \tag{7.122}$$

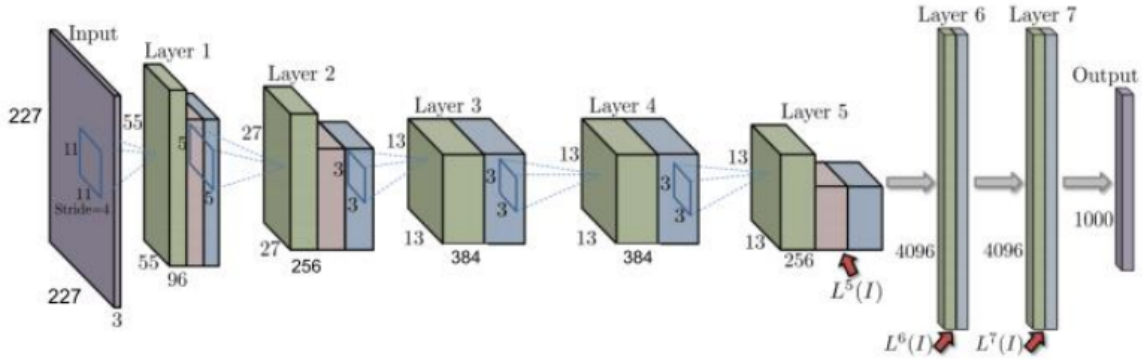


Figure 7.18: CaffeNet. Source: [72].

7.8 CaffeNet

CaffeNet (2014) is a variant of AlexNet. Its architecture is reported in Figure 7.18, where, for the Layers 1, 2, 3, 4 and 5, the colors refers to:

- Purple: Input ($227 \times 227 \times 3$ images) and Output with 1000 nodes, as it has been trained on ImageNet-1000.
- Green: Convolution.
- Red: Max-Pooling.
- Blue: ReLU activation.

Other properties are:

- Layer 6, 7 and 8 are fully-connected. Layer 8 (the output) has a softmax activation.
- The stride is 4 for the first convolutional layer, then it is always 1.
- Filter size is 11 for the first convolutional layer, 5 for the second, 3 for the others.
- Layer 5, 6 and 7 denote reusable features for other problems.
- Total number of parameters: 60 millions.

7.9 EnzyNet

EnzyNet (2018) is a network for enzyme classification. The input are volumes of size $32 \times 32 \times 32$ containing the structural coordinates of the enzymes encoded in *voxels* (three-dimensional equivalent of the pixel). The inputs go through:

1. Convolutional layer of 32 filters of size $9 \times 9 \times 9$ (since the inputs are volumes, we are doing 3D convolutions) with stride 2.
2. Convolutional layer of 64 filters of size $5 \times 5 \times 5$ with stride 1.
3. Max-pooling layer of size $2 \times 2 \times 2$ with stride 2.
4. Two fully-connected layers of 128 and six (the number of classes) hidden nodes respectively, concluded by a softmax activation that outputs class probabilities.

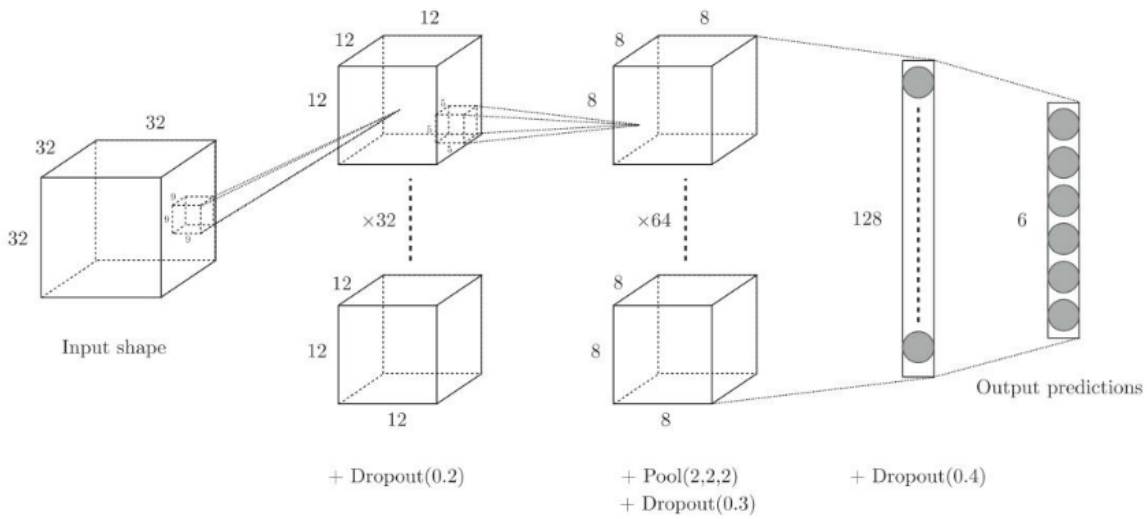


Figure 7.19: EnzyNet. Source: [72].

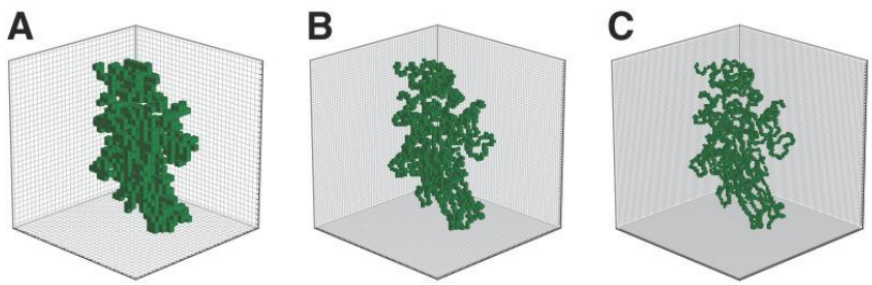


Figure 7.20: Illustration of enzyme 2Q3Z for gride sizes $l = 32$ (A), $l = 64$ (B) and $l = 96$ (C). Source: [72].

7.9.1 Voxels representation

Enzymes are represented in voxels fitted in a cube V of a fixed grid size l (above defined to be 32) with respect to the three dimensions. Continuity between voxels are achieved by nearest neighbor interpolation, such that for $(i, j, k) \in \{0, 1, 2, \ldots, l-1\}^3$, a voxel of vertices

$$(i + \Delta x, j + \Delta y, k + \Delta z), \text{ such that } (\Delta x, \Delta y, \Delta z) \in \{0, 1\}^3 \tag{7.123}$$

takes the value 1 if the backbone of the enzyme passes through the voxel, and 0 otherwise.

Preprocessing

To construct this shape representation, some preprocessing steps are necessary. First, protein structure is mapped to a grid of a predefined resolution. The selection of grid resolution determines the level of complexity/scale retained for the enzymatic structures, A full resolution is not preferred due to high data dimensionality and because fine local details are less relevant in characterizing enzymes' chemical reactions. Thus, in order to avoid to get trapped into local minima, side chains are ignored and enzymes are represented exclusively through their *backbone* atoms that are carbon, nitrogen, and calcium.

Additionally, we note that enzymes do not possess any absolute spatial orientation. Unlike objects such as chairs or boats that appear usually with a specifc orientation, proteins can have any orientation in 3D conformational space, thus the Cartesian coordinates defined by the model stored in the *Protein Data Bank* (PDB) represent only a frozen in space and time snapshot of an overall highly dynamic structural diversity. The orientation is irrelevant to the properties of the protein. This observation underlines the need of either a rotation invariant representation of a convention that makes output

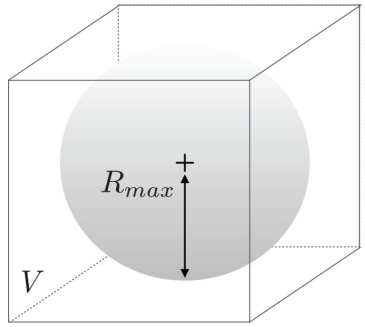


Figure 7.21: Illustration of the meaning of $R_{\max}$ with respect to volume V. Source: [5].

structures comparable one to another based on the definition of an intrinsic coordinate system. We define as origin of this intrinsic coordinate system the consensus barycenter of the protein as it is defined by taking into account only the four atoms of the backbone for each residue, and as axes the principal directions of each enzyme calculated by PCA. Each structure is rotated around its center and the three principal directions of the enzyme aligned with the three axes of the Cartesian coordinate system. Our source files provide the coordinates of the enzymes at a same scale. After all, proteins are comprised of various combinations of equally sized amino acids. This scaling issue is therefore equivalent to determining a maximum radius $R_{\max}$ so that the atom occupancy information contained in the sphere centered on the barycenter of the enzyme and of radius $R_{\max}$ fits into V. This situation is illustrated in Figure 7.21.

$R_{\max}$ has to be large enough so that a sufficient number of enzymes fit the most of their volume inside V. Conversely, it also has to be small enough so that most enzymes are represented at a satisfactory resolution. As a result, a homothetic transformation with center S and ratio λ, where S is the center of V and

$$\lambda = \left\lfloor \frac{l}{2} - 1 \right\rfloor \cdot \frac{1}{R_{\max}} \tag{7.124}$$

is performed on all enzymes to scale them to the desired size. When l is low, the grid is coarse enough so that the voxels of the structure have a contiguous shape. On the other hand, big volumes tend to separate voxels, which engender "holes". In that case, consecutive atoms $\mathbf{A}_i$ and $\mathbf{A}_{i+1}$ (vectors) are interpolated by p regularly spaced new points $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_p$ such that:

$$\mathbf{v}_k = \frac{(p-k+1)\mathbf{A}_i + k\mathbf{A}_{i+1}}{p+1}. \tag{7.125}$$

The entire procedure is shown in Figure 7.22.

Data: N training enzymes, grid size of l, homothetic transformation ratio λ, p interpolations, probability p_{flip} to flip with respect to each axis.
Input: Raw coordinates contained in PDB files
Output: Volumes of binary voxels representing backbone atoms occupancy

foreach *of the N enzymes of the training set* **do**
  Step 1: structural information extraction
  Extract coordinates of backbone atoms from its PDB file
  Step 2: holes completion
  Interpolate consecutive backbone atoms by p new points
  Step 3: size adjustment
  Center barycenter S of the coordinates on (0, 0, 0)
  Homothetic transformation of each point with center S and ratio λ
  Step 4: enzyme orientation
  Principal component analysis (PCA) transformation
  Step 5: random augmentation
  if `True` *with probability* p_{flip} **then**
    Flip coordinates with respect to the origin along *x—axis*
  if `True` *with probability* p_{flip} **then**
    Flip coordinates with respect to the origin along *y—axis*
  if `True` *with probability* p_{flip} **then**
    Flip coordinates with respect to the origin along *z—axis*
  Step 6: voxelization
  Center barycenter S of the coordinates on $\left(\frac{l}{2}, \frac{l}{2}, \frac{l}{2}\right)$
  Transform coordinate points into binary voxels

Figure 7.22: Summary of the preprocessing steps done to each enzyme at training time. Source: [72].

7.10 GoogLeNet

In 2014, *GoogLeNet* won the ImageNet Challenge, proposing a structure that combined the strenghts of *Network in Network* (NiN) and paradigms of repeated blocks. One focus of the original paper was to address the question of which size convolution filters are best. Previous popular networks employed choice as small as 1×1 and as large as 11×11. The paper states that sometimes it can be advantageous to employ a combination of variously-sized filters.

7.10.1 Inception block

The basic convolutional block in GoogLeNet is the *Inception block*, reported in Figure 7.23. It consists of four parallel paths:

- The first three paths use convolutional layers with window of sizes of 1×1, 3×3 and 5×5 to extract information from different spatial sizes.
- The middle two parts perform a 1×1 convolution on the input to reduce the number of channels (recall Figure 7.7), reducing the model's complexity.
- The fourth path uses a 3×3 max-pooling layer followed by a 1×1 convolutional layer to change the number of channels.
- Finally, the outputs along each path are concatenated along the channel dimension and comprise the block's output. In fact, all four paths all use appropriate padding to give the input and output the same height and width.

To gain some intuition for why this network works so well, consider the combination of the filters. They explore the image in a variety of filter sizes. This means that details at different extents can be recognized efficiently by filters of different sizes. At the same time, we can allocate different amounts of parameters for different filters.

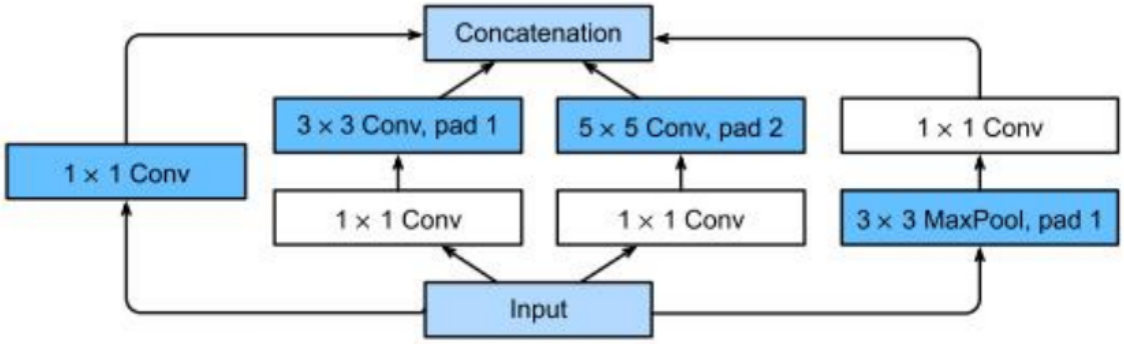


Figure 7.23: Structure of the Inception block. Source: [72].

7.11 ResNet

Before describing *ResNet*, we introduce other concepts.

7.11.1 Batch normalization for fully-connected layers

To motivate *batch normalization*, let us review a few practical challenges that arise when training ML models and neural networks in particular. For a typical MLP or CNN, as we train, the variables (e.g., affine transformation outputs in MLP) in intermediate layers may take values with widely varying magnitudes: both along the layers from the input to the output, across units in the same layer, and over time due to our updates to the model parameters. The inventors of batch normalization postulated informally that this drift in the distribution of such variables could hamper the convergence of the network. Intuitively, we might conjecture that if one layer has variable values that are 100 times that of another layer, this might necessitate compensatory adjustments in the learning rates.

Deeper networks are complex and easily capable of overfitting. This means that regularization becomes more critical. A form of regularization appears to be batch normalization. Batch normalization is applied to individual layers (optionally, to all of them) and works as follows: In each training iteration, we first normalize the inputs (of batch normalization) by subtracting their mean and dividing by their standard deviation, where both are estimated based on the statistics of the current mini-batch. Next, we apply a scale coefficient and a scale offset. Note that if we tried to apply batch normalization with mini-batches of size 1, we would not be able to learn anything. That is because after subtracting the means, each hidden unit would take value 0. In fact, with large enough mini-batches, the approach proves effective and stable. One takeaway here is that when applying batch normalization, the choice of batch size may be even more significant than without batch normalization.

Formally, denoting $\mathbf{x} \in B$ an input to batch normalization (operator BN below) that is from a mini-batch B, batch normalization transforms $\mathbf{x}$ according to the following expression:

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \cdot \frac{\mathbf{x} - \boldsymbol{\mu}_B}{\boldsymbol{\sigma}_B} + \boldsymbol{\beta}, \tag{7.126}$$

where $\boldsymbol{\mu}_B$ is the mean and $\boldsymbol{\sigma}_B$ is the standard deviation of B. Note that $\mathbf{x}$ is an input to a generic layer (i.e. we are considering $\mathbf{x} = \mathbf{net}$). It corresponds to a training sample only for the first layer. After applying standardization, the resulting mini-batch has zero mean and unit variance. Because the choice of unit variance (vs. some other magic number) is an arbitrary choice, we commonly include element wise scale parameter $\boldsymbol{\gamma}$ and shift parameter $\boldsymbol{\beta}$ that have the same shape as $\mathbf{x}$. Note that $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are parameters that need to be learned jointly with the other model parameters. Consequently, the variable magnitudes for intermediate layers cannot diverge during training because batch normalization actively centers and rescales them back to a given mean and size. One piece of practitioner intuition or wisdom is that batch normalization seems to allow for more aggressive learning rates. Formally, we calculate $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ as follows:

$$\boldsymbol{\mu}_B = \frac{1}{|B|} \sum_{\mathbf{x} \in B} \mathbf{x}, \tag{7.127}$$

$$\boldsymbol{\sigma}_B^2 = \frac{1}{|B|} \sum_{\mathbf{x} \in B} (\mathbf{x} - \boldsymbol{\mu}_B)^2 + \epsilon, \tag{7.128}$$

where $\epsilon > 0$ is a small constant added to the variance estimate to ensure that we never attempt division by 0. The estimates $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ counteract the scaling issue by noisy estimates of mean and variance. As already mentioned in Section 6.3, noise can be beneficial, acting as a form of regularization.

Why do not we use the entire dataset to estimate the mean and the variance? During training, such exact calculation is infeasible because the intermediate variables for all data change every time we update our model. However, once the model is trained, we can compute the means and variances for each layer based on the entire dataset. Indeed, this is standard practice. In fact, for models employing batch normalization layers, those functions differently in training mode (normalizing by mini-batch statistics) and in prediction mode (normalizing by dataset statistics).

In general, we apply batch normalization after the affine transformation and before the non-linear activation function. Denoting the input to the fully-connected layer by $\mathbf{x}$, the affine transformation $\mathbf{net} = \mathbf{W}\mathbf{x} + \mathbf{b}$, and the activation function by f, we can express the computation of a batch-normalization-enabled, fully-connected layer output $\mathbf{h}$ as:

$$\mathbf{h} = f(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b})). \tag{7.129}$$

7.11.2 Batch normalization for convolutional layers

Batch normalization for fully-connected layers and convolutional layers are slightly different. Similarly, with convolutional layers, we can apply batch normalization after the convolution and before the nonlinear activation function. When the convolution has multiple output channels, we need to carry out batch normalization for each of the outputs of these channels, and each channel has its own scale and shift parameters, both of which are scalars. Assume that our mini-batches contain m examples and that for each channel, the output of the convolution has height p and width q. For convolutional layers, we carry out each batch normalization over the $m \cdot p \cdot q$ elements per output channel simultaneously. Thus, we collect the values over all spatial locations when computing the mean and variance and consequently apply the same mean and variance within a given channel to normalize the value at each spatial location.

7.11.3 Why batch normalization works

Intuitively, batch normalization is thought to make the optimization landscape smoother. However, we must be careful to distinguish between speculative intuitions and true explanations for the phenomena that we observe when training deep models. Recall that we do not even know why simpler deep neural networks (MLPs and conventional CNNs) generalize well in the first place. Even with dropout and weight decay, they remain so flexible that their ability to generalize to unseen data cannot be explained via conventional learning-theoretic generalization guarantees. In the original paper proposing batch normalization, the authors offered an explanation for why it works: by reducing *internal covariate shift*. Presumably by internal covariate shift the authors meant the notion that the distribution of variable values changes over the course of training. However, there were two problems with this explanation:

- This drift is very different from *covariate shift*, rendering the name a misnomer. Covariate shift is a type of causal ($\mathbf{x}$ causes y) distribution shift, which assumes that the distribution of input can change over time, while the labeling function stays the same [23].
- The explanation offers an underspecified intuition but leaves the question of why *precisely* this technique works an open question wanting for a rigorous explanation.

Following the success of batch normalization, its explanation in terms of internal covariate shift has repeatedly surfaced debates in the technical literature and broader discourse about how to present machine learning research. Ali Rahimi used internal covariate shift as a focal point in an argument likening the modern practice of deep learning to alchemy. Subsequently, the example was revisited in detail in a position paper outlining troubling trends in machine learning. Other authors have proposed alternative explanations for the success of batch normalization, some claiming that batch normalizations success comes despite exhibiting behavior that is in some ways opposite to those claimed in the original paper. We note that the internal covariate shift is no more worthy of criticism than any

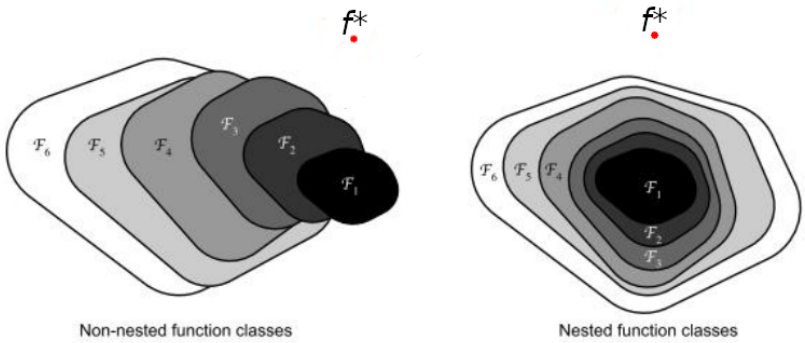


Figure 7.24: Non-nested function classes (left) and nested function classes (right). The red point indicates the truth function $f^\star$. Source: [72].

of thousands of similarly vague claims made every year in the technical machine learning literature. Likely, its resonance as a focal point of these debates owes to its broad recognizability to the target audience. Batch normalization has proven an indispensable method, applied in nearly all deployed image classifiers, earning the paper that introduced the technique tens of thousands of citations.

7.11.4 Function classes

We need to understand how adding layers can increase the complexity and expressiveness of the network. Even more important is the ability to design networks where adding layers makes networks strictly more expressive rather than just different.

Consider $\mathcal{F}$ be the class of functions (the hypothesis class $\mathcal{H}$ of statistical learning) that a specific network architecture (together with learning rates and other hyperparameters) can reach. That is, for all $f \in \mathcal{F}$ there exists some set of parameters that can be obtained through training on a suitable dataset. Let us assume that $f^\star$ is the "truth function" that we would like to find. If $f^\star \in \mathcal{F}$, we are in a good shape, but typically we will not be so lucky. Instead, we try to find some $f^\star_{\mathcal{F}}$ which is our best bet within $\mathcal{F}$: we pick $f^\star_{\mathcal{F}}$ that minimizes the error on the training set

If we design a more powerful architecture $\mathcal{F}'$ it is reasonable to assume that we should arrive at a better outcome. In other words, we should expect that $f^\star_{\mathcal{F}'}$ is better than $f^\star_{\mathcal{F}}$. However, if $\mathcal{F} \not\subseteq \mathcal{F}'$ there is no guarantee that this should even happen. In fact, $f^\star_{\mathcal{F}'}$ might well be worse. As illustrated in Figure 7.24 for non-nested function classes, a larger function class does not always move closer to the truth function $f^\star$. For instance, on the left of Figure 7.24, though $\mathcal{F}_3$ is closer to $f^\star$ than $\mathcal{F}_1$, the largest function class $\mathcal{F}_6$ moves away and there is no guarantee that further increasing the complexity can reduce the distance from $f^\star$. With nested function classes where $\mathcal{F}_1 \subseteq \cdots \subseteq \mathcal{F}_6$ on the right of Figure 7.24 we can avoid the aforementioned issue from the non-nested function classes.

7.11.5 ResNet

Thus, only if larget function classes contain the smaller ones we are guaranteed that increasing them strictly increases the expressive power of the network. For DNNs, if we can train a newly-added layer into an identity function $f(\mathbf{x}) = \mathbf{x}$ ($\mathbf{x}$ input to the layer), the new model will be as effective as the original model. As the new model may get a better solution to fit the training set, the added layer might make it easier to reduce training errors.

With this in mind, He et al. proposed the *Residual Network*, also called *ResNet* (2016). The main idea is that every additional layer should easily contain the identity function as one of its elements.

7.11.6 Residual blocks

Consider the network on the left of Figure 7.25, composed of a block (dotted line) of two convolutional layers and an activation layer. Denote the input of network by $\mathbf{x}$. We assume that the desired underlying mapping we want to obtain by learning is $f(\mathbf{x})$, to be used as the input to the activation function on the top (therefore f is not the activation function in this case). This means that, since

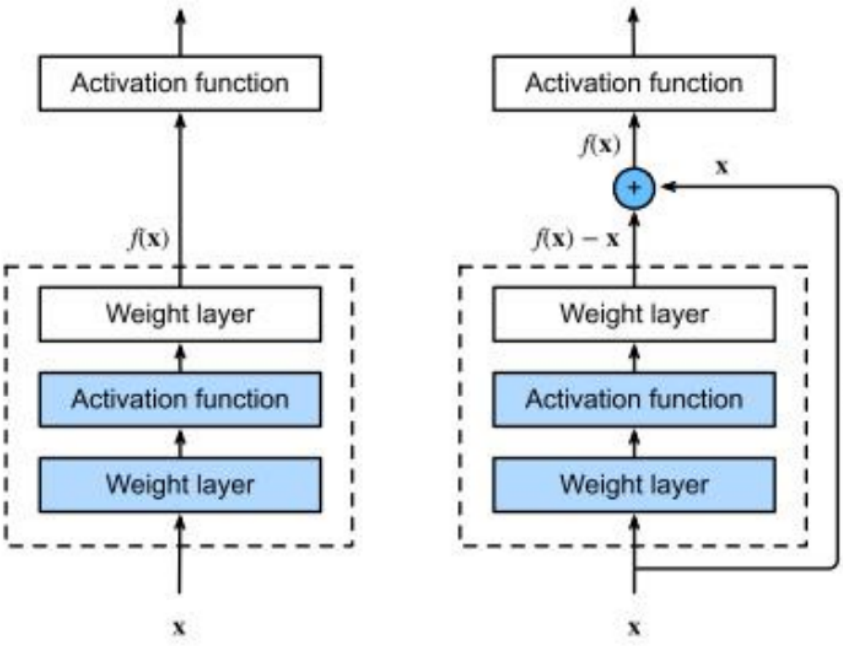


Figure 7.25: A regular block (left) and a residual block (right). Source: [72].

convolutional operations are performed, $f(\mathbf{x})$ outputs a tensor. On the left of Figure 7.25 must directly learn the mapping $f(\mathbf{x})$. On the right, the portion within the dotted line box needs to learn the *residual mapping*

$$f(\mathbf{x}) - \mathbf{x}. \tag{7.130}$$

This means the residual block decomposes $f(\mathbf{x})$ into

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}), \tag{7.131}$$

where $g(\mathbf{x})$ is a nonlinear transformation. That is, ResNet decomposes the function into a simple linear term and a more complex nonlinear one. The solid line carrying the layer input $\mathbf{x}$ to the addition operator is called a *residual connection* (or *shortcut connection*). With residual blocks, inputs can forward propagate faster through the residual connections across layers. If the block output is 0, we have $f(\mathbf{x}) = \mathbf{x}$. ResNet follows VGG's full 3×3 convolutional layer design. The residual block has two 3×3 convolutional layers with the same number of output channels as input channels, stride 1 and padding 1. In this way, according to Equation (7.23), the shape of the output of the dotted-line region is the same of $\mathbf{x}$. For example, if $\mathbf{x}$ has 5 channels, then each convolutional layer will have 5 filters. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function. This kind of design requires that the output of the two convolutional layers has to be of the same shape as the input, so that they can be added together. If we want to change the number of channels, we need to introduce an additional 1×1 convolutional layer to transform the input into the desired shape for the addition operation. A complete representation of the residual block is reported in Figure 7.26.

7.12 DenseNet

ResNet significally changed the view of how to parametrize the functions in deep networks. *Dense convolutional network*, or *DenseNet* (2017) is the logical extension of ResNet. Recall the Taylor expansion for functions $\mathbb{R} \to \mathbb{R}$. For $x = 0$, it holds:

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \dots. \tag{7.132}$$

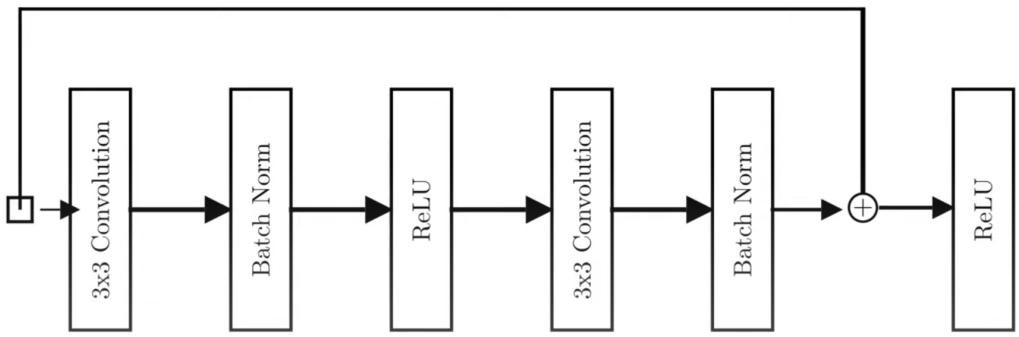


Figure 7.26: Residual block. Source: [95].

The function is decomposed into increasingly higher order terms. If we want to capture information beyond two terms, we need the DenseNet. As shown in Figure 7.27, the key difference is that in the latter case outputs are *concatenated* (denoted by $[\cdot]$) rather than added.

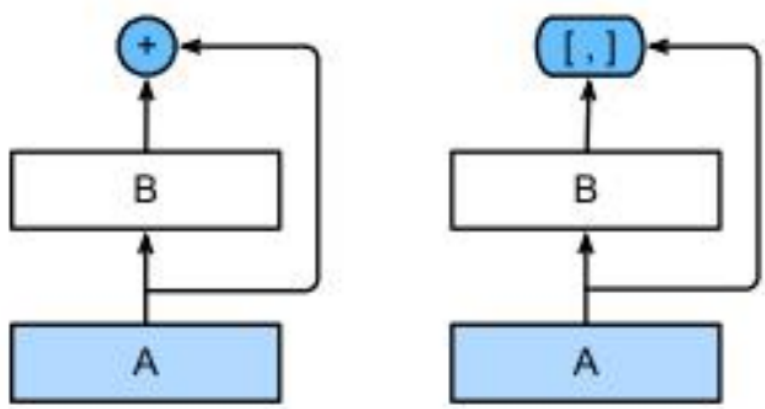


Figure 7.27: The main difference between ResNet (left) and DenseNet (right) in cross-layer connections. It is the use of addition in ResNet and the use of concatenation in DenseNet. Source: [72].

Figure 7.27 shows a dense block with only one concatenation. However, the dense block usually contains more concatenations, as shown in Figure 7.28. As a result, we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions (i.e. DenseNet decomposes $f(\mathbf{x})$ into):

$$f(\mathbf{x}) = \left[\mathbf{x}, \quad f_1(\mathbf{x}), \quad f_2([\mathbf{x}, f_1(\mathbf{x})]), \quad f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \quad \ldots\right]. \tag{7.133}$$

Each $f_1, f_2, \ldots$ performs convolution, batch normalization and ReLU [53].

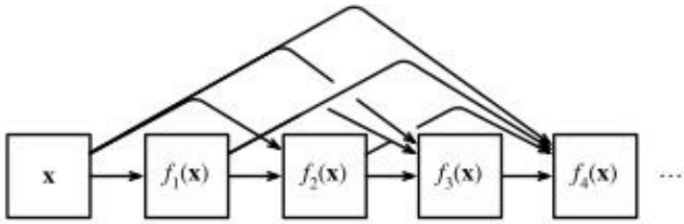


Figure 7.28: Dense connections of a single dense block. Source: [72].

In the end, all these functions are combined in MLP to reduce the number of features again. In

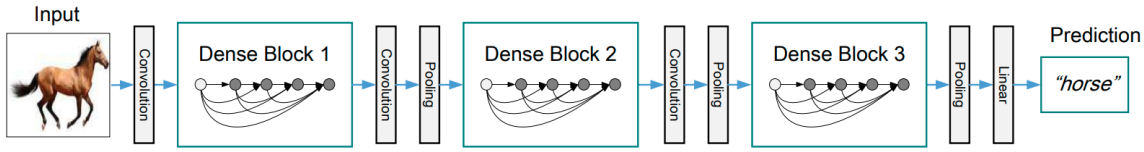


Figure 7.29: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling. Source: [53].

terms of implementation this is quite simple: rather than adding terms, we concatenate them. The name DenseNet arises from the fact that the dependency graph between variables becomes quite dense. The last layer of such a chain is densely connected to all previous layers.

The main components of a DenseNet are *dense blocks* and *transition layers*. The former define how the inputs and outputs are concatenated, while the layer control the number of channels so that it is not too large (Figure 7.29).

7.13 Automatic differentiation

Automatic differentiation (auto-differentiation, autodiff, or AD), is a set of techniques to evaluate the partial derivative of a function specified by a computer program [6]. Consider the cost function associated with a sample $x \in \mathbb{R}$ with label $y \in \mathbb{R}$:

$$C(y, w, x, b) = z = y - \max(0, wx + b). \tag{7.134}$$

Automatic differentiation can only calculate the partial derivative of an expression on a certain point, we have to assign initial values to each of the variables. Consider $y = 5$, $w = 2$, $x = 1$ and $b = 1$.

7.13.1 Computational graph

Before deriving the expression, it must be converted into a *computational graph* (note that in this section we are generalizing the same concept seen in Section 7.7.1). A computational graph turns each operation into a node and connects them through edges. The computational graph for out example function is shown in Figure 7.30.

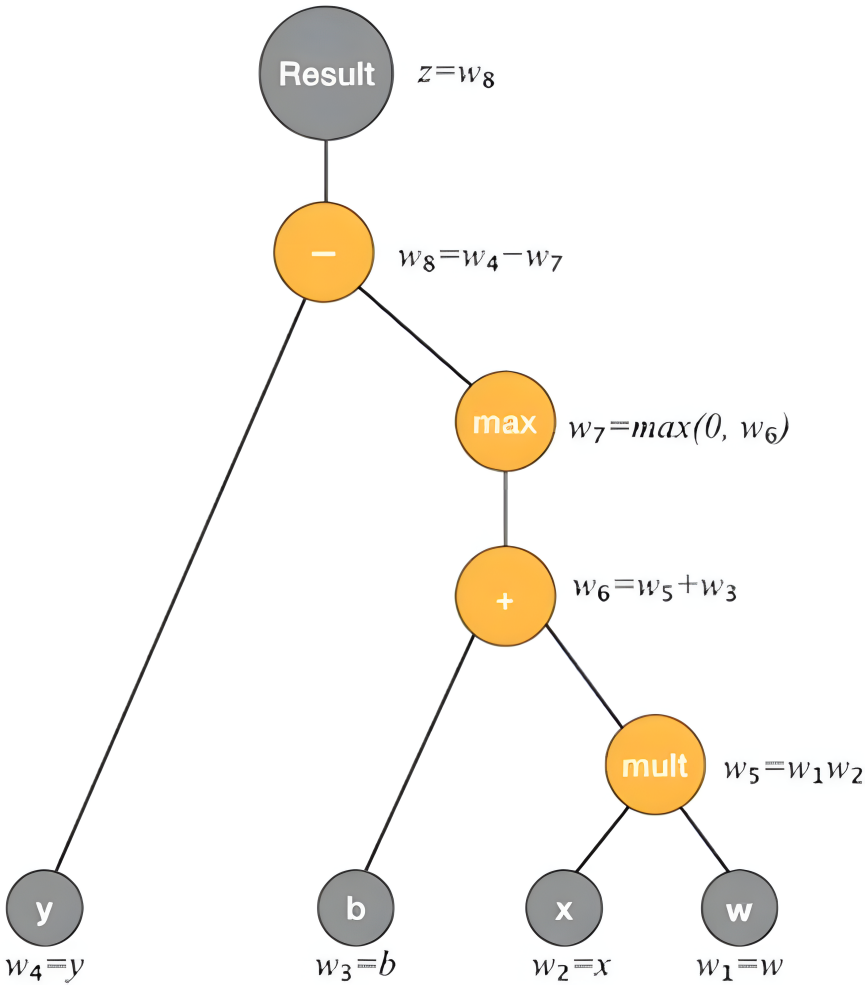


Figure 7.30: Computational graph for Equation (7.134). Source: [72].

First, let us calculate the values of each node, propagating from the bottom (the input variables) to the top (the output function). This is what we get (Figure 7.31):

Node	Expression	Value
w_1	w	2
w_2	x	1
w_3	b	1
w_4	y	5
w_5	$w_1 \cdot w_2$	2
w_6	$w_5 + w_3$	3
w_7	$\max(0, w_6)$	3
w_8	$w_4 - w_7$	2
z	w_8	2

Figure 3: Values of each node

Figure 7.31: Values of each node. Source: [72].

Next, we need to calculate the partial derivatives of each connection between operations, represented by the edges. These are the calculations of the partials of each edge (Figure 7.32):

$\frac{\delta w_5}{\delta w_1} = \frac{\delta w_1 w_2}{\delta w_1} = w_2$	$\frac{\delta w_5}{\delta w_2} = \frac{\delta w_1 w_2}{\delta w_2} = w_1$
$\frac{\delta w_6}{\delta w_5} = \frac{\delta w_5 + w_3}{\delta w_5} = 1$	$\frac{\delta w_6}{\delta w_3} = \frac{\delta w_5 + w_3}{\delta w_3} = 1$
$\frac{\delta w_7}{\delta w_6} = \frac{\delta \max(0, w_6)}{\delta w_6} = \begin{cases} 0, & w_6 < 0 \\ 1, & w_6 > 0 \end{cases}$	$\frac{\delta w_8}{\delta w_7} = \frac{\delta w_4 - w_7}{\delta w_7} = -1$
$\frac{\delta w_8}{\delta w_4} = \frac{\delta w_4 - w_7}{\delta w_4} = 1$	$\frac{\delta z}{\delta w_8} = 1$

Figure 7.32: Partial derivatives table. [72].

Now we can move on to calculating the partials. Let us find the partial of with respect to the weights. As seen in Figure 7.33, there is just one line that connects the result to the weights.

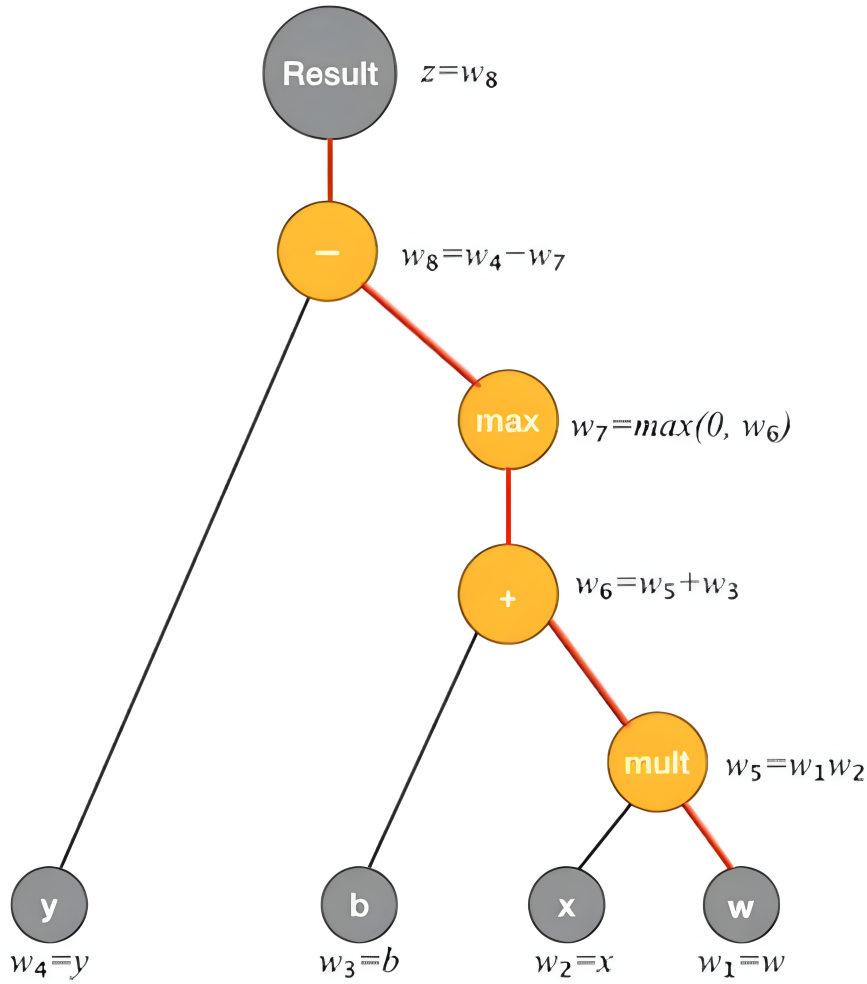


Figure 7.33: Computational graph for Equation (7.134) with path to compute $\frac{\partial z}{\partial w}$. Source: [72].

Therefore:

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial w_1} = \frac{\partial z}{\partial w_8}\frac{\partial w_8}{\partial w_7}\frac{\partial w_7}{\partial w_6}\frac{\partial w_6}{\partial w_5}\frac{\partial w_5}{\partial w_1}. \tag{7.135}$$

Then:

$$\frac{\partial z}{\partial w} = 1 \cdot (-1) \cdot \begin{cases} 0 & \text{if } w_6 \leq 0 \\ 1 & \text{if } w_6 > 0 \end{cases} \cdot 1 \cdot w_2 = -1. \tag{7.136}$$

This whole process can be completed automatically, and allows computers to compute the partial derivative of a value of a function accurately and quickly.

However, note that there is not always only one path between z and the parameter. For instance, try to create the graph related to $z = \frac{x}{1+|x|}$. In these cases, we have to resort to Equation (7.57) as we did in Figure 7.17.

7.14 Transfer Learning

Training complex CNNs on large datasets (e.g. ImageNet) can take days or weeks of machine time, even if performed on GPU. In addition, training a CNN on a new problem requires a large labeled training set (often not available). Fortunately, once the network has been trained, the time required for classifying a new pattern (forward propagation) is usually fast. As an alternative to training from scratch, we can pursue two *transfer learning* approaches:

- Fine-Tuning.
- Deep Features.

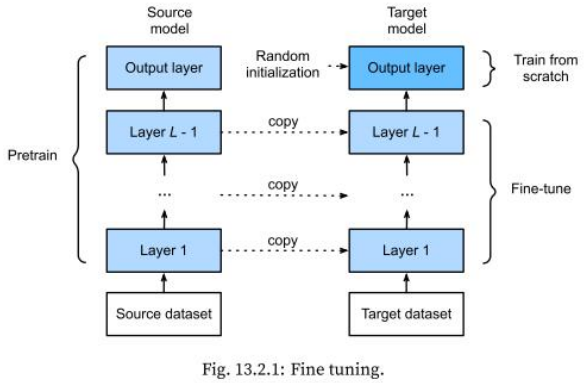


Figure 7.34: Fine-Tuning. Source: [72].

7.14.1 Fine-Tuning

In *Fine-Tuning* we start with a pre-trained network trained on a similar problem and:

1. Replace the output level with a new softmax output level (adjusting the number of classes).
2. the initial values of the weights are used those of the pre-trained network, except for the connections between the penultimate and last level whose weights are initialized at random.
3. New training iterations are performed to optimize the weights with respect to the peculiarities of the new dataset (it does not need to be very large).

The procedure is reported in Figure 7.34.

7.14.2 Deep Features

An existing pre-trained network is used without further fine-tuning. The features generated by the network during the forward step are *extracted* at intermediate levels (see layer 5, 6 and 7 of Figure 7.18). These features are concatenated and (Figure 7.35) used to train an external classifier (e.g. SVM) to classify the patterns of the new application domain.

We present now two possible variations.

Deep Features with innermost layers

Even the innermost layers of a deep network can be used to train a classifier. However, the problem is the very high dimensionality of these levels. A possible solution is to reduce this dimensionality using computationally manageable methods such as PCA or *Discrete Cosine Transform* (DCT), as shown in Figure 7.35. In fact, DCT can be used for dimensionality reduction [55] as it is at the foundation of JPEG compression.

Deep Features with different external classifier

Instead of concatenating the various feature vectors extracted from different layers and from different networks (e.g. different topologies), for each single feature extraction we can train a different SVM and then the results are combined (e.g. by sum rule). It is already shown in the literature that in many applications different topologies carry partially independent information, therefore combining different CNNs allows increased performance (Figure 7.36).

7.15 AutoEncoders

An *AutoEncoder (AE)* is a type of artificial neural network used to learn efficient codings of unlabeled data (unsupervised learning). An autoencoder learns two functions: an *encoding function* that transforms the input data ($\mathbf{x}$), and a *decoding function* that recreates the input data ($\mathbf{o}$) from the encoded

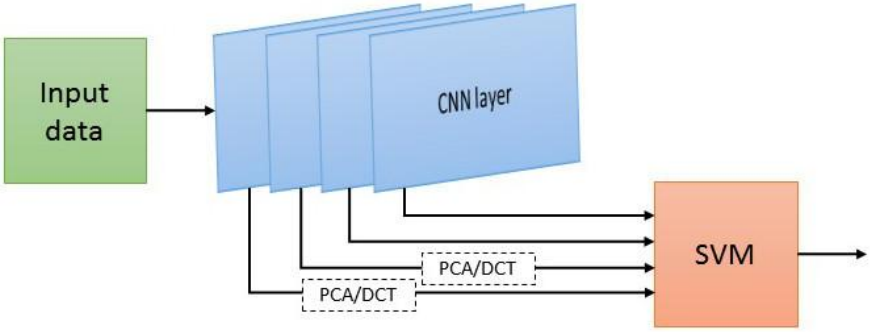


Figure 7.35: The various feature vectors are extracted from different layers and then their dimension is reduced using PCA or DCT. After dimensionality reduction, they are used to train a SVM. Source: [72].

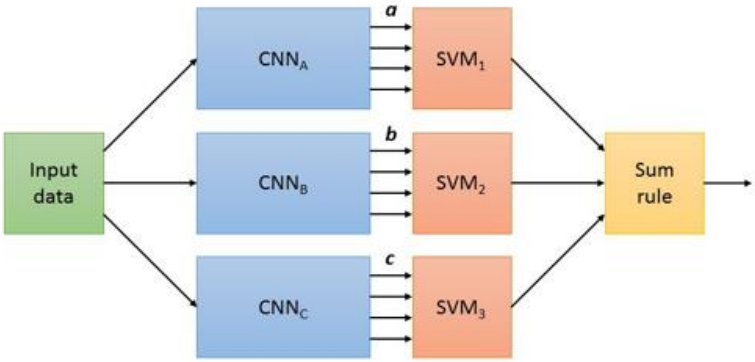


Figure 7.36: A scheme for deep features. The same input is sent to three different CNNs, whose output are processed by three SVMs. The output of the SVMs are finally combined through the sum rule. Source: [72].

representation ($\mathbf{s}$). The autoencoder learns an efficient representation (encoding) for a set of data, typically for dimensionality reduction. The network is commonly trained by having it learn to ignore the noise.

Therefore, autoencoders transform the input into a new representation (called *code* or *latent-space representation*) and then reconstruct the output from this representation. The latent-space representation lives in the *latent-space*, also known as latent feature space or embedding space. It is an embedding of a set of patterns within a manifold, in which patterns resembling each other (e.g. belong to the same class) are positioned closer to one another.

An AE is composed of two parts (also shown in Figure 7.37).

- *Encoder (E)*. It implements the encoding function $E : \mathbb{R}^n \to \mathbb{R}^k$, hence compressing the input ($\mathbf{x}$) and producing the code ($\mathbf{s}$).
- *Decoder (D)*. It implements the decoding function $D : \mathbb{R}^k \to \mathbb{R}^n$, hence reconstructing the input ($\mathbf{o}$) starting from the code ($\mathbf{s}$).

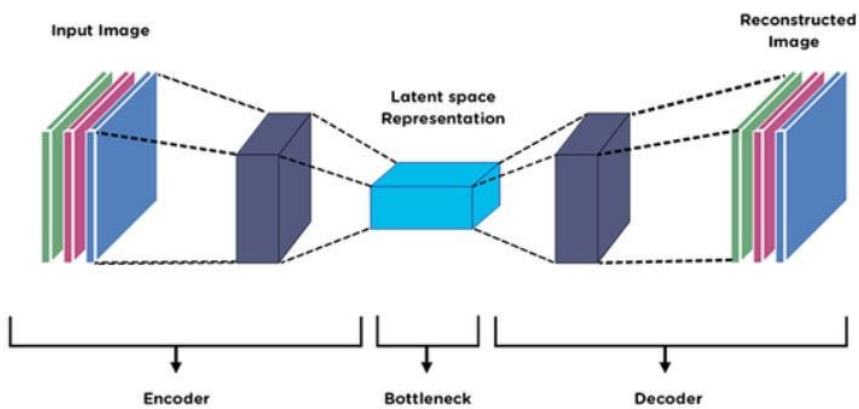


Figure 7.37: Architecture of an autoencoder. Source: Unavailable.

The simplest form of an AE is a feed-forward neural network having an input layer and an output layer with the same number of neurons, and one or more hidden layers connecting them.

An AE can be trained by minimizing the *reconstruction error*, defined as $J(\mathbf{x}, \mathbf{o})$, which measures the difference between the input and its reconstruction. However, if the only purpose of AE was to copy the input to the output, they would be useless. Instead, the hope is that during training the latent representation will take on useful properties. The risk is that the AE could learn the so-called *identity function*, so the output equals the input, and AE does not perform any useful representation learning or dimensionality reduction. So the goal is that position within the latent space can be viewed as being defined by a set of latent variables that emerge from the resemblances from the objects. Basically, the idea is that in latent space it is easier to extract useful information from the training set.

To avoid the risk of learning the identity function, the simplest solution is to use a bottleneck layer which forces a compressed knowledge representation of the original input constraining the amout of information that can traverse the full network, meaning $k < n$.

Another solution is to use *denoising AEs*, that prevent the network from learning the idendity function by corrupting the input data on purpose (adding noise or masking some of the input variables) and making the AE recover the original noise-free data. The schema of these AEs is shown in Figure 7.38. The input corruption is performed only during the training phase. Once the model has learnt the optimal parameters, in order to extract the representations from the original data no corruption is added.

Sparse AEs represent an alternative method to avoid that the model learns the identity function, without a reduction in the number of neurons, by using a *sparsity constraint*. A penalty term is added to the loss function such that only a fraction of the neurons become active:

$$J(\mathbf{x}, \mathbf{o}) + \Omega(\mathbf{h}), \tag{7.137}$$

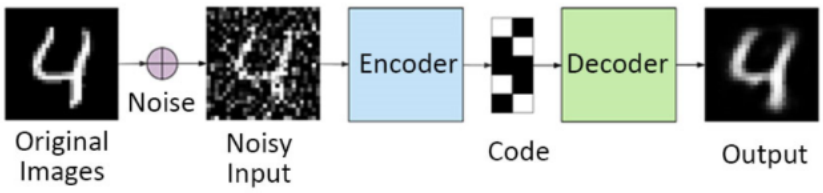


Figure 7.38: Example of denosing AE, that cannot simply copy the input to its output. It is forced to extract useful features that constitute better higher-level representations of the input. Source: [72].

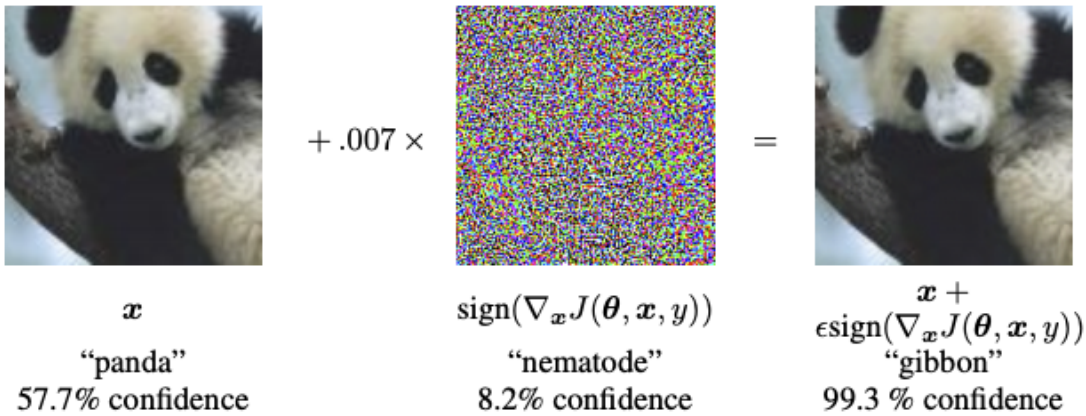


Figure 7.39: Starting with the image of a panda, the attacker adds small perturbations (distortions) to the original image, which results in the model labelling this image as a gibbon, with high confidence. Source: [45].

where $\Omega(\mathbf{h})$ is a penalty function on hidden layer activations $\mathbf{h}$. This forces the AE to represent each input as a combination of small number of neurons and, hence, to discover interesting structure in the data. Because the penalty term is related to the activation of the neurons of the inner layers and the output of the encoder is the latent space. If many inner neurons are not activated, it is easier to have many neurons of the latent space with a value of zero (or close to zero).

7.16 Adversarial examples

Adversarial examples [3] are specialised inputs created with the purpose of confusing a neural network, resulting in the misclassification of a given input. These notorious inputs are indistinguishable to the human eye, but cause the network to fail to identify the contents of the image. One of the most famous examples of an adversarial image shown below is taken from [45] and reported in Figure 7.39.

7.16.1 Generation of an adversarial example

The generation pipeline of an adversarial example from $\mathbf{x} \in \mathbb{R}^m$ with label y is reported in Figure 7.39 and its mathematical model is[102]:

$$\arg\min_{\boldsymbol{\delta}} |\boldsymbol{\delta}|, \text{ such that } f(\mathbf{x} + \boldsymbol{\delta}) \neq y, \ \mathbf{x} + \boldsymbol{\delta} \in [0, 1]^m, \tag{7.138}$$

where $\mathbf{x} + \boldsymbol{\delta}$ is an adversarial example composed of clean samples $\mathbf{x}$ and perturbations $\boldsymbol{\delta}$ and y is the true label of $\mathbf{x}$. f represents the classification model and maps the input image to a label output. The minimizer $\boldsymbol{\delta}$ might not be unique. Let $\boldsymbol{\delta}'$ be the minimizer obtained by solving Equation 7.138. In

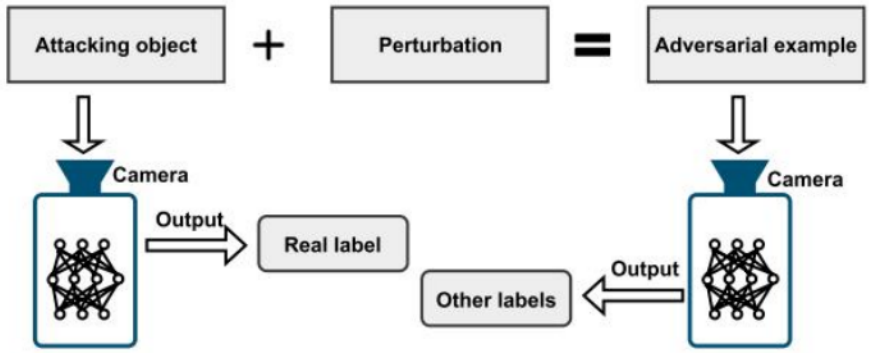


Figure 7.40: Physical world attack. The attack object with added perturbation is captured by the sensor, such as a camera, and then passed into the model to be predicted as other labels instead of the real label. Source: [72].

general, the exact computation of $\boldsymbol{\delta}'$ is a hard problem, so we approximate it by solving [102]:

$$\arg\min_{\boldsymbol{\delta}} J(f(\mathbf{x}+\boldsymbol{\delta}), y') + \text{QualityPenalty}(\boldsymbol{\delta}), \tag{7.139}$$

where J is the loss function, y' is the target label (desired outcome class) and QualityPenalty($\boldsymbol{\delta}$) is usually a kind of norm of $\boldsymbol{\delta}$, resembling l_2 regularization.

7.16.2 Types of attacks

We briefly introduce some type of attacks.

- *Non-target attacks.* The model output is any category other than the real category. In Equation (7.138), we impose $f(\mathbf{x}+\boldsymbol{\delta}) \neq y$.
- *Targeted attacks.* The model not only outputs an error, but the output is a category specified by the attacker. Targeted attacks are harder to achieve than non-targeted attacks. In Equation (7.138), we impose $f(\mathbf{x}+\boldsymbol{\delta}) = y'$ and $y' \neq y$, where y' is the prediction category that the attacker wants the model output.
- *White-box attacks.* The attacker knows the internal structure, weights and training algorithm of the attack model.
- *Black-box attacks.* In contrast to white-box attacks, the attacker knows nothing about the attack model. A type of black-box attack is a *query-based attack*, where the attacker estimates the gradient by querying the model multiple times.

7.16.3 Physical world

In the digital world, the attacker has "digital level" access to the input. For example, the attacker can make arbitrary pixel-level modifications to the input image of the classifier. However, in real applications, attackers cannot control the sensors and data pipes of the system. At the same time, the perturbation must survive in various environmental transformations. This part mainly introduces the challenge of adversarial attacks in the physical world and the mitigation methods proposed by some work. Figure 7.40 shows scheme of a physical world attack, while Figure 7.41 reports an example of attack.

7.17 Distribution shift

Distribution shift means the data in production diverges from the used for training the model. Many failed machine learning deployments can be traced back to this failure. Sometimes models appear to

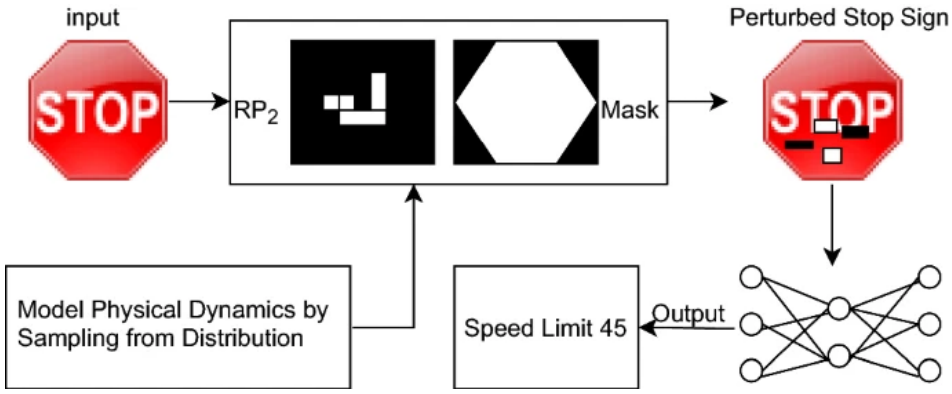


Figure 7.41: Black and white stickers of road signs attack the pipeline. Adversarial examples can mislead the model to recognize the stop sign as a speed limit sign of 45. Source: [72].

perform marvelously as measured by test set accuracy but fail catastrophically in deployment when the distribution of data suddenly shifts. More insidiously, sometimes the very deployment of a model can be the catalyst that perturbs the data distribution. Say, for example, that we trained a model to predict who will repay rather than default on a loan, finding that an applicant's choice of footwear was associated with the risk of default (Oxfords indicate repayment, sneakers indicate default). We might be inclined thereafter to grant a loan to any applicant wearing Oxfords and to deny all applicants wearing sneakers. As soon as we began making decisions based on footwear, customers would catch on and change their behavior.

We assume that our training data was sampled from some distribution $p_S(\mathbf{x}, y)$ but that our test data will consist of unlabeled examples drawn from some different distribution $p_T(\mathbf{x}, y)$. Already, we must confront a sobering reality. Absent any assumptions on how and relate to each other, learning a robust classifier is impossible.

Consider a binary classification problem, where we wish to distinguish between dogs and cats. If the distribution can shift in arbitrary ways, then our setup permits the pathological case in which the distribution over inputs remains constant: $p_S(\mathbf{x}) = p_T(\mathbf{x})$, but the labels are all flipped: $p_S(y|\mathbf{x}) = 1 - p_T(y|\mathbf{x})$. In other words, if in the future all "cats" are now dogs and what we previously called "dogs" are now cats, without any change in the distribution of inputs $p_S(\mathbf{x}) = p_T(\mathbf{x})$, then we cannot possibly distinguish this setting from one in which the distribution did not change at all. Fortunately, under some restricted assumptions on the ways our data might change in the future, principled algorithms can detect shift and sometimes even adapt on the fly, improving on the accuracy of the original classifier.

7.17.1 Covariate shift

Among categories of distribution shift, *covariate shift* may be the most widely studied. Here, we assume that while the distribution of inputs may change over time, the labeling function, i.e., the conditional distribution $p(y|\mathbf{x})$ does not change. It is called covariate shift because the problem arises due to a shift in the distribution of the covariates (features). Formally [34]:

$$p_S(\mathbf{x}) \neq p_T(\mathbf{x}), \tag{7.140}$$

$$p_S(y|\mathbf{x}) = p_T(y|\mathbf{x}). \tag{7.141}$$

While we can sometimes reason about distribution shift without invoking causality, we note that covariate shift is the natural assumption to invoke in settings where we believe that $\mathbf{x}$ causes y.

Consider the challenge of distinguishing cats and dogs. Our training data might consist of images of the kind in Figure 7.42.

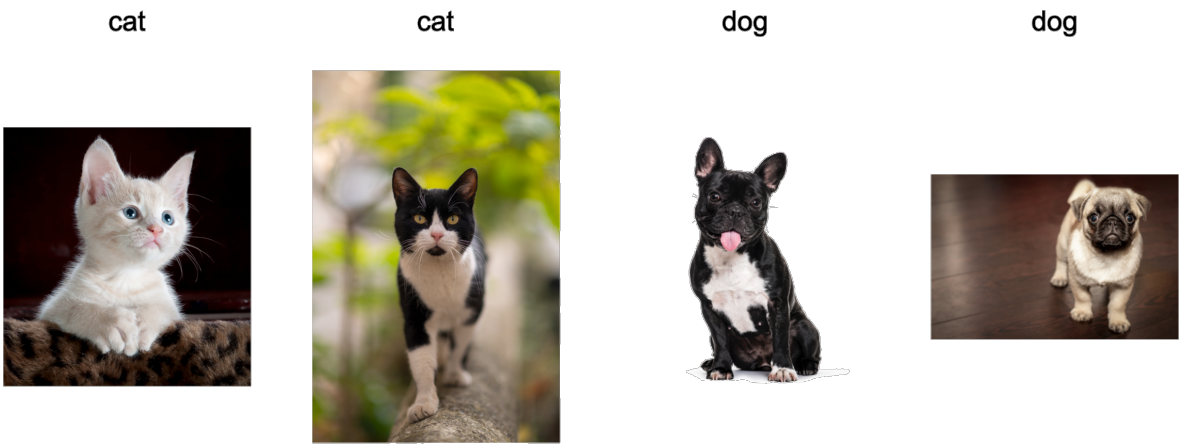


Figure 7.42: Training data for distinguishing cats and dogs. Source: [34].

However, at test time we are asked to classify the images in Figure 7.43.

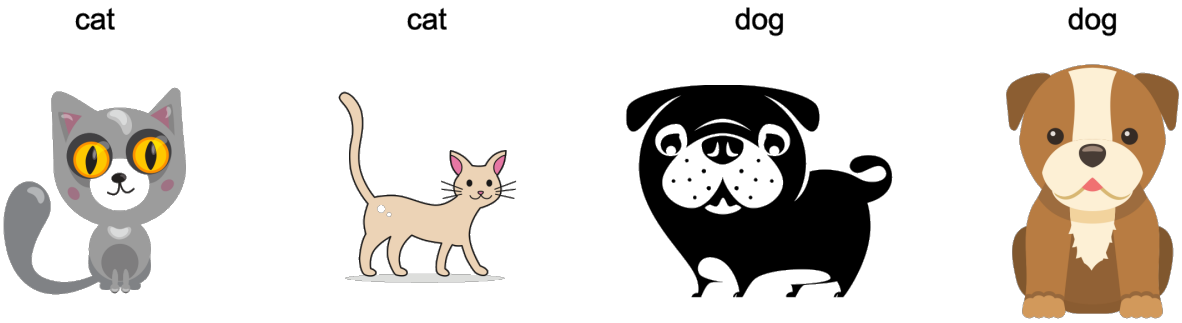


Figure 7.43: Training data for distinguishing cats and dogs. Source: [34].

In other words, the training set consists of photos, while the test set contains only cartoons. Training on a dataset with substantially different characteristics from the test set can spell trouble absent a coherent plan for how to adapt to the new domain.

7.17.2 Label shift

Label shift describes the converse problem. Here, we assume that the label marginal $p(y)$ can change but the class-conditional distribution $p(y|\mathbf{x})$ remains fixed across domains. Formally [34]:

$$p_S(y) \neq p_T(y), \tag{7.142}$$

$$p_S(\mathbf{x}|y) = p_T(\mathbf{x}|y). \tag{7.143}$$

Label shift is a reasonable assumption to make when we believe that y causes $\mathbf{x}$. For example, we may want to predict diagnoses given their symptoms (or other manifestations), even as the relative prevalence of diagnoses are changing over time. Label shift is the appropriate assumption here because diseases cause symptoms. In some degenerate cases the label shift and covariate shift assumptions can hold simultaneously. For example, when the label is deterministic, the covariate shift assumption will be satisfied, even when y causes $\mathbf{x}$. Interestingly, in these cases, it is often advantageous to work with methods that flow from the label shift assumption. That is because these methods tend to involve manipulating objects that look like labels (often low-dimensional), as opposed to objects that look like inputs, which tend to be high-dimensional in deep learning.

7.17.3 Concept shift

Concept shift arises when the very definitions of labels can change. Formally [34]:

$$p_S(y|\mathbf{x}) \neq p_T(y|\mathbf{x}). \tag{7.144}$$

Some classification tasks cannot suffer this kind of shift. As an example, in cat and dog classification, a cat will always be a cat. However, other categories are subject to changes in usage over time. Diagnostic criteria for mental illness, what passes for fashionable, and job titles, are all subject to considerable amounts of concept shift. It turns out that if we navigate around the United States, shifting the source of our data by geography, we will find considerable concept shift regarding the distribution of names for soft drinks.

Chapter 8

Reinforcement Learning

The goal of *reinforcement learning* is to learn optimal behavior from past experiences. An agent performs actions that modify the environment, causing changes from one state to another. When the agent obtains positive results, it receives a reward which, however, may be temporally delayed with respect to the action, or the sequence of actions, which determined it.

8.1 Setting of reinforcement learning

Let $\mathcal{S}$ be a set of states and $\mathcal{A}$ a set of actions. In each state $s_{t-1} \in \mathcal{S}$, the agent chooses and action $a_{t-1} \in \mathcal{A}$ to perform on the enviroment, which gives to the agent a reward r_t. The agent moves to a state s_t.

We define an *episode* or a *game* to be a finite sequence of states, actions and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n. \tag{8.1}$$

A *transition* at time t is the quatern

$$s_t, a_t, r_{t+1}, s_{t+1}. \tag{8.2}$$

In each state s_{t-1} the goal is to choose the optimal action a_{t-1} maximizing the *future reward* R_t

$$R_t = r_t + r_{t+1} + \cdots + r_n = \sum_{i=t}^{n} r_i. \tag{8.3}$$

In many real applications the environment is stochastic, meaning that the same action does not always determines the same sequence of states and rewards. For example, we define the probability of transition from state s_{t-1} to state s_t under the action a_{t-1}. Therefore, if we apply the logic of "better an egg today than a hen tomorrow", reinforcement learning approaches weigh more the rewards temporally close. Therefore we define the *discounted future reward*, denoted R_t the future reward, as:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-t} r_n = \sum_{i=t}^{n} \gamma^{i-t} r_i, \tag{8.4}$$

where r_{t+1} is the reward for transitioning from state s_t to s_{t+1}, $0 \leq \gamma < 1$ is the *discount factor*. γ is less than 1, so rewards in the distant future are weighted less than rewards in the immediate future. Note that, differently from Equation (3.1), we now impose a finite number n of transitions or steps. Equation (8.4) can be defined recursively as:

$$R_t = r_t + \gamma R_{t+1}. \tag{8.5}$$

8.2 Q-learning

Q-learning[1] is one of the best known and most used approaches in reinforcement learning. Let Q be a function

$$Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}. \tag{8.6}$$

$Q(s, a)$ measures the *quality* of the action a when it is applied at state s. The objective is to maximize the discounted future reward:

$$Q(s_t, a_t) = \max R_{t+1}. \tag{8.7}$$

Assuming that Q is known, the optimal policy is the one that chooses $a^\star$ such that:

$$a^\star = \arg\max_a Q(s, a). \tag{8.8}$$

Therefore our goal is to learn (estimate) the Q function (hence the name Q-learning). Given a transition $s_t, a_t, r_{t+1}, s_{t+1}$ we can write:

$$\begin{aligned}
Q(s_t, a_t) &= \max R_{t+1} && \text{by Equation (8.7)} \\
&= \max(r_{t+1} + \gamma R_{t+2}) && \text{by Equation (8.5)} \\
&= r_{t+1} + \gamma \max R_{t+2} && \text{since } r_{t+1}, \gamma \text{ are fixed} \\
&= r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) && \text{by Equation (8.7)}
\end{aligned}$$

We choose the action a_{t+1} with the current policy. In particular, we choose a_{t+1} that maximize Q. In formulas:

$$Q(s_t, a_t) = r_{t+1} + \gamma \max_a Q(s_{t+1}, a), \tag{8.9}$$

which is known as *Bellman's equation*.

The values of $Q(s, a)$, $\forall s \in \mathcal{S}, a \in \mathcal{A}$ are stored in a table [88]. This can be a practical problem, since it may be really big. In particular, it contains $|\mathcal{S}| \cdot |\mathcal{A}|$ elements. The update at each t becomes:

$$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a)], \tag{8.10}$$

where α is the learning rate, and we perform a convex combination of the two terms. If $\alpha = 1$ the update corresponds to Equation (8.9). If $\alpha < 1$, the update goes in the direction suggested by the Bellman equation but with smaller steps. Note that Equation (8.10) can be expressed as

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \tag{8.11}$$

In other way, we are minimizing the following loss function

$$J = \frac{1}{2}(y_t - Q(s_t, a_t))^2 = \frac{1}{2}(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))^2. \tag{8.12}$$

Typical initial values are $\gamma = 0.99$ and $\alpha = 0.5$, which are progressively reduced during training. Algorithm 8.2.1 reports the Q-learning algorithm, according to the implementation of [89].

8.3 Robotic arm example

Consider the example in Figure 8.1, a robotic arm (the agent), with three joints and gripper, must take a ball placed on a pedestal of random position (x, y). The possible states of the agent encode the relative position $(\Delta x, \Delta y)$ of the gripper with respect to the ball and whether the gripper is closed or open. The possible actions are the rotation to right or left of each of three joints ($3 \cdot 2 = 6$ actions) or invert the gripper status (1 action). The possible rewards are:

[1] The matter is far more complex than what described in Section 8.2. For instance, the $Q^\pi(s, a)$ function is rigorously defined as an expectation over the cumulative reward obtained by taking action a in state s and then by following the policy π. For simplicity, take what most of what is written in this Section as granted. To better understand this, we first suggest studying *Markov Decision Processes* (MDPs), policy iteration, value iteration, Monte Carlo policy evaluation, TD learning.

Algorithm 8.2.1: Q-learning algorithm

Data: Learning rate α, discount rate γ, reward r, for each s, a. $maxNumberOfEpisodes$ and number of steps n in each episode.

$Q(s,a) = 0,\ \forall s \in \mathcal{S}, a \in \mathcal{A}$;

for $e \in \{1, \ldots, maxNumberOfEpisodes\}$ **do**

  Select state s_0 randomly;

  for $t \in \{0, 1, \ldots, n\}$ **do**

    $a_t = \arg\max_a Q(s_t, a)$;

    Take action a_t and obtain reward r_{t+1} and state s_{t+1};

    $Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a)]$.;

    if s_{t+1} *is the terminal state* **then**

      Break;

    end

  end

end

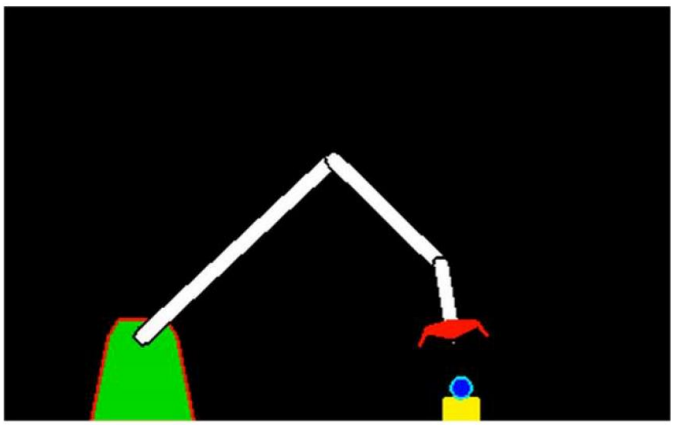


Figure 8.1: Robotic arm example. Source: [72].

- Approach the ball: $r = +1$.
- Move away from the ball: $r = -1$.
- Move the gripper when it is closed: $r = -0.2$.
- Ball capture: $r = +100$.

The task of specifying the rewards for a given environment is often referred to as *reward engineering*.

8.4 Deep Q-Learning

8.4.1 Videogame reinforcement learning

In 2014, researchers from the Deep Mind company (immediately acquired by Google) published the article "Playing Atari with Deep Reinforcement Learning" [70] where reinforcement learning algorithms are successfully used to train a computer to play (at a super-human level) numerous Atari console games. This in itself would not be extraordinary, except for the fact that the state s observed by the agent does not consist of game-specific numerical features (e.g. the position of the spacecraft, its speed), but of simple screen images (raw pixel). Among other things, this allows the same algorithm to learn different games simply by playing. In [70], the input of the model is a set of 4 grayscale images with 86×86 spatial resolution and 8-bit intensity resolution, so the number of states is $256 \times 86 \times 86 \times 4 \sim 1067970$, more than the number of atoms in the known universe. It is impossible to explicitly handle a Q table of this size. For this reason, *Deep Q networks* (DQN) were developed.

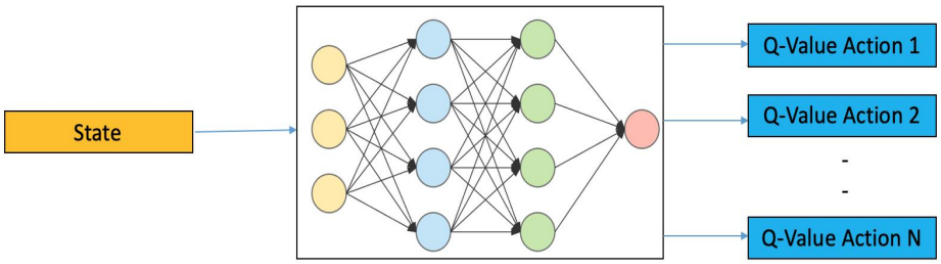


Figure 8.2: Scheme of Deep Q-learning. Given an input s to the network, the output node associated to action a estimates $Q(s, a)$. Source: Unavailable.

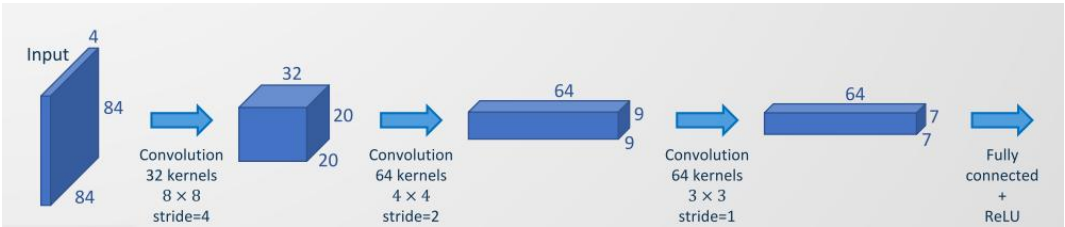


Figure 8.3: The DQN consists in 3 convolutional layers and 2 fully connected layers. Note that there are no pooling layers because they introduce translation invariance, and the network would become insensitive to the location of an object in the image. Source: [72].

8.4.2 Deep Q networks

The main idea negin DQNs consists in approximating the Q function with a CNN which, for each input state, provides an estimation of the quality value for each possible action. In fact, every output node of the CNN represents an action, and its score is the estimated Q-value. This approach is called Deep Q-learning and makes use of DQNs [70], illustrated in Figure 8.2 and, in detail, in Figure 8.3.

The DQN is a fairly standard neural network architecture and could be as simple as a linear network with a couple of hidden layers if your state can be represented via a set of numeric variables. If the state data is represented as images or text, we could employ a regular CNN or RNN architecture.

8.4.3 Experience replay

There is an issue when using a DQN as Q approximator: the transitions are very correlated since they are all extracted from the same episode reducing the overall variance. As a result, the network tends to forget the previous transitions as it overwrites them with new ones, resulting in a network overfitted on the current episode.

To remove correlations and make the DQN training more stable, the *experience replay* technique can be used. During training, all transitions are stored in a *replay memory*. When the network is updated, mini-batches are randomly sampled from the replay memory and used instead of the most recent transition.

8.4.4 ϵ-greedy approach

The approach ϵ-*greedy*, usually selects the action with the highest estimated reward most of the time (that maximizes the Q function), but with a small probability of ϵ we choose to explore new ways, i.e., not to exploit what we have learned so far. In this case, the action is selected randomly, independent of the action-value estimates.

8.4.5 Double DQN

Instead of using one neural network for learning, we use two: a *target network* and a *prediction network*. We use a target network to estimate the target, which has the same architecture as the function

approximator but with frozen parameters. Every C iterations (a hyperparameter), the parameters from the prediction network are copied to the target network.

Chapter 9

Sequence based neural networks

The feedforward networks studied so far (MLP and CNN) operate on input of a predetermined size. For example, the input is an image vector and the output a probability vector of the classes. This case is represented by the first column of Figure 9.1 as a *one to one sequence*. Different applications require that the input and output can be sequences of variable length. The other kinds of networks are:

- *One to many*. Used for image captioning where the input is an image and the output a sentence in natural language that describes it.
- *Many to one*. Used for sentiment analysis where the input is a text (e.g. a product review) and the output a continuous value that expresses the sentiment or judgment (positive or negative).
- *Many to many*. Used for language translation where the input is a sentence in english and the output is its translation in italian.

9.1 Recurrent Neural Networks

Recurrent neural networks or RNNs are a family of neural networks for processing sequential data. Much as a CNN is a neural network that is specialized for processing a grid of values $\mathbf{X}$ such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values $\mathbf{x}_{(1)}, \ldots, \mathbf{x}_{(\tau)}$ [43].

At each time instant t, in addition to the input $\mathbf{x}_{(t)}$, the layer receives the output from the previous time instant $\mathbf{y}_{(t-1)}$ (Figure 9.2). This allows the network to make decisions based on past history or on all the elements of a sequence and on their reciprocal position. Note that the use of the output of a RNN instead of the hidden units resembles a *Jordan network* [93] (however, it is not equivalent).

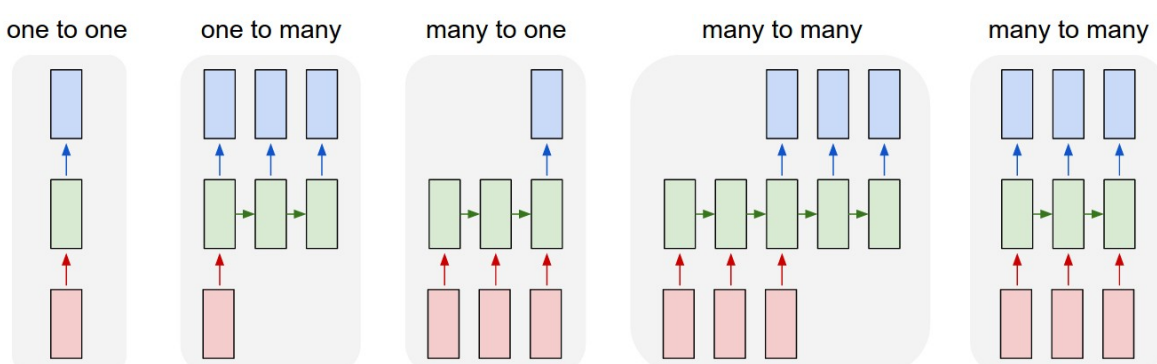


Figure 9.1: Different kinds of input and output sequences. Input vectors are in red, output vectors are in blue and green vectors hold the network state. Source: [72].

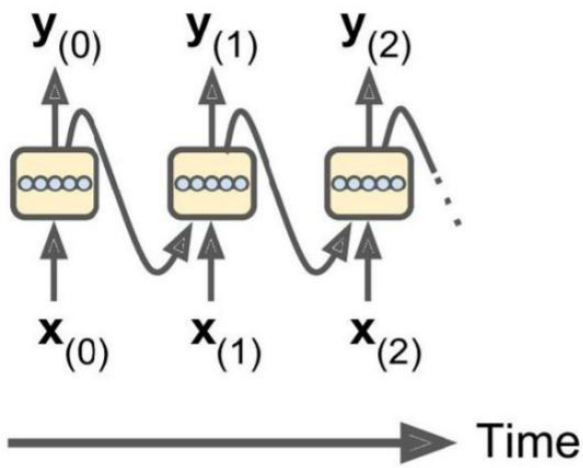


Figure 9.2: RNN with one layer. The output of the network $\mathbf{y}_{(t-1)}$ at time $t-1$ is concatenated with the input $\mathbf{x}_{(t)}$ at time t. Note that there is *one* neural network, which is represented in multiple time frames. Source: [72].

However, we can slightly modify the definition above, such that we use the hidden units $\mathbf{h}_{(t-1)}$ instead of the networks's output $\mathbf{y}_{(t-1)}$. At each time instant t, in addition to the input $\mathbf{x}_{(t)}$, the layer receives the output from the previous time instant $\mathbf{h}_{(t-1)}$ (Figure 9.3). Again, this allows the network to make decisions based on past history or on all the elements of a sequence and on their reciprocal position. The use of the hidden units instead of the network's output resembles a *Elman network* [93] (however, it is not equivalent).

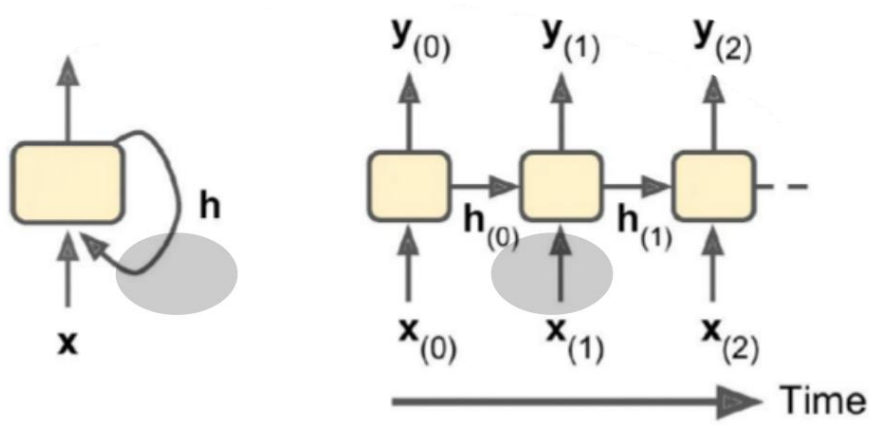


Figure 9.3: Unfolding of a RNN with one layer. The hidden units $\mathbf{h}_{(t-1)}$ at time $t-1$ is concatenated with the input $\mathbf{x}_{(t)}$ at time t. Note that there is *one* neural network, which is represented in multiple time frames. Source: [72].

Consider, for example, the task of processing some sentence. It is not the presence of specific words that is relevant, but also how the words are linked together. On the other hand, the classification of videos is not necessarily based on the interrelationships of the individual frames. In fact, to understand if a video is a documentary on animals, the detection of animals in a few frames is sufficient, so no RNN is required. However, to understand sign language it is necessary to analyze the interrelationships of hand movements in the frames, therefore it is useful to use a RNN.

9.1.1 Unfolding in time

Consider the RNN in Figure 9.3. A *cell* is a part of a RNN that preserves an internal state (or memory) $\mathbf{h}_{(t)}$. It consists of a predetermined number of neurons and can be seen as a layer. Therefore, $\mathbf{h}_{(t)}$ depends on the input $\mathbf{x}_{(t)}$ and on the hidden units of the previous time stamp $\mathbf{h}_{(t-1)}$. In formulas, we have

$$\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)}), \tag{9.1}$$

where f is a function of $\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)}$. More specifically, we can write:

$$\mathbf{h}_{(t)} = \phi(\mathbf{W_x} \cdot \mathbf{x}_{(t)} + \mathbf{W_h} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}), \tag{9.2}$$

where $\mathbf{W_x}$ and $\mathbf{W_h}$ are the weight matrices and $\mathbf{b}$ is the bias vector to be learned, ϕ is the activation function. Note that, like in the original RNNs, the network shares weights over time, which means that the $\mathbf{W_x}$ and $\mathbf{W_h}$ weights are the same for all cells. The decision to share parameters in an RNN was made when any serious computation was a problem (1980s). The trainable parameters of an RNN include the weights and biases of all layers, which are replicated at each time step. In bidirectional networks (see Section 9.3), however, they are not shared in the forward and backward steps. If we are considering the RNN in Figure 9.2, we have

$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)}. \tag{9.3}$$

We can visualize this formulas by *unrolling* over the input sequence the network on the left of Figure 9.3 into the graph on the right of the same figure. This operation is called *unfolding in time*. Unfolding in time is necessary for the training of a RNN. In addition, we have to establish a priori the number of time steps on which we want to perform the analysis. For example, $\mathbf{h}_{(t)}$ could depend on $\mathbf{h}_{(t-1)}, \ldots, \mathbf{h}_{(t-5)}$.

9.1.2 Exploding gradient and gradient clipping

An unfolded RNN on 20 time steps is equivalent to a feedforward DNN with 20 layers. Therefore, training RNNs, that appear relatively simple, can be very expensive and critical for convergence problem. The resulting product of matrices is the source of exploding and vanishing gradients.

Gradient clipping is a technique that tackles exploding gradients. The idea of gradient clipping is very simple: If the gradient gets too large, we rescale it to keep it small. More precisely, if

$$|\nabla J| \geq c, \tag{9.4}$$

then

$$\nabla J \leftarrow c \cdot \frac{\nabla J}{|\nabla J|}, \tag{9.5}$$

where c is a hyperparameter. Since $\frac{\nabla J}{|\nabla J|}$ is a unit vector, after the rescaling ∇J will have norm c. On the other hand, if $|\nabla J| < c$, the gradient remains unchanged.

9.1.3 Deep RNN

In Figure 9.4 we report a deep RNN consisting of many layers. With more layers, more complex data relationships can be learned. If the input are images, each $\mathbf{x}_{(t)}$ generally does not correspond to pixels, but to high-level features extracted from a CNN or a feature extractor. Or again, if the inputs are words, each $\mathbf{x}_{(t)}$ could be the result of *word embedding*.

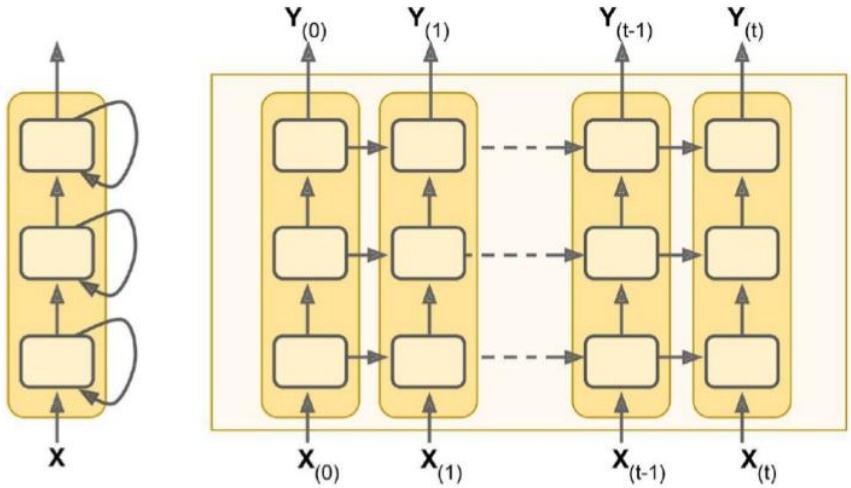


Figure 9.4: Unfolding of a deep RNN. Source: [72].

9.2 Long Short-Term Memory

Long Short-Term Memory (LSTM) is a type of RNN. A cell at time t of this network contains three gates:

- Input gate i_t, which outputs a vector $\mathbf{i}_t$.
- Output gate o_t, which outputs a vector $\mathbf{o}_t$.
- Forget gate f_t, which outputs a vector $\mathbf{f}_t$.

Each one of these gates consist of one sigmoid activation function. A cell also contains a single layer network *candidate* $\tilde{\mathbf{C}}_t$, which has a tanh activation function. Each cell has 4 state vectors:

- Memory state $\mathbf{C}_t$.
- Previous memory state $\mathbf{C}_{t-1}$.
- Hidden state $\mathbf{h}_t$.
- Previous hidden state $\mathbf{h}_{t-1}$.

Note that we indicate the two memory state vectors with capital letters, even if they are not matrices. This is because most sources refers to them with this notation.

Let $\mathbf{x}_t$ be the input vector to the cell, $\mathbf{h}_{t-1}$ be the previous hidden state and $\mathbf{C}_{t-1}$ be the previous memory state. Let $\mathbf{U}$, $\mathbf{W}$ (weight matrices) and $\mathbf{b}$ (bias vector) be the learnable parameters of the cell, independent from t. The LSTM cell computes $\mathbf{C}_t$ and $\mathbf{h}_t$ with the following procedure. We compute the gates' values in the following way:

$$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{x}_t + \mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{b}_f), \tag{9.6}$$

$$\mathbf{i}_t = \sigma(\mathbf{U}_i \mathbf{x}_t + \mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{b}_i), \tag{9.7}$$

$$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{x}_t + \mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{b}_o), \tag{9.8}$$

where σ is the sigmoid activation function. The argument of each gate is an affine transformation of the vector $[\mathbf{x}_t, \mathbf{h}_t]^T$. The candidate $\tilde{\mathbf{C}}_t$ is computed as:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{U}_c \mathbf{x}_t + \mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{b}_c), \tag{9.9}$$

and the memory state $\mathbf{C}_t$ is:

$$\mathbf{C}_t = \mathbf{f}_t * \mathbf{C}_{t-1} + \mathbf{i}_t * \tilde{\mathbf{C}}_t, \tag{9.10}$$

where $*$ is the element-wise multiplication. The output is:

$$\mathbf{h}_t = \mathbf{o}_t * \tanh(\mathbf{C}_t). \tag{9.11}$$

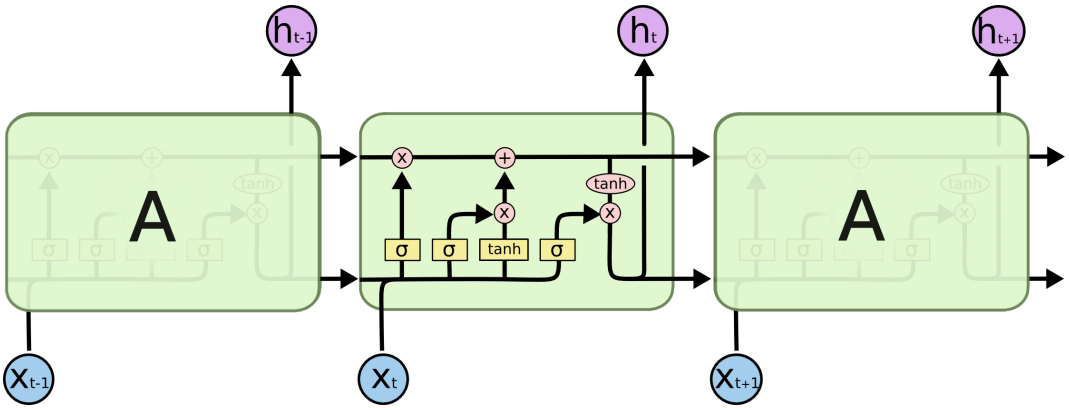


Figure 9.5: LSTM cell. Source: [112].

If $\mathbf{f}_t$ has values near 1, then the cell must "remember" $\mathbf{C}_{t-1}$. On the other hand, if it has values near 0, it can forget about it. This intuition is represented by Equation (9.10). Conversely, $\mathbf{i}_t$ represents how much the candidate memory $\tilde{\mathbf{C}}_t$ will impact the new memory state $\mathbf{C}_t$.

9.3 Bidirectional Long Short-Term Memory

Bidirectional Long Short-Term Memory (Bi-LSTM) is a type of RNN that processes sequential data in both forward and backward directions. It combines the power of LSTM with bidirectional processing, allowing the model to capture both past and future context of the input sequence. A scheme of a Bi-LSTM is reported in Figure 9.6.

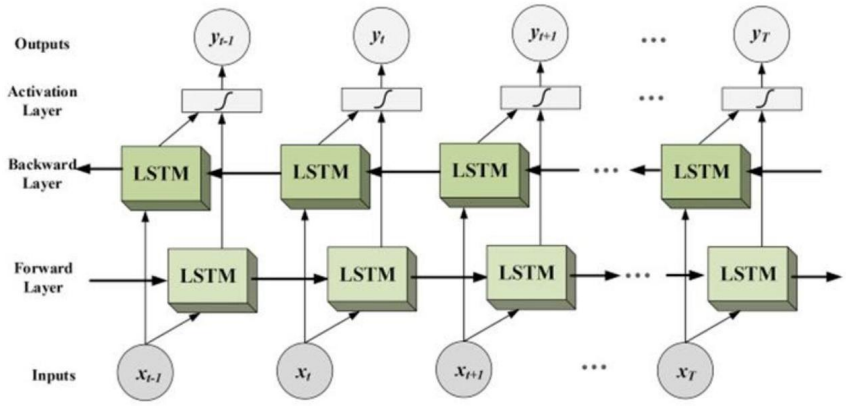


Figure 9.6: Scheme of a Bi-LSTM. Before the activation layer, we can combine the outputs from both LSTM layers in several ways, such as average, sum, multiplication, or concatenation. Source: [72].

With the regular LSTM we can make input flow in one direction, either backwards or forward. However, in Bi-LSTM, we can make the input flow in both directions to preserve the future and the past information. Consider the following example. In the sentence

The boys go to $\cdots$

we cannot fill the rightmost blank spaces. However, if the sentence is of the kind

The boys go to $\cdots$ boys come out of school.

we can easily predict the past blank space.

9.4 Gated Recurrent Unit

Gated Recurrent Unit (GRU) is a simplified version of LSTM, which tries to maintain its advantages, reducing parameters and complexity.

The basic component of a GRU cell are:

- Reset gate r_t, which outputs a vector $\mathbf{r}_t$. It determines how much old information to forget.
- Update gate z_t, which outputs a vector $\mathbf{z}_t$. It determines what information should forget and what information should pass the output. It combines the forget and input gates of LSTM [112].

Each one of these gates consist of one sigmoid activation function. A cell also contains a single layer network *candidate* $\tilde{\mathbf{h}}_t$, which has a tanh activation function. Each cell has 2 state vectors:

- Hidden state $\mathbf{h}_t$.
- Previous hidden state $\mathbf{h}_{t-1}$.

The GRU cell computes $\mathbf{h}_t$ with the following procedure. We compute the gates' values in the following way:

$$\mathbf{z}_t = \sigma(\mathbf{U}_z\mathbf{x}_t + \mathbf{W}_z\mathbf{h}_{t-1} + \mathbf{b}_z), \tag{9.12}$$

$$\mathbf{r}_t = \sigma(\mathbf{U}_r\mathbf{x}_t + \mathbf{W}_r\mathbf{h}_{t-1} + \mathbf{b}_r), \tag{9.13}$$

where σ is the sigmoid activation function. The candidate $\tilde{\mathbf{h}}_t$ is computed as:

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{U}_h\mathbf{x}_t + \mathbf{W}_h(\mathbf{r}_t * \mathbf{h}_{t-1}) + \mathbf{b}_h), \tag{9.14}$$

where * is the element-wise multiplication. The output is:

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{z}_t) * \mathbf{h}_{t-1} + \mathbf{z}_t * \tilde{\mathbf{h}}_t, \tag{9.15}$$

where $\mathbf{1}$ is a vector of only ones. The update gate vector $\mathbf{z}_t$ measures the amount of old and new information to keep.

9.5 Temporal Convolutional Network

Temporal convolutional network (TCN) is a kind of network for many to many modeling. The general TCN architecture consists of multiple residual blocks (Figure 9.7) each containing two sets of *dilated causal convolution layers* with the same *dilation factor*, followed by batch normalization, ReLU and spatial dropout layers.

9.5.1 Dilated causal convolution layer

The main building block of a TCN is a *dilated causal convolutional layer*, which operates over the time steps of each sequence. In this context, "causal" means that the activation computed for a particular time step cannot depend on activations from future time steps.

We now define the dilated convolution for a sequence of real values as proposed in [8]. Let $\mathbf{x} = (x_1, \ldots, x_n)$ be a sequence, $x_i \in \mathbb{R}$ and let $\mathbf{w} = \{w_0, \ldots, w_{f-1}\}$ be a filter (w_i is a weight of the filter). Then the dilated convolution on element s of the sequence is

$$\sum_{i=0}^{f-1} w_i \cdot x_{s-d\cdot i}, \tag{9.16}$$

where d is the dilation factor, f is the filter size, and $s-d\cdot i$ accounts for the direction of the past. Figure 9.9 shows two numerical examples of dilated convolution. Dilation is thus equivalent to introducing a fixed step between every two adjacent filter taps. When $d = 1$, a dilated convolution reduces to a regular convolution. Using larger dilation enables an output at the top level to represent a wider range of inputs, thus effectively expanding the receptive field of a CNN. Figure 9.8 shows a graphical representation of dilated convolution.

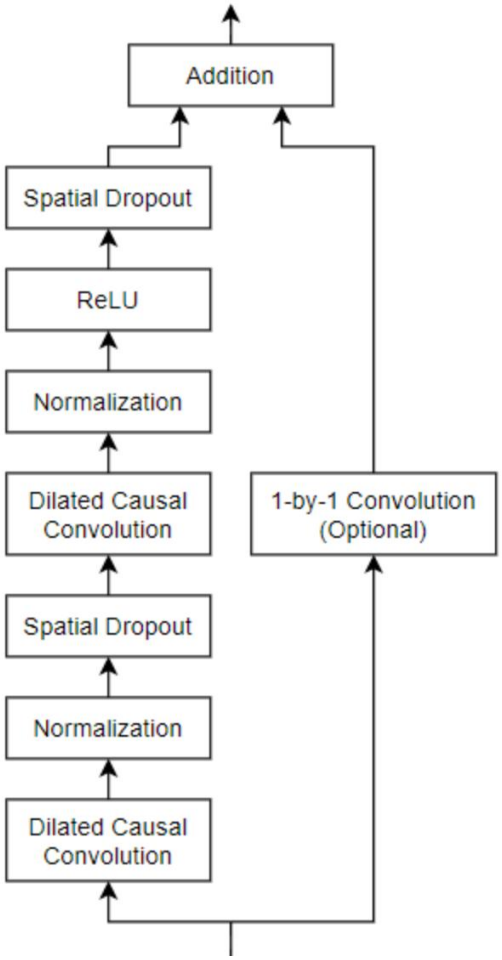


Figure 9.7: Residual block of a TCN. Source: [72].

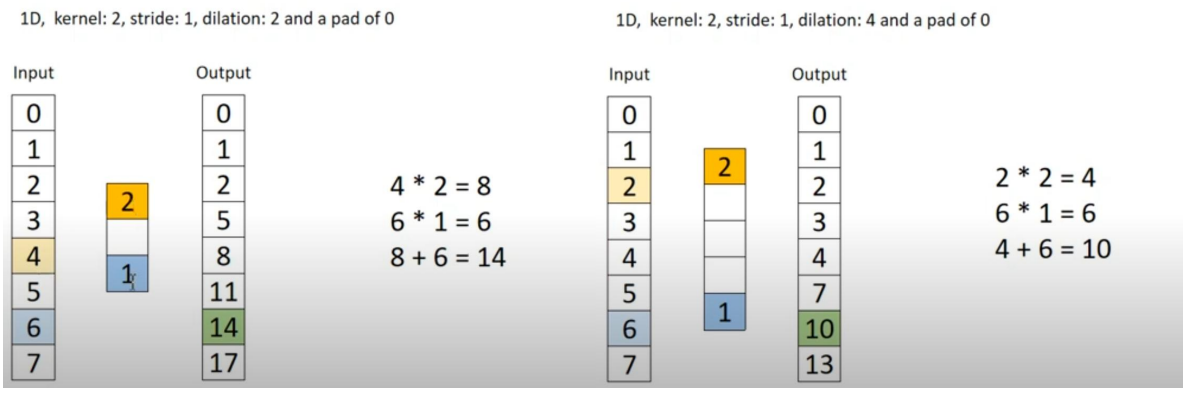


Figure 9.9: Numerical examples of dilated convolution. Source: [72].

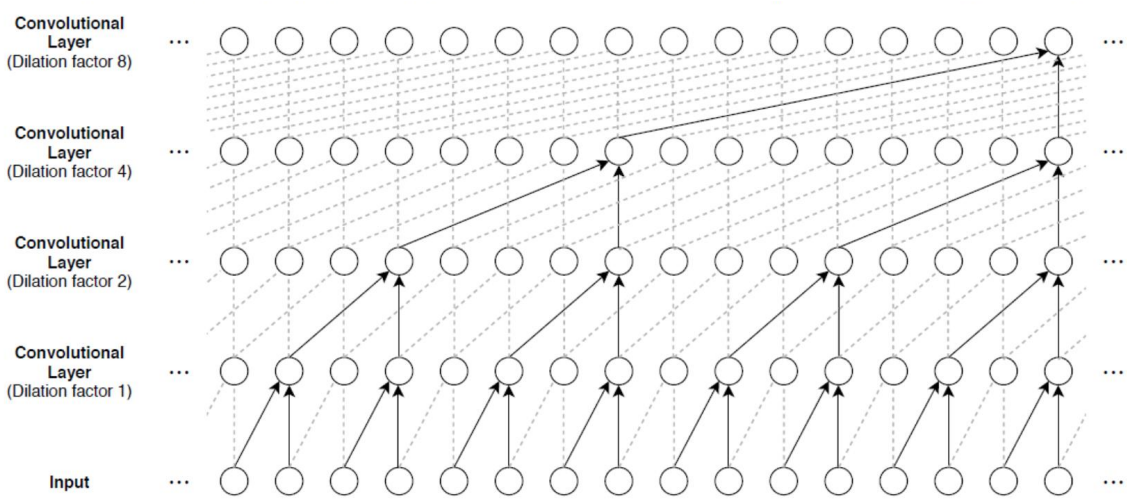


Figure 9.8: Four dilated convolutional layer, with varying d for each layer and $f = 2$ for each layer. Each circle in the input row correspond to one x_i. Each arrow corresponds to a filter weight or component w_i. Each vertical rows represent each time step t. Source: [72].

Note that it is possible to extend the definition for vectors or matrices.

To build up context from previous time steps, multiple convolutional layers are typically stacked on top of each other. To achieve large receptive field sizes, the dilation factor of subsequent convolution layers is increased exponentially as shown in the Figure 9.8. Assuming that the dilation factor of the k-th convolutional layer is 2^{k-1} and the stride is 1, then the *receptive field size* R of such a network can be computed as

$$R = (f - 1)(2^k - 1) + 1, \tag{9.17}$$

where f is the fiter size and k is the number of convolutional layers. Change the filter size and number of layers to easily adjust the receptive field size and the number of learnable parameters as necessary for the data and task at hand.

9.5.2 Disadvantages

One of the disadvantages of TCNs compared to RNNs is that they have a larger memory footprint during inference. In fact, the entire raw sequence is required to compute the next time step. To reduce inference time and memory consumption, especially for step-ahead predictions, train with the smallest sensible receptive field size R and perform prediction only with the last R time steps of the input sequence.

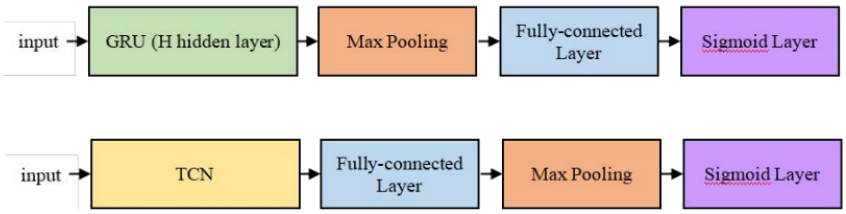


Figure 9.10: Two examples of multilabel recurrent DNNs. Source: [72].

9.6 Recurrent DNNs for multilabel classification

We create a DNN architecture based on GRU and TCN both adapted to a multilabel classification problem. The base schema of the model is a GRU with N hidden units (e.g. number N of hidden units is set to 50), followed by a max pooling layer and a fully connected layer. The output layer is a sigmoid that provides multiclass classification: given a set of output nodes, it computes the sigmoid for each one of them. The TCN base approach has a similar architecture but with max pooling following the fully connected layer. The loss function for a single sample $\mathbf{x}$ is the Binary Cross-Entropy Loss of Equation (4.42).

9.7 Time series

A time series can be related to the trend over time, for example a stock exchange, the temperature of an environment, the energy consumption of a system, etc. We can consider a time series as a function $f(t)$ sampled in several time instants. Knowing the values up to instant t, we are interested in predicting the value at $t+1$. Note that the input is not only one-dimensional, but could be constituted also by vectors or matrices.

Chapter 10

Generative Adversarial Networks

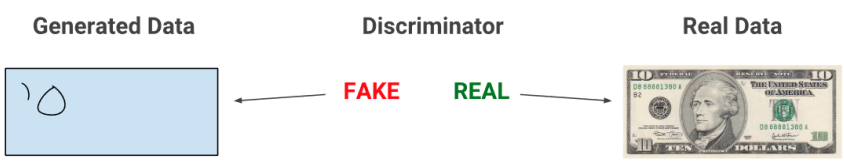
Generative models are a class of machine learning models that can generate new data that is similar to the data they were trained on. These models are trained on a dataset and learn to understand the underlying patterns and structure of the data. Once trained, they can generate new data that is similar to the training data. There are many types of generative models, such as *Generative Adversarial Networks* (GANs), *Variational Autoencoders* (VAEs), and normalizing flow models. These models can be used to generate a wide variety of data, including images, text, and audio.

10.1 Overview

A GAN is composed of two parts:

- The *generator* learns to generate plausible data. The generated instances become negative training examples for the discriminator.
- The *discriminator* learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.

Consider the following example from [82]. When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake:



As training progresses, the generator gets closer to producing output that can fool the discriminator:

Finally, if generator training goes well, the discriminator gets worse at telling the difference between real and fake. It starts to classify fake data as real, and its accuracy decreases. A picture of the whole system is reported in Figure 10.1. Both the generator and the discriminator are neural networks.

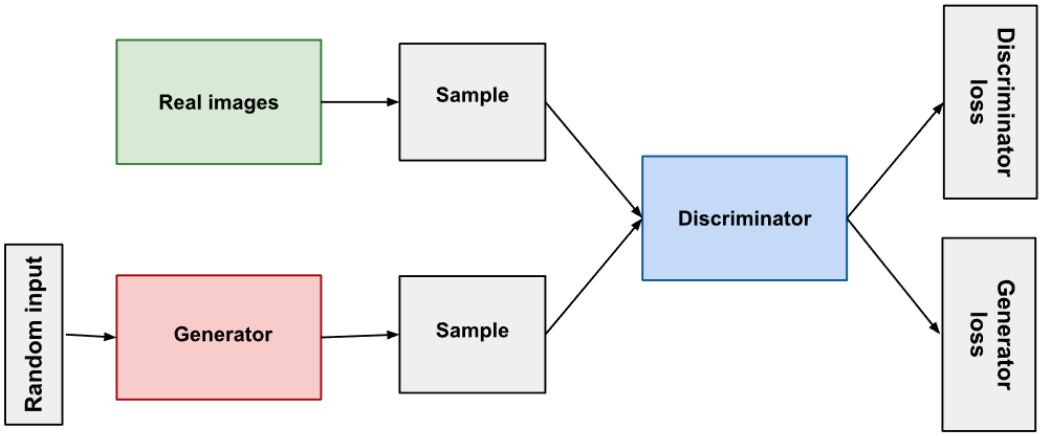


Figure 10.1: Overview of a GAN. Source: [82].

The generator output is connected directly to the discriminator input. Through backpropagation, the discriminator's classification provides a signal that the generator uses to update its weights.

10.2 Distribution estimation

We firstly consider *Generative Matching Networks* (GMNs) instead of GANs, a simplified version of GANs based only on a generator. Suppose that we are interested in generating black and white square images of dogs with a size of $n \times n$ pixels. We can reshape each data as a $N = n \times n$-dimensional vector (by stacking columns on top of each others) such that an image of dog can then be represented by a vector. However, it does not mean that all vectors represent a dog once shaped back to a square. So, we can say that the N-dimensional vectors that effectively give something that look like a dog are distributed according to a very specific *probability distribution* over the entire N-dimensional vector space (some points of that space are very likely to represent dogs where as it is highly unlikely for some others). In the same spirit, there exists, over this N-dimensional vector space, probability distributions for images of cats, birds and so on.

Then, the problem of generating a new image of dog is equivalent to the problem of generating a new vector following the "dog probability distribution" over the N-dimensional vector space. So we are, in fact, facing two problems:

- The problem of generating a random vector image that looks like dog and others that does not.
- We obviously do not know how to express explicitly this distribution.

Both previous points make the process of generating random variables from this distribution pretty difficult. Let us then try to tackle these two problems in the following.

10.2.1 Random vector generation

Our first problem when trying to generate our new image of dog is that the "dog probability distribution" over the N-dimensional vector space is a very complex one and we do not know how to directly generate complex random variables. As most of the time in these cases, very complex function naturally implies neural network modeling. Then, the idea is to model the *transform function* by a neural network that takes as input a simple N-dimensional uniform random vector and that returns as output another N-dimensional vector that should follow, after the training, the "dog probability distribution". An illustration of this procedure is reported in Figure 10.2. This is the main idea of GMNs.

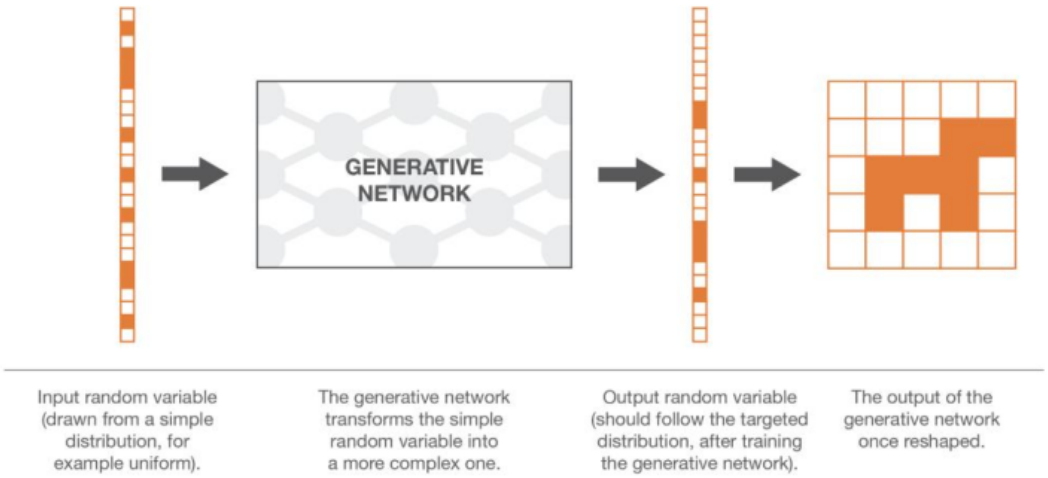


Figure 10.2: Illustration of the notion of generative models using neural networks. Obviously, the dimensionality we are really talking about are much higher than represented here. Source: [72].

10.2.2 Comparing two probability distributions based on samples

As mentioned above, the idea of GMNs is to train the generative network by directly comparing the generated distribution to the true one. However, we do not know how to express explicitly the true "dog probability distribution". In addition, the generated distribution is far too complex to be expressed explicitly. So, comparison based on explicit expressions are not possible.

However, if we have a way to compare probability distributions based on samples, we can use it to train the network. Indeed, we have a sample of true data and we can, at each iteration of the training process, produce a sample of generated data. A popular distance measure based on samples is the *Maximum Mean Discrepancy* (MMD).

10.2.3 Direct training method

Given a random vector with uniform probability distribution as input, we want the probability distribution of the generated output to be the "dog probability distribution". The idea of GMNs is to optimize the network as described in Algorithm 10.2.1. The training procedure of a GMN is reported

Algorithm 10.2.1: GMNs training

for $t \in \{1, 2, \dots\}$ **do**
  Generate some random vectors with uniform probability distribution;
  Make these inputs go through the generator network and collect the generated outputs;
  Compare the true probability distribution and the generated one based on the available samples (e.g. using MMD);
  Use backpropagation to make one step of GD to lower the distance (e.g. MMD) between true and generated distributions;
end

in Figure 10.3. This is called *direct training* as we directly compare the generated distribution to the true one when training the generative network.

10.2.4 Indirect training method

Now we consider GANs again. In fact, the idea that rules GANs consists in replacing this direct comparison by an *indirect* one that takes the form of a *downstream task* over the two distributions. The training of the GAN is done with respect to this task such that it forces the generated distribution to get closer and closer to the true distribution.

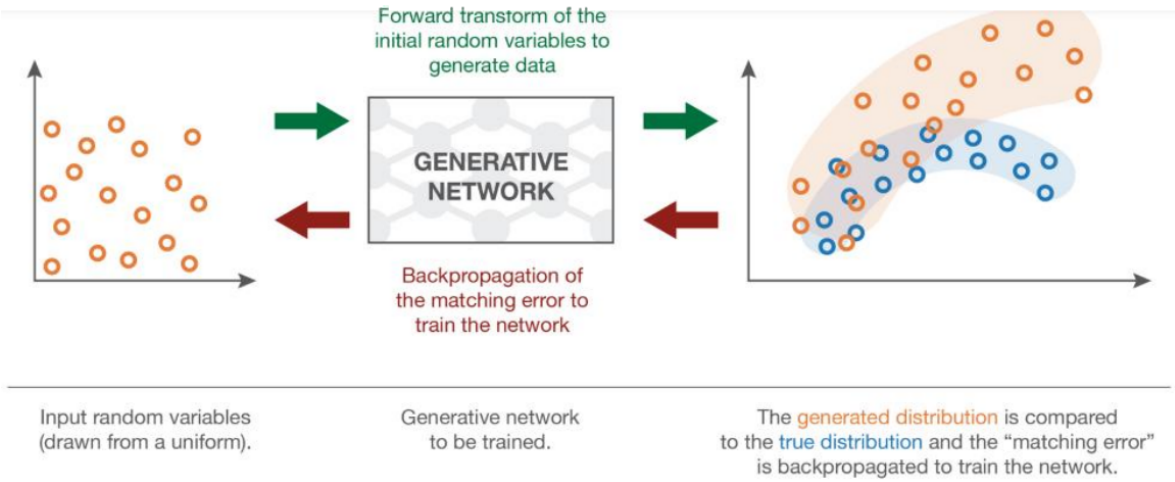


Figure 10.3: Training of a GMN. GMNs take simple random inputs, generate new data, directly compare the distribution of the generated data to the distribution of the true data and backpropagate the matching error to train the network. Source: [72].

The downstream task of GANs is a *discrimination* task between true and generated samples. Informally, we want the discriminator to fail as much as possible. So we have a *discriminator* that takes sample of true and generated data and that tries to classify them as well as possible, while the generator is trained to fool the discriminator as much as possible.

10.2.5 The ideal case: perfect generator and discriminator

In order to better understand why training a generator to fool a discriminator will lead to the same result as training directly the generator to match the target distribution, lets take a simple one dimensional example. We forget, for the moment, how both generator and discriminator are represented and consider them as abstract notions. Moreover, both are supposed "perfect" (with infinite capacities) in the sense that they are not constrained by any kind of parametric model.

Suppose that we have a true distribution, for example a one-dimensional gaussian, and that we want a generator that samples from this probability distribution. The *direct* training method would then consist in adjusting iteratively the generator (gradient descent iterations) to correct the measured difference or error between true and generated distributions, Finally, assuming the optimization process perfect, we should end up with the generated distribution that matches exactly the true distribution. An illustration of this procedure is reported in Figure 10.4

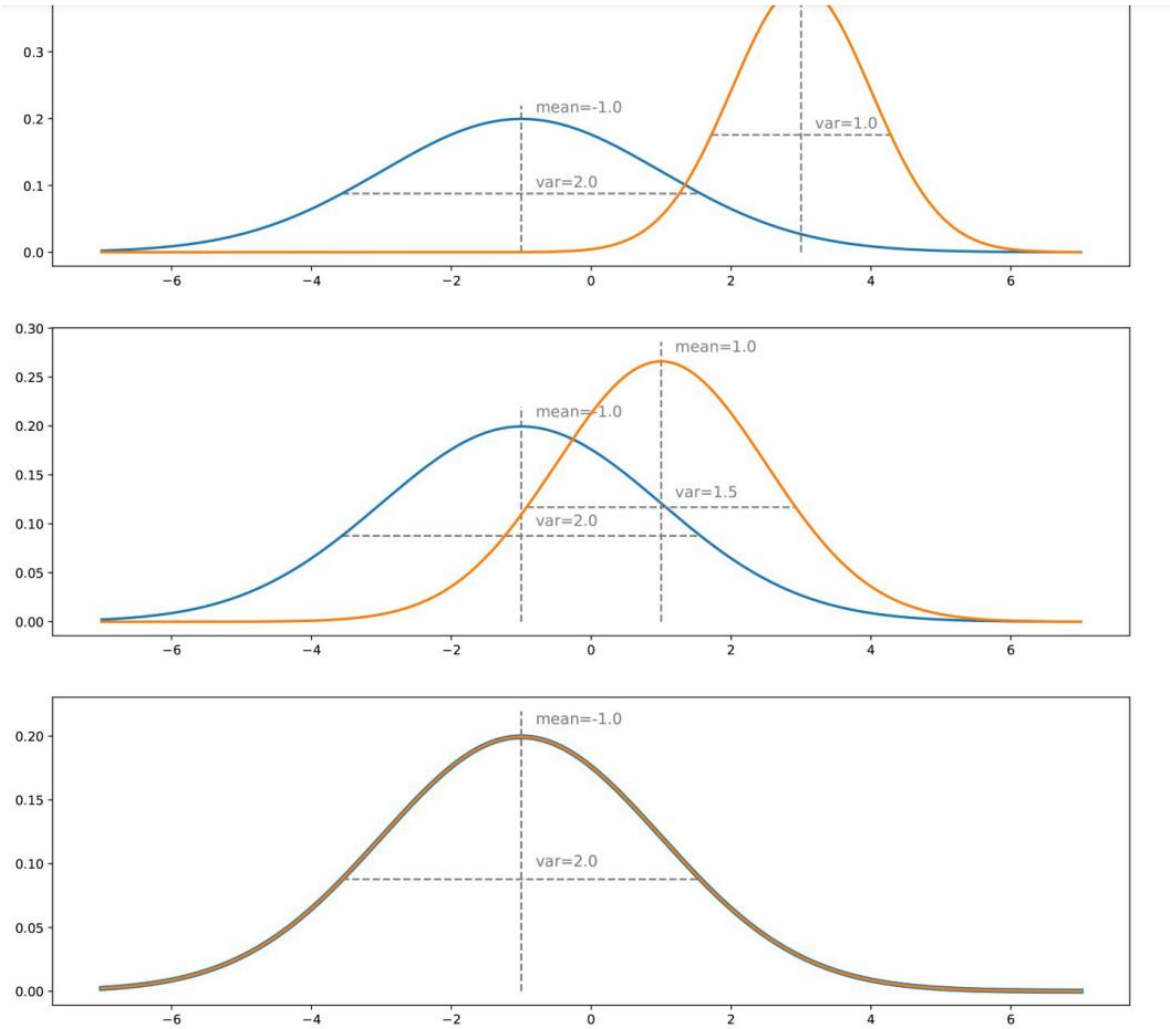


Figure 10.4: Illustration of the concept of direct matching method. The distribution in blue is the true one while the generated distribution is depicted in orange. Iteration by iteration, we compare the two distributions and adjust the networks weights through GD steps. Here the comparison is done over the mean and the variance (similar to a truncated moments matching method). Note that this example is so simple that it does not require an iterative approach: the purpose is only to illustrate the intuition given above. Source: [72].

For the *indirect* approach, we have to consider also a discriminator. We assume for now that this discriminator is a kind of oracle that knows exactly what are the true and generated distribution and that is able, based on this information, to predict a class, "true" or "generated", for any given point. If the two distributions are far apart, the discriminator will be able to classify easily and with a high level of confidence most of the samples we present to it. If we want to fool the discriminator, we have to bring the generated distribution close to the true one. The discriminator will have the most difficulty to predict the class when the two distributions will be equal in all points: in this case, for each point there are equal chances for it to be "true" or "generated". In this case, the discriminator cannot do better than being true in one case out of two in average. An illustration of this procedure is reported in Figure 10.5

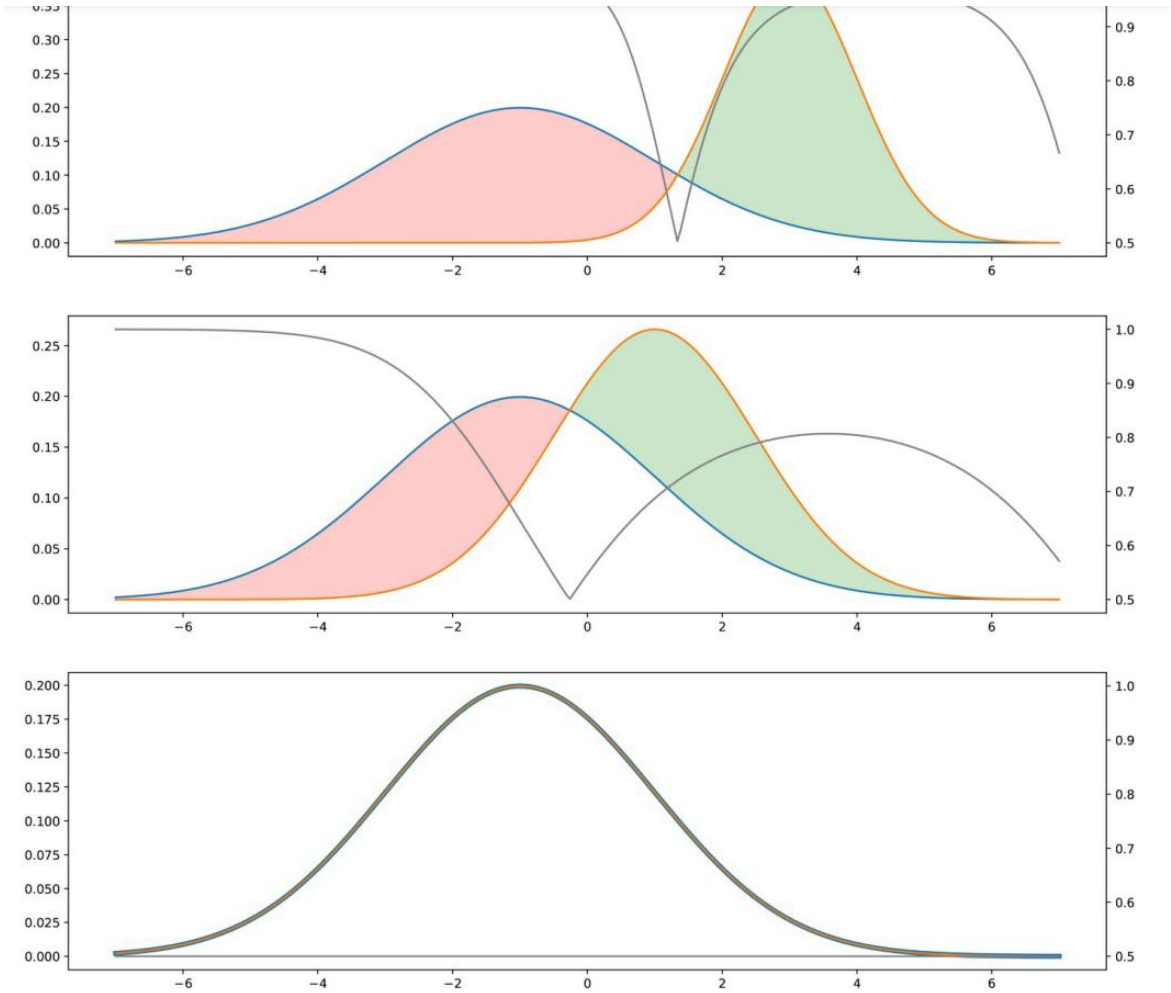


Figure 10.5: Illustration of the concept of indirect or adversarial method. The blue distribution $t(x)$ is the true one, while the orange $g(x)$ is the generated one. Given a point x, the discriminator chooses "true" if $t(x) > g(x)$, and it chooses "false" if $t(x) < g(x)$ (assuming "true" and "generated" data are equal in proportions). In other words, it chooses the class with largest density in that point. Following this policy, in grey, with corresponding y-axis on the right, we display the probability of the discriminator of being right. The closer the two distributions are, the more often the discriminator is wrong. When training, the goal is to "move the green area" towards the red area. In the bottom-most figure, the grey line is always 0.5, as the two distributions are completely overlapping, so each sample has the same probability for both distributions. Source: [72].

At this point, it seems legit to wonder whether this indirect method is really a good idea. Indeed, it seems to be more complicated (we have to optimise the generator based on a downstream task instead of directly based on the distributions) and it requires a discriminator that we consider here as a given oracle but that is, in reality, neither known nor perfect. For the first point, the difficulty of directly comparing two probability distributions based on samples counterbalances the apparent higher complexity of indirect method. For the second point, it is obvious that the discriminator is not known. However, it can be learned.

10.2.6 The approximation: adversarial neural networks

We now describe the specific form that take the generator and the discriminator in the GANs architecture. The generator is a neural network that models a transform function. It takes as input a simple random variable and must return, once trained, a random variable that follows the targeted distribution. As it is very complicated and unknown, we decide to model the discriminator with another

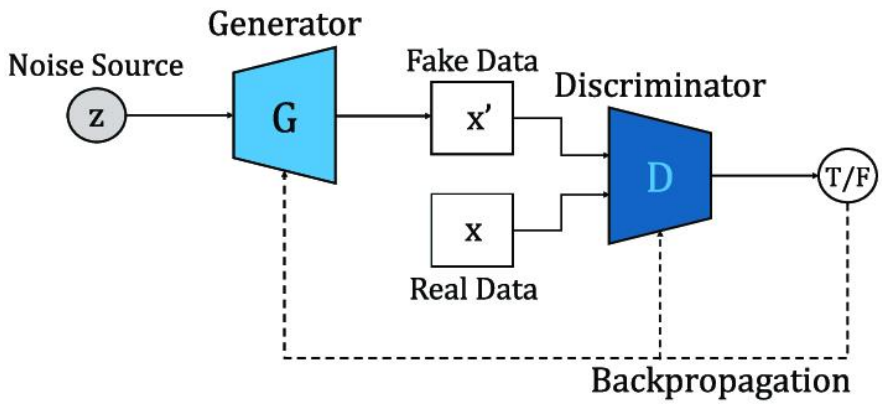


Figure 10.6: Structure of a GAN. Given a noise sample $\mathbf{z}$, the generator generates a fake sample $\mathbf{x}'$. The discriminator tries to classify $\mathbf{x}'$ based on real data $\mathbf{x}$. The weights of both networks are updated through backpropagation. Source: [72].

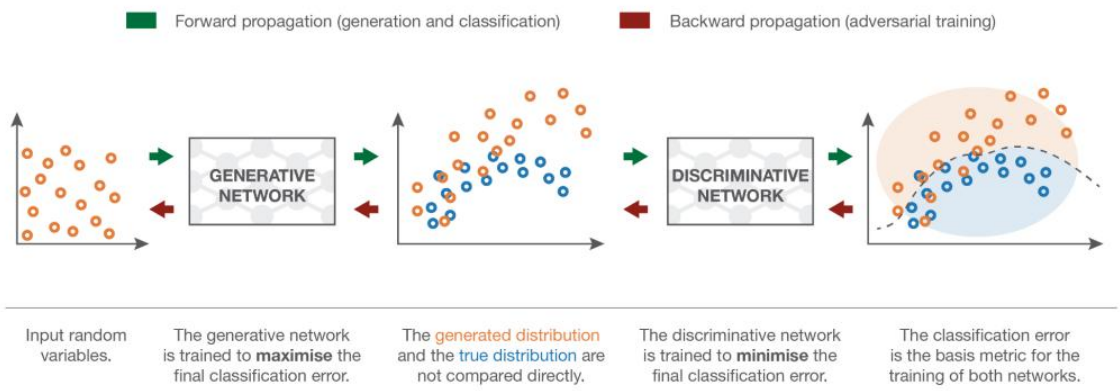


Figure 10.7: Training scheme of a GAN. Source: [72].

neural network. This neural network models a discriminative function. It takes as input a point (in our dog example a N-dimensional vector) and returns as output the probability of this point to be a "true" one.

Note that the fact that we impose now a parametric model to express both the generator and the discriminator, instead of the idealized versions of Section 10.2.5 has, in practice, not a huge impact on the theoretical argument given above:

- The goal of the generator is to fool the discriminator, so the generative neural network is trained to maximize the final classification error (between true and generated data).
- The goal of the discriminator is to detect fake generated data, so the discriminative neural network is trained to minimize the final classification error.

So, at each iteration of the training process:

- The weights of the generative network are updated in order to increase the classification error (error gradient ascent over the generator's parameters).
- The weights of the discriminative network are updated so that to decrease this error (error gradient descent over the discriminator's parameters).

The structure of a GAN is reported in Figure 10.6, while a scheme of its training is shown in Figure 10.7. The opposite goals and the implied notion of adversarial training of the two networks explains the name of *adversarial networks*: both networks try to beat each other and, by doing so, they are both becoming better and better. The competition between them makes these two networks progress with respect their respective goals.

10.3 Mathematical details

Let G be the generating MLP with parameters $\boldsymbol{\theta}_G$, where $G(\mathbf{z})$ is a mapping from the noise space to data space. Let D be the discriminating MLP with parameters $\boldsymbol{\theta}_D$, where $D(\mathbf{x})$ is a mapping from the data space to a scalar value, representing the probability that $\mathbf{x}$ comes from the real data. Let $p_{\text{data}}(\mathbf{x})$ be the data generating distribution over data $\mathbf{x}$, $p_{\mathbf{z}}(\mathbf{z})$ be the noise generating distribution and $p_g(\mathbf{x})$ be the distribution of the vectors produced by the generator network of the GAN [24] (i.e. it consists in the step after the noise generation $\mathbf{x} = G(\mathbf{z})$).

We train D to maximize the probability of assigning the correct label to both training examples and samples from G:

$$\mathop{\mathbb{E}}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x})] + \mathop{\mathbb{E}}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z}))]. \tag{10.1}$$

Simultaneously, we train G to minimize (fool G):

$$\mathop{\mathbb{E}}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z}))]. \tag{10.2}$$

Therefore the goal of a GAN is to train G and D solving

$$\min_G \max_D V(D, G) = \mathop{\mathbb{E}}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x})] + \mathop{\mathbb{E}}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z}))]. \tag{10.3}$$

In order to do this, we update D to have:

$$\max\{\log D(\mathbf{x}) + \log(1 - D(G(\mathbf{z})))\}, \tag{10.4}$$

and to update G to minimize the same equation:

$$\min\{\log D(\mathbf{x}) + \log(1 - D(G(\mathbf{z})))\}. \tag{10.5}$$

However, note that the component $\log(D(x))$ is not related to the weights of G, so in the backward step its gradient will be zero while updating G.

In practice, optimizing D upon completion in the inner loop of training is computationally prohibitive, and on finite datasets would result in overfitting. Instead, we alternate between k steps of optimizing D and one step of optimizing G. This results in D being maintained near its optimal solution, as long as G changes slowly enough. The training algorithm pseudo code is reported in Figure 10.8.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 10.8: Training algorithm pseudocode. Source: [44].

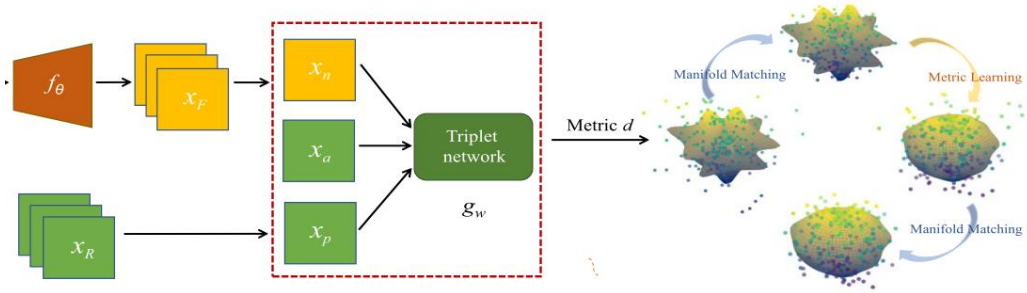


Figure 10.9: x_R and x_F represent samples in real data set S_R and generated data set S_F, respectively. The distribution generator f_θ outputs samples $x_F \in S_F$ based on manifold matching criteria under learned metric. For Triplet metric learning (see Section 11.3), anchor samples x_a and positive samples x_p are randomly selected from S_R, and negative samples x_n are randomly selected from S_F, without labelled data involved in this step. Learned distance metric is then used for manifold matching. Manifold matching step makes the fake samples (dots) condense around the real data manifold (surface), while metric learning step tries to "straighen" or "flatten" the real data manifold. These two steps goes interchangeably until convergence. Source: [26].

10.4 Manifold Matching via Metric Learning

The capabilities of GANs in Computer Vision tasks are limited by two main factors:

- Firstly, these neural networks model distributions using statistical characteristics, such as mean and moments, rather than geometric characteristics.
- Secondly, traditional GANs represent the loss of the discriminator network only in the form of a one-dimensional scalar value corresponding to the Euclidean distance between the real and fake data distributions.

Because of these two problems, it is impossible to directly apply metric training methods, as well as apply new loss functions. In *Manifold Matching via Metric Learning* (MvM), two networks are trained against each other:

1. The *metric generator* network learns to determine the best metric for the distribution generator network.
2. The *distribution generator* network learns to create negative examples for the metric generator network.

Metric learning is the task of learning a distance function over objects [69]. In practice, it means that we can train a model that tells a number for any pair of given objects representing a degree or score of similarity or dissimilarity between those given objects. As an example, given two images $\mathbf{a}$ and $\mathbf{b}$, metric learning aims to learn the dissimilarity d, where:

$$d(\mathbf{a}, \mathbf{b}) = \|\mathbf{Ga} - \mathbf{Gb}\|, \tag{10.6}$$

where the matrix $\mathbf{G}$ is learnable.

Through competitive learning, MvM creates a distribution generator network that can create a fake data distribution close to the real data distribution, and a metric generator network that can provide an effective metric to capture the internal geometric structure of the data distribution. The procedure is shown in Figure 10.9.

Chapter 11

Siamese Networks

Siamese Neural Network (SNN) is a class of neural network architectures that contain two or more identical subnetworks. The word "identical" means that they have the same configuration with the same parameters and weights and, usually, the inputs of the two networks are different. Parameter updating is mirrored across both sub-networks. It is used to find the similarity of the inputs by comparing its feature vectors, that is, the output vectors of the networks, so these networks are used in many applications. The employed *distance metric* o *similarity metric* depends on the application [98]. The distance metric can be used to build a loss function, how it is shown in Figure 11.1.

11.1 Cosine similarity

The SNN in Figure 11.2 uses the *cosine similarity* (in this case it is the cosine distance multiplied by -1) as similarity metric. The algorithm compares the output of the upper neural network and the output of the lower neural network through a distance or similarity metric. Through this similarity measure, the neural network states that the two profiles are different (cosine similarity value in $[-1, 0]$) or similar (cosine similarity value in $[0, 1]$). The algorithm then labels the data instance as positive if in the former case, or as negative if in the latter case. The final output can be finally compared with its corresponding ground truth.

Given the two output vectors of the twin networks $\mathbf{z}_A$ and $\mathbf{z}_B$, their cosine similarity is defined as:

$$S(\mathbf{z}_A, \mathbf{z}_B) = \frac{\langle \mathbf{z}_A, \mathbf{z}_B \rangle}{|\mathbf{z}_A||\mathbf{z}_B|}, \tag{11.1}$$

that is the cosine of the angle between the two vectors.

11.2 Binary Cross-Entropy loss

It is popular to choose the Binary Cross-Entropy loss as distance metric and for training the network. Let the two output vectors of the twin networks $\mathbf{z}_A$ and $\mathbf{z}_B$. We first compute the absolute value of their difference, obtaining the vector:

$$\mathbf{z}' = [|z_{A,1} - z_{B,1}|, |z_{A,2} - z_{B,2}|, \ldots, |z_{A,s} - z_{B,s}|]^T, \tag{11.2}$$

where both vectors are in $\mathbb{R}^s$. We then pass the obtained vector through a fully-connected layer with one output node, obtaining the value z''. The predicted confidence of the SNN is

$$y = \sigma(z''). \tag{11.3}$$

and the Binary Cross-Entropy value is:

$$-(t \log(y) + (1 - t) \log(1 - y)), \tag{11.4}$$

where t is the ground-truth similarity between the inputs (e.g. $t = 1$ if they are similar, $t = 0$ otherwise). The procedure is represented by the SNN in Figure 11.3.

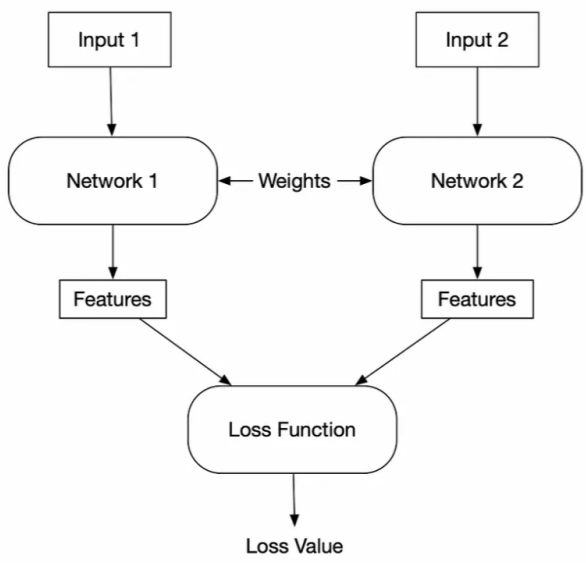


Figure 11.1: Generic SNN. Source: [17].

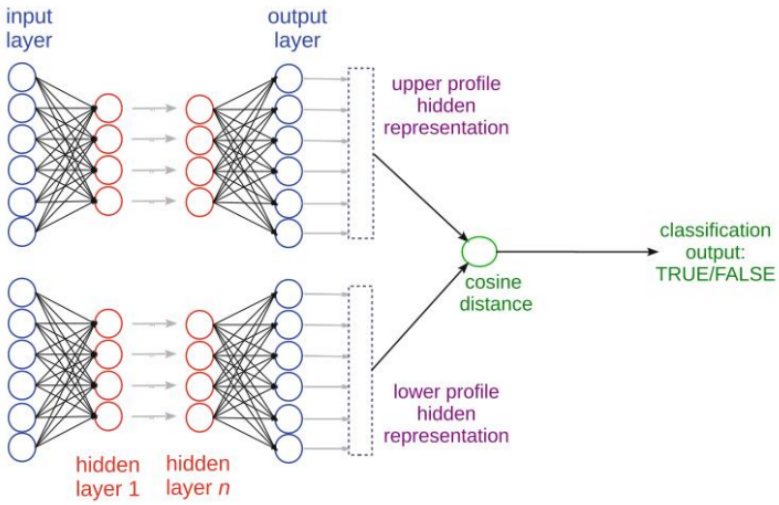


Figure 11.2: Representation of the structure of the SNN model. The data are processed from left to right. The value of the cosine distance is a measure of the similarity between the input pair of data instances, as final output.

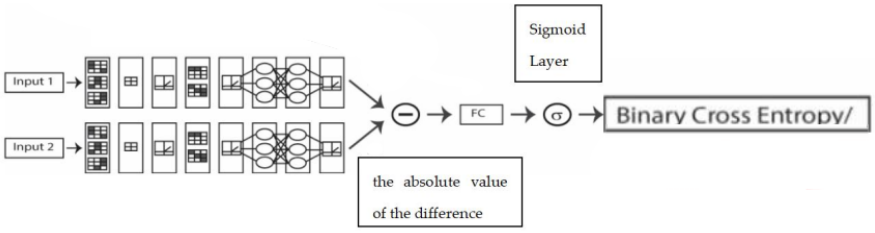


Figure 11.3: SNN with Binary Cross-Entropy as loss. Source: [72].

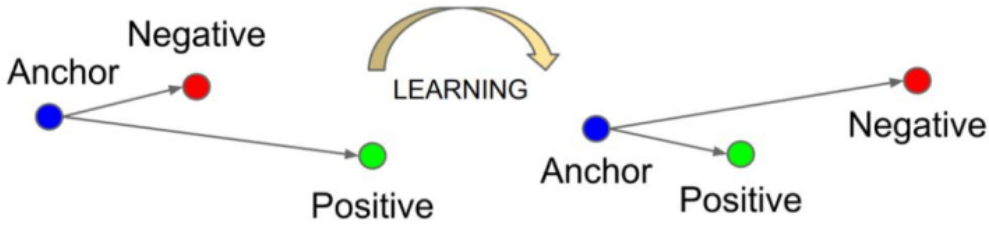


Figure 11.4: The triplet loss minimizes the distance between an anchor and a positive and maximizes the distance between the anchor and a negative. Source: [72].

11.3 Triplet loss

Triplet loss is a loss function where a reference input, the *anchor*, is compared to a matching input, called *positive*, and a non-matching input, called *negative* [108]. In the training phase, for every triplet of anchor, positive and negative (A, P, N) in the training set, we compute the feature vectors $\mathbf{z}_A, \mathbf{z}_P, \mathbf{z}_N$ and pass them through a sigmoid activation to obtain $\mathbf{y}_A, \mathbf{y}_P, \mathbf{y}_N$. The loss function to minimize for that triplet is:

$$J = \max(\|\mathbf{y}_A - \mathbf{y}_P\| - \|\mathbf{y}_A - \mathbf{y}_N\|, \ -\xi), \tag{11.5}$$

where $\xi > 0$ is the margin between positive and negative pairs [108]. In general, $\xi = 1$ turned out to give the best results. In other words, the loss function encourages the network to create similar representations for samples in the same class and different representation for samples in different classes. The inituition behind the triplet loss is reported in Figure 11.4.

11.4 Contrastive loss

The *contrastive loss* is defined as:

$$J = y\frac{1}{2}D_w^2 + (1-y)\frac{1}{2}\max(0, m - D_w)^2, \tag{11.6}$$

where y is the ground-truth similarity between the inputs (e.g. $y = 1$ if they are similar, $y = 0$ otherwise), D_w is the distance of the feature vectors using a predefined distance metric (e.g. the Euclidean distance) and m is the margin. With contrastive loss, we want to approach the patterns of the same class and try to distance the patterns of the other class. The first term of the objective therefore corresponds to trying to minimize intra-class distances, while the second term corresponds to trying to maximize the interclass distances.

The parameter m is the margin, that is the radius of the circumference within which all the patterns of the same class are to be found. Consider the case in which $y = 0$, meaning that the two classes are dissimilar. If $m - D_w < 0$ then the two patterns are at a greater distance than the margin and therefore all the loss is equal to 0. If instead $m - D_w > 0$ then it means that the two patterns belong to different classes, but their distance lies within the circumference of radius m: the loss will therefore take on a value other than 0 and the weights of the network will be updated to try to avoid this situation from happening again for the next patterns.

11.5 Dissimilarity space using SNNs

In [74] SNNs are employed for image classification, where a dissimilarity space (see Section 2.6) is learned by a set of clustering methods and a set of SNNs with different CNN backbones. The training method is composed of two phases:

1. A set of clusters is generated through k-means, producing a set of *prototypes*. These prototypes are the centers from k-means on the vector representation.

2. A dissimilarity space is generated by a SNN that learns a distance measure from the prototypes that minimizes differences between pairs within classes while also maximizing differences of pairs

between different classes, a process that produces a feature vector that is trained on SVMs using a *one-vs-all* approach [75].

In the test stage, an unknown pattern is projected onto the dissimilarity space that was learned by the SNN, which generates the feature vector that is then fed into the trained SVMs for a decision.

Chapter 12

Segmentation

We now specify the main difference between types of segmentation.

12.1 Image segmentation

Image segmentation consists in dividing an image into several constituent regions. The methods for this type of problem usually make use of the correlation between pixels in the image. It does not need label information about image pixels during training, and it cannot guarantee that the segmented regions will have the semantics that we hope to obtain during prediction. Taking the image in Figure 12.1 as input, image segmentation may divide the dog into two regions: one covers the mouth and eyes which are mainly black, and the other covers the rest of the body which is mainly yellow.

12.2 Instance segmentation

Instance segmentation consists in recognizing the pixel-level regions of each object instance in an image. Differently from semantic segmentation, instance segmentation needs to distinguish not only semantics, but also different object instances. For example, if there are two dogs in the image, instance segmentation needs to distinguish which of the two dogs a pixel belongs to.

12.3 Panoptic segmentation

If we want to distinguish each instance on one image but also recognize that a set of instances belongs to the same category, the task is called *panoptic segmentation*.

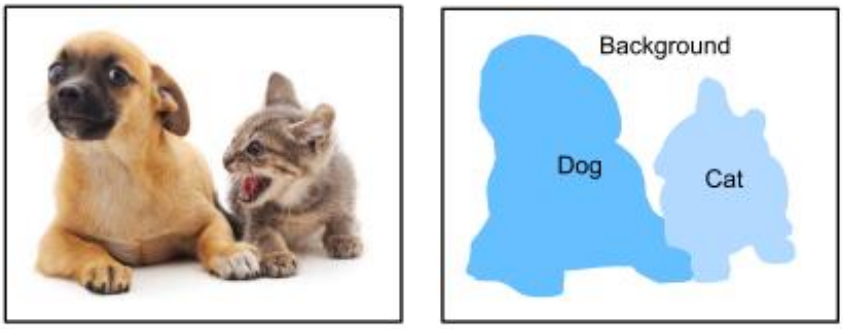


Figure 12.1: Image segmentation of a picture of a cat and a dog. Source: [72].

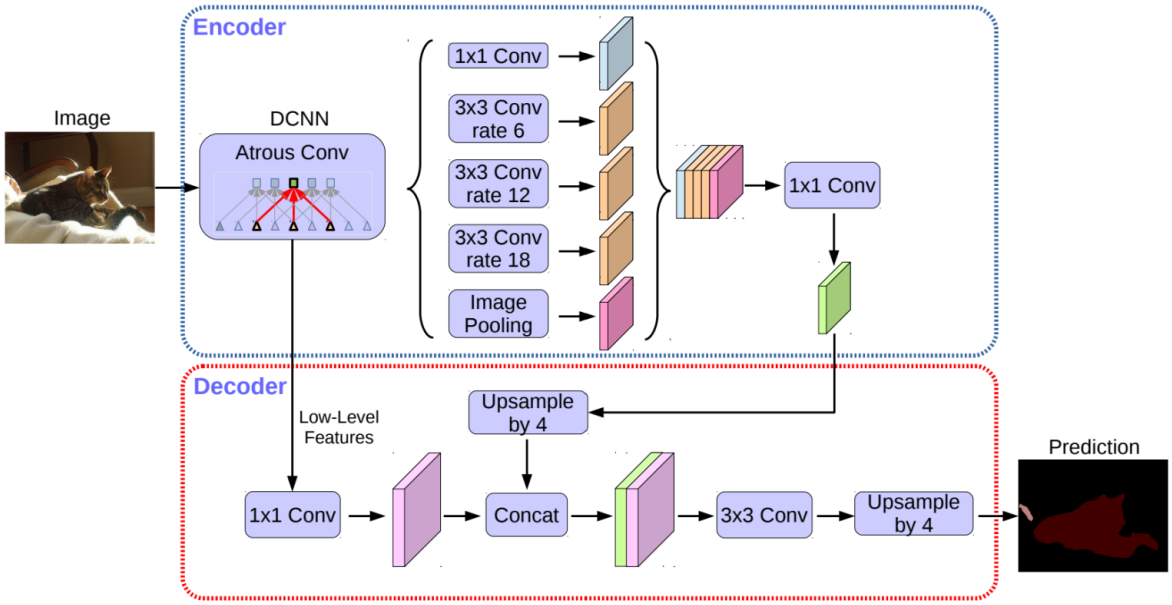


Figure 12.2: Architecture of DeepLabV3. Source: [72].

12.4 DeepLabV3+

DeepLabv3+ is a DNN architecture for segmentation tasks. Its structure is reported in Figure 12.2. It is composed by an *encoder* that gradually reduces the feature maps and captures higher semantic information, and by a *decoder* that recovers the spatial information upsampling them. This structure was the key point added by DeepLab V3+ with respect to the predecessor DeepLabV3.

12.4.1 Atrous convolution

Atrous convolution is the same as dilated convolution. We just insert zeros into the convolution filter to increase the size of the filter without increasing the number of learnable parameters [29]. It allows us to explicitly control the resolution of features computed by DNNs and adjust the filter's field-of-view in order to capture multi-scale information.

In the case of two-dimensional signals, for each location $\mathbf{i} = (i_1, i_2)$ on the output feature map $\mathbf{y}$ and a convolutional filter $\mathbf{w}$, atrous convolution is applied over the input feature map $\mathbf{x}$ as follows:

$$\mathbf{y}[\mathbf{i}] = \sum_{\mathbf{k}} \mathbf{x}[\mathbf{i} + r \cdot \mathbf{k}] \cdot \mathbf{w}[\mathbf{k}], \tag{12.1}$$

where the atrous rate r determines the stride with which we sample the input signal. In the sum, $\mathbf{k}$ assumes all possible positions (k_1, k_2) in the filter $\mathbf{w}$. For example, if $\mathbf{w}$ is a 3×3 filter, $\mathbf{k}$ will assume the values $(1,1), (1,2), (1,3), (2,1), (2,2), \ldots, (3,3)$. Note that the standard convolution is a special case in which $r = 1$. The filter's field-of-view is adaptively modified by changing the rate. Figure 12.3 shows the difference of the use of a standard convolution and the atrous convolution.

12.4.2 Depthwise separable convolution

Depthwise separable convolution is obtained by factorizing a standard convolution into a *depthwise convolution* followed by a *pointwise convolution* (convolution that uses a 1×1 kernel [84]). Depthwise convolution [33] is a type of convolution in which each input channel is convolved with a different kernel (called a depthwise kernel). Unlike a regular 2D convolution, depthwise convolution does not mix information across different input channels.

DeepLabV3+ uses atrous depthwise separable convolution [103]. The authors found that atrous separable convolution significantly reduces the computation complexity of proposed model while maintaining similar (or better) performance [13].

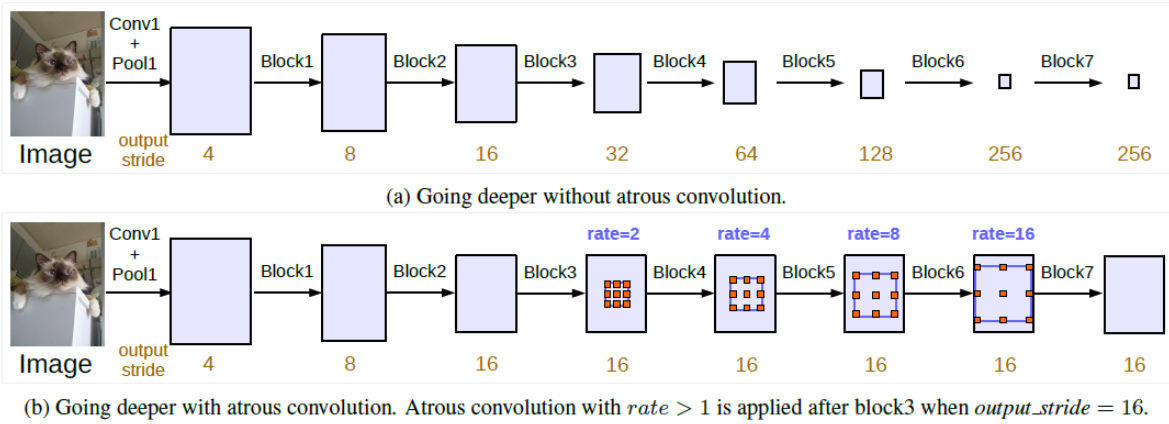


Figure 12.3: (a) Without Atrous Convolution: Standard convolution and pooling are performed which makes the output stride increasing, i.e. the output feature map smaller, when going deeper. However, consecutive striding is harmful for semantic segmentation because location and spatial information is lost at the deeper layers.
(b) With Atrous Convolution: With atrous convolution, we can keep the stride constant but with larger field-of-view without increasing the number of parameters or the amount of computation. And finally, we can have larger output feature map which is good for semantic segmentation. Source: [14].

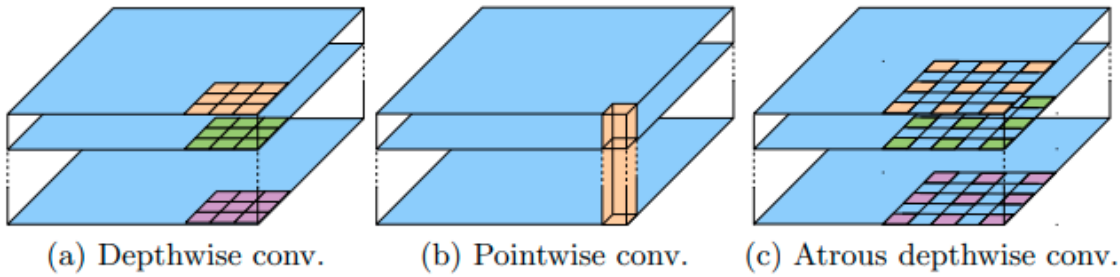


Figure 12.4: 3×3 Depthwise separable convolution decomposes a standard convolution into (a) a depthwise convolution (applying a single filter for each input channel) and (b) a pointwise convolution (combining the outputs from depthwise convolution across channels). In this [13], the authors explored atrous separable convolution where atrous convolution is adopted in the depthwise convolution, as shown in (c) with rate $r = 2$. Source: [72].

12.4.3 Decoder

We refer to the *low-level features* as the the feature extracted (see Section 7.14.2) from the backbone DCNN (downward arrow in Figure 12.2), and to the *encoder features* as the encoder output. In fact, the encoder module allows us to extract features at an arbitrary resolution by applying atrous convolution [13], corresponding to the curly braced convolutions in Figure 12.2 and performing *Atrous Spatial Pyramid Pooling* (ASPP).

The decoder performs the following steps:

- The encoder features are first bilineary upsampled by a factor of 4 and then concatenated with the corresponding low level features.
- A 1×1 convolution is performed on the low-level features before concatenation to reduce the number of channels. In fact, the corresponding low-level features usually contain a large number of channels (e.g. 256 or 512) which may outweight the importance of the rich encoder features.
- After concatenation, we apply a few 3×3 convolutions to refine the features followed by another bilinear upsampling by a factor of 4.

12.5 Region growing

Region growing is a region-based image segmentation method, it involves the selection of initial seed points. This approach to segmentation examines neighboring pixels of initial seed points and determines whether the pixel neighbors should be added to the region. The process is iterated on, in the same manner as general data clustering algorithms.

12.6 Loss functions for segmentation

12.6.1 Dice loss

The *Dice loss* comes from the Sørensen–Dice coefficient, a metric widely used to estimate the performance of semantic segmentation models. It shows how similar two images are to each other. Let $\mathbf{Y}$ be the prediction of the model and $\mathbf{T}$ be the target, such that:

$$Y_{km} = \begin{cases} 1 & \text{if } k \text{ is the predicted class of pixel } m \\ 0 & \text{otherwise} \end{cases} \tag{12.2}$$

and

$$T_{km} = \begin{cases} 1 & \text{if } k \text{ is the true class of pixel } m \\ 0 & \text{otherwise} \end{cases} \tag{12.3}$$

The *generalized Dice loss* is:

$$J = 1 - \text{DiceCoefficient} = 1 - 2\frac{\sum_{k=1}^{K} w_k \sum_{m=1}^{M} Y_{km} T_{km}}{\sum_{k=1}^{K} w_k \sum_{m=1}^{M} (Y_{km} + T_{km})}, \tag{12.4}$$

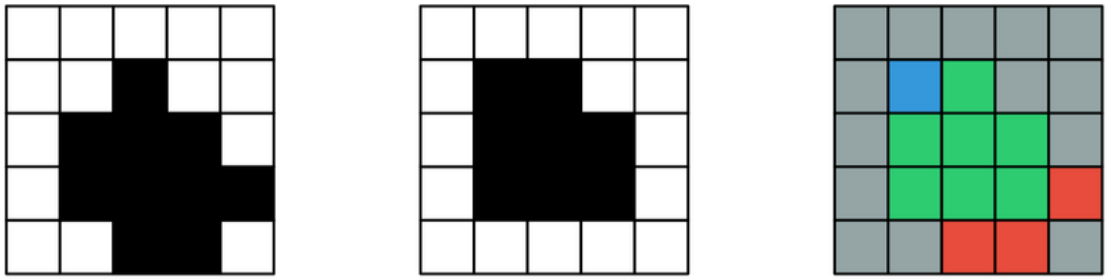
where

$$w_k = \frac{1}{(\sum_{m=1}^{M} T_{mk})^2}, \tag{12.5}$$

K is the number of classes and M is the number of pixels. The weighting factor w_k is introduced to help the network focus on a small region. In fact, it is inversely proportional to the frequency of the labels of a given class k. However, if each image has 2 classes (e.g. object and background) a more popular, but not totally equivalent, formulation of DiceCoefficient in Equation 12.4 is:

$$\text{DiceCoefficient} = \frac{2TP}{2TP + FP + FN}. \tag{12.6}$$

Figure 12.5 clearly explains the meaning of TP, FP and FN. The *Dice coefficient* shows how similar



(a) Ground truth binary mask **(b)** Predicted binary mask **(c)** Overlap between the masks.

Figure 12.5: Overlap scenario with (a) the ground truth, (b) the predicted binary masks, and (c) the overlap. In (a) and (b), black and white pixels denote the foreground and the background respectively. In (c), green, grey, blue, and red pixels denote TP, TN, FP, and FN pixels respectively. Source: [42].

two images are to each other. It holds

$$\text{DiceCoefficient} \in [0, 1]. \tag{12.7}$$

It is equal to 0 if no pixel is correctly classified by the model, while it is equal to 1 if all pixels are correctly classified.

12.6.2 Tversky loss

Tversky loss was introduced to tackle the problem of predominance of one class over another. Fix one class k, and consider the segmentation task with 2 classes, with class k as positive class and the other classes as negative class. The *Tversky index* for the class k is:

$$\text{TverskyIndex}_k = \frac{TP}{TP + \alpha FN + \beta FP}. \tag{12.8}$$

Again, Figure 12.5 represents this scenario with class k as positive class and the other classes as negative class. The *Tvwerky loss* is:

$$J = \sum_{k=1}^{K} 1 - \text{TverskyIndex}_k, \tag{12.9}$$

where K is the number of classes.

Chapter 13

Object detection

Object detection involves identifying instances of objects within an image in a *bounding box* and in classifying them as belonging to a certain class (e.g. humans, animals or cars). From an application point of view, it is possible to group object detection into two categories:

- *General object detection*, in which the goal is to investigate methods for identifying different types of objects using a single framework, in order to simulate human vision and cognition.
- *Detection applications*, that is, recognition, under specific application scenarios, of objects of a certain class: this is the case of applications for pedestrian detection, face detection or for text detection.

Currently the models for object detection can be divided into two macro-categories:

- *Two-stage detectors*, models that divide the task of identifying objects into several phases, following a "coarse to fine" policy.
- *One-stage detectors*, models whose process tries to complete the recognition in a single step with the use of a single network.

13.1 Region-based CNN

A *region-based CNN* (R-CNN) is a two-stage detector for image detection. The following explanation is mainly taken from [91]. R-CNN can be broken down into 4 steps:

1. Generating region proposals.
2. Extracting CNN features.
3. Classification of region proposals using SVMs.
4. Bounding Box regression.

A scheme of R-CNN is reported in Figure 13.1.

13.1.1 Region proposals

R-CNN starts by dividing the input image into multiple regions or subregions. These regions are referred to as *region proposals*. The region proposal step is responsible for generating a set of potential regions in the image that are likely to contain objects. R-CNN does not generate these proposals itself, instead, it relies on external methods like *selective search* [120], illustrated in Figure 13.2. The algorithm generates approximaly 2000 regions. For each region, the best fitting rectangle is selected.

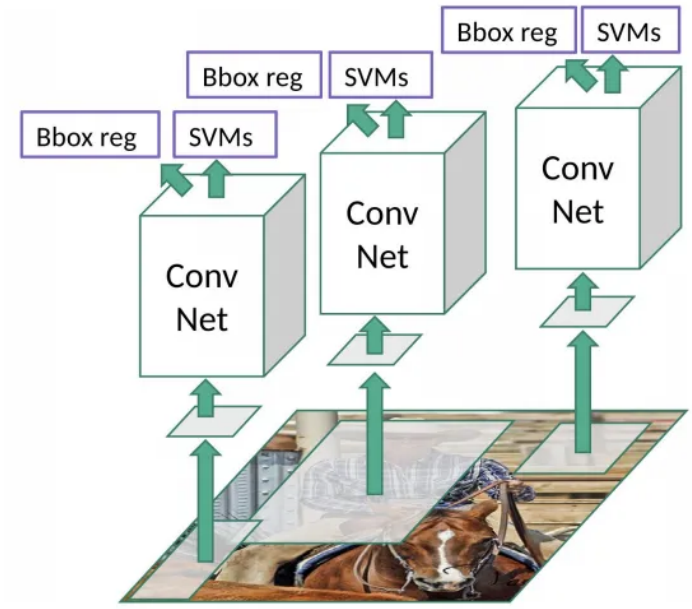


Figure 13.1: Scheme of R-CNN. Source: [81].

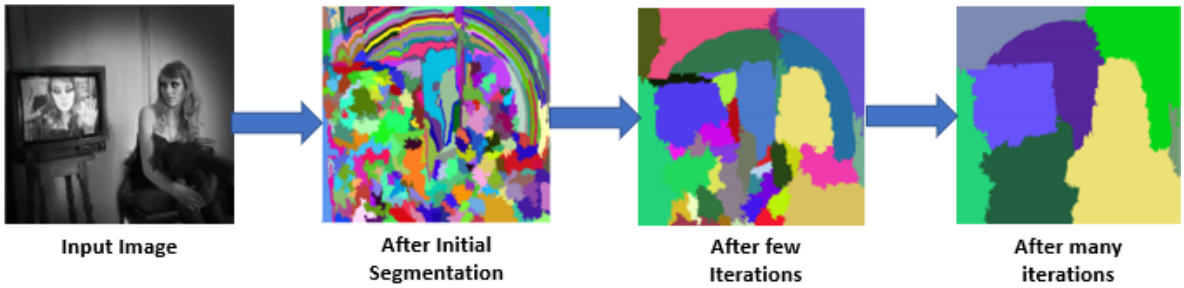


Figure 13.2: Example of the selective search algorithm applied to an image. Source: [120].

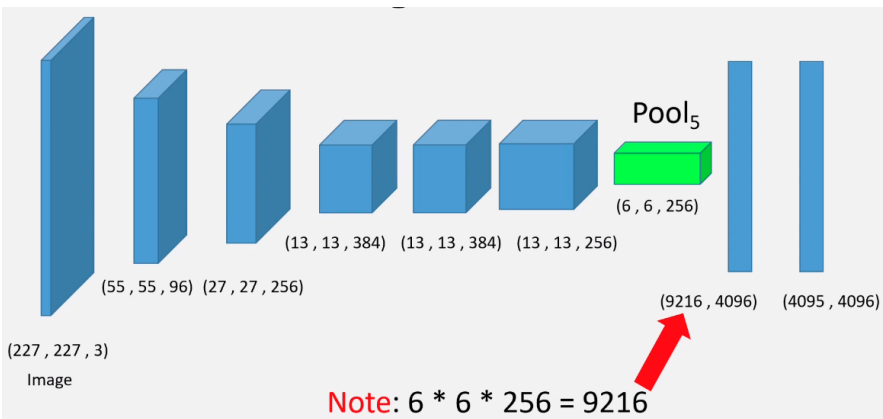
13.1.2 Extracting features from the CNN

We pick a pre-trained CNN trained on image classification as feature extractor. The original authors uses AlexNet. The CNN is then fine-tuned for the object detection task. For this reason, we have to:

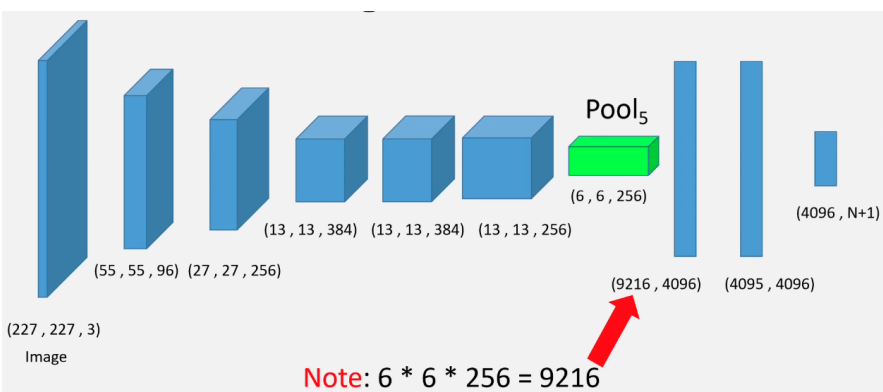
- Replace the last layer.
- Redefine the dataset and classify the region proposals.

Replace the last layer

Given the original AlexNet architecture (trained on 1000 class of ImageNet):



we replace the last layer as follows:



where N is the number of classes in the object detection task. The $+1$ term is introduced to classify a region proposal as *background*. In fact, the CNN goal is to classify a region proposal as one of the N classes or as the background. The CNN will receive as input a rescaled version of a region proposal. Note that, when a region proposal is passed to the CNN, we obtain a 4096-dimensional feature vector before the classification.

Redefine the dataset and classify the region proposals for fine-tuning AlexNet

Since object detection is a supervised learning task, ground-truth boxes are given in the dataset. As an example, consider the case in which the classes are "dog" and "cat" ($N = 2$). An example of the training set could be given by Figure 13.3 together with the ground-truth coordinates and, of course, the classes.

Now we need to "classify" each region proposal for the training purpouse (in fact, we are still building the training set). Given a region proposal, we compute the *Intersection Over Union* (IOU) with each ground-truth bounding box. If IOU > 0.5, then the region proposal becomes a *positive*

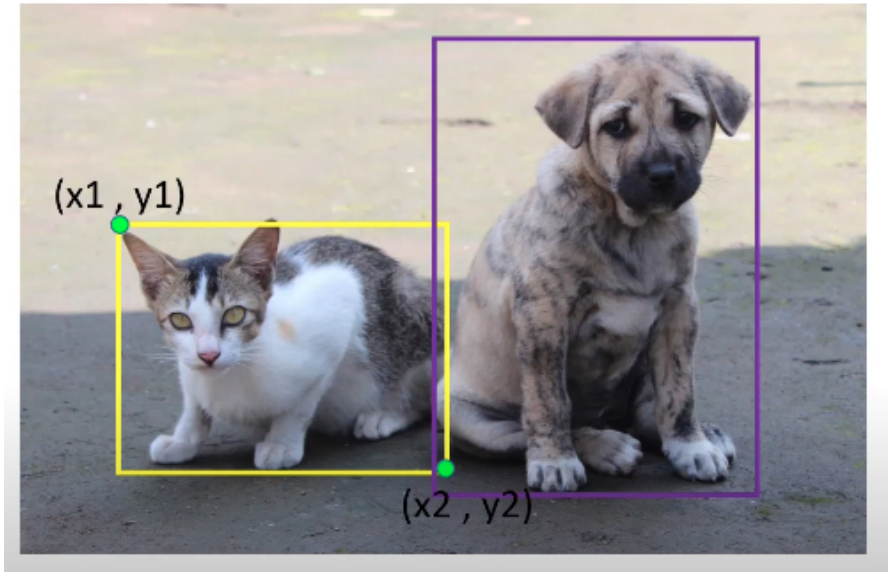
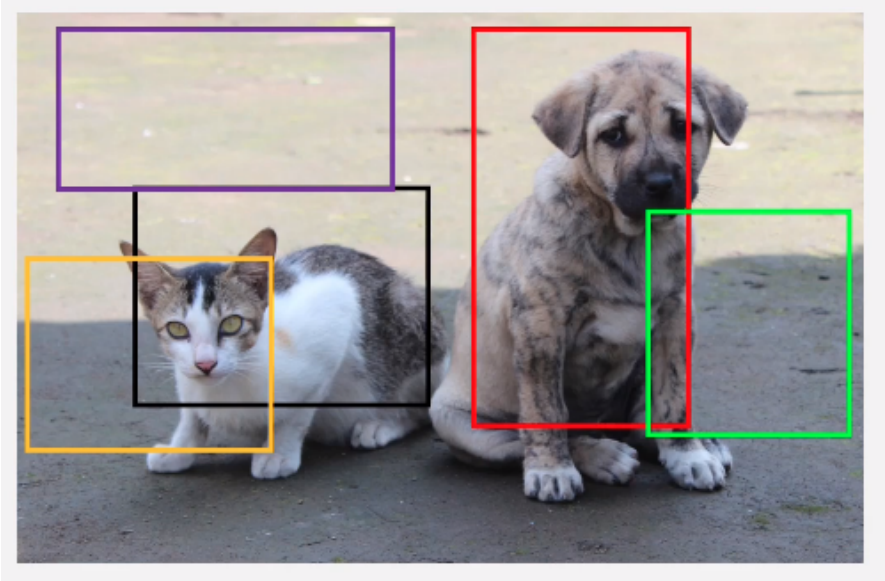
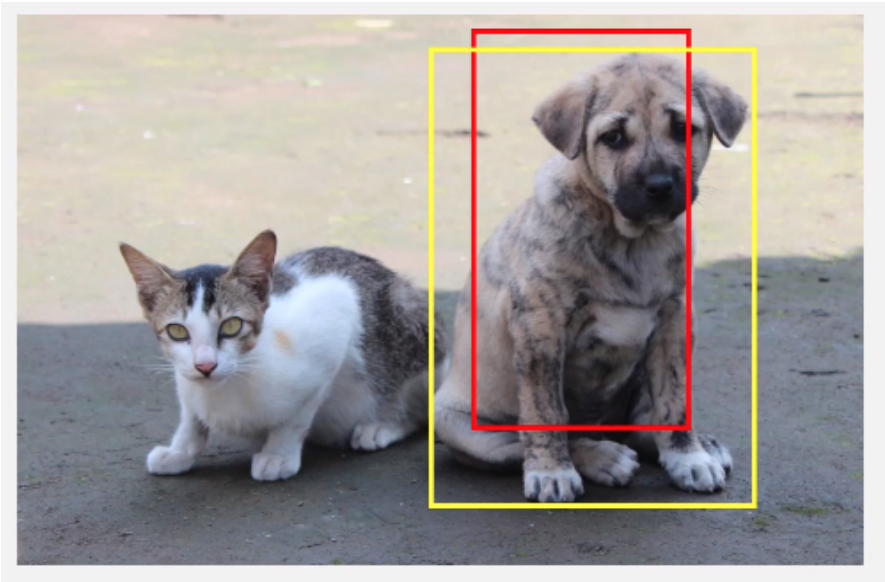


Figure 13.3: Example of training set example. In this case, there are two ground-truth bounding boxes: one for the cat and one for the dog. Each ground-truth box has a ground-truth class assigned. Source: [91].

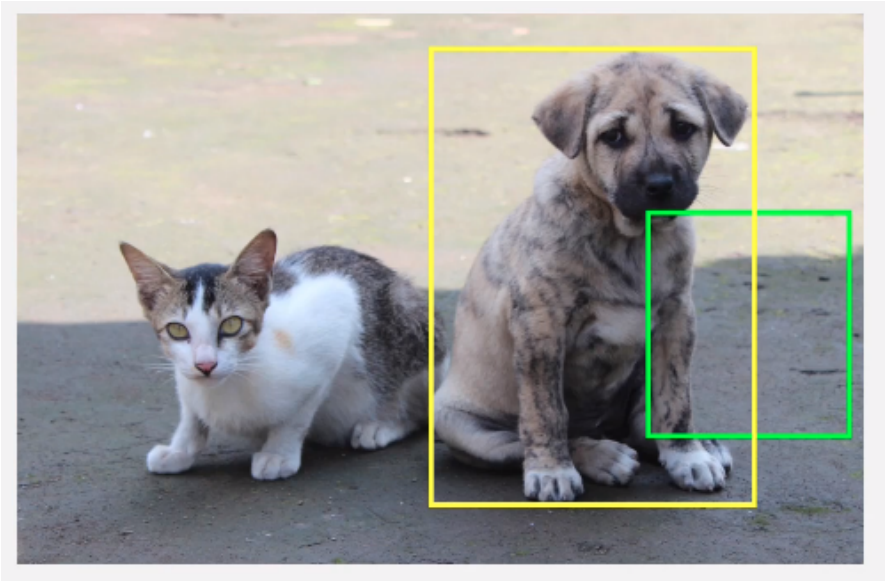
example for that class. Otherwise, it becomes a *negative example*. Suppose we apply selective search to the Figure 13.3, obtaining the following region proposals:



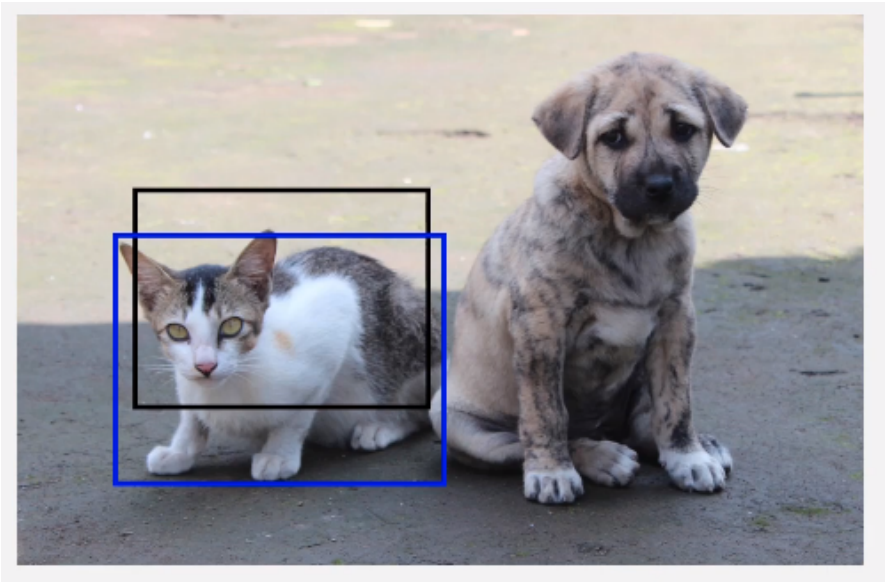
The red-boxed region proposal has an IOU = 0.8 with the ground-truth bounding box of the dog (yellow in the figure below), therefore it will be a positive example for the class dog.



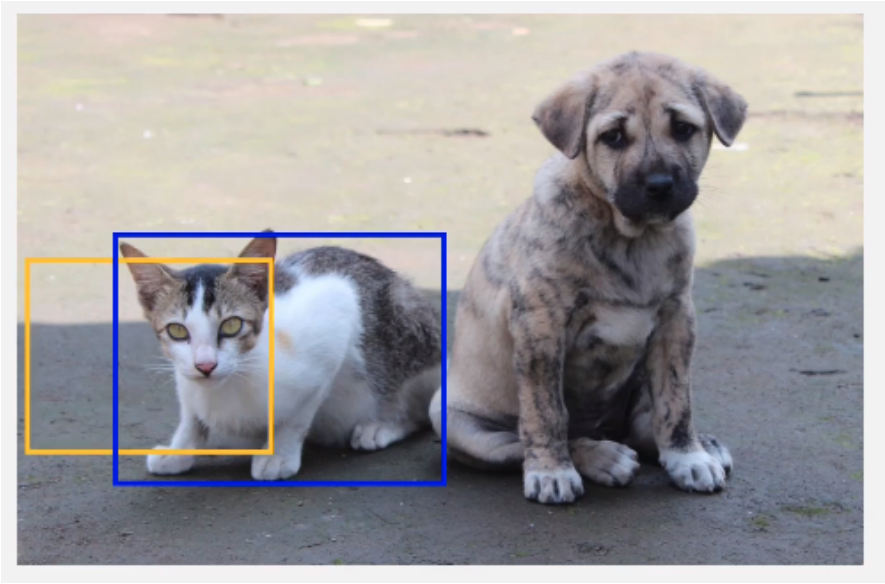
However, the green-boxed region proposal has an IOU = 0.3, therefore it will be a negative example.



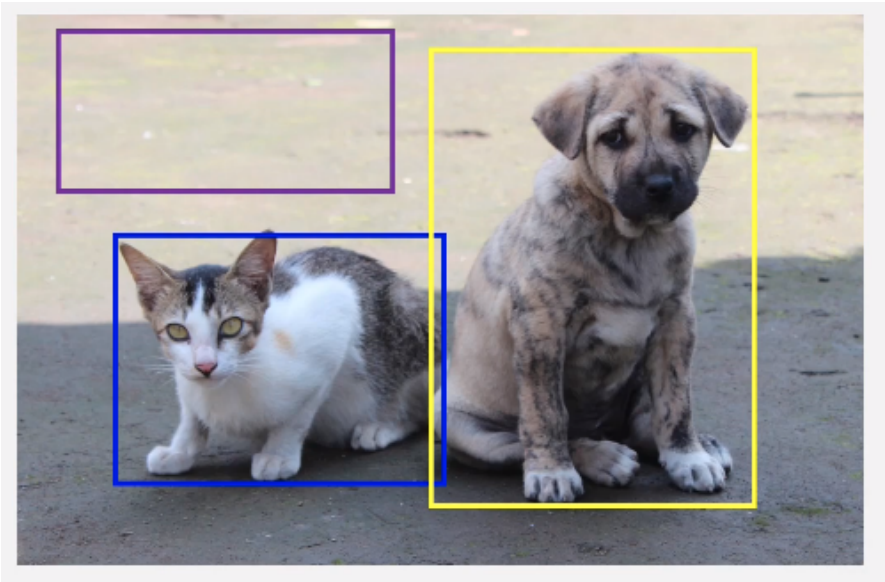
Similarly, the black-boxed region proposal has an IOU = 0.7 with the ground-truth bounding of the cat (blue in the figure below), therefore it will be a positive example for the class cat.



However, the orange-boxed region proposal has an IOU = 0.4, therefore it will be a negative example.



Lastly, the purple-boxed region proposal has an IOU = 0 with every ground-truth bounding box, therefore it will be a negative example.



The built dataset is used to *fine-tune the CNN*. It is reported in Figure 13.4.

Region	x1	y1	x2	y2	IOU	class (neuron)
Red	300	20	70	290	0.8	dog (1)
Green	375	180	60	70	0.3	negative (3)
Black	...	...	...	...	0.7	cat (2)
orange	...	...	...	...	0.4	negative (3)
purple	...	...	...	...	0.0	negative (3)

Figure 13.4: Dataset for fine-tuning the CNN.

The region proposals are scaled for the CNN (Figure 13.5). In fact, CNNs accepts images of constant size. Each image is resized into a $(227 \times 227 \times 3)$ tensor.

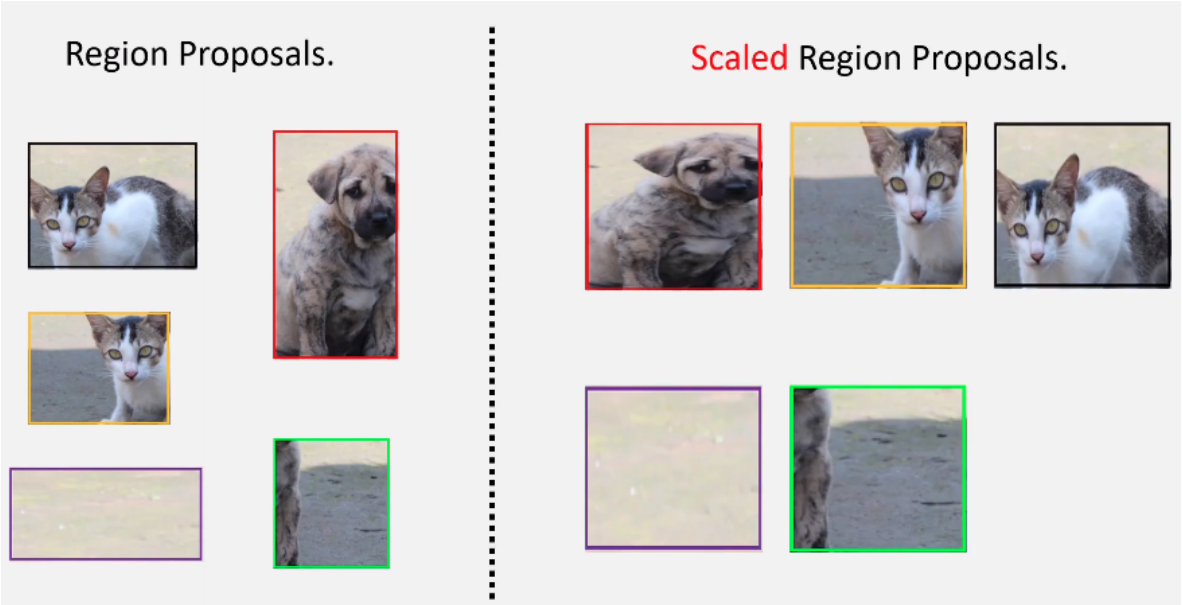


Figure 13.5: Region proposals and scaled region proposals.

13.1.3 Training

When we input the region proposals into the dataset and, for each of them, obtain a 4096-dimensional vector from the AlexNet as follows:

4096

Region	feature_0	feature_1	...	feature_4095	IOU	class
Red	0.25	1.96	-2.66	8.4	0.8	dog
Green	...	...	...	...	0.3	negative
Black	...				0.7	cat
orange		...			0.4	negative
purple			...		0.0	negative
...				...		
...					...	
...	...	...	...	...	...	...

The training consists in training the SVMs for classification and the bounding box regressor.

Training the SVMs

There is one SVM for each class. In other words, there are N SVMs. The goal of a SVM of a certain class is to find whether an object of that class is present in that region proposal or not. Consider the training of the "Dog" SVM. From the dataset in Figure 13.4, we take all the region proposals that have IOU < 0.3 with the dog ground-truth box (e.g. the black-boxed region proposal will be picked because it has IOU $= 0$ with the dog ground-truth box). In our example, we pick the following region proposals:

	Region	x1	y1	x2	y2	IOU	class (neuron)
✖	Red	300	20	70	290	0.8	dog
✓	Green	375	180	60	70	0.3	negative
✓	Black	...	...	...	...	0.7	cat (negative for dog SVM.)
✓	orange	...	...	...	...	0.4	negative
✓	purple	...	...	...	...	0.0	negative

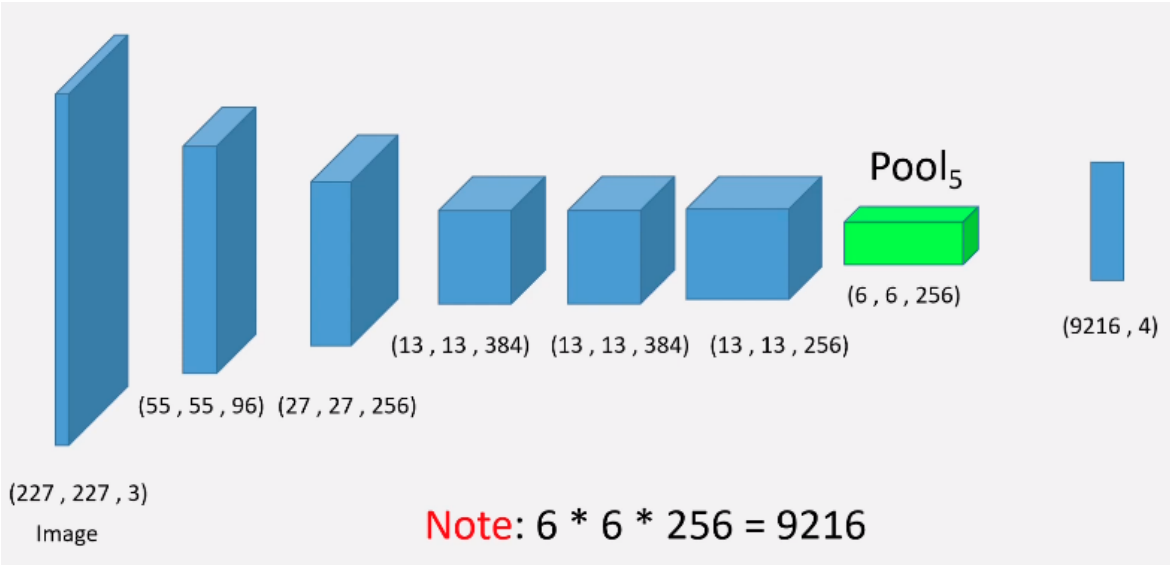
The picked region proposals become the *negative examples* for the "Dog" SVM. The positive examples are the ground-truth boxes for the dog class. In this example, the only positive example is the yellow ground-truth box the dog. The dataset for training the "Dog" SVM is:

4096

Region	feature_0	feature_1	...	feature_4095	IOU	class
Bounding box	0.8	-3.62	...	1.86	1.0	1 (positive)
Green	...	...	...	...	0.3	0 (negative)
Black	...				0.7	0 (negative)
orange		...			0.4	0 (negative)
purple			...		0.0	0 (negative)

Since there will be multiple images in the dataset, the number of positive class will be actually greater than 1.

Similarly, for the "Cat" SVM cat dataset we pick the following region proposals as negative:



	Region	x1	y1	x2	y2	IOU	class
✓	Red	300	20	70	290	0.8	negative
✓	Green	375	180	60	70	0.3	negative
✖	Black	...	...	...	...	0.7	non negative
✖	orange	...	...	...	...	0.4	non negative
✓	purple	...	...	...	...	0.0	negative

The dataset for training the "Cat" SVM is:

Region	feature_0	feature_1	...	feature_4095	IOU	class
Bounding box	-1.56	8.12	...	3.77	1.0	1 (positive)
Green	...	...	...	...	0.3	0 (negative)
purple			...		0.0	0 (negative)
Red	0.25	1.96	-2.66	8.4	0.8	0 (negative)

Training the bounding box regressor

For training the bounding box regressor we pick only the region proposals with IOU > 0.6 with the ground truth boxes.

	Region	x1	y1	x2	y2	IOU	class (neuron)
✓	Red	300	20	70	290	0.8	dog
✖	Green	375	180	60	70	0.3	negative
✓	Black	...	...	...	...	0.7	cat
✖	orange	...	...	...	...	0.4	negative
✖	purple	...	...	...	...	0.0	negative

The regressor is built on top the CNN starting from the pooling layer of AlexNet: The bounding box regressor has 4 output notes $(\hat{t}_x, \hat{t}_y, \hat{t}_w, \hat{t}_h)$ since they corresponds to the predicted translation ($\hat{t}_x$ and $\hat{t}_y$) and scaling ($\hat{t}_w$ and $\hat{t}_h$) we should apply to the region proposal's bounding box to match the intersecated ground-truth box (Figure 13.6). The objective function is the MSE between the needed transformations (easily computable) and the predicted ones.

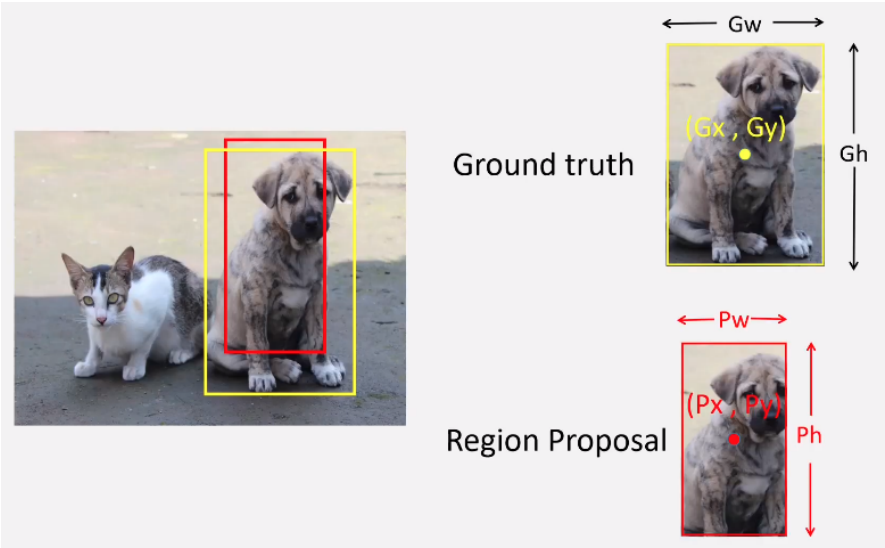


Figure 13.6: The bounding box regressor outputs the predicted trasformation (translation and scaling) needed for turning the red box (region proposal) into the yellow box (ground-truth).

13.2 Non-maximum suppression

Out of those many bounding boxes, a lot of them would be redundant and overlapping bounding boxes which need to be removed. To accomplish that, *non-maximum suppression* algorithm is used.

The algorithm accepts as input:

- A list of proposal region boxes B.
- A list of scores S corresponding to the proposal region boxes B obtained from the SVMs.
- Overlap threshold N ([72] uses $N = 0.7$).

The algorithm returns a list of filtered proposals D from B. The procedure is the following:

1. Set $D = \emptyset$.
2. Select the proposal with highest confidence score, remove it from B and add it to the final proposal list D.
3. Now compare this proposal with all the proposals — calculate the IOU of this proposal with every other proposal. If the IOU is greater than the threshold N, remove that proposal from B.
4. This process is repeated until there are no more proposals left in B.

Figure 13.7 reports an intuition of this procedure.

13.3 Fast R-CNN

The same authors of R-CNN proposed a faster algorithm, called *fast R-CNN*. The general architecture of Fast R-CNN is shown in Figure 13.8. Informally, given an image and a region proposal box (red box in the image), we input the full image to a CNN and we *project* the box to the feature map, referred to as region of interest. Then we apply the *region of interest pooling* (RoI pooling).

Figure 13.7: Example of non-maximum suppression. Source: [72].

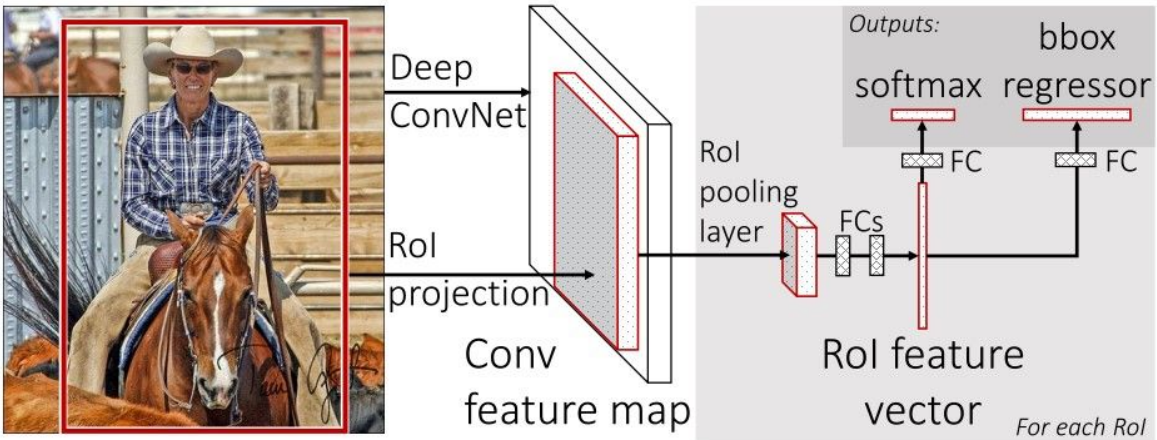


Figure 13.8: Fast R-CNN structure. Source: [41].

RoI pooling layer works differently from pooling layer. In the pooling layer, we indirectly control the output shape by specifying sizes of the pooling window, padding and stride, as in Equation (7.23). In contrast, in RoI pooling layer we can directly specify the output shape. Consider the following example, in which we specify a 2×2 output shape:

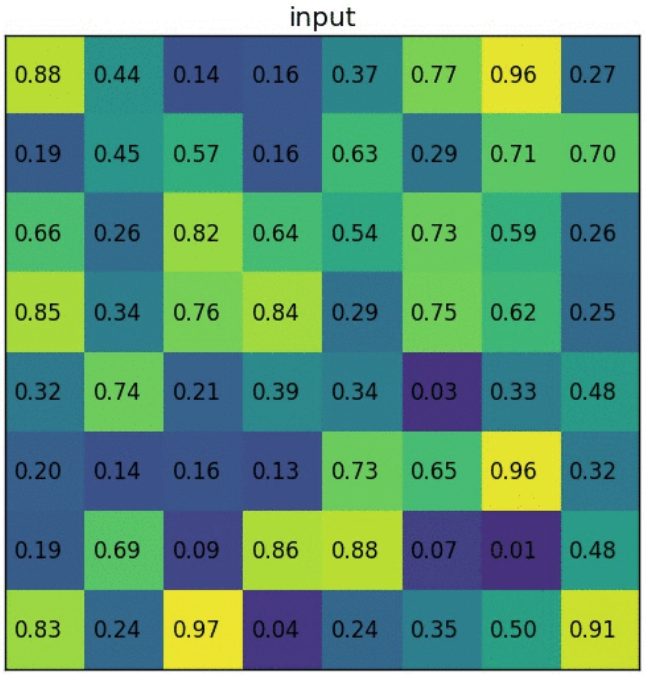


Figure 13.9: Source: [72].

We highlight the projected region proposal in black.

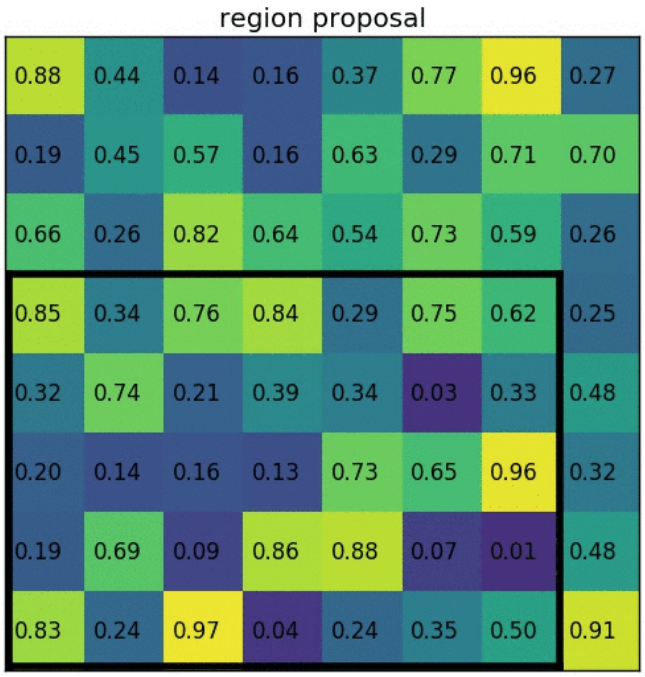


Figure 13.10: Source: [72].

Note that the size of the region of interest does not have to be perfectly divisible by the number of pooling sections (in this case our RoI is 7×5 and we have 2×2 pooling sections).

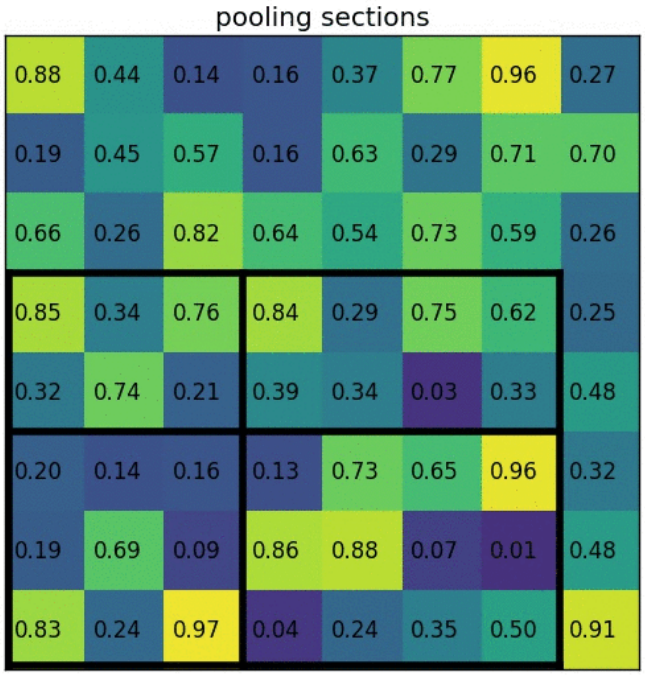


Figure 13.11: Source: [72].

The max values in each of the sections are:

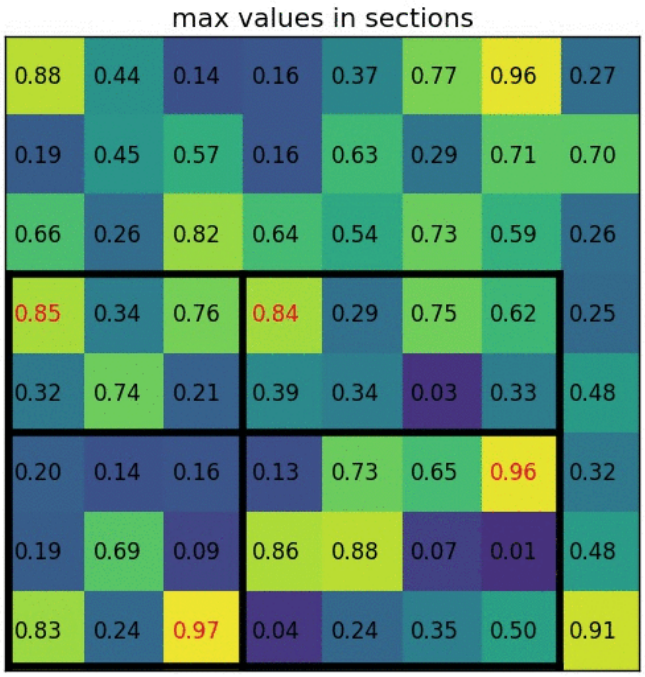


Figure 13.12: Source: [72].

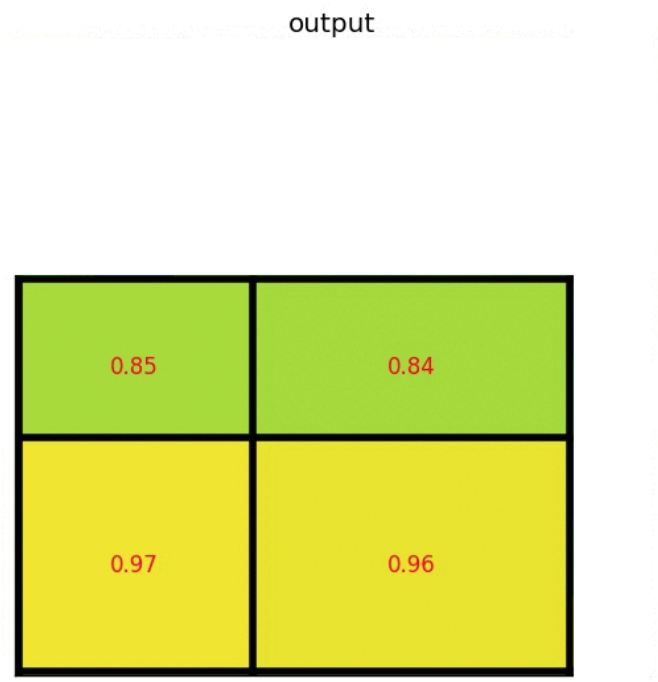


Figure 13.13: Source: [72].

Formally, the procedure is the following:

1. Given an input image, let the shape of the CNN output be $1 \times c \times h_1 \times w_1$, where c is the number of channels.

2. Suppose that selective search generates n region proposals. These region proposals (of different shapes) mark regions of interest (of different shape) on the CNN output. The CNN output is input into the RoI layer, outputting concatenated features of shape $n \times c \times h_2 \times w_2$.

3. Using a fully-connected layer, transform the concatenated features into an output of shape $n \times d$, where d depends on the model design.

4. Predicts the class and bounding box for each of the n region proposals. In class and bounding box prediction, transform the fully-connected layer into an output of shape $n \times q + 1$ (with q number of classes, $+1$ refers to the background) and an output shape $n \times 4$ (analogous to Section 13.1.3), respectively. The class prediction uses softmax regression.

13.3.1 RoI pooling layer size

As already mentioned above, in the RoI pooling layer we can directly specify the output shape. Let us specify the output height and width for each region as h_2 and w_2, respectively. For any region of interest window of shape $h \times w$, this window is divided into a $h_2 \times w_2$ grid of subwindows, where the shape of each subwindows is approximately $\frac{h}{h_2} \times \frac{w}{w_2}$. In practice, the height and width of any subwindow shall be rounded up, and the largest element shall be used as the output of the subwindow (since we are applying max-pooling). Therefore, the RoI pooling layer can extract features of the same shape even when regions of interest have different shapes.

13.3.2 CNN training

We present how the CNN is trained in a R-CNN.

1. Pick a pre-trained CNN on a image classification task.

2. Propose regions by selective search.

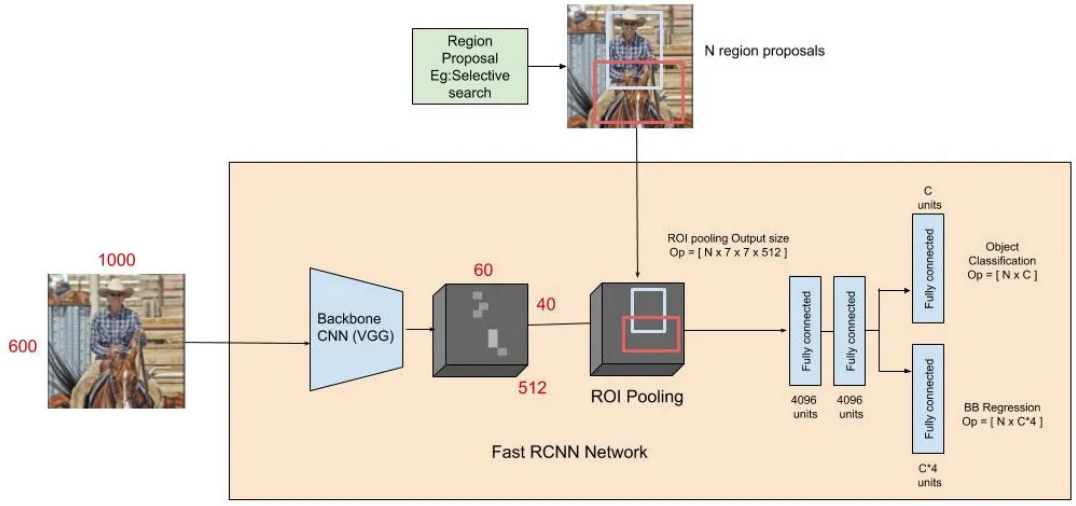


Figure 13.14: The Fast R-CNN pipeline. Source: [40].

3. After the pre-trained CNN:
   (a) Replace the last max-pooling layer with a RoI pooling layer. Note that sharing the CNN computation makes a lot of sense, as many region proposals of the same image are highly overlapped.
   (b) Replace the last fully connected layer and the softmax layer with a fully-connected layer and softmax over $q + 1$ classes (+1 indicates the background).
4. Finally, the model branches into two output layers:
   - A softmax estimator of $q + 1$ classes, outputting the estimated discrete probability distribution per region of interest.
   - A bounding box regression model which predicts offsets and scales (t_x, t_y, t_w, t_h) for each one of the q classes.

The network architecture is reported in Figure 13.8 where C is the number of classes.

13.4 Faster R-CNN

To be more accurate in object detection, the fast R-CNN model usually has to generate a lot of region proposals in selective search. To reduce region proposals without loss of accuracy, the *faster R-CNN* proposes to replace selective search with a *region proposal network*.

13.5 You Only Look Once

Compared to pre-existing object detection models, *You Only Look Once* (YOLO) is significantly faster. This is mainly possible due to the fact that YOLO does not divide the recognition into several phases, but predicts bounding boxes, probabilities and classes of the objects present in the input image in a *single phase* (it is, in fact, the first one-stage detector). Compared to other object detection systems, YOLO makes more localization errors, but at the same time it is less likely to recognize false-positives in the background of the image, as well as being considerably faster.

YOLO unifies the different components of object detection systems in a single network. This force the network to reason over the entire image at the same time, rather than specific regions,

providing a greater understanding of the environment in which the different classes of objects are located. Furthermore, by doing so, it creates an implicit link between effectively related classes of objects.

13.5.1 Execution

The model divides the image into $S \times S$ cells. For each cell it predicts B bounding box, each with a confidence score and C classes with relative probabilities. In non-maximal suppression, YOLO suppresses all bounding boxes that have lower probability scores. A scheme of YOLO is reported in Figure 13.15.

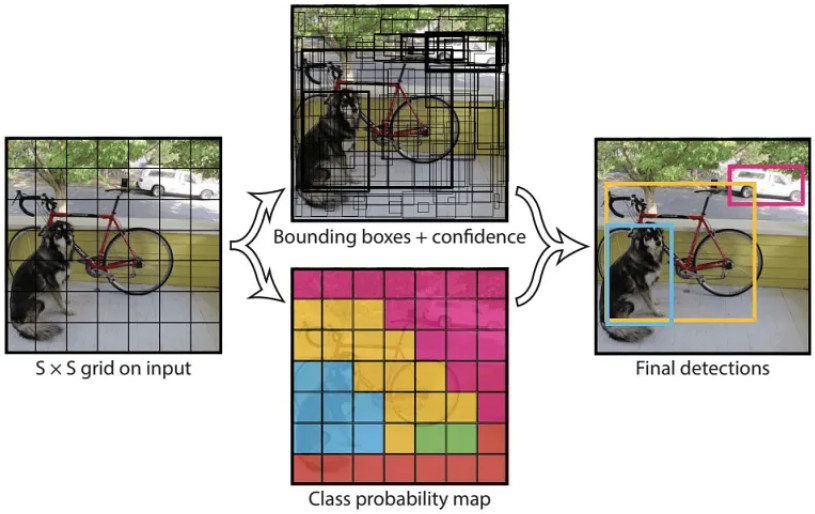


Figure 13.15: Scheme of YOLO. Source: [94].

YOLO algorithm does not search for interesting regions in our image that could potentially contain an object. Instead, the model splits the image into cells, now typically using a 19×19 grid (original YOLO uses 7×7). Each cell is responsible for predicting 5 bounding boxes (in case there is more than one object in this cell). Therefore, we arrive at a large number of $19 \times 19 \times 5 = 1805$ bounding boxes for one image. In fact, many bounding boxes are generated, even for an image with a small number of recognizible objects. However, during inference, the number of boxes is trimmed down using non-maximal suppression.

13.5.2 Unified detection

Fixed S and B, the model divides the image into a grid of $S \times S$ cells. The grid cell containing the center of an object is defined as the *cell responsible* for it (see figure below). Each cell of the grid predicts the B bounding boxes and assigns each of them a *confidence score*. Formally [94] we define confidence as

$$\Pr(\text{Object}) \cdot \text{IOU}. \tag{13.1}$$

If no object exists in that cell, the confidence scores should be zero. Otherwise, we want the confidence score to equal the IOU between the predicted box and the ground truth.

Each bounding box consists of 5 predictions: x, y, w, h, and confidence. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image. Finally the confidence prediction is $\Pr(\text{Object}) \cdot \text{IOU}$ [72] as Equation (13.1).

Each grid cell also predicts C conditional class probabilities:

$$\Pr(\text{Class}_i \mid \text{Object}). \tag{13.2}$$

These probabilities are conditioned on the grid cell containing an object. We only predict one set of class probabilities per grid cell, regardless of the number of boxes B. In this way we create the class probability map reported in Figure 13.15.

At test time we multiply the conditional class probabilities and the individual box confidence predictions,

$$\Pr(\text{Class}_i \mid \text{Object}) \cdot \Pr(\text{Object}) \cdot \text{IOU} = \Pr(\text{Class}_i) \cdot \text{IOU}, \tag{13.3}$$

which gives us class-specific confidence scores for each box. These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object.

The predictions of the network are encoded as a $(S \times S \times (B \cdot 5 + C))$ tensor. C corresponds to the predicted class probabilities.

13.5.3 Training

The training is divided in two-phases:

1. Pretraining of the CNN on image classification, only of deepest levels.
2. Training of the entire network on detection.

The architecture is pretrained on ImageNet1000, then converted to perform detection by adding 4 convolutional layers and two fully-connected layers. The pretrained is performed by increasing the input resolution of the network from 224×224 to 448×448.

The final layer predicts both class probabilities and bounding box coordinates. We normalize the bounding box width and height by the image width and height so that they fall between 0 and 1. We parametrize the bounding box x and y coordinates to be offsets of a particular grid cell location so they are also bounded between 0 and 1. We use a linear activation function for the final layer and all other layers use leaky ReLU with $a = 0.1$.

The YOLO loss function consists of:

- Classification loss.
- Localization loss.
- Confidence loss.

Classification loss

Calculates the error in the classification of a specific object. That is, it indicates whether or not the recognized class corresponds to the object identified in the image. Given a cell in the grid, if it is considered responsible for a certain object, then it is also responsible for its classification. The error in this operation is given by the quadratic function which calculates the conditional probabilities of the recognized class with respect to each class:

$$\sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2, \tag{13.4}$$

where $\mathbb{1}_i^{\text{obj}}$ is 1 if and only if there is an object in the i-th cell, 0 otherwise, $p_i(c)$ is the conditional probability of the c-th class to appear in the i-th cell (Equation (13.2)).

Localization loss

It measures the error in the prediction of the bounding boxes, i.e. the height, width and coordinates of the center. YOLO calculates this error only for the bounding boxes responsible for recognizing an object. The error in the localization calculated as follows:

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2], \tag{13.5}$$

where $\mathbb{1}_{ij}^{\text{obj}}$ is 1 if and only if the j-th bounding box of the i-th cell is responsible for recognizing an object (i.e. an object is located in that bounding box), 0 otherwise, $\lambda_{\text{coord}} = 5$. Note that the error with respect to the size of the bounding box is calculated on the square root of the values. This is in order not to give the same weight to errors, regardless of the size of the bounding box: for example, a 2-pixel error could be negligible if found on a 500-pixel bounding box, but important on a 30-pixel bounding box. The square root downscales high values while less affecting low values of width and height.

Confidence loss

It is used to quantify the objectivity of the prediction made on a single bounding box. If, given a bounding box, the model recognizes an object in it, the confidence loss is calculated as follows:

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} (C_{ij} - \hat{C}_{ij})^2, \tag{13.6}$$

where C_{ij} is the confidence score of the j-th bounding box of the i-th cell (Equation (13.1)).

Most bounding boxes contain no objects. This causes an imbalance of the model, which in practice trains more frequently to recognize backgrounds rather than objects. To remedy this, the confidence loss for bounding boxes that do not contain objects is changed by a $\lambda_{\text{no obj}} = 0.5$ factor. We therefore get:

$$\lambda_{\text{no obj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{no obj}} (C_{ij} - \hat{C}_{ij})^2, \tag{13.7}$$

where $\mathbb{1}_{ij}^{\text{no obj}}$ is the complement of $\mathbb{1}_{ij}^{\text{obj}}$.

Additional details

To reduce overfitting, the model has a dropout layer with a 0.5 reduction rate placed after the first fully-connected layer to prevent coadaptation between the fully connected layers. Additionally, data augmentation techniques have also been introduced: before using the images taken as input, the model processes them by adding translations and changing the aspect ratio of the image up to 0.2 compared to the original characteristics. Finally, the images are modified by changing exposure and saturation by a factor between 0 and 1.5.

Regarding the learning rate, the authors noted that the model often diverges towards unstable gradients when trained from the beginning with high levels of this parameter. For this reason the learning rate starts from a low value, 10^{-3}, then following this scheme (for the second training phase of 135 epochs):

1. The rate slowly increases in the first epochs (how many epochs depends on the implementation, it is not specified in the original paper) until it reaches the value of 10^{-2}.
2. Remains constant until epoch 75.
3. For the following 30 epochs a rate of 10^{-3} is applied.
4. For the last 30 epochs a learning rate of 10^{-4} is used.

13.6 YOLOv2

YOLOv2 strongly outperforms YOLO, and is based on YOLO adding some "tricks", described in the following.

13.6.1 Batch normalization

Batch normalization leads to significant improvements in convergence while eliminating the need for other forms of regularization: we eliminate dropout without overfitting. We get more than 0.02 improvement in mAP.

13.6.2 High resolution classifier

YOLOv2 fine-tunes the classification network at the full 448×448 resolution for 10 epochs on ImageNet. This gives the network time to adjust its filters to work better on higher resolution input. We then fine-tune the resulting network on detection. This high resolution classification gives an increase of 0.04 mAP.

13.6.3 DarkNet-19

The network DarkNet-19 is used. It is composed of 19 convolutional layers and 5 max-pooling layers. After each convolutional layer, we perform batch normalization with leaky ReLU activation function, $a = 0.01$.

The last convolution layer has been removed and instead the authors added the following layers:

Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7

13.7 YOLOv3

A further performance improvement is obtained with YOLOv3, it improves YOLOv2 using some heuristics; anyway the most important difference is the new net: Darknet-53, based on 53 convolutional layers. Moreover, YOLOv3 uses multi-label classification. For example, the output labels may be "pedestrian" and "child", which are not non-exclusive.

YOLOv3 replaces the softmax function of DarkNet-19 with independent logistic classifiers to calculate the likeliness of the input belonging to a specific label. YOLOv3 predicts an "objectness" score for each bounding box using logistic regression. In addition, the model changes the cost function.

Chapter 14

Neural style transfer

Neural style transfer (NST) refers to a class of software algorithms that manipulate digital images, or videos, in order to adopt the appearance or visual style of another image [79]. *Style transfer* is the problem of applying the style of one image to another image. In fact, this task needs two input images: one is the *content image* and the other is the *style image*. Figure 14.1 shows an example of neural style transfer. We can use neural networks to modify the content image to make it close to the style image in style.

14.1 CNN-based style transfer

Figure 14.2 illustrates the CNN-based style transfer method with a simplified example. First, we initialize the *synthesized image*, for example, as the content image. Differently from what we have already seen, the synthesized image is the only variable that needs to be updated during the style transfer process, i.e. the model parameters to be updated during training. Then we choose a pretrained CNN to extract image features and freeze its model parameters during training. The deep CNN uses multiple layers to extract hierarchical features for images. We can choose the output of some of these layers as *content features* or *style features*. As an example, the CNN in Figure 14.2 has 3 convolutionals layers, where the second layer outputs the content features, while the first and third layers output the style features.

Next, we calculate the loss function of style transfer through forward propagation and update the model parameters (i.e. the synthesized image for output) through backpropagation (direction of the dashed arrows). The loss function commonly used in style transfer consists of three parts:

- The *content loss* makes the synthesized image and style image close in content features.
- The *style loss* makes the synthesized image and style image close in style features.
- The *total variation loss* helps to reduce the noise in the synthesized image.

Finally, then the model training is over, we output the model parameters of the style transfer to generate the final synthesized image.

Figure 14.1: Neural style transfer applied to the Mona Lisa. Source: unavailable.

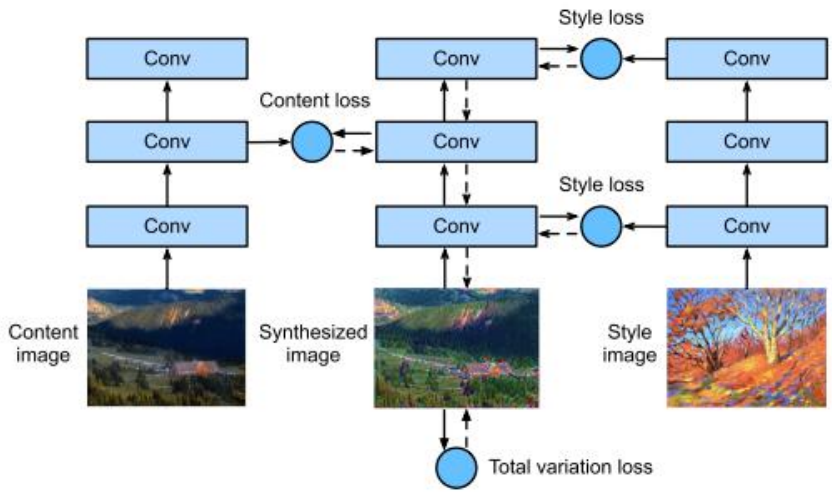


Figure 14.2: CNN-based style transfer process. Solid lines show the direction of forward propagation and dotted lines show backpropagation. Source: [129].

14.1.1 Content and style layers

As already seen before, in order to extract the content features and style features from the image, we can select the output of certain layers of the CNN. Generally speaking, the closer to the input layer, the easier to extract details of the image, and viceversa, the easier to extract global information of the image. In order to avoid excessively retaining the details of the content image in the synthesized image we choose a CNN layer that is closer to the output as the *content layer* to output the content features of the image. We also collect the output of different CNN layers for extracting style features, called *style layers*.

14.1.2 Content loss

Let f_{ij}^l be the activation function of the the l-th layer, i-th feature map and j-th position. Let $\mathbf{g}$ be the synthesized (generated) image and let $\mathbf{c}$ be the content image. Then the content loss for the l-th layer is:

$$J_{\text{content},l} = \frac{1}{2} \sum_{i,j} \left(f_{ij}^l(\mathbf{g}) - f_{ij}^l(\mathbf{c}) \right)^2 . \tag{14.1}$$

Therefore [78]:

$$J_{\text{content}} = \sum_l \frac{1}{2} \sum_{i,j} \left(f_{ij}^l(\mathbf{g}) - f_{ij}^l(\mathbf{c}) \right)^2 . \tag{14.2}$$

14.1.3 Style loss

Consider the style features output of a style layer. Suppose the output has 1 example (i.e. we are considering one image), c channels (feature maps), height h and width w. We can transform this output into a $c \times hw$ matrix $\mathbf{X}$. The row vector $\mathbf{x}_i$ represents the style feature of channel $i \in \{1, \ldots, c\}$.

We define the *Gram matrix* of $\mathbf{X}$ as the $c \times c$ matrix $\mathbf{X}\mathbf{X}^T$. In particular, the element in row i and column j is the dot product of vectors $\mathbf{x}_i$ and $\mathbf{x}_j$. It represents the correlation of the style features of channels i and j. Note that a larger value of hw leads to larger values in the Gram matrix. In addition, the height and the width of the matrix are both c. To allow style loss to not be affected by these values, we divide the Gram matrix by the number of its elements $c \cdot h \cdot w$.

Consider the i-th feature map and j-th position. Let $\mathbf{G}^{\mathbf{g},l}$ and $\mathbf{G}^{\mathbf{s},l}$ be the Gram matrices of the output of the l-th layer for the synthesized (generated) image $\mathbf{g}$ and style image $\mathbf{s}$, respectively. Then

the style loss is:

$$J_{\text{style}} = \sum_{l} \sum_{i,j} \left(\frac{1}{chw} G_{i,j}^{\mathbf{g},l} - \frac{1}{chw} G_{i,j}^{\mathbf{s},l} \right)^2 . \tag{14.3}$$

14.1.4 Total variation loss

Sometimes the learned synthesized image has a lot of high-frequency noise, i.e. particularly bright or dark pixels. One common noise refuction method is the *total variation denoising*. Denote by $x_{i,j}$ the pixel value at coordinate (i, j). Total variation denoising consists in minimizing the total variation loss:

$$J_{\text{total variation}} = \sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}|. \tag{14.4}$$

Minimizing $J_{\text{total variation}}$ makes the values of the neighboring pixels on the synthesized image closer.

Chapter 15

Capsule networks

A *capsule network* (CapsNet) is a type of artificial neural network that can be used to better model hierarchical relationships. In CNNs, the convolution layer holds a very important function that is to detect the features from an image. Deeper layers detect the simple features like edges and color. Although the performance of CNNs is really good, they still have some drawbacks. As quoted by Hinton, "The pooling operation used in CNNs is a big mistake, and the fact that it works so well is a disaster". In fact, there is a loss of valuable information when we are using pooling layers. CapsNets are networks able to fetch spatial information and more important features, overcoming the loss of information seen in pooling operations.

CapsNet introduces the *capsule*, which is different from a neuron of an ANN. Capsules returns a vector as an output that has a *direction*. For example, if we change the direction of an image, the vector moves in the same direction. Conversely, as we already know, the output of a neuron is a scalar quantity that does not tell anything about some direction.

15.1 CNNs problems

Consider the following example, reported in Figure 15.1. Say we rotate a boat or a face upside down and then feed it to a CNN: it would not be able to identify features like the eyes, nose or mouth. Similarly, if we reconstruct specific regions of the face, e.g. switch the positions of the nose and eyes, the network will still be able to recognize the face, even though it is not exactly a face anymore. In short, CNNs can learn the patterns of the images statistically, but not what the actual image looks like in different orientations. In deep learning, the activation level of a neuron is often interpreted as the likelihood of detecting a specific feature.

In summary, internal data representation of a CNN does not take into account important spatial hierarchies between simple and complex objects. This kinds of network cannot extrapolate their understanding of geometric relationships to new viewpoints.

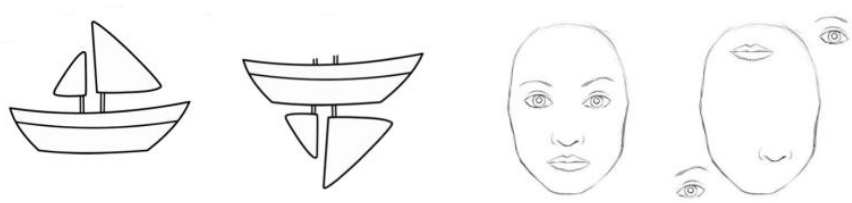


Figure 15.1: We can fool a *simple* CNN model in believing that the fourth draw is a good sketch. The boat is not recognized as a boat, but the weird face is classified as a face. Source: [72].

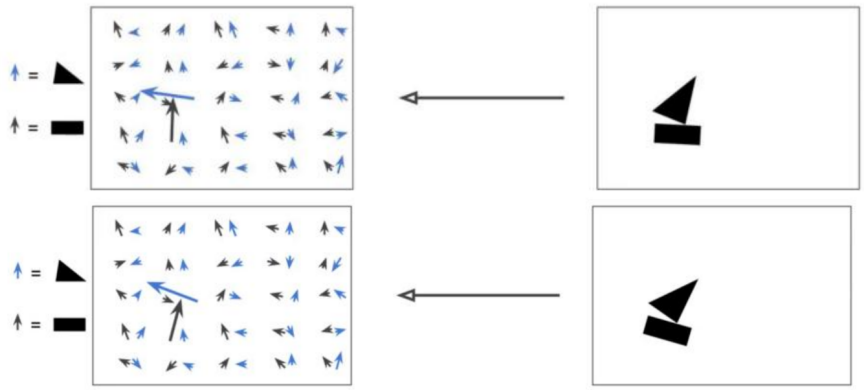


Figure 15.2: Example of equivariance. If the triangle and the rectangle rotate of some degrees, the probability (length of vector) is the same but the orientations of the vectors change. We say that it is an activities equivariance since the probability of the given capsule does not change, only the orientation changes. Source: [72].

15.2 Invariance and equivariance

Informally:

- An operator is *invariant* with respect to a transformation when the effect of the transformation is not detectable in the operator output.
- An operator is *equivariant* with respect to a transformation when the effect of the transformation is detectable in the operator output.

As we have already seen, pooling tries to make the neural network operations invariant to small changes in viewpoint. However, in this context, this is the wrong goal. It is better to aim for equivariace: we want that changes in the viewpoint lead to corresponding changes in neural activities.

15.3 Capsule network

A *capsule* is a group of neurons that perform a lot of internal computation and then encapsulate the results of these computations into a small vector of highly informative outputs. Each capsule learns to recognize an implicitly defined visual entity over a limited domain of viewing conditions and deformations. It outputs two things, embedded in the vector:

1. The probability that the entity is present within its limited domain.
2. A set of *instantiation parameters*, the generalized pose of the object. That may include the precise position, lighting and deformation of the visual entity relative to an implicitly defined canonical version of that entity.

Capsules encode probability of detection of a feature as the length of their output vector [96], while the state of the detected feature is encoded as the direction in which that vector points to. So when detected feature moves around the image or its state somehow changes, the probability still stays the same (length of vector does not change), but its orientation changes, as shown in Figure 15.2.

15.4 Computation

Let $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_I$ be the input vectors to the layer. We want to compute the j-th output vector of the layer $\mathbf{v}_j$, that is the capsule j. We compute $\hat{\mathbf{u}}_{j|1}, \hat{\mathbf{u}}_{j|2}, \ldots, \hat{\mathbf{u}}_{j|I}$ as:

$$\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i, \tag{15.1}$$

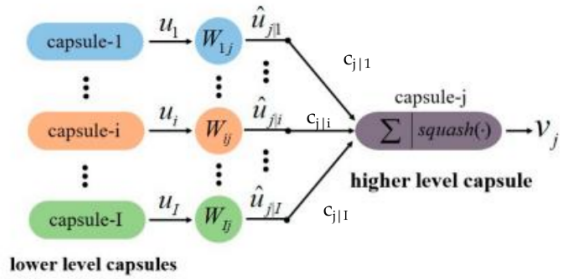


Figure 15.3: Capsule computation. Source: [72].

where $\mathbf{W}_{ij}$ is the weight matrix from capsule i to capsule j, learned during training. We compute $\mathbf{s}_j$ as:

$$\mathbf{s}_j = \sum_{i=1}^{I} c_{ij} \hat{\mathbf{u}}_{j|i}. \tag{15.2}$$

where $c_{ij} \in [0, 1]$ are coupling coefficients that are determined by the iterative dynamic routing process, such that

$$\sum_j c_{ij} = 1. \tag{15.3}$$

Conceptually, c_{ij} measures how likely capsule i may activate capsule j. Finally, we apply a non-linear *squashing* function to ensure that short vectors get shrunk to almost zero length and long vectors get shrunk to a length slightly below 1.

$$\mathbf{v}_j = \frac{\|\mathbf{s}_j\|^2}{1 + \|\mathbf{s}_j\|^2} \cdot \frac{\mathbf{s}_j}{\|\mathbf{s}_j\|}. \tag{15.4}$$

This procedure is graphically shown in Figure 15.3.

15.5 Spatial information

Consider the following figure:

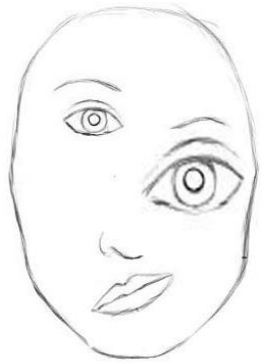


Figure 15.4: Source: [72].

We now briefly discuss how capsule are able to provide spatial information. The face in the figure does not look quite right for a face, even though it has all the necessary components to make up a face. We know that this is not how faces are supposed to look, but because CNNs only look for features in images and do not pay attention to their pose, it is hard for them to notice a difference between that face and a real face. How capsule networks solve this problem is by implementing groups of elements that encode spatial information as well as the probability of an object being present. The length of a capsule vector is the probability of the feature existing in the image and the direction of the

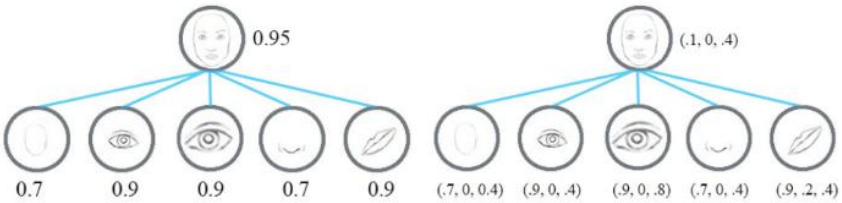


Figure 15.5: (Left) How a regular CNN would detect facial features. (Right) How a capsule network would detect features. While CNNs only "consider" probabilities (with some hypothesis on the activation function), capsule networks "consider" vectors that encode $[\text{probability}, \text{orientation}, \text{size}]^T$. For example, an eye is huge compared to the face (size 0.8 of the large eye with respect to 0.4 of the right size eye). Since a capsule contains information under the form of a vector, it can learn a richer representation of images. Source: [72].

vector would represent its pose information. We referred to this kind of information as instantiation parameters.

15.5.1 Instantiation parameters intuition

Intuitively, instantiation parameters may indicates that at position (x, y) there is an equilateral triangle of sides s rotated by an angle θ. In computer graphics applications such as design and rendering, objects are often created by giving some sort of parameter which it will render from. However, capsules networks aim to do the opposite, that is looking at an image and trying to predict what the instantiation parameters for it are.

15.5.2 Example of capsule information

As an example, consider that a capsule represent a vector of the form:

$$[\text{probability},\ \text{orientation},\ \text{size}]^T. \tag{15.5}$$

A regular CNN, with scalar as output of nodes, might work something like the diagram on the left of Figure 15.5, detecting a face even though an eye is huge relative to the face with 0.95 probability. Capsules, on the other hand, due to their richer vector information, see that the sizes of the features are different and therefore output a lower likelihood for the detection of the face. In fact, the previously considered face is classified as face with 0.1 probability.

15.6 Sample complexity

Capsules also have the advantage of being able to achieve good accuracy with far less training data than CNN. Since a capsule has vector information such as orientation, size, etc, it can use that information to learn a richer representation of the features as it is trained. It does not need 50 examples of the same rotated dog (see Chapter 17), it just needs one with a vector representation which can easily be transformed. By forcing the model to learn the feature variant in a capsule, we may extrapolate possible variants more effectively with less training data.

15.7 Affine transformation

We further analyze Equation 15.1, which is an affine transformation. Note that no bias is explicited. The input vectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_I$ represent either the initial input, or an input provided by an earlier layer in the network. These vectors are first multiplied by the weight matrices $\mathbf{W}_{1j}, \mathbf{W}_{2j}, \ldots, \mathbf{W}_{Ij}$, which are able to capture spatial relationships. In particular, these matrices model a *part-whole relationship* between lower and higher level entities [72, slide 354]. For instance, assume that one object is centered around another, and they are equally proportioned in size. For instance, if the low-level features are nose, mouth, left eye, and right eye, and if the predictions of the four low-level

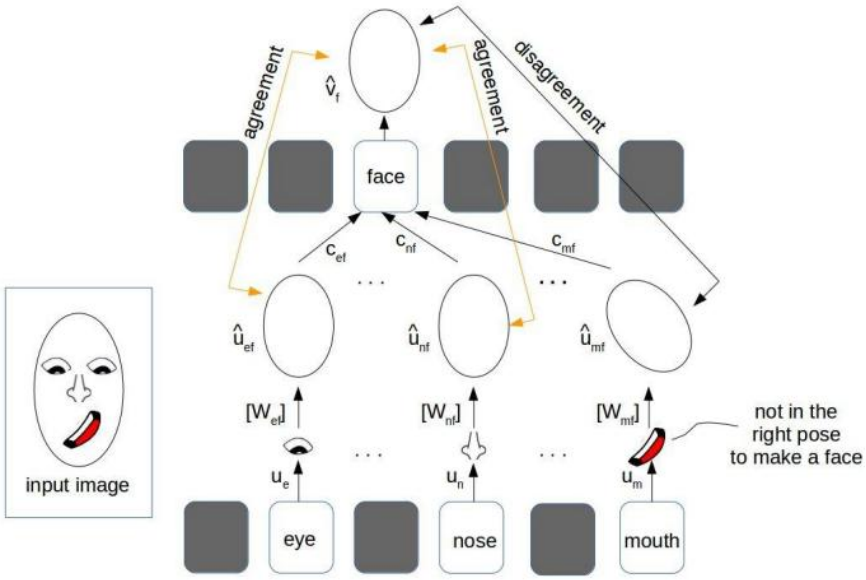


Figure 15.6: Example of routing by agreement. Source: [72].

features point to the same orientation and are reasonable with respect the size of a face, then a face will be predicted by the network.

Assume that $\mathbf{W}_{1j}$ encodes relationship between the nose and face. This relationships may be:

- The face is centered around its nose.
- The face has a size 10 times the size of the nose.
- The face orientation in space corresponds to the orientation of the nose, since they both lie on the same plane.

Similar intuitions can be drawn for the other matrices. After the affine transformations, we get the predicted position of the higher level feature. In other words, $\mathbf{W}_{1j}\mathbf{u}_1$ represents where the face should be according to the detected position of the nose, $\mathbf{W}_{2j}\mathbf{u}_2$ represents where the face should be according to the detected position of the mouth and $\mathbf{W}_{3j}\mathbf{u}_3$ represents where the face should be according to the detected position of the left eye, and so on.

15.8 Routing by agreement

When a visual stimulus is triggered, the human brain has an inbuilt mechanism to "route" low-level visual data to parts of the brain where it believes can handle it best. The analogous components of the routing by agreement (or dynamic routing, as we are going to see below) in CNN is max pooling. Max pooling, in some sense, is a coarse way of selectively route information in the previous layer to the next - among a small pool of neurons, only the maximum get routed to the next layer. Capsule networks are different, as they try to send low-level information to the high-level information capsule that is best at dealing with it. Another interpretation is that data is passed to the higher-level capsule that "agrees" the most with the lower-level capsule. This concept is referred to as *routing by agreement*. Concretely, "agreement" between a low-level capsule and an high-level capsule means an high value of c_{ij}. Conversely, "disagreement" between a low-level capsule and an high-level capsule means a low value of c_{ij}. An example of routing by agreement is shown in Figure 15.6.

15.8.1 Dynamic routing

Dynamic routing is an algorithm to compute the coupling coefficients proposed in [96]. They are computed for every layer at each forward pass [12], hence the name "dynamic". This means that, after

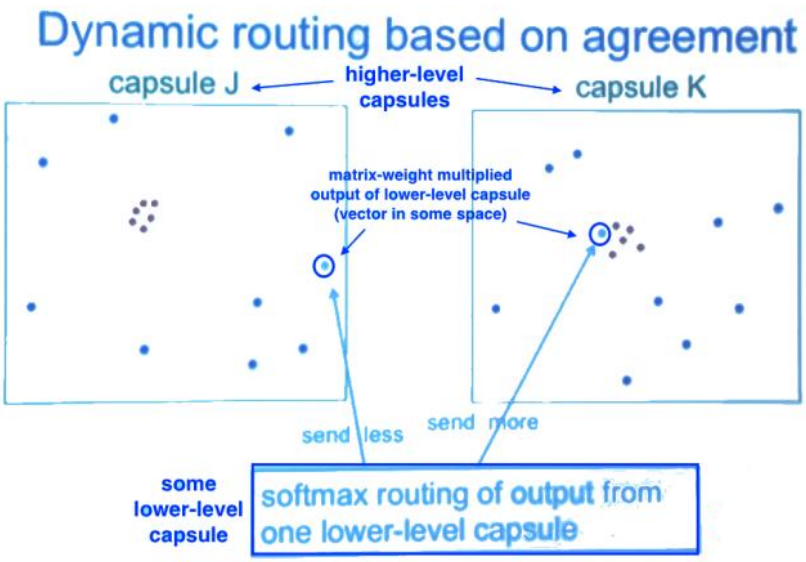


Figure 15.8: Agreement between capsules representation. The output of this lower-level capsule i is close to the cluster center of the output of other lower-level capsules of the capsule k. On the other hand, the output of this lower-level capsule i is far from the cluster center of the output of other lower-level capsules of the capsule j. Therefore b_{ij} will decrease, while b_{ik} will increase. In addition, we will probably have $c_{ij} < c_{ik}$. Source: [72].

computing $\mathbf{u}_{j|i}$, we compute the coupling cefficients and contextually $\mathbf{v}_j$. The algorithm is reported in Figure 15.7.

1: **procedure** ROUTING($\hat{\mathbf{u}}_{j|i}, r, l$)
2: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
3: **for** r iterations **do**
4: for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$
5: for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
6: for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$
7: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
return $\mathbf{v}_j$

Figure 15.7: Source: [96].

Now we are going to break down the algorithm. Before the r iterations, we initialize the *logits* $\mathbf{b}_{ij}$ as 0 (line 2), for every lower-level capsule i and higher-level capsule j. Then we update the coupling coefficients r times inside the loop. We also compute $\mathbf{v}_j$ this many times, however only the last is the output of capsule j. Let $\mathbf{b}_i = [b_{i1}, b_{i2}, \dots]^T$ and $\mathbf{c}_i = [c_{i1}, c_{i2}, \dots]^T$ refer to the capsule i. Then we apply the softmax activation function (line 5), i.e.

$$c_{ij} = \frac{e^{b_{ij}}}{\sum_k e^{b_{ik}}}. \tag{15.6}$$

In this way Equation 15.3 holds. Then, for each j, we compute $\mathbf{s}_j$ and $\mathbf{v}_j$ as already discussed.

In line 7 we update b_{ij} with the *agreement* a_{ij} between layer i and j:

$$a_{ij} = \langle \hat{\mathbf{u}}_{j|i}, \mathbf{v}_j \rangle. \tag{15.7}$$

Intuitively, if the "activity" vector $\mathbf{v}_j$ has close similarity with the prediction vector $\hat{\mathbf{u}}_{j|i}$, we conclude that capsule i is highly related with the capsule j (e.g. the eye capsule is highly related to the face capsule). We can visualize this concept in Figures 15.8.

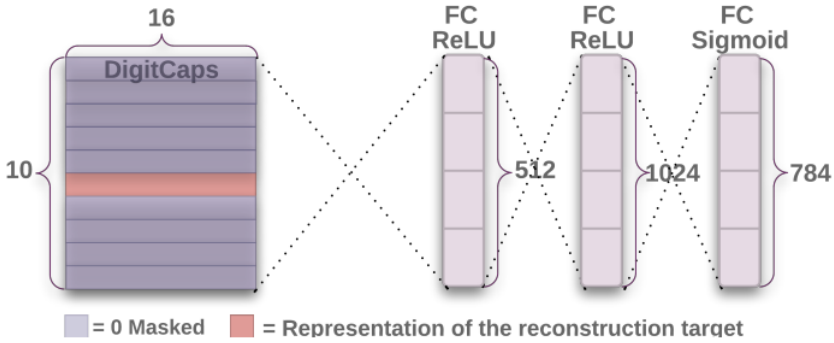


Figure 15.9: Consider the MNIST dataset, where each class is a digit from 0 to 9, totaling 10 classes. We refer to each output capsule as *DigitCaps*. The figure shows the decoder structure to reconstruct a digit from the DigitCaps layer representation. The euclidean distance between the image and the output of the sigmoid layer is minimized during training. We use the true label as reconstruction target during training. Each input image has a shape of 28×28 [71], and the output layer of the decoder has 784 nodes. Each node represents a pixel of the reconstructed 28×28 image, in fact $\sqrt{784} = 28$. Source: [72].

15.9 Loss function

We now present the *margin loss* and the *reconstruction loss*. At the end, the full loss used to train the network is:

$$J_{\text{total}} = J_{\text{margin}} + \alpha J_{\text{reconstruction}}, \tag{15.8}$$

where $\alpha = 0.0005$.

The general loss function will be now presented taking as a case study digit classification, further detailed in Section 15.10.

15.9.1 Margin loss

Consider the example $(\mathbf{x}, \mathbf{t})$, where $\mathbf{t} = [t_1, t_2, \cdots]^T$ is the ground-truth vector of labels, $t_c \in \{0, 1\}$. $t_c = 1$ if and only if $\mathbf{x}$ has class c. The loss function with respect to a single class is

$$J_c = t_c \max(0, m^+ - \|\mathbf{v}_c\|)^2 + \lambda(1 - t_c) \max(0, \|\mathbf{v}_c\| - m^-)^2. \tag{15.9}$$

Where $m^+ = 0.9$, $m^- = 0.1$ and $\lambda = 0.5$, down-weighting for absent digit classes, since $t_c = 1$ if the class is c. Intuitively, if the ground-truth class is c, then $\|\mathbf{v}_c\|$ should be no less than 0.9. Otherwise, $\|\mathbf{v}_c\|$ should be no more than 0.1. The complete *margin loss function* for a sample $\mathbf{x}$ is the sum of J_c for each c [12], i.e.

$$J = \sum_c t_c \max(0, m^+ - \|\mathbf{v}_c\|)^2 + \lambda(1 - t_c) \max(0, \|\mathbf{v}_c\| - m^-)^2. \tag{15.10}$$

15.9.2 Reconstruction loss

Capsule networks employ a *reconstruction loss function* as a regularization method to encourage the output capsules to *encode the instantiation parameters* of the input. We refer to the capsule network as *encoder*, and to a neural network with 3 fully-connected layer as *decoder*. During training, we mask out all but the activity vector of the correct output capsule (i.e. true label). Then we use this activity vector to reconstruct the input image. The output of the digit capsule is fed into a decoder consisting of 3 fully connected layers that model the pixel intensities as described in Figure 15.9. We minimize the sum of squared differences between the outputs of the logistic units and the pixel intensities. We scale down this reconstruction loss by 0.0005 so that it does not dominate the margin loss during training.

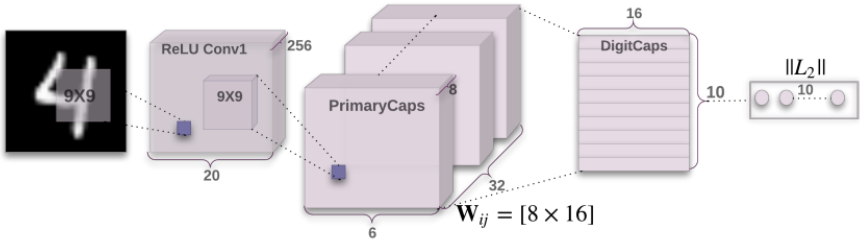


Figure 15.10: Capsule network architecture in [96]. The length of the activity vector of each capsule in DigitCaps layer indicates presence of an instance of each class and is used to calculate the classification loss. $\mathbf{W}_{ij}$ is a weight matrix between $\mathbf{u}_i$, $i \in (1, 32 \times 6 \times 6)$ in PrimaryCapsules and $\mathbf{v}_j$, $j \in (1, 10)$. Source: [96].

15.10 Capsule network for image recognition on MNIST

[96] proposed a capsule network architecture for digit classification on the MNIST dataset, where each class is a digit from 0 to 9, totaling 10 classes. The architecture is shown in Figure 15.10. The architecture is shallow (i.e. it has few hidden layers) with only two convolutional layers and one fully-connected layer. Conv1 has 256 kernels of size 9×9 with a stride of 1 and ReLU activation. This layer converts pixel intensities to the activities of local feature detectors that are then used as inputs to the *primary* capsules.

The primary capsules are the lowest level of multi-dimensional entities and, from an inverse graphics perspective, activating the primary capsules corresponds to inverting the rendering process. This is a very different type of computation than piecing instantiated parts together to make familiar wholes, which is what capsules are designed to be good at.

The second layer (PrimaryCapsules) is a convolutional capsule layer with 32 channels of convolutional 8D capsules (i.e. each primary capsule contains 8 convolutional units with a 9×9 kernel and a stride of 2). Each primary capsule output sees the outputs of all 256×81 Conv1 units whose receptive fields overlap with the location of the center of the capsule. In total PrimaryCapsules has $(32 \times 6 \times 6)$ capsule outputs (each output is an 8D vector) and each capsule in the (6×6) grid is sharing their weights with each other. One can see PrimaryCapsules as a Convolution layer with Equation (15.4) as its block non-linearity. The final Layer (DigitCaps) has one 16D capsule per digit class and each of these capsules receives input from all the capsules in the layer below.

We have routing only between two consecutive capsule layers (e.g. PrimaryCapsules and DigitCaps). Since Conv1 output is 1D, there is no orientation in its space to agree on. Therefore, no routing is used between Conv1 and PrimaryCapsules. All the routing logits (b_{ij}) are initialized to zero. Therefore, initially a capsule output (u_i) is sent to all parent capsules ($\mathbf{v}_0, ..., \mathbf{v}_9$) with equal probability (c_{ij}).

15.10.1 PrimaryCaps layer

We further explain the second convolutional layer or PrimaryCaps layer of the network, since it may be of hard interpretation. Firstly, the PrimaryCaps layer produces 256 activation maps of 6×6.

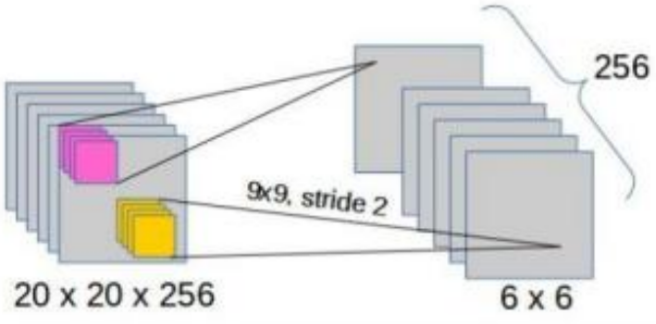


Figure 15.11: Source: [72].

Secondly, the output of the PrimaryCaps layer of shape $6 \times 6 \times 256$ is interpreted as a set of 32 *capsule activation maps* with capsule dimension 8.

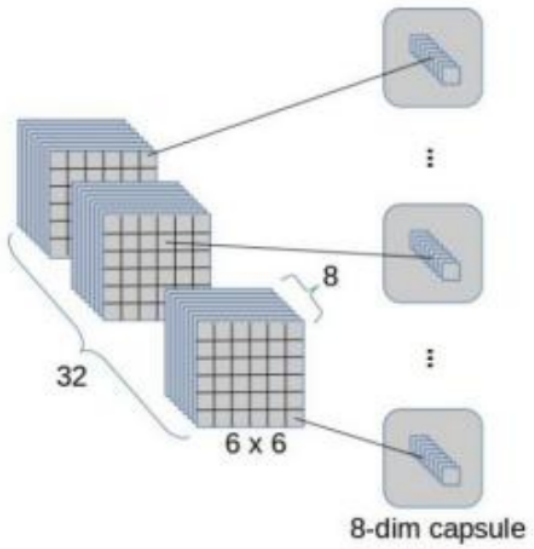


Figure 15.12: Source: [72].

A total of $6 \cdot 6 \cdot 32 = 1152$ 8-dimensional capsules are created. The 1152 lower level capsules are connected to the 10 higher level. Since the weights are shared among the capsules j of DigitCaps, there are 1152 c_{ij} coupling coefficients, 1152 b_{ij} coefficients and 1152 weight matrices $\mathbf{W}_{ij}$.

15.11 Other capsule-based network

Several other capsule-based networks have been proposed in the literature. One of them is *CNN-CapsNet*, which can be divided into two parts: a CNN and a CapsNet. First, an image is fed into a CNN model, and the initial feature maps are extracted from its layer. Then, the initial feature maps are fed into CapsNet described in Section 15.10.

We name a few other architectures and techniques realted to capsules:

- Transformer-based capsule routing algorithm.
- Ensemble of CNNs and capsule networks.
- Siamese capsule networks.
- Capsule networks for NLP.

Chapter 16

Ensemble of networks

Ensemble learning is the process by which multiple models, such as classifiers or experts, are strategically generated and combined to solve a particular computational intelligence problem. These models are usually used in *parallel*, but sometimes also in *cascade* or in a *hierarchical* way. Ensemble learning is primarily used to improve the (classification, prediction, function approximation, etc.) performance of a model, or reduce the likelihood of an unfortunate selection of a poor one [39].

The idea is the following:

- Train several models separately for the same task.
- At inference time we average the results. For this reason ensemble learning is often called *model averaging*.

The intuition behind ensemble learning is that different models make different errors on the test set. However, by averaging the results of, we obtain a more robust estimate without having a better model. This method works best if the models are maximally uncorrelated. The clear drawback is that it requires the evaluation of multiple models at inference time.

It has been demonstrated (theoretically but above all in practice) that the use of combinations of classifiers can improve performance, even a lot. In practice, investing time in the "push" optimization of a single classifier is generally less convenient than alongside (the initial classifier) other classifiers.

16.1 Independence

The combination of models is effective only when the individual classifiers are (at least partially) *independent* from each other, i.e. they do not make the same mistakes. Independence (or diversity) is normally achieved in the following ways:

- Using different features for training the classifiers.
- Using different image preprocessing approaches.
- Using different classification algorithms.
- Using different loss functions.
- Training the same classification algorithm on different portions of the training set (*bagging*).
- Insisting on training on incorrectly classified patterns (*boosting*).

The combination can be done at the decision level or at the confidence level.

16.2 Bagging

Bagging consists in training the same classification algorithm on different portions of the training set. Each portion has the same size of the original training set, therefore there are many repetition [7]. It consists of 4 phases [7]:

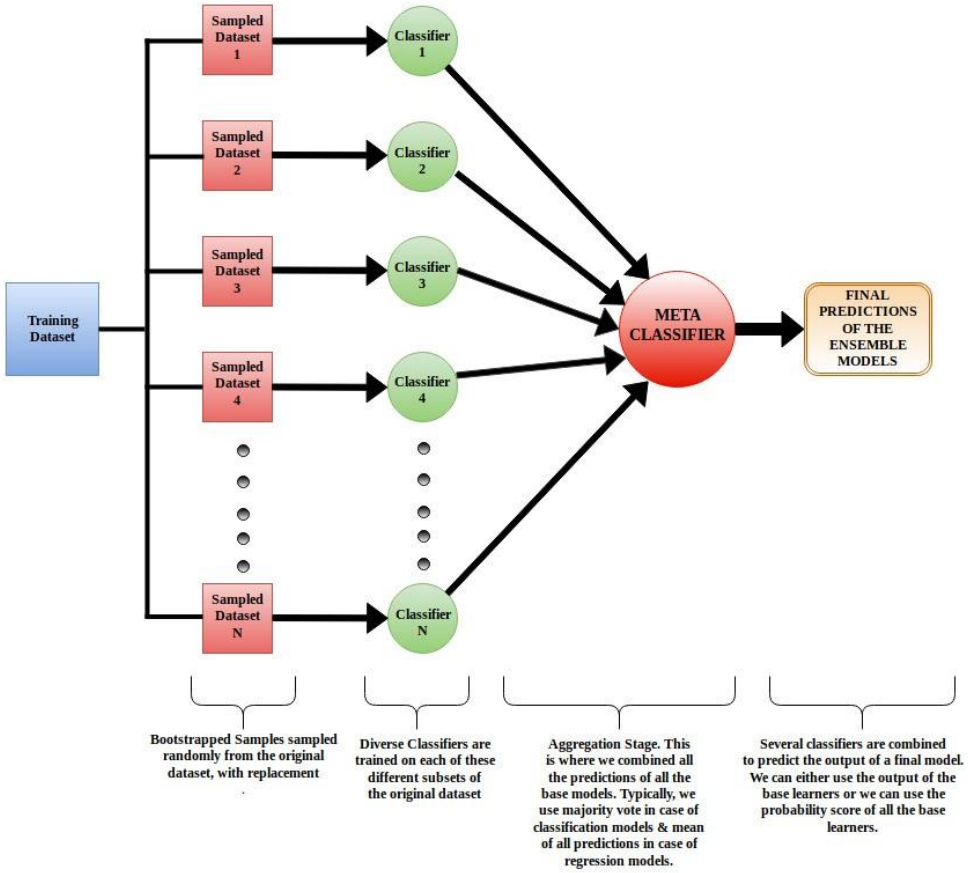


Figure 16.1: Illustration of the bagging method. Source: [72].

1. *Bootstrap sampling*, process of randomly sampling a dataset with replacement to generate various subsets known as bootstrap samples. The size of each subset is the same as the original dataset.

2. Base model training.

3. *Aggregation*, where the result of each base model is then aggregated via aggregation, which is commonly accomplished by taking the average for regression tasks or the majority for classification tasks.

4. Prediction.

An illustration of this procedure is shown in Figure 16.1.

16.3 Majority voting

In the following discussion, we assume that only the class labels are available from the classifier outputs. Let us define the decision of the t-th classifier as $d_{t,j} \in \{0, 1\}$, $t = 1 \ldots, T$ and $j = 1, \ldots C$, where T is the number of classifiers and C is the number of classes. If classifier t chooses class j, then $d_{t,j} = 1$, and 0, otherwise

There are three versions of majority voting, where the ensemble choose the class:

1. on which all classifiers agree (*unanimous voting*);

2. predicted by at least one more than half the number of classifiers (*simple majority*);

3. that receives the highest number of votes, whether or not the sum of those votes exceeds 50% of the total votes (*plurality voting* or *majority voting*). The ensemble decision for the plurality

voting chooses class $j^\star$:

$$j^\star = \max_{j \in \{1,\ldots,C\}} \sum_{t=1}^{T} d_{t,j}. \tag{16.1}$$

Figures 16.2 and 16.3 shows two examples of majority voting.

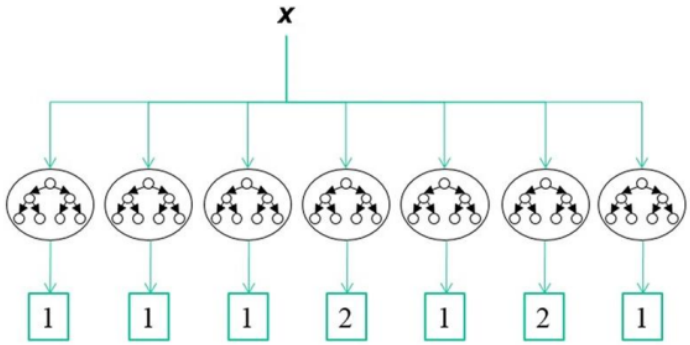


Figure 16.2: An ensemble of seven networks. Every network provides its own classification for the instance $\mathbf{x}$. Using majority voting, instance $\mathbf{x}$ should be classified as class 1 since five network out of seven vote for 1. Source: [72].

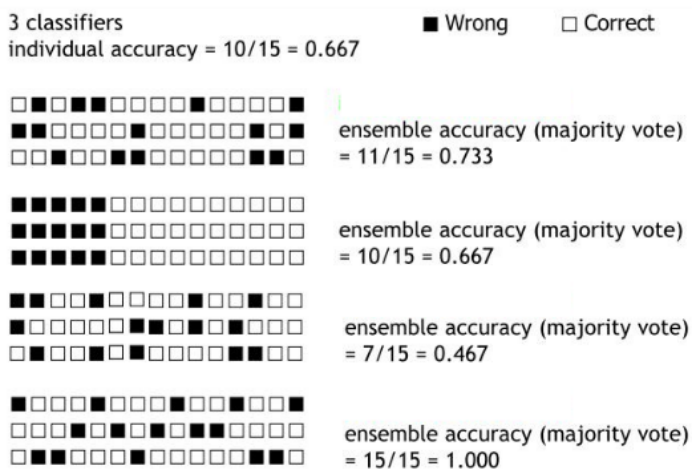


Figure 16.3: Example of majority voting in an ensemble of 3 classifiers. Each one of the four 3×15 tables of black and white squares represents whether the three classifiers (one per row) correctly classifiy a sample (one per column). Every table represents a different configuration of the classifiers. For example, table 2 represents the situation in which all three classifiers perform the exact same classification on all 15 samples. This means that, by majority voting, one sample is classified correctly if and only if in its column there are at least 2 white squares out of 3. As expected, the combination of models is effective only when the individual classifiers are independent from each other. Source: [72].

16.3.1 Classifiers weight

We can also assign a weight to each classifier, as shown in Figure 16.4. This weight is chosen during training or validation (e.g. using grid search).

16.4 Q-statistic

Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ be a labelled data set, coming from the classification problem in question. For each classifier D_i we design an n-dimensional output vector $\mathbf{y}_i = [y_{1,i}, \ldots, y_{n,i}]^T$ of correct classifica-

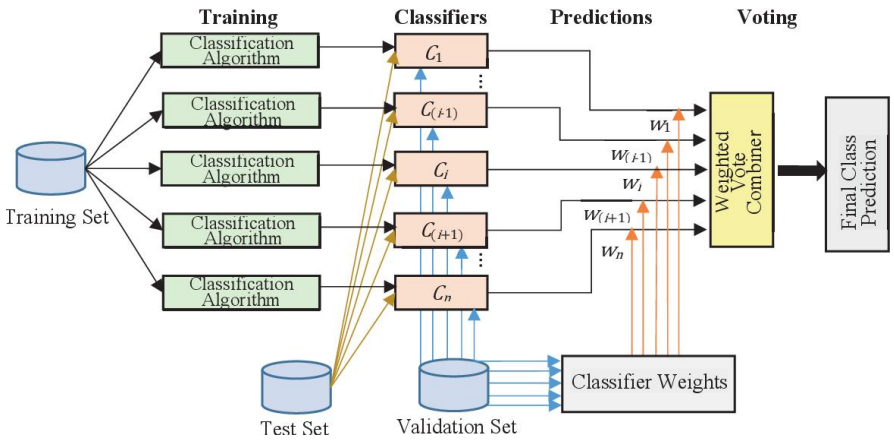


Figure 16.4: Schema of n weighted networks in an esemble. The weights are $W_1, W_2, \ldots, W_n$. Source: [72].

	D_k correct (1)	D_k wrong (0)
D_i correct (1)	N^{11}	N^{10}
D_i wrong (0)	N^{01}	N^{00}

Figure 16.5: A 2×2 table of relationship between a pair of classifiers.

tion, such that $y_{j,i} = 1$ if D_i recognises correctly $\mathbf{x}_j$ and 0, otherwise. There are various statistics to assess the similarity of D_i and D_k. Yule suggested that the Q-statistic should be used as a measure of association. The Q-statistic for two classifiers is [60]:

$$Q_{i,k} = \frac{N^{11}N^{00} - N^{01}N^{10}}{N^{11}N^{00} + N^{01}N^{10}}, \tag{16.2}$$

where N^{ab} is the number of elements $\mathbf{x}_j \in \mathbf{X}$ for which $y_{j,i} = a$ and $y_{j,k} = b$. For example, $N^{01} = 30$ means that there are 30 samples that are classified wrongly by D_i and correctly by D_k, as shown in Figure 16.5. The lower is Q, the more diverse the classifiers are [60].

For statistically independent classifiers D_i and D_j, we need $Q_{i,k} = 0$. Q is in $[-1, 1]$. Classifiers that tend to classify the same patterns correctly will have $Q > 0$, and those which commit errors on different patterns will have $Q < 0$ [73]. The correlation between two binary classifiers outputs $\mathbf{y}_i$ and $\mathbf{y}_j$ is

$$\rho_{i,k} = \frac{N^{11}N^{00} - N^{01}N^{10}}{\sqrt{(N^{11} + N^{10})(N^{01} + N^{00})(N^{11} + N^{01})(N^{10} + N^{00})}}. \tag{16.3}$$

For any two classifiers, Q and ρ have the same sign, and it can be proved that $|\rho| \leq |Q|$. [60] chooses Q to measure dependency because it has been designed for 2×2 contingency tables. It is also simpler to calculate from the table entries. Figure 16.6 shows the relationship between individual accuracy of each classifier, their diversity and the ensemble accuracy.

16.5 Combining continuous outputs

We propose now some alternative methods to 16.3 for combining the supports (i.e. the scores) of the networks of the ensemble. Let $d_{t,j}(\mathbf{x})$ be the support given by the t-th classifier to the j-th class for instance $\mathbf{x}$, w_j be the weight of the j-th classifier, T be the total number of classifiers, $\mu_j(\mathbf{x})$ be the total support for the j-th class for instance $\mathbf{x}$.

1. *Sum rule.* The total support for a class is calculated as the sum of the supports given to that class by all the classifiers. After calculating the total supports for all classes, the class with the

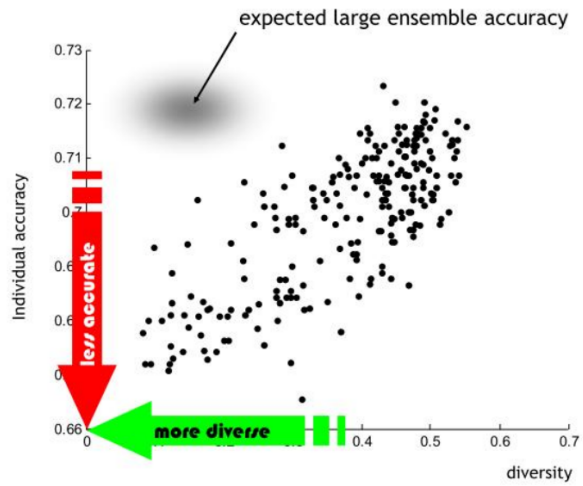


Figure 16.6: Relationship between individual accuracy of each classifier, their diversity and the ensemble accuracy. In the black spot we expect a large ensemble accuracy because the classifiers are diverse and have an high individual accuracy. Source: [72].

highest support is selected as the final output. In formulas:

$$\mu_j(\mathbf{x}) = \sum_{t=1}^{T} d_{t,j}(\mathbf{x}). \tag{16.4}$$

2. *Mean rule.* This rule is similar to the sum rule, but the total support is normalized by $1/T$:

$$\mu_j(\mathbf{x}) = \frac{1}{T} \sum_{t=1}^{T} d_{t,j}(\mathbf{x}). \tag{16.5}$$

3. *Weighted sum rule.* By this rule, the support for a class is calculated by the sum of the product of the classifiers' weight and their respective supports:

$$\mu_j(\mathbf{x}) = \sum_{t=1}^{T} w_t d_{t,j}(\mathbf{x}). \tag{16.6}$$

4. *Product rule.* The supports provided by the classifiers to a particular class are multiplied to obtain the final support for that class. This rule is very sensitive to the pessimistic classifiers as a low support (or even zero support) can remove any chance of getting selected by that class. The product is the most correct method by a probabilistic point of view, since we are computing a joint probability. However, the joint probability equals the product of probabilities in the case of statistical independence. Therefore, the sum is often preferable to the product as it is more robust.

$$\mu_j(\mathbf{x}) = \prod_{t=1}^{T} d_{t,j}(\mathbf{x}). \tag{16.7}$$

5. *Maximum rule.* This rule selects the maximum of all the supports of the different classifiers for a particular class:

$$\mu_j(\mathbf{x}) = \max_{t \in \{1,\ldots,T\}} d_{t,j}(\mathbf{x}). \tag{16.8}$$

6. *Minimum rule.* This rule selects the minimum of all the supports of the different classifiers for a particular class:

$$\mu_j(\mathbf{x}) = \min_{t \in \{1,\ldots,T\}} d_{t,j}(\mathbf{x}). \tag{16.9}$$

7. *Median rule.* This rule selects the median of the supports of the different classifiers for a particular class.

$$\mu_j(\mathbf{x}) = \operatorname*{median}_{t \in \{1,\ldots,T\}} d_{t,j}(\mathbf{x}). \tag{16.10}$$

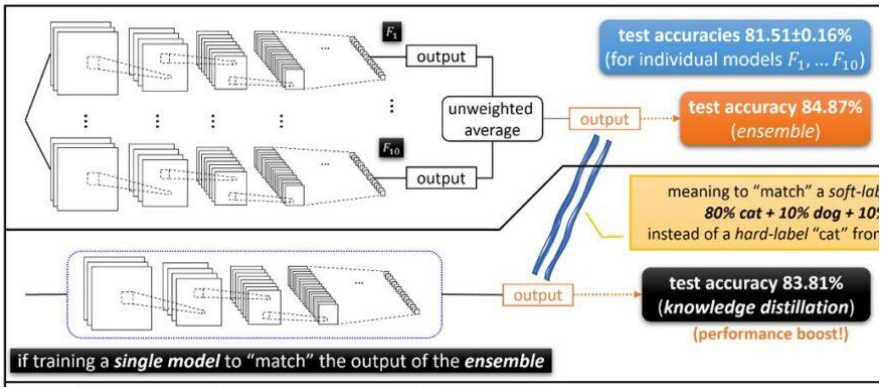


Figure 16.7: Applying knowledge distillation to a network to match the output of a 10 networks ensemble. Source: [72].

16.6 Knowledge distillation

An ensemble of n networks becomes n times slower during inference. To address this issue, we can use *knowledge distillation*, that trains a new randomly initialized model to match the output of the ensemble.

This method is also known as *student-teacher learning*. Where the teacher is the ensemble of networks and the student is the new model.

Chapter 17

Data augmentation

Data augmentation is a technique used to increase the amount of data by adding slightly modified copies of already existing training pattern or newly created synthetic data from existing patterns. It helps to reduce overfitting when training a machine learning model.

A problem present in the real world is *intra-class variations*, that is the appearance of a very wide variation between images of the same class, e.g. an animal may be presented from different angles or in different positions. To improve the classification performance we should train the network using images representing different variants of the given class, these training patterns are not always available, for this reason it is important to increase the size of the training set with artificially created images. By doing so the aim is to make the model invariant (not affected by) to input variations.

So the goal of data augmentation is to create "fake" data from the existing data (on the fly) and add it to the training set. This process is "on the fly" as we do not store the augmented images. The augmented images are generated randomly (by applying random transformations) from each image in our batch at the beginning of every training iteration.

New data must preserve semantics i.e. the augmented data of a class should not change the semantic meaning of the class category. For instance, augmenting the images of a label "dog" to look like cats is not a good idea.

Even simple operations like rotations, translations, adding per-pixel noise or the ones shown in Figure 17.1, can greatly improve generalization.

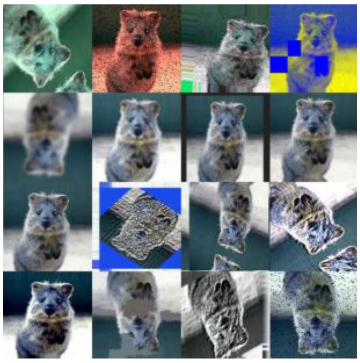


Figure 17.1: Simple random data augmentation examples. Source: [72].

However, data augmentation should be run considering the given problem. For example the standard approaches used in computer vision for image augmentation deteriorate the performance in audio classification. Since spectrograms have time and frequency on their axes, standard image augmentation techniques, e.g. reflection, will hardly make sense. For example, if a bird call is characterized by an increasing pitch over the duration of the call, its spectrogram augmented by reflection would represent a completely different sound where the pitch decreases over the duration of the call. This shows the importance of using specific augmentation techniques for audio signals and their spectrograms.

17.1 Data augmentation techniques

In the following part many data augmentation techniques will be presented. Additional information are given by [59] and by the popular GitHub library [28] providing also implementation.

17.1.1 Geometric transformation

We put in the category of geometric transformation the following operations:

- *Image cropping.* Images are randomly cropped and then re-scaled to original image size. It is important to not crop regions that are too small because semantics of the image category could be lost.
- *Horizontal image flipping.* Flip the given image horizontally based on a specified probability. It cannot be always used naively, as it is possible that semantic meaning of an image changes after flipping. For example, in a street scene by applying horizontal flipping notion of traffic lanes would change from right sided traffic to left sided traffic and vice-versa leading to possible undesirable results. Also depending on the task, images can be flipped upside-down.
- *Affine transformation.* Consists in applying a single affine/linear transformation to the 2D image space i.e. scale, rotate, shear an image. When blank spaces are left after the transformation they can be filled using different strategies like using constant color, expand image colors, replicate colors, etc. An example is presented by Figure 17.2.
- *Piecewise affine transformation.* Similar to Affine transformation, albeit now a single transformation is not applied to the whole image. The image is perceived as an underlying grid and a piece-wise affine transformation is applied independently to each grid cell. This causes different regions in the image to be distorted differently. An application is show in Figure 17.3.
- *Perspective transformation.* It is similar to Affine transformation, but it has more degrees of freedom. As a consequence some regions of the image end up being squeezed and others regions enlarged. An example is provided by Figure 17.4.

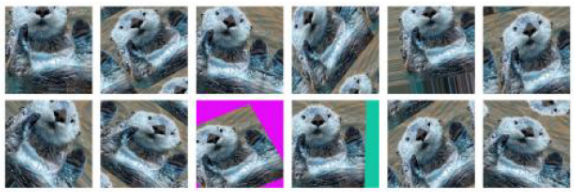


Figure 17.2: Example of affine transformation with filling of blank spaces. Source: [72].

Figure 17.3: Example of piecewise affine transformation. Source: [72].

17.1.2 Local filters

Inside the *local filters* category fall the following techniques.

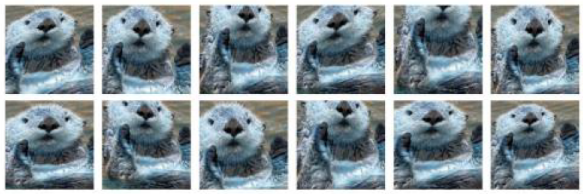


Figure 17.4: Example of perspective transformation. Source: [72].

- *Gaussian blur.* Applies the Gaussian blur filter (i.e. convolution of the image with a Gaussian function) onto the given image with the σ of the Gaussian distribution being chosen randomly from an interval specified by the user. It helps the model to recognize objects at different resolutions and to help the model being robust to camera-induced variations. In fact the process of capturing data through camera induce small amount of blur.
- *Image sharpening.* It does the opposite of introducing blur (i.e. introduces sharpness into input images).
- *Edge detection.* It uses the edge detection version of an image. An example is provided by Figure 17.6. It is important to verify whether the edge detected version are still reasonable and do not make it impossible to decipher semantics of the class.

Figure 17.5: Example of Gaussian blur application according to different σ values. Source: [72].

Figure 17.6: Example of edge detection application. Source: [72].

17.1.3 Adding noise

It is a popular data augmentation technique which typically involves introducing a more structured random per-pixel noise. In fact DNNs are highly sensitive to noise in images. Thus, it is very likely that adding even a small amount of noise (just to the test data) would lead to drastic reduction in

performance, even though to the human eye the "noisy" test set would look almost similar to the original test dataset.

Following are some of the most popular additive-noise augmentation techniques to improve generalization performance:

- *Gaussian noise*. Add a fixed Gaussian noise to each pixel in an image. The additive noise is chosen randomly for different images. Notice that this is different from Gaussian blur presented in Section 17.1.2, which used the convolution operation.
- *Salt and pepper noise*. Each pixel of an image is left as it is or turned into a black pixel or a white pixel depending on a specified probability.

17.1.4 Color transformation

Color transformations have played a key-role in the success of most NN that work well on the ImageNet dataset. In fact, cameras produce different color spectrum's based on the type of sensors used and the kind of white-balancing performed. A more important reason behind the success of these transformations is the change in lighting conditions while capturing data, e.g. the colors captured by a camera would change dramatically for the same scene depending on the time of the day (i.e. sunset would have warmer colors, mid-day would instead have brighter colors). We thus want our models to be invariant to such variations so that our generalization performance increases. This is exactly why color transformations are an essential form of data augmentation.

However, it is almost impossible for the training set to have all the possible variations of light that can occur in the images. Digital image data is usually encoded as a tensor of the dimension (height $\times$ width $\times$ color_ channels). Performing augmentations in the color channels space is another strategy that is very practical to implement. The RGB values can be easily manipulated with simple matrix operations to increase or decrease the brightness of the image. More advanced color augmentations come from deriving a color histogram describing the image. Changing the intensity values in these histograms results in lighting alterations such as what is used in photo editing applications.

Following are some of the most popular color transformation techniques.

- *Contrast*. Change the contrast of an image to either have it become faint or have stronger contrast.
- *Brightness*. Change the brightness of the whole images to make them either darker or brighter.
- *Brightness per channel*. Instead of changing brightness of the whole image change brightness for each channel instead. This results in images such as the ones presented in Figure 17.7.

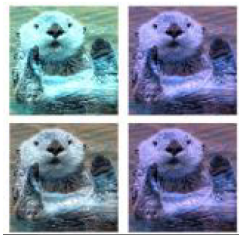


Figure 17.7: Example of changing brightness only for some channels. Source: [72].

17.1.5 Weathers

As the name suggests the idea is to simulate the images as if a specific weather was present when the image was taken. And even though they might sound complex they are still very easy to compute and can be done on-the-fly without requiring any sophisticated graphics engine. Some examples are introducing snow, clouds or fog. In Figure 17.8 is shown a simple example of snow-like effect.

Figure 17.8: Example of snow weather effect. Source: [72].

17.1.6 Elastic deformation

Elastic deformation transforms a given image by applying a randomly generated *displacement field* (i.e. set of vectors indicating a movement of a point) to its pixels by a value extracted from the standard uniform distribution in the range $[-1, 1]$. The resulting horizontal and vertical displacement fields are passed through a low-pass filter to smooth the image. An example of elastic transformation application is given by Figure 17.9.

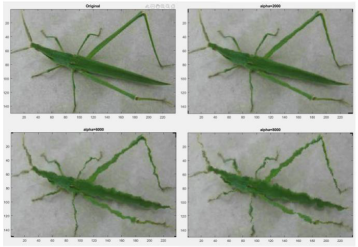


Figure 17.9: Example of elastic transformation. Source: [72].

17.2 Applications

Data augmentation should be considered as a part of the network design. On the other hand, it is important when comparing two networks to make sure you use the same augmentation. This is because data augmentation is an extremely powerful strategy and could significantly improve performance. If the same augmentation strategy is not used in both networks, one could be misled into thinking that their idea was responsible for better performance, whereas in reality it was just a better data augmentation strategy.

Choosing a data augmentation strategy means specifying the right distributions (often done empirically), choose some of the techniques with their hyperparameters and combine them. The combinations are then applied on-the-fly to different images drawn randomly from within a mini-batch. Because not all kind of transformations help improving performance they are chosen empirically, as well as the hyperparameters.

The idea of data augmentation can be combined with ensemble, as shown in Figure 17.10.

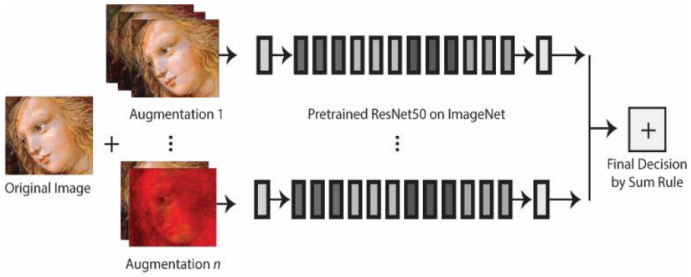


Figure 17.10: Combination of data augmentation with ensamble. Source: [72].

An example is provided by [76]. The novelty of this research is to consider different strategies for data augmentation to generate training sets from which to train several classifiers that are combined into an ensemble. Specifically, the idea is to create an ensemble based on a kind of bagging of the training set, where each model is trained on a different training set obtained by augmenting the original training set with different approaches. They built ensembles on the data level by adding images generated by combining fourteen augmentation approaches, with three based on Fourier Transfrom, PCA, and others proposed here for the first time. Pretrained ResNet50 networks are finetuned on training sets that include images derived from each augmentation method. These networks and several fusions are evaluated and compared across eleven benchmarks. Results show that building ensembles on the data level by combining different data augmentation methods produce classifiers that not only compete competitively against the state-of-the-art but often surpass the best approaches reported in the literature.

17.3 AutoAugmented

In [25] a new way, called AutoAugmented, to automatically find augmenting strategies has been introduced. This is done thanks to reinforcement learning.

Chapter 18

Graph Convolutional Networks

18.1 Graphs

Graphs have several applications like representing knowledge, chemical compounds, images, malware API calls and many others.

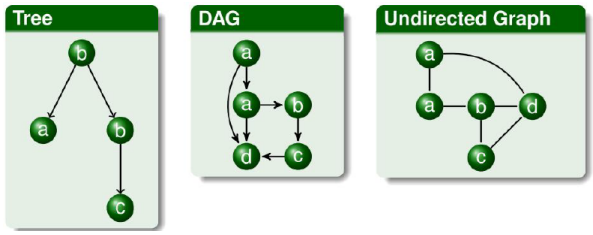


Figure 18.1: Commonly used types of graphs: Trees, Directed Acyclic Graphs (DAG) and undirected graphs. Source: [72].

Data that can be described as sequences such as vectors, grids, and relational databases we say we are working with *structured data*. Graphs, by contrast, are *unstructured*. They can take any shape or size and contain any kind of data. An idea of the difference between structured and unstructured data is given in Figure 18.2.

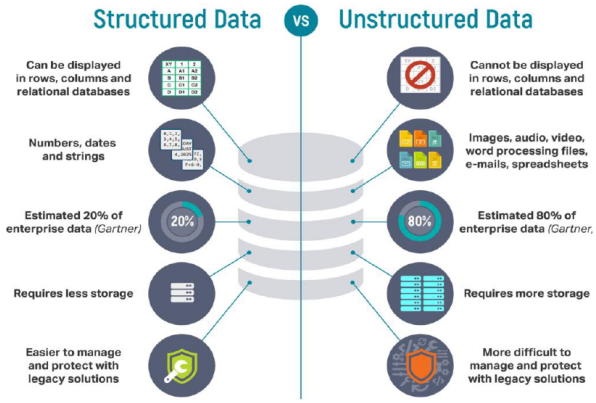


Figure 18.2: Considerations about structured and unstructured data. Source: [72].

Images or words are structured data only when such objects are described by means of a vector, matrix or tensor; but they are generally not considered structured data. So graphs can contain them.

A graph can be represented as a triplet $G = (V, E, W)$ with *vertices* V, *edges* E and *weights* W. Vertices (or nodes) are defined as a set of N numerical labels $V = \{1, \ldots, N\}$. Edges are ordered pairs of these numerical labels (i, j), where $(i, j) \in E$ is interpreted as "node i is influenced by node j".

Weights $w_{ij} \in \mathbb{R}$ are numbers associated to edges (i, j) that determine the strength of the influence that node j has on node i. Depending on these basic properties, a graph can be classified as a directed graph or an undirected (or symmetric) graph.

In the case of a *directed graph* the edge (i, j) differs from edge (j, i). Thus, a connection between two nodes can be one-way only, meaning that $(i, j) \in E$ and $(j, i) \notin E$. Furthermore, if a connection between two nodes is bidirectional $\{(i, j), (j, i)\} \subseteq E$, their weights can be different $w_{ij} \neq w_{ji}$.

The *undirected* or *symmetric graph* is a directed graph with the special property that its edge set and its weights are symmetric. Therefore, if the edge set E contains (i, j) it implies that $(j, i) \in E$ as well.

A special case in directed and undirected graphs is the *unweighted graph*, where all existing connections $(i, j) \in E$ have weights $w_{ij} = 1$. Since the weights of a graph convey valuable information as a prior for GNN, most graphs that are encountered in the context of DL are weighted.

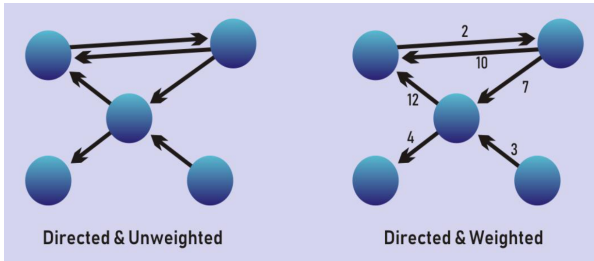


Figure 18.3: Examples of directed weighted and unweighted graphs. Source: [72].

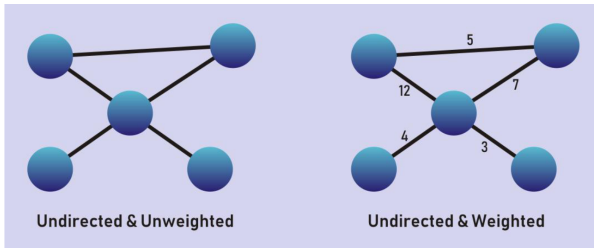


Figure 18.4: Examples of undirected weighted and unweighted graphs. Source: [72].

18.1.1 Graph tasks

GNN have gained popularity also thanks to their applications in fields such as Computer Vision and Natural Language Processing. For the first, GNN architectures can be applied to image classification problems or *scene graph generation*, in which the model aims to parse an image into a semantic graph that consists of objects and their semantic relationships. So, given an image, scene graph generation models detect and recognize objects and predict semantic relationships between pairs of objects. While for NLP, we know that the text is a type of sequential data which can be described by an RNN or an LSTM. However, graphs are heavily used in various NLP tasks, due to their naturalness and ease of representation. Recently, there has been a surge of interest in applying GNNs for a large number of NLP problems like text classification, exploiting semantics in machine translation, user geolocation, relation extraction, or question answering. Every node is an entity and edges describe relations between them.

Example of graphs tasks (also shown in Figure 18.5) are:

- *Graph classification.* Given a graph, predict to which of a set of classes it belongs.
- *Node classification.* Given a graph with incomplete node labeling, predict the class of the remaining nodes.
- *Link prediction.* Given a graph with incomplete adjacency matrix, predict for each pair of nodes whether they are connected.

- *Community detection (a.k.a. Graph clustering).* Given a graph, partition its nodes into clusters based on its edge structure.
- *Graph embedding.* Given a graph, map it into a vector while preserving relevant information.
- *Graph generation.* Learn distribution of a set of given graphs, and sample from this distribution to generate new similar graphs.

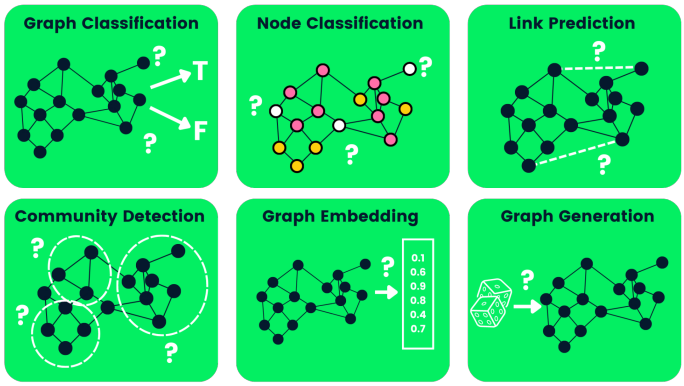


Figure 18.5: Some of the types of GNN tasks. Source: [16].

Graph Neural Networks (GNNs) are special types of neural networks capable of working with a graph data structure. The majority of GNN are *Graph Convolutional Networks* (GCNs)[16].

18.2 Graph Convolutional Networks

18.2.1 Graph matrix representation

The visual graph with its connections can be mathematically represented with the help of different matrices. A common representation of the edges of a graph G is the *adjacency matrix* $\mathbf{A}$. In this sparse $N \times N$ matrix, each row and each column represent a vertex and each entry A_{ij} contains the weight w_{ij} of all connections $(i, j) \in E$. Notice that we will not allow multi-edges between two nodes. Furthermore if the graph is undirected it holds that $\mathbf{A} = \mathbf{A}^T$.

Particular care has to be given to the interpretation of entry w_{ij} in the adjacency matrix, because the meaning of w_{ij} can vary depending on the convention used in different sources or contexts. Both interpretations are valid, but they represent different conventions and can lead to different interpretations of the matrix. In one convention, w_{ij} represents the weight of the edge connecting vertex i to vertex j. A different notation, where w_{ij} represents the weight of the edge connecting vertex j to vertex i, is sometimes referred to as the *transpose adjacency matrix notation* (Figure 18.6).

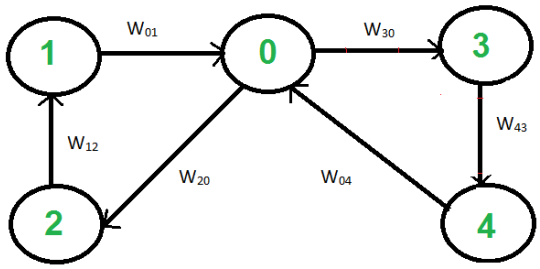


Figure 18.6: Example of transpose adjacency matrix notation. Source: unavailable.

When considering unweighted and undirected graphs, the *degree matrix* $\mathbf{D}$[1] is a diagonal matrix that contains information about the degree of each vertex, that is, the number of edges attached to

[1] In the lecture notes [72] and in the exercises, we often have $\mathbf{D} = \mathbf{D}_{\text{in}}$.

each vertex. While when considering unweighted directed graphs, the term degree may refer to in-degree (the number of incoming edges at each vertex) or out-degree (the number of outgoing edges at each vertex). For weighted directed graphs the above definitions have been extended by considering, instead of the number of edges, the sum of the weights. As a consequence, for directed weighted graphs the *out-degree* of the node i is obtained by the following sum (considering w_{ij} as the cost of arc from i to j):

$$d_i = \sum_{j \in n(i)} w_{ij}. \tag{18.1}$$

Where

$$n(i) = \{j \mid (i,j) \in E\}. \tag{18.2}$$

The *in-degree* can be computed in the same way, but using transpose adjacency matrix notation that was described above. The diagonal D_{ii} therefore contains the degree d of vertex i: $D_{ii} = d_i$.

The *Laplacian matrix* $\mathbf{L}$, when working with undirected graphs, combines matrices $\mathbf{D}$ and $\mathbf{A}$ as follows (Figure 18.7):

$$\mathbf{L} = \mathbf{D} - \mathbf{A}. \tag{18.3}$$

If we are considering no self-loop it therefore contains $\mathbf{D}$ on its diagonal and $\mathbf{A}$ negated on its off-diagonals. So, written explicitly in terms of graph weights the entries consist of $L_{ij} = -A_{ij} = -w_{ij}$ and $L_{ii} = d_i$.

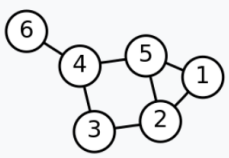
Labelled graph	Degree matrix	Adjacency matrix	Laplacian matrix
	$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$

Figure 18.7: Computations of the adjacency, degree and Laplacian matrices for undirected unweighted graph. Source: [63].

The definition is generalized for directed graphs (Figure 18.8) considering the out-degree matrix $\mathbf{D}_{\text{out}}$ and the in-degree matrix $\mathbf{D}_{\text{in}}$. Thus we have

$$\mathbf{L}_{\text{out}} = \mathbf{D}_{\text{out}} - \mathbf{A}, \tag{18.4}$$

$$\mathbf{L}_{\text{in}} = \mathbf{D}_{\text{in}} - \mathbf{A}. \tag{18.5}$$

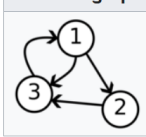
Labelled graph	Adjacency matrix	Out-Degree matrix	Out-Degree Laplacian	In-Degree matrix	In-Degree Laplacian
	$\begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 2 & -1 & -1 \\ 0 & 1 & -1 \\ -1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & -1 & -1 \\ 0 & 1 & -1 \\ -1 & 0 & 2 \end{pmatrix}$

Figure 18.8: Computations of the adjacency, in-degree, out-degree, in-degree Laplacian and out-degree Laplacian matrices for directed unweighted graph. Source: [61].

Both the Laplacian and adjacency matrix can be normalized to achieve a more homogeneous representation of a graph. That is especially helpful for asymmetric graphs, where some nodes have a lot of neighbors and/or a high degree and some nodes have only a few neighbors and/or a low degree. The *normalized adjacency matrix* expresses weights relative to node degrees. Let $\bar{\mathbf{A}}$ be the normalized adjacency matrix, then

$$\bar{A}_{ij} = \frac{w_{ij}}{\sqrt{d_i \cdot d_j}}. \tag{18.6}$$

The *normalized Laplacian matrix* is similarly defined as:

$$\bar{\mathbf{L}} = \mathbf{I} - \bar{\mathbf{A}}, \tag{18.7}$$

where $\mathbf{I}$ is the identity matrix.

18.2.2 Graph signal

The Laplacian matrix, the adjacency matrix and the normalized forms of both are in the following represented by the *Graph Shift Operators* (GSO) $\mathbf{S}$, that is a $N \times N$ matrix. That is, because for the theoretical analysis of GNN the specified graph matrix representation is irrelevant. Nevertheless during deployment the specific representation matters and leads to different results. It also holds that if G is symmetric, then $\mathbf{S} = \mathbf{S}^T$. Many other GSO are proposed, for example [27] introduces a new way to represent graphs thanks to Parametrized Graph Shift Operators (PGSO). The PGSO is suggested as a replacement of the standard GSOs that are used in state-of-the-art GNN architectures and the optimisation of the PGSO parameters is seamlessly included in the model training.

Graph Signal Diffusion

We now describe *Graph Signal Diffusion*. Given the mathematical representation of a graph G as $\mathbf{S}$ that captures the structure of this graph, a *graph signal* can be defined as a vector $\mathbf{x} \in \mathbb{R}^N$ that assigns a value $x_i \in \mathbb{R}$ to every node i. Graph signal can also be seen as a function $f : V \to \mathbb{R}$. To emphasize that the graph is instric to the signal we may write the signal as a pair $(\mathbf{S}, \mathbf{x})$. We can consider $\mathbf{S}$ to encode the expected proximity or similarity between components of $\mathbf{x}$. A multiplication $\mathbf{Sx}$ yields to a *diffused signal* $\mathbf{y}$ over G:

$$\mathbf{y} = \mathbf{Sx}. \tag{18.8}$$

If $\mathbf{S}$ is the adjacency matrix that would yield to $y_i = \sum_j w_{ij} \cdot x_j$. While, when $\mathbf{S}$ is the degree matrix we have a local operation $y_i = \sum_{j \in n(i)} w_{ij} \cdot x_j$, only the nodes of the neighborhood of the given node are considered for updating its value. For further explanation see Figure 18.9.

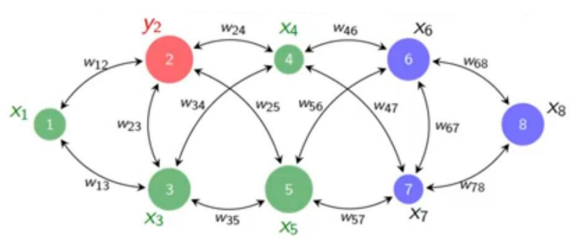


Figure 18.9: Source: [72].

The operation mixes the neighboring values of x_i and diffuses its signal along the edges over G. Since one application of $\mathbf{S}$ moves the signal of $\mathbf{x}$ only one edge at a time a *diffusion sequence* over k steps can be defined as:

$$\mathbf{x}_{k+1} = \mathbf{Sx}_k \tag{18.9}$$

with $\mathbf{x}_0 = \mathbf{x}$ or equivalently in form of a power sequence:

$$\mathbf{x}_{k+1} = \mathbf{S}^k \mathbf{x} \tag{18.10}$$

where $\mathbf{A}^0 = \mathbf{I}$, $\mathbf{A}^1 = \mathbf{A}$, $\mathbf{A}^k = \underbrace{\mathbf{AA} \ldots \mathbf{A}}_{\text{k times}}$.

In practice we can see graph signal $\mathbf{x}$ as a vector whose elements are stored in the nodes and the graph as an expectation of similarity between the elements of $\mathbf{x}$, that is, if there is an edge between two elements of $\mathbf{x}$ we can assume that these two elements are more similar than two nodes that are not connected by an edge.

18.2.3 Graph convolution

Graph convolution filters are the basic building block of a GCN. Given $\mathbf{S}$ and *filter coefficients* h_k, a graph convolution is a polynomial on $\mathbf{S}$:

$$\mathbf{H}(\mathbf{S}) = \sum_{k=0}^{\infty} h_k \mathbf{S}^k. \tag{18.11}$$

Where the filter coefficient h_k determines the importance of a particular *diffusion state* k. The result of applying filter $\mathbf{H}(\mathbf{S})$ to the signal $\mathbf{x}$ is the signal:

$$\mathbf{y} = \mathbf{H}(\mathbf{S})\mathbf{x} = \sum_{k=0}^{\infty} h_k \mathbf{S}^k \mathbf{x}. \tag{18.12}$$

In practice a finite amount of diffusion states $K-1$ is observed and it yields:

$$\mathbf{y} = \mathbf{H}(\mathbf{S})\mathbf{x} = \sum_{k=0}^{K-1} h_k \mathbf{S}^k \mathbf{x}. \tag{18.13}$$

As a short hand notation this is written as

$$\mathbf{y} = \mathbf{h} \star_{\mathbf{S}} \mathbf{x}, \tag{18.14}$$

where $\star_{\mathbf{S}}$ denotes the *graph convolution operation* of filter $\mathbf{h}$, on signal $\mathbf{x}$ and graph shift operator $\mathbf{S}$. A graph convolution can be visualized in a *block diagram* (Figure 18.10).

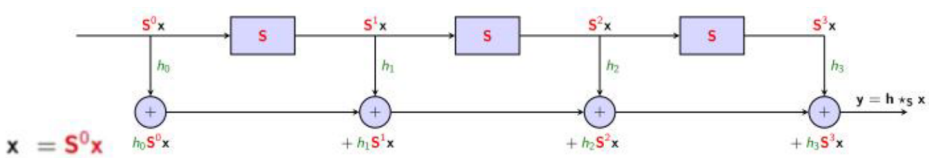


Figure 18.10: Diagram block in which iteratively a shift of $\mathbf{S}$, a weighting of $\mathbf{S}^k\mathbf{x}$ with h_k and a summation of the weighted terms is applied. Source: [72].

Local and global information tradeoff

Equation (18.13) embeds a tradeoff between local and global information. As shown in Figure 18.11, the k-th element of the diffusion sequence $\mathbf{x}_k$ diffuses information to *k-hop neighborhoods*.

Similitude between convolution in CNN and GCN

It can help to intuitively think that there is a big similitude between convolution in CNN and GCN, as shown in Figure 18.12.

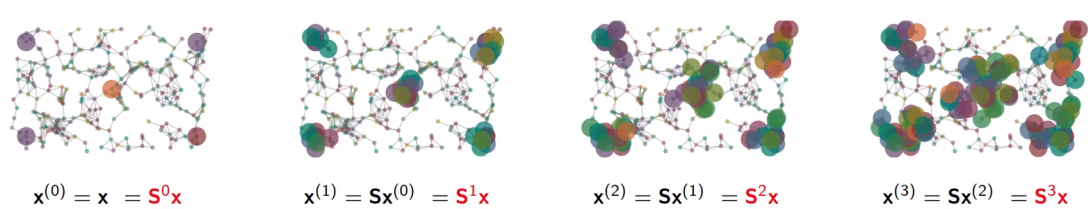


Figure 18.11: Local and global information tradeoff. Source: [72].

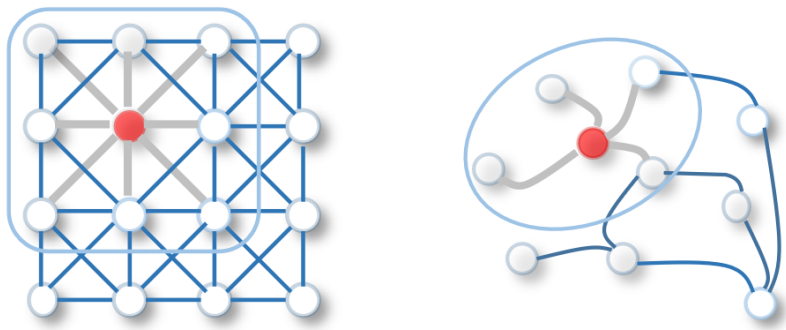


Figure 18.12: To the left we have 2D convolution. 2D Convolution. Analogous to a graph, each pixel in an image is taken as a node where neighbors are determined by the filter size. The 2D convolution takes the weighted average of pixel values of the red node along with its neighbors. The neighbors of a node are ordered and have a fixed size. To the right is pictured convolution on graphs. To get a hidden representation of the red node, one simple solution of the graph convolutional operation is to take the average value of the node features of the red node along with its neighbors. Different from image data, the neighbors of a node are unordered and variable in size. Source: [127].

18.2.4 Permutation equivariance

We now show that graph convolutional filters are *equivariant to permutations*. We say that a square matrix $\mathbf{P}$ is a *permutation matrix* if it has binary entries so that $\mathbf{P} \in \{0,1\}^{n\times n}$ and has exactly one entry of 1 in each row and each column with all other entries 0. An $n \times n$ permutation matrix represents a permutation of n elements.

The product $\mathbf{P}^T\mathbf{x}$ *reorders* the entries of the vector $\mathbf{x}$. The product $\mathbf{P}^T\mathbf{S}\mathbf{P}$ is a consistent reordering of the rows and columns of $\mathbf{S}$. If $(\mathbf{S}, \mathbf{x})$ is a graph signal, then $(\mathbf{P}^T\mathbf{S}\mathbf{P}, \mathbf{P}^T\mathbf{x})$ is a *relabeling* of $(\mathbf{S}, \mathbf{x})$. It represents the same signal, but with different node names.

Consider consistent permutations of the shift operator $\hat{\mathbf{S}} = \mathbf{P}^T\mathbf{S}\mathbf{P}$ and input signal $\hat{\mathbf{x}} = \mathbf{P}^T\mathbf{x}$. Then:

$$\mathbf{H}(\hat{\mathbf{S}})\hat{\mathbf{x}} = \mathbf{P}^T\mathbf{H}(\mathbf{S})\mathbf{x}. \tag{18.15}$$

This means that graph filters are equivariant to permutations. If we permute or shift the input and transform the GSO accordingly it results in a permutation or shift of the output.

18.2.5 Learning graph filters

After knowing what graph convolutions are, an algorithm for learning the filter coefficients $\mathbf{h}$ of this graph convolution operation can be defined. This is equivalent to training a GCN with a single layer only. Let $\mathcal{S} = \{(\mathbf{x}_i, \mathbf{y}_i, \mathbf{S}_i)\}$ denote a dataset of input signals $\mathbf{x}_i$, output signals $\mathbf{y}_i$ and GSO $\mathbf{S}_i$. It is important to note that a dataset can contain various GSO $\mathbf{S}_i$. Thus, a single graph filter can be trained on multiple graphs in order to become more robust during deployment even for unseen graph representations. The graph convolution is denoted by $f_{\mathbf{h}}(\mathbf{x}, \mathbf{S})$. A prediction of an output signal $\hat{\mathbf{y}}$ can be calculated using the following equation:

$$\hat{\mathbf{y}} = f_{\mathbf{h}}(\mathbf{x}, \mathbf{S}) = \sum_{k=0}^{K-1} h_k \mathbf{S}^k \mathbf{x}. \tag{18.16}$$

During the training a loss J is minimized between the predicted output signal $\hat{\mathbf{y}} = f_{\mathbf{h}}(\mathbf{x}, \mathbf{S})$ and the original output signal $\mathbf{y}$ in order to find the optimal values for the filter coefficients $\mathbf{h} = (h_0, \ldots, h_{K-1})^T$:

$$\mathbf{h}^* = \arg\min_{\mathbf{h}} \sum_{(\mathbf{x},\mathbf{y},\mathbf{S})\in\mathcal{S}} J(f_{\mathbf{h}}(\mathbf{x}, \mathbf{S}), \mathbf{y}) \tag{18.17}$$

18.2.6 Graph perceptron

The graph filters as defined above have limited expressive power as they can only learn linear mappings. In order to achieve a higher expressive power, the graph filters are combined with pointwise non-

linearities $g(\cdot)$, such as sigmoid, tanh or ReLU activation functions. This function is named *graph perceptron*, since it introduces the same features for GCN as the perceptron for general neural network.

Graph perceptron can be expressed as

$$f_{\mathbf{h}}(\mathbf{x}, \mathbf{S}) = g\left(\sum_{k=0}^{K-1} h_k \mathbf{S}^k \mathbf{x}\right). \tag{18.18}$$

Because of the introduced non-linearity, the graph perceptron is able to express a larger function class.

18.2.7 Graph Convolution Networks

A GCN can now be defined by stacking multiple layers of graph perceptrons on top of each other. Similar to a MLP, the GCN is recursively composed

$$\mathbf{x}_\ell = g\left(\sum_{k=0}^{K-1} h_{\ell k} \mathbf{S}^k \mathbf{x}_{\ell-1}\right), \tag{18.19}$$

where it is assumed that the input to the first layer is set to the input signal $\mathbf{x}_0 = \mathbf{x}$. Note that compared to MLP, in GCN $\mathbf{x}$ and h_k have index ℓ now to denote the layer. A short hand notation for the recursive application of the graph perceptron over L layers is:

$$\hat{\mathbf{y}} = f_{\mathcal{H}}(\mathbf{x}, \mathbf{S}) = \mathbf{x}_L, \tag{18.20}$$

where $\mathcal{H}$ denotes a set of L vectors of trainable filter coefficients $\mathcal{H} = \{\mathbf{h}_1, \ldots, \mathbf{h}_L\}$.

The filter parameters of a GCN are learned in same way as in the single-layer case, but with the filter coefficients of all layers $\mathcal{H}$ as the optimization objective:

$$\mathcal{H}^* = \arg\min_{\mathcal{H}} \sum_{(\mathbf{x},\mathbf{y},\mathbf{S})\in\mathcal{S}} J(f_{\mathcal{H}}(\mathbf{x}, \mathbf{S}), \mathbf{y}). \tag{18.21}$$

18.3 GCN for node classification

As previously stated, one of the possible tasks to compute on a graph is node classification (Figure 18.13). A graph convolutional layer encapsulates each node's hidden representation by aggregating feature information from its neighbors. After feature aggregation, a non-linear transformation is applied to the resulted outputs. By stacking multiple layers, the final hidden representation of each node receives messages from a further neighborhood. Node-level outputs relate to node regression and node classification tasks. GCN can extract high-level node representations by information propagation/graph convolution. In the output level we have a feature vector that describe each node, now we can use these feature vectors for training a classifier for node classification or use them for clustering the nodes.

18.4 GCN for graph classification

For graph classification (Figure 18.14 and Figure 18.15) we introduce the pooling layer. A distinction is often found in the literature between *regular* and *global pooling*, which is so strong to the point where global pooling is usually referred to as a separate operation called *readout*.

Pooling layer are used to coarsen a graph into sub-graphs so that node representations on coarsened graphs represent higher graph-level representations. A pooling layer can be any graph clustering algorithm that group sets of nodes together. A max pooling takes the max values from the cluster. We talk about readout/global pooling to indicate those methods that reduce a graph to a single node discarding topological information. For example, it takes the sum or mean of hidden representations of sub-graphs. By using a readout layer we obtain a vector representation, that can for example be used as input for a classification algorithm.

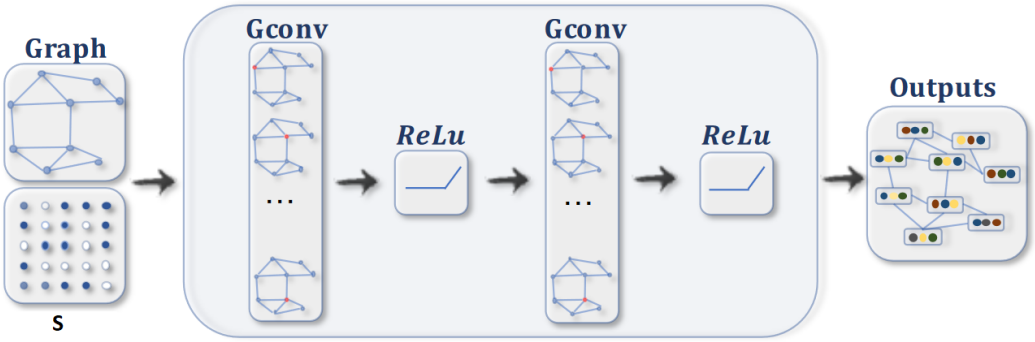


Figure 18.13: Example of GCN which, thanks to the use of two convolutional layer and using ReLU activation function, produces as output a feature vector for each node. So that for example classification/regression on one node can be done. Source: [127].

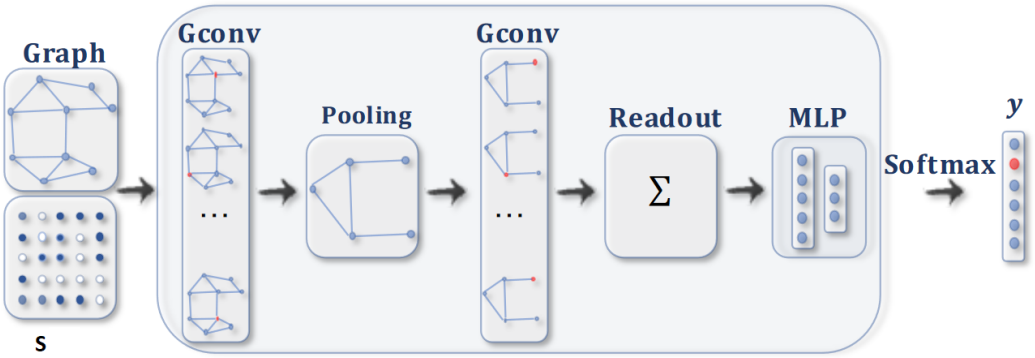


Figure 18.14: A GCN with pooling and readout layers for graph classification. After using convolutional and pooling layers a readout layer is used to produce a vector. After this process of feature extraction, we have final fully connected layers. Source: [127].

18.5 GCN examples

18.5.1 Crop and weed classifier

[56] presents a method to recognize weed (unwanted plants) and crop (wanted plants) in a smart farm. The work exploits both CNN and GCN to obtain a classifier able to outperformed the state-of-the-art methods. Assuming N_{train} labeled images and N_{test} unlabeled images, the proposed method consists of 2 parts. The first uses a CNN model to extract features from each image (both labeled and unlabeled), producing a vector of features for each image (Figure 18.16). We now consider each vector to be the signal $\mathbf{x}_i$ associated to a vertex of a graph. We thus obtain the vertex set V divided in V_{train} and V_{test}. The Graph convolution operator $\mathbf{S}$ considered is the adjacency matrix obtained by considering A_{ij} as the euclidean distance between $\mathbf{x}_i$ and $\mathbf{x}_j$.

Now a GCN is used is to obtain the label information of unlabeled vertices. The GCN model naturally combines graph structures and vertex features in the convolution, and propagated over the graph through multiple layers. Through the graph convolution layer by layer, unlabeled vertices update their features. Then the final obtained features of unlabeled vertices are fed into a softmax classifier, and the output is label information of unlabeled vertices. Thus all unlabeled images are recognized by a softmax classifier trained based on features of labeled vertices V_{train}.

18.5.2 Pairwise drug combination effect on cancer cells

Drug combination therapy has become a increasingly promising method in the treatment of cancer. However, the number of possible drug combinations is so huge that it is hard to screen synergistic

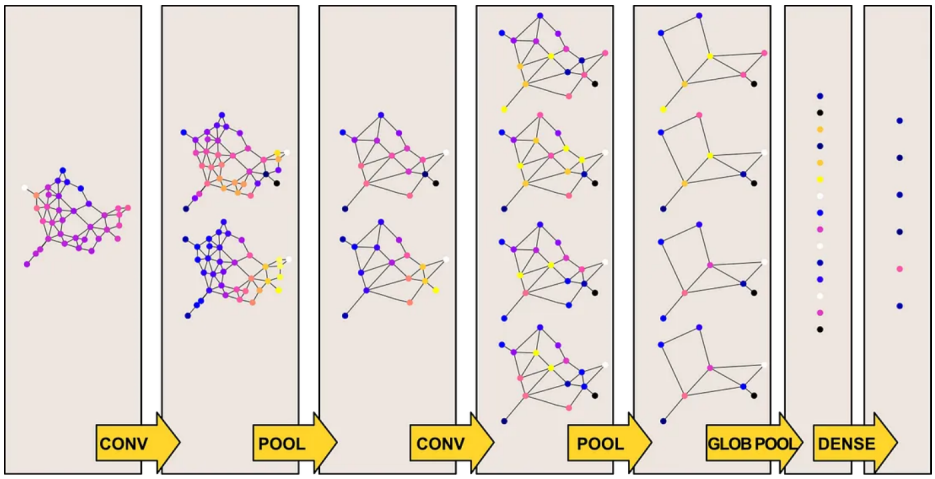


Figure 18.15: The image provides an alternative view of the same GCN shown in Figure 18.14. Source: [46].

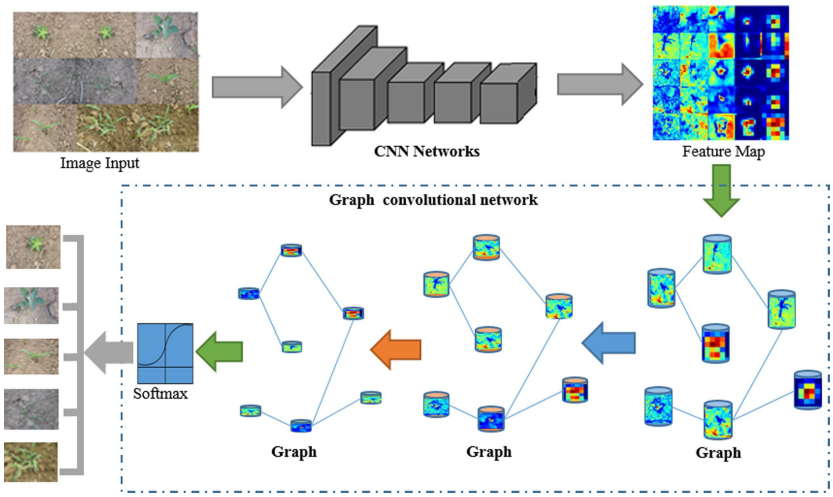


Figure 18.16: Process to classify weed and crop images using CNN and GCN. Source: [56].

drug combinations through wet-lab experiments. Therefore, computational screening has become an important way to prioritize drug combinations.

[116] proposes a deep learning model based on graph neural networks and attention mechanism to identify drug combinations that can effectively inhibit the life of specific cancer cells (Figure 18.17). For each pairwise drug combination, the input layer firstly receives the molecular graphs of two drugs and gene expression profiles of one cancer cell line that was treated by these two drugs. Two type of Graph Neural Networks (GNN): graph attention network (GAT) and graph convolution network (GCN), were tested to extract features of drugs. The genomic feature representation of cancer cells is encoded by a MLP. The embedding vectors are subsequently concatenated as the final feature representation of each drug-pair-cell-line combination, which is propagated through the fully-connected layers for the binary classification of drug combinations (synergistic or antagonistic). For example, triple-negative breast cancer is a malignant tumor with strong invasiveness, high metastasis rate and poor prognosis. Lapatinib or Rapamycin alone has little therapeutic effect, but their combined treatment has been reported to significantly increase the apoptosis rate of triple-negative breast cancer cells.

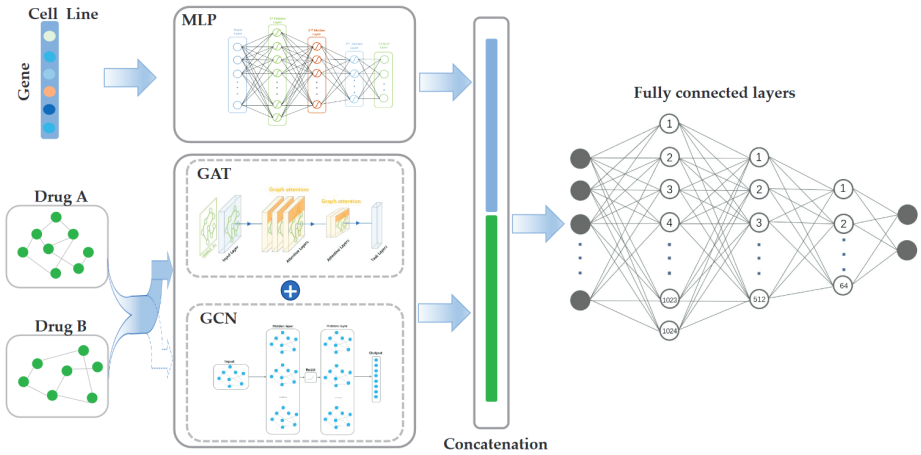


Figure 18.17: Process that compares GCN and GAT to test pairwise effect of drugs on a cell line, from which features are extracted thanks to MLP. Source: [116].

18.6 GraphSage

Recently, there has been a surge of interest in applying GNNs for a large number of NLP problems like text classification, exploiting semantics in machine translation, user geolocation, relation extraction, or question answering. We know that every node is an entity and edges describe relations between them. In NLP research, the problem of question answering is not recent. But it was limited by the existing database. Although, with techniques like *GraphSage* [47], the methods can be generalized to previously unseen nodes.

Chapter 19

Transformer

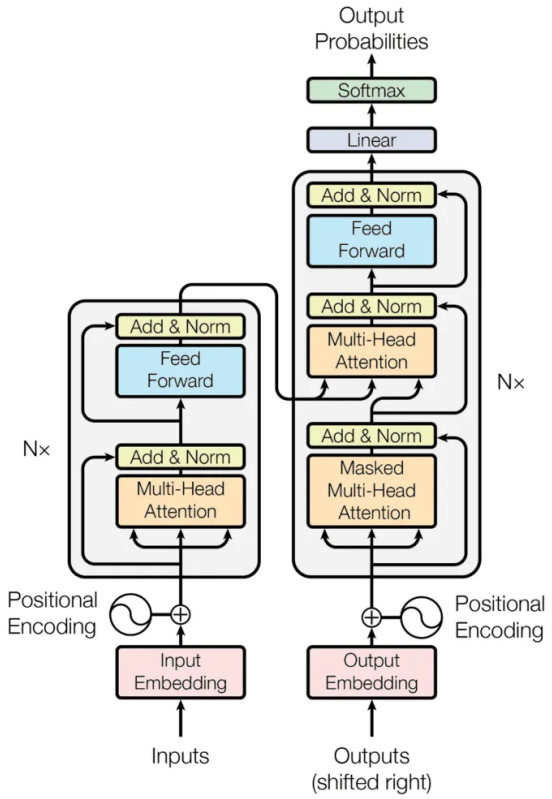


Figure 19.1: Original transformer architecture proposed by the milestone paper *Attention is all you need* [115].

Transformers in the last years have gained tremendous popularity due to the rise of models like OpenAI's ChatGPT and BERT from Google. A *transformer* [117] is a neural network that learns context and thus meaning by tracking relationships in sequential data like the words in this sentence. It was introduced in 2017 by the paper [115]. Transformer models apply an evolving set of mathematical techniques, called *attention* or *self-attention*, to detect subtle ways even distant data elements in a series influence and depend on each other. Attention is so key to transformers that the Google researchers that introduced transformer model almost used the term "Attention Net" as the name, but they discarded it because it didn't sound "exciting".

Attention is a rather general mechanism that can be used in a wide variety of problem domains. Consider the task of machine translation using RNN model. Also, consider the problem of image classification using a basic CNN model. While a RNN produces a sequence of hidden state vectors,

a CNN creates feature maps, where each region in the image is represented by a feature vector. The RNN hidden states are organized sequentially, while the CNN feature maps are organized spatially. Yet, attention can still be applied in both situations, since the attention mechanism does not inherently depend on the organization of the feature vectors. This characteristic makes attention easy to implement in a wide variety of models in different domains.

19.1 Model architecture

Most competitive neural sequence transduction models have an encoder-decoder structure. Here, the encoder maps an input sequence of symbol representations $\mathbf{x} = (x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$. Given $\mathbf{z}$, the decoder then generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.

The *encoder* is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, positionwise fully connected feed-forward network. A residual connection is employed around each of the two sub-layers, followed by layer normalization. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

The *decoder* is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, residual connections are employed around each of the sub-layers, followed by layer normalization. Also the self-attention sub-layer is modified in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i (the masking will be explained more in detail later).

19.2 Tokenization

Tokenization is the process of encoding a string of text into transformer-readable token ID integers. An example of tokenization is provided by Figure 19.2.

Text is divided into smaller units called tokens, which can be words, phrases, subwords, or characters [87]. In the context of Transformer models, tokenization is a crucial step in preprocessing text data for use in NLP tasks.

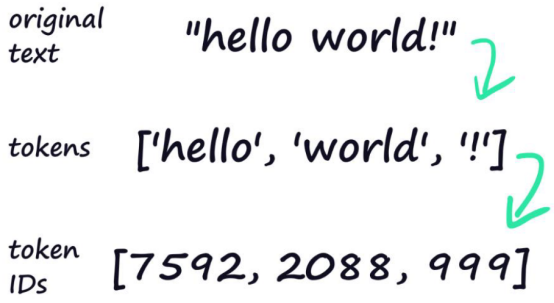


Figure 19.2: Tokenization of a sentence. Source: [72].

19.3 Embedding

Now the tokenized words are passed to the *embedding layer* of the encoder which produces the embeddings/vector representation of words. The embedding layer transform each token ID to a 512 dimensional vector. So, during inference/training, these token IDs are read by an embedding layer

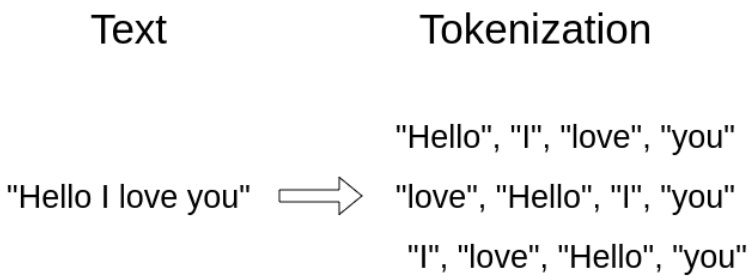


Figure 19.3: The order of the obtained embeddings is irrelevant, therefore the three sets on the right are equivalent. Source: [106].

in our transformer model which maps the token ID to a dense vector representation of that token (imagine a numerical representation of the meaning of the initial word), i.e. each word is represented with a feature vector. There are many methods for representing words as vectors, further details are provided by [125].

By representing words as vectors, embeddings enable the models to capture the similarities between words. For example, consider the words "king", "queen","man", and "woman". These words are related, and their relationships can be captured using embeddings. In an embedding space, the vector difference between "king" and "queen" would be similar to the vector difference between "man" and "woman".

Instead of a sequence of elements, we now have a set (meaning that the order is irrelevant [106], Figure 19.3) denoted as $X = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_N\}$ where $\mathbf{x}_i \in \mathbb{R}^{d_{\text{in}}}$. Each $\mathbf{x}_i$ is a token representation as a vector, N is the size of the input sequence and d_{in} the size of the vector associated to one token.

19.4 Positional encoding

Following tokenization and embedding is the *Positional encoding*, as shown in Figure 19.4.

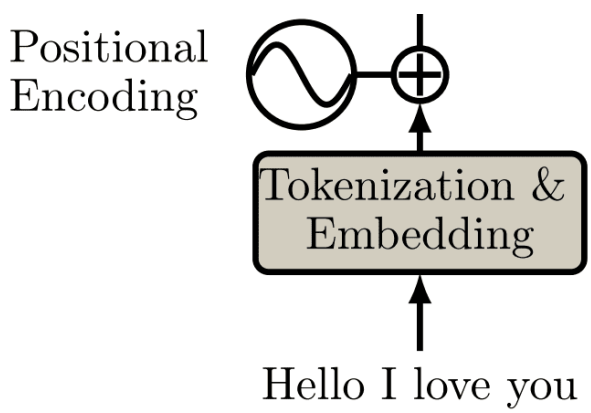


Figure 19.4: Tokenization and embedding layer, with the addition of the Positional encoding. Source: [52].

Officially, positional encoding is a set of small constants, which are added to the word embedding vector before the first self-attention layer. So if the same word appears in a different position, the actual representation will be slightly different, depending on where it appears in the input sentence.

In the transformer paper, the authors came up with the sinusoidal function for the positional encoding. The sine function tells the model to pay attention to a particular wavelength λ. Given a signal $y(x) = \sin(kx)$ the wavelength will be $\lambda = \frac{2\pi}{k}$. In our case the λ will be dependent on the position in the sentence.

Mathematically:

$$\mathbf{p}_t = \begin{bmatrix} \sin(\omega_1 \cdot t) \\ \cos(\omega_1 \cdot t) \\ \\ \sin(\omega_2 \cdot t) \\ \cos(\omega_2 \cdot t) \\ \\ \vdots \\ \\ \sin(\omega_{\frac{d_{\text{in}}}{2}} \cdot t) \\ \cos(\omega_{\frac{d_{\text{in}}}{2}} \cdot t) \end{bmatrix}. \tag{19.1}$$

where:

$$\omega_k = \frac{1}{1000^{2k/d_{\text{in}}}}, \tag{19.2}$$

d_{in} is the feature size associated to each word and t is the position of the word in the sentence. In the following part we will consider $d_{\text{in}} = 512$.

The $\mathbf{p}_t \in \mathbb{R}^d$ is added to the feature vector used to describe each word, as shown in Figure 19.5.

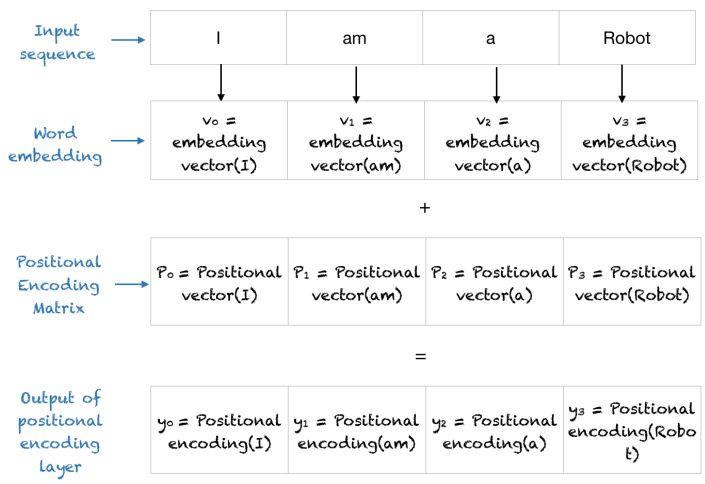


Figure 19.5: Positional encoding summed with embedding. Source: [2].

By considering the set X as a matrix $\mathbf{X}$ where each row is an embedding vector. As a consequence $\mathbf{X}$ has size $n \times d_{\text{in}}$, where n is the length of the input sequence (i.e. each row is a word). We can consider the positional encoding as summing $\mathbf{X}$ to a matrix indicating positional encoding. An idea is given by Figure 19.6

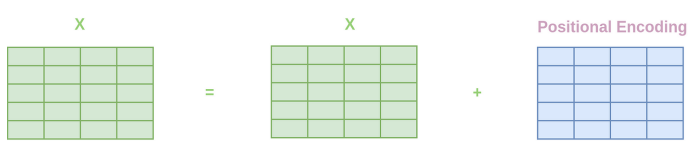


Figure 19.6: $\mathbf{X}$ summed to the matrix associated to positional encoding. Source: [114].

We can represent the heatmap of the position encoding matrix that we will add to the input that is to be given to the first encoder. Figure 19.7 shows the heatmap for the first 300 positions and Figure 19.8 for the first 3000 positions. We can see that there is a distinct pattern that we provide to our Transformer to understand the position of each word. And since we are using a function comprised of sin and cos, we are able to add positional embeddings for a high number of positions.

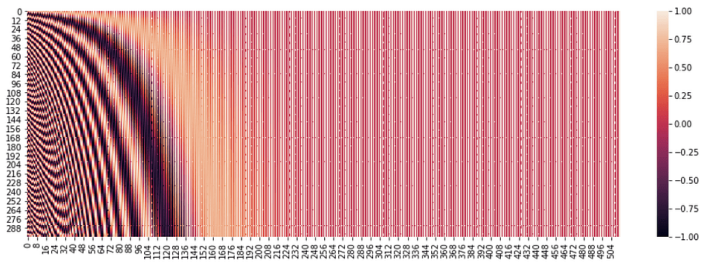


Figure 19.7: Heatmap of positional embedding matrix with a sequence of 300 tokens. Source: [114].

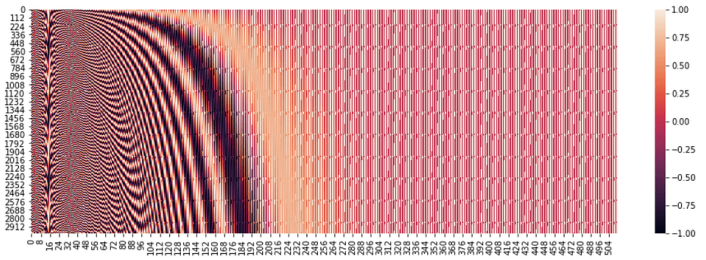


Figure 19.8: Heatmap of positional embedding matrix with a sequence of 3000 tokens. Source: [114].

19.5 Self-attention

Self-Attention finds meaning. For example, in the sentence: She poured water from the pitcher to the cup until it was full. We know "it" refers to the cup, while in the sentence: She poured water from the pitcher to the cup until it was empty. We know "it" refers to the pitcher. So, meaning is a result of relationships between things, and self-attention is a general way of learning relationships.

It was defined as follows by the creators: "Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence." [115].

19.5.1 Key-Value-Query Concept

Let's start with an example of searching for a video on YouTube. When you search (query) for a particular video, the search engine will map your query against a set of *keys* (video, title, description, etc.) associated with possible stored videos. Then the algorithm will present you the best-matched videos (*values*). This is the foundation of content/feature-based lookup.

Bringing this idea closer to the transformer's attention we have something like the Figure 19.9.

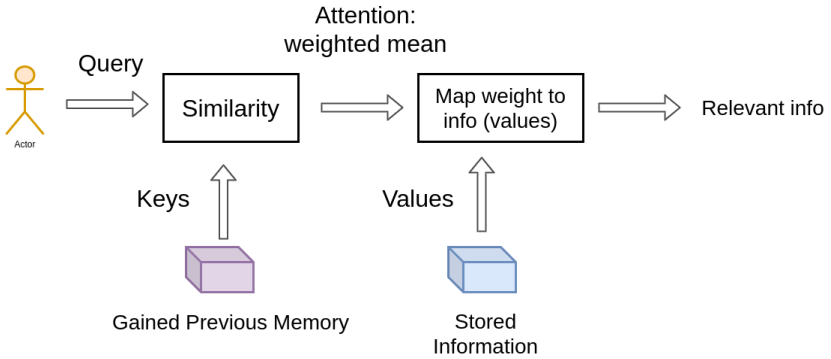


Figure 19.9: Query process with attention based information. Source: [52].

In the single video retrieval, the attention is the choice of the video with a maximum relevance score. But we can relax this idea. To this end, the main difference between attention and retrieval

systems is that we introduce a more abstract and smooth notion of *retrieving* an object. By defining a degree of similarity (weight) between our representation (videos for YouTube) we can weight our query.

So, by moving one step forward, we further split the data into key-value pairs. We use the keys to define the attention weights to look at the data and the values as the information that we will actually get.

For the so-called mapping we need to quantify similarity, that is high dimensional spaces is computed as normalized inner product:

$$\text{sim}(\mathbf{a}, \mathbf{b}) = \cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{ab}}{|\mathbf{a}|\,|\mathbf{b}|} \tag{19.3}$$

We can associate the similarity between vectors that represent anything (i.e. animals) by calculating the scaled dot product. As a matter of fact, recall that the angle between two vectors can be computed as $\arccos\left(\frac{\mathbf{ab}}{|\mathbf{a}||\mathbf{b}|}\right)$.

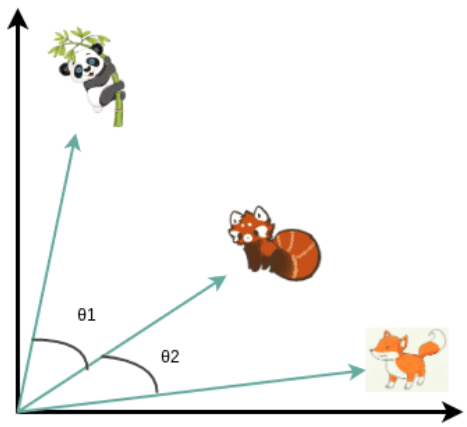


Figure 19.10: Angle between the objects tells how different they are. Source: [52].

In transformers, this is the most basic operation and is handled by the self-attention layer as we will see.

19.5.2 Self-attention layer

In practice, the self-attention idea may be written in a compact way as follows using 3 different representations: the Queries, Keys and Values. By multiplying our input $\mathbf{X}$ with 3 different weight matrices $\mathbf{W}_Q$, $\mathbf{W}_K$ and $\mathbf{W}_V$ the result is

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \tag{19.4}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V, \tag{19.5}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K. \tag{19.6}$$

The shapes of the matrices are defined by the queries and keys dimension d_k and values dimension d_v, i.e.:

$$\mathbf{W}_K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \tag{19.7}$$

$$\mathbf{W}_Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \tag{19.8}$$

$$\mathbf{W}_V \in \mathbb{R}^{d_{\text{model}} \times d_v}. \tag{19.9}$$

Having the Query, Value and Key matrices, we can now apply the self-attention layer as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{19.10}$$

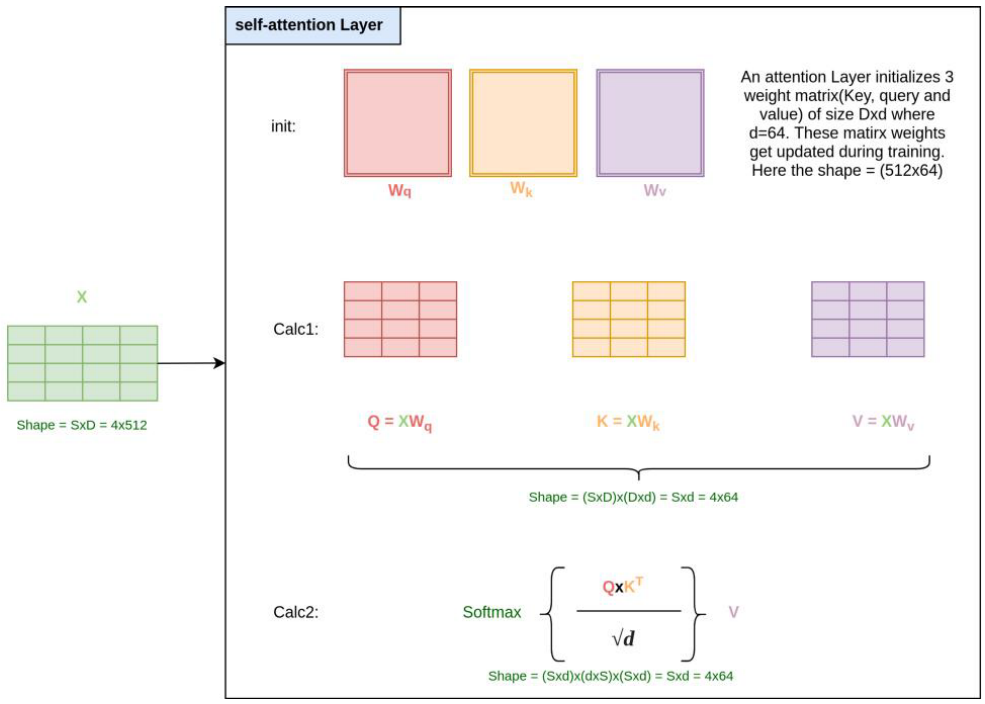


Figure 19.11: Visualization of the attention layer computation. Cite: [72].

where d_k is the dimension of the key and query vectors (that are on row of the corresponding matrices). The whole process is visualized by Figure 19.11.

In essence, it is just a matrix multiplication in the original word embeddings, where the resulted dimension will be smaller. If we consider what would happen to the single vector the concept may be clearer. Every input token must have three representations (see Figure 19.12). These representations are called key, query and value. They are obtained by multiplying the input vector with the key, query and value weight matrices. For this example, assume that we have $n = 4$ words and that we want these representations to have a dimension of $d_v = d_k = 3$ (output). Because every input has a dimension of 4, each set of the weights must have a shape of 4×3. To obtain the *attention scores*, we start with taking a dot product between Input 1's query with the key associated to each word, including itself. Since there are 3 key representations (because we have 3 input tokens), we obtain 3 attention scores. Take the softmax across these attention scores. If for the query $\mathbf{q}_i$ and the key $\mathbf{k}_j$ we have an high value of $\langle \mathbf{q}_i, \mathbf{k}_j \rangle$, we say that the embedding of the j-th word *attends to* the embedding of the i-th word. The softmaxed attention scores for each input is multiplied by the value vector corresponding to that token. This results in 3 *alignment vectors*. We'll refer to them as *weighted values*. Take all the weighted values and sum them element-wise. The resulting vector $[2.0, 7.0, 1.5]^T$ is Output 1, which is based on the query representation from Input 1 interacting with all other keys, including itself. The process described is repeated but using the query for Input 2 and, then, for Input 3. Therefore, the number of vectors coming out of the self-attention mechanism will be equal to the number of inputs vectors.

19.5.3 Multi-head attention

In the original paper, the authors expand on the idea of self-attention to multi-head attention. Instead of performing a single attention function with d_{model}-dimensional keys, values and queries, they found beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values they then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values. The whole process is visualized by Figure 19.13.

Each time, we map the independent set of Key, Query, Value matrices into different lower dimen-

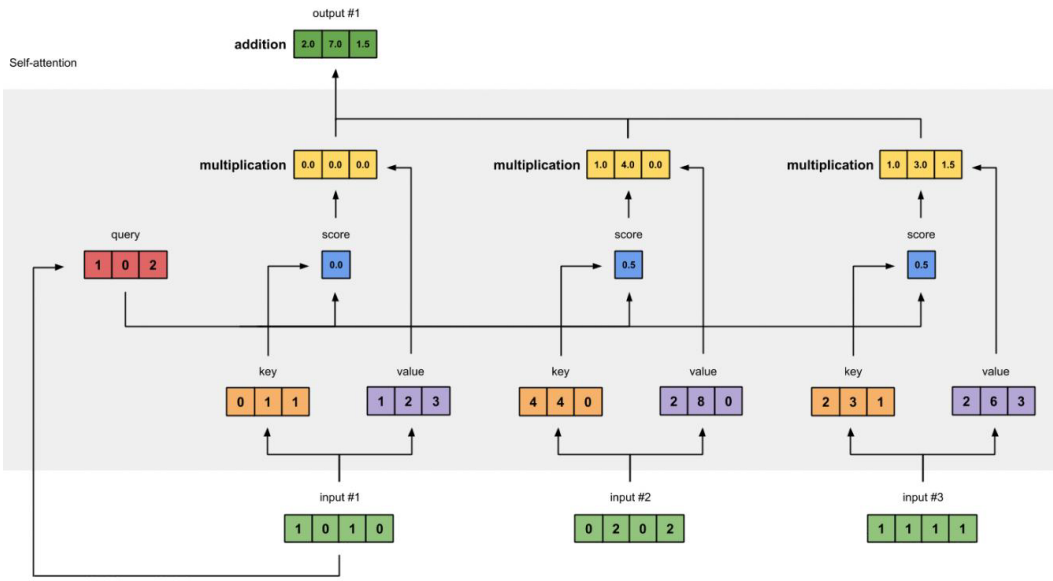


Figure 19.12: Self attention process for Input 1, with key (orange), query (red), value (purple), attention scores (blue) and alignment vectors (yellow). In this model the computation of the softmax is done implicitly when computing the attention scores. After applying the softmax, the obtained attention scores for Input 1 are $[0.0633, 0.4683, 0.4683]$, then rounded to $[0, 0.5, 0.5]$. Source: [72].

sional spaces and compute the attention there (the output is called a "head"). The mapping is achieved by multiplying each matrix with a separate weight matrix, denoted as $\mathbf{W}_i^K$, $\mathbf{W}_i^Q$ and $\mathbf{W}_i^V$. We have that:

$$\mathbf{W}_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \tag{19.11}$$

$$\mathbf{W}_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \tag{19.12}$$

$$\mathbf{W}_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}. \tag{19.13}$$

To compensate for the extra complexity, the output vector size is divided by the number of heads. Specifically, in the vanilla transformer, they use an input size of 512 and 8 heads, which gives us vector representations of 64. Now, the model has multiple independent paths (ways) to understand the input. The heads are then concatenated and transformed using a square weight matrix $\mathbf{W}^O \in \mathbb{R}^{h \cdot d_v \times d_{\text{model}}}$ (h number of heads), with the goal of reducing the many heads to the size that would come from a single head.

Putting it all together we get:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\mathbf{head}_1, ..., \mathbf{head}_h)\mathbf{W}^O, \tag{19.14}$$

$$\mathbf{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{19.15}$$

Recall that n is the length of the input sequence. The output of Equation (19.15) is a $n \times d_v$ matrix as in the single-headed case. Therefore, the Concat$(\cdot)$ in Equation 19.14 outputs a $n \times (h \cdot d_v)$ matrix because of the concatenation. Finally, the output of the multi head attention block is a $n \times d_{\text{model}}$ matrix. Note that $\mathbf{W}_i^Q$, $\mathbf{W}_i^K$ and $\mathbf{W}_i^V$ are conceptually the same as $\mathbf{W}^Q$, $\mathbf{W}^K$ and $\mathbf{W}^V$. Therefore, the MultiHead$(\cdot)$ is invoked as:

$$\text{MultiHead}(\mathbf{X}, \mathbf{X}, \mathbf{X}). \tag{19.16}$$

Then, each of the h heads performs its own linear mapping and performs Attention$(\cdot)$[1]. A good visualization of the process is given by Figure 19.14.

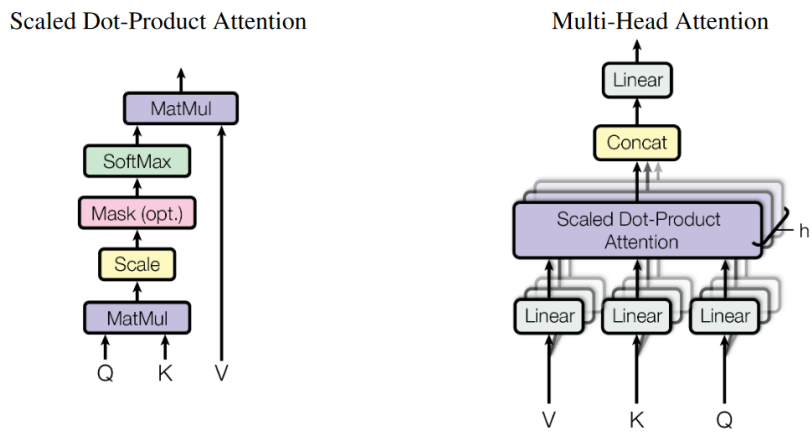


Figure 19.13: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel. Source: [115].

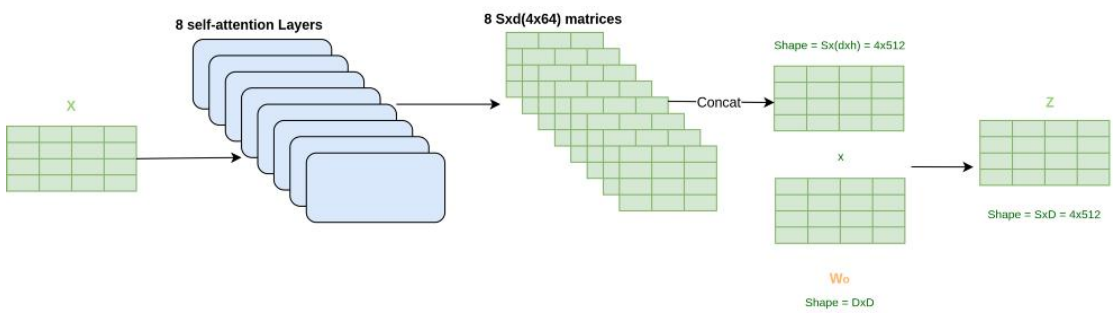


Figure 19.14: Visualization of multi-head attention. In this example $h = 8$, $S = n = 4$, $D = d_{\text{model}} = 512$, $d = d_k = d_v = 64$. Source: [72].

19.6 Addition and Layer Normalization

As shown in Figure 19.15, following the attention layer is the addition and layer normalization. In practice, the attention layer output is summed to the input of the attention layer through a residual connection. Then layer normalization is applied. The architecture that goes from input to the output of the normalization layer is shown in Figure 19.16.

Like batch normalization, layer normalization aims to reduce the effect of covariant shift. In other words, it prevents the mean and standard deviation of embedding vector elements from moving around, which makes training unstable and slow (i.e., we can't make the learning rate big enough). Unlike batch normalization, layer normalization works at each embedding vector (not at the batch level). Considering that i indexes the pattern, in this case the words of the input sentence and k indexes the feature elements. The statistics are computed through the feature vector of a given pattern, so they are independent with respect to those calculated for other patterns. So normalization can be described mathematically as follows:

$$\mu_i = \frac{1}{K}\sum_{k=1}^{K} x_{ik}, \tag{19.17}$$

$$\sigma_i^2 = \frac{1}{K}\sum_{k=1}^{K} (x_{ik} - \mu_i)^2, \tag{19.18}$$

$$\hat{x}_{ik} = \frac{x_{ik} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}. \tag{19.19}$$

[1]Note the difference in notation with Section 19.5.2.

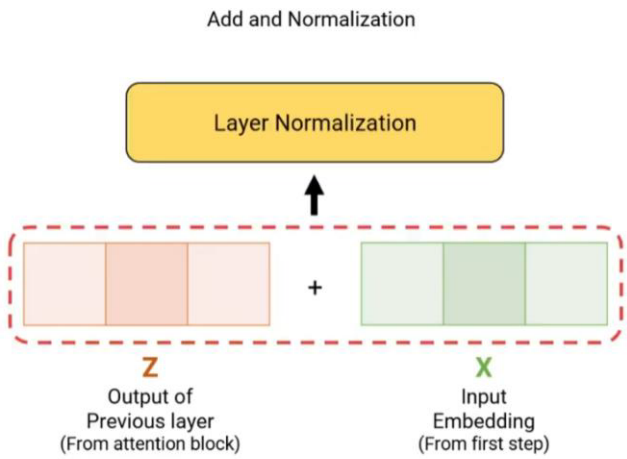


Figure 19.15: The output of the attention layer is added to the input embedding. Then normalization is performed. Source: [72].

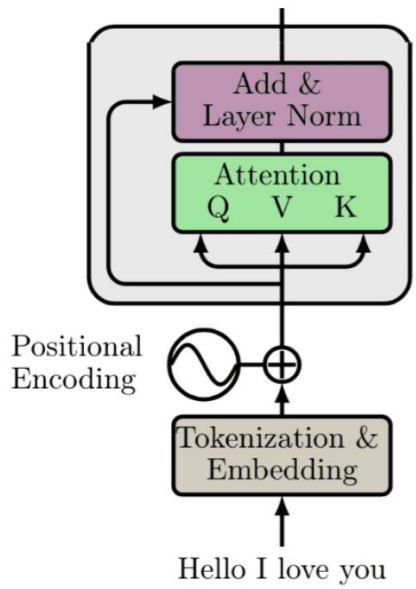


Figure 19.16: Part of the architecture that goes from the input to the layer normalization. Source: [72].

19.7 Position-wise Feed-Forward Networks

Each of the layers in encoder and decoder contains a fully connected feed-forward network, which is applied in parallel to each position separately and identically. Hence, the name feed-forward position-wise network. The single feed-forward network consists of two linear transformations (dropout is applied) with a ReLU activation in between.

$$\mathrm{FFN}(\mathbf{x}) = \max(0, \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2. \tag{19.20}$$

While the linear transformations are the same across different positions, they use different parameters from layer to layer. The dimensionality of input and output is $d_{\text{model}} = 512$, and the inner-layer has dimensionality $d_{ff} = 2048$.

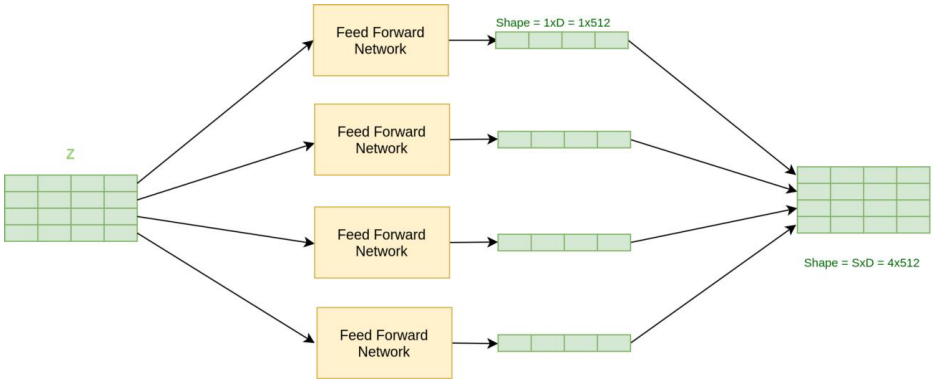


Figure 19.17: In this figure it is clear that each token (each row of this matrix) goes into a different feed-forward network, these networks can operate in parallel, moreover, all these networks share internal weights. The linear level output is the same size as the linear level input. Source: [72].

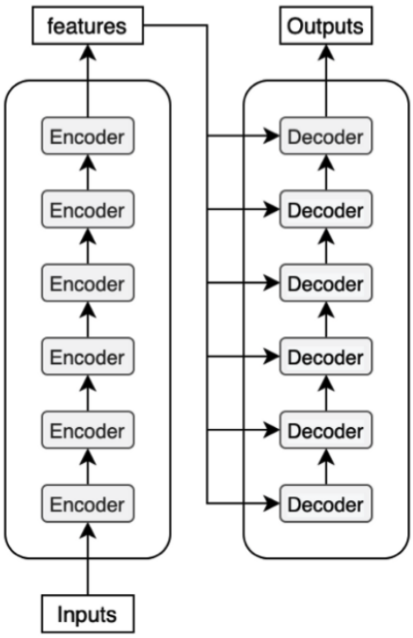


Figure 19.18: Each decoder block receives the features from the encoder. Source: [107].

19.8 Encoder-decoder

Until now the focus has been on defining the structure of a single block of the encoder. However, the encoder in the transformer consists of multiple encoder blocks [107].

The transformer uses an *encoder-decoder architecture*. The encoder extracts features from an input sentence, and the decoder uses the features to produce an output sentence (translation). Therefore, input sentence goes through the encoder blocks, and the output of the last encoder block becomes the input features to the decoder. The decoder also consists of multiple decoder blocks, each receiving the same input from the encoder, as shown in Figure 19.18. The paper uses "$N\times$" (N-times) to indicate multiple blocks. In this way a more concise view can be used.

Then, the output vector from the decoder goes through a linear transformation that changes the dimension of the vector from the embedding vector size (512) into the size of vocabulary (say, 10,000). The softmax layer further converts the vector into 10,000 probabilities, and the highest score is the chosen word.

So, the decoder outputs one token at a time. An output token becomes the subsequent input to the decoder. In other words, a previous output from the decoder becomes the last part of the

next input to the decoder. This kind of processing is called *auto-regressive* — a typical pattern for generating sequential outputs and not specific to the transformer. It allows a model to generate an output sentence of different lengths than the input.

However, we have no previous output at the beginning of a translation. So, we pass the *start-of-sentence* marker ⟨SOS⟩ (as known as the beginning-of-sentence marker ⟨BOS⟩) to the decoder to initiate the translation. The decoder uses multiple decoder blocks to enrich ⟨SOS⟩ with the contextual information from the input features. In other words, the decoder transforms the embedding vector ⟨SOS⟩ into a vector containing information helpful to generating the first translated word (token).

Then the predicted word is concatenated with the previous output. The concatenation is then fed back to the decoder to produce the next token. While, the output of the encoder does not need to be recomputed since the input of the encoder did not change from the previous computation. The process repeats until the model predicts the end-of-sentence ⟨EOS⟩ as the most probable output.

Because of the ⟨SOS⟩ inserted in the output sequence, the first output token becomes the second input token. This explains why the original diagram says "Outputs (shifted right)".

It may look like a slow process as we must generate one output at a time, especially for training. However, during training label tokens (rather than predicted ones) are fed to the network, making learning more stable. It also makes the training run faster as attention masks can be used, allowing parallel processing.

19.8.1 Masking

Masking is needed to prevent the attention mechanism of a transformer from cheating in the decoder when training (on a translating task for instance). This kind of cheating-proof masking is not present on the encoder side (because we want the encoder to be able to see the whole initial phrase that has to be translated).

If we have the following sequence as an input for our decoder: "⟨SOS⟩ I love it", then the expected prediction for the token at position one ("⟨SOS⟩") is the token at the next position ("I"). Similarly the expected prediction for the tokens "⟨SOS⟩ I" is "love". We do not want the attention mechanism to share any information regarding the token at the next positions, when giving a prediction using all the previous tokens. To ensure that this is done, future positions are masked (setting them to $-\infty$) before the softmax step in the self-attention calculation. An example of mask is shown in Figure 19.19 In this way the softmax within the attention mechanisms effectively assigns zero probability to masked positions. Intuitively it is as if we were gradually increasing the visibility of output sentences by the masks.

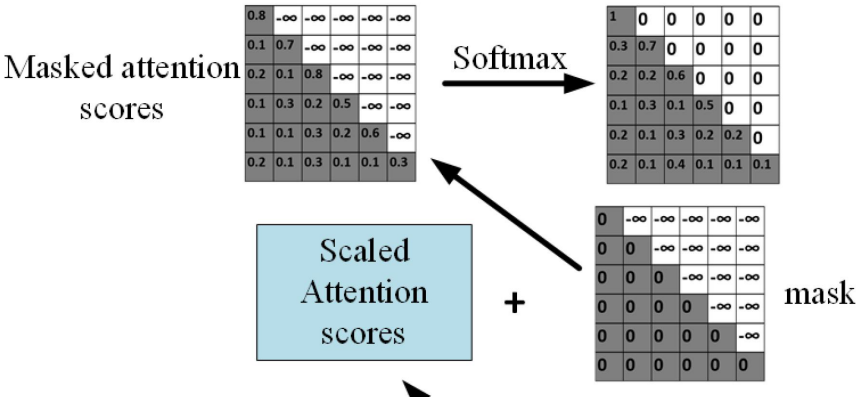


Figure 19.19: Example of masking where the mask is subtracted to the initial attention scores. By having $-\infty$ in the mask the result is having 0 in the softmaxed result. The initial attention score is obtained by $\mathbf{QK}^T$ of Equation (19.10). If $j > i$, then the i-th query cannot attend to the j-th key, since it would "see into the future". Source: [72].

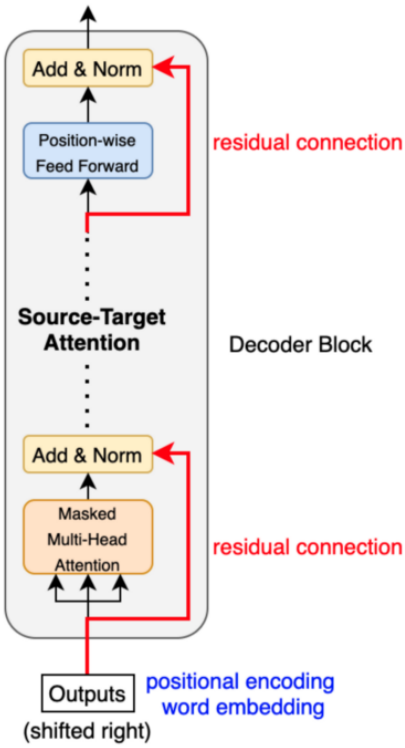


Figure 19.20: The decoder block general structure. Notice how the Source-Target Attention is the only difference from the encoder block. Source: [72].

19.9 Decoder block

Until now decoder blocks were seen as black-boxes. Now focus will shift on the internals of the decoder blocks.

The decoder block is similar to the encoder block, except it calculates the source-target attention (as shown in Figure 19.20). As mentioned before, an input to the decoder is an output shifted right, which becomes a sequence of embeddings with positional encoding. So, we can think of the decoder block as another encoder generating enriched embeddings useful for translation outputs. The *source-target attention* is another multi-head attention that calculates the attention values between the features (embeddings) from the input sentence and the features from the output (yet partial) sentence. Figure 19.21 provides a good example of the final structure of the decoder block.

The intuition behind the encoder-decoder attention layer is to combine the input and output sentence. The encoder's output encapsulates the final embedding of the input sentence. It is like our database. So we will use the encoder output to produce the Key and Value matrices. On the other hand, the output of the Masked Multi-head attention block contains the so far generated new sentence and is represented as the Query matrix in the attention layer. Again, it is the "search" in the database.

Hence, if we consider $\mathbf{X}$ to be the data flow from the decoder and $\mathbf{Z}$ to be the memory from the encoder. The resulting output of the encoder-decoder attention layer will be

$$\text{Attention}(\mathbf{X}\mathbf{W}_Q, \mathbf{Z}\mathbf{W}_K, \mathbf{Z}\mathbf{W}_V). \tag{19.21}$$

It will eventually determine how related each English word is with respect to the French words. This is essentially where the mapping between English and French is happening.

19.10 Decoder only

Modern Language Models as GPT-3 use decoder-only transformer architectures, which apply a sequence of layers consisting of masked self-attention and feed forward transformations to the model's

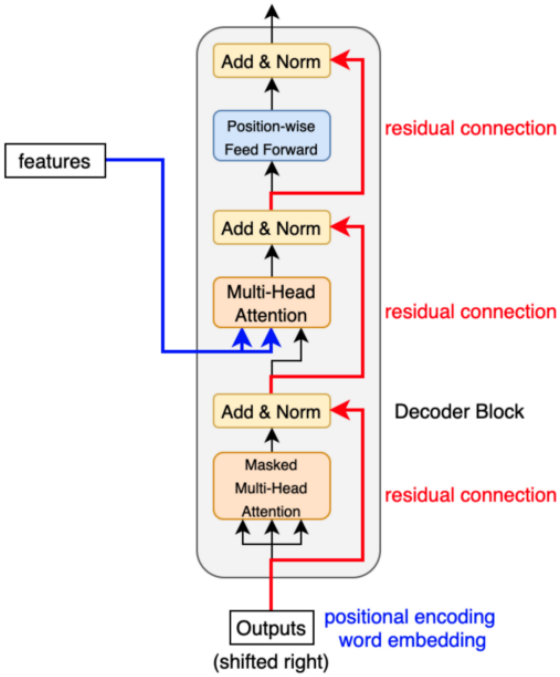


Figure 19.21: Full decoder block architecture. Source: [72].

input. As activation function Gaussian Error Linear Unit (Gelu) is used, it can be thought of as a smoother ReLU, GELUs are used in GPT-3, BERT, and most other Transformers. For further details see [48].

19.11 Vision Transformer

First introduced in [36], *Vision Transformers (ViTs)* have shown that a vanilla Transformer, once adapted to deal with data from the visual domain, can compete with some of the most performant CNNs developed up to that point.

The model's performance was acceptable when trained on ImageNet (1M images), great when pre-trained on ImageNet-21k (14M images), and state-of-the-art when pretrained on Google's internal JFT-300M dataset (300M images).

The striking performance improvement was due to the reduced inductive bias that characterizes Vision Transformers. By making fewer assumptions about the data, Vision Transformers could better adapt themselves to the given task. However, this ability came at a cost – when the sample size was too small (such as in the ImageNet case), the models overfit, resulting in degraded performance.

The goal for many follow-up papers would be that of matching (and surpassing) the performance of the best convolutional models in the "small" data regime – ImageNet (which is after all over a million images).

19.11.1 Model architecture

The ViT converts an image to tokens such that each token represents a local area (or patch) of the image. The paper describes reshaping an image of height H, width W, and channels C into N tokens with patch size P. The patch is then reshaped into a token of length $P^2 \cdot C$. This linearly flattened patch is further passed through a Feed-Forward layer with a linear activation function to get a linear patch projection of the length D, where D is the hyperparameter called as *embedding dimension* used throughout the transformer.

For classification purposes, taking inspiration from the original BERT paper, we concatenate a *learnable class embedding* with the other patch projections, whose state at the output serves as class information. This extra class token is added to the set of image tokens which is responsible for

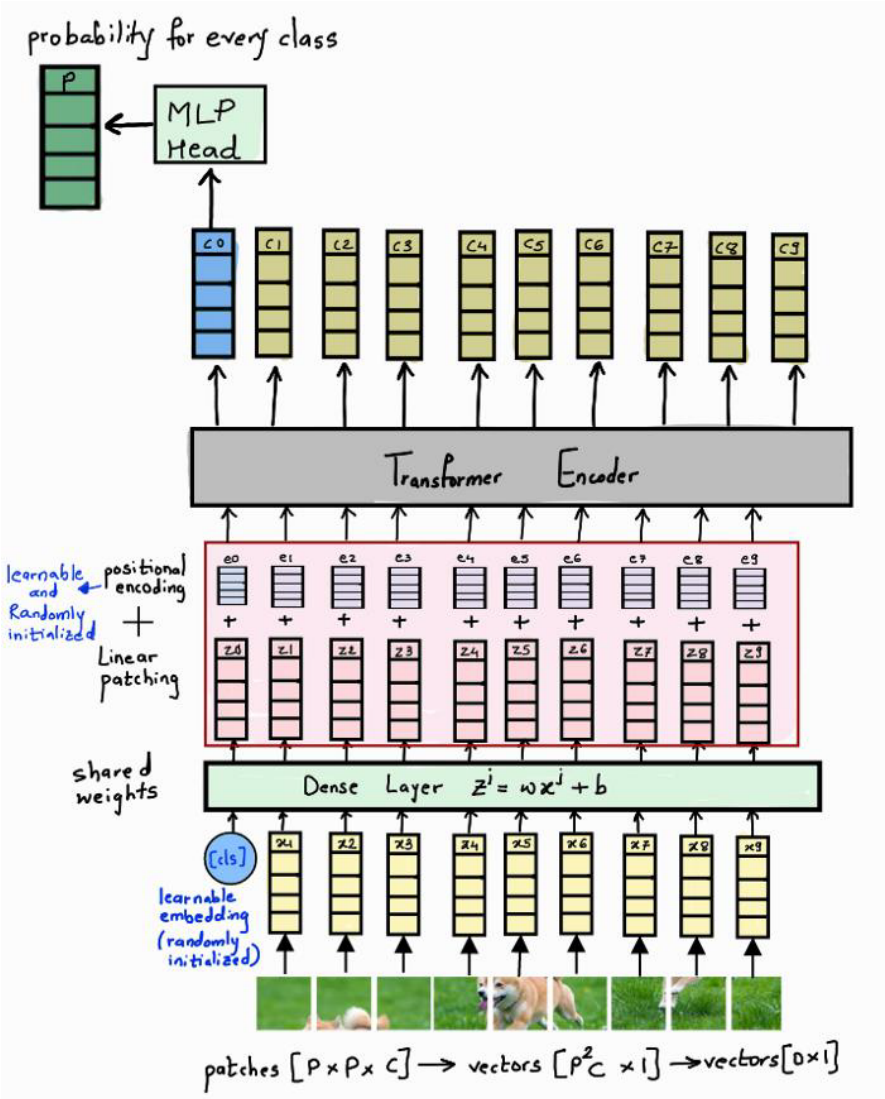


Figure 19.22: Model overview. Image is split into fixed-size patches, linearly embedded, summed to positional embeddings, and fed to a standard transformer encoder. In order to perform classification an extra learnable classification toke is added to the sequence. Source: [72].

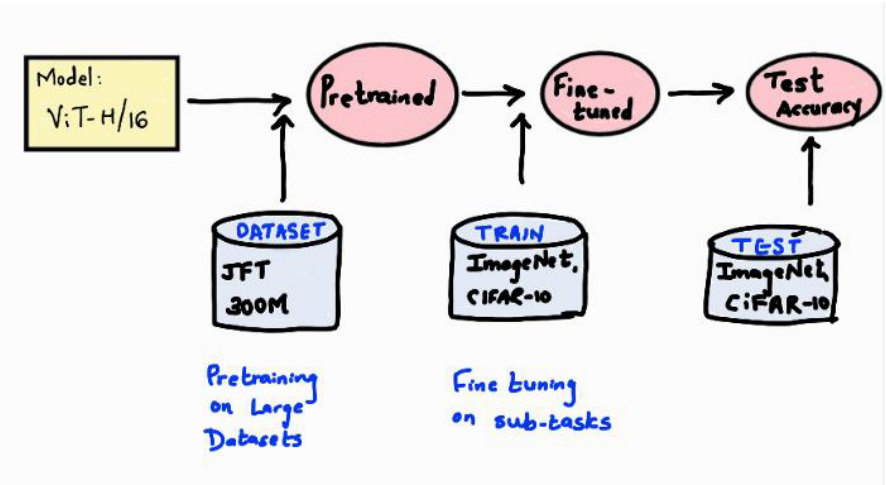


Figure 19.23: In this example, the ViT H/16 model is first trained on JFT 300M dataset and then fine-tuned on ImageNet or CIFAR datasets. Source: [72].

aggregating global image information and final classification. It is able to learn this global aggregation while it passes and learns through the attention layers.

This additional token is needed because transformers, by default are sequence to sequence networks. As there is no decoder layer in ViT, then the length of input sequence (number of patches) equals the length of output sequence. So If the goal is classification, there is two choices. The first possibility is to apply a fully connected layer on top of the transformer (which is not a good idea because then we have to fix the number of patches, translating to input image resolution). Another solution is to apply the classification layer on one item of the output sequence, but which one? The best answer here is none of them. We don't want to be biased toward any of the patches. So the best solution here is to add a dummy input, call it class token and apply the classification layer on the corresponding output item.

Equivalently for what was explained for vanilla transformers, position embedding are added to the tokens. The position embedding allows the transformer to understand the order of the image tokens.

Then comes the Transformer Encoder, that consists of multiple encoder blocks equivalent to the one seen for the original transformers. In fact, the encoder block is similar to the original transformer. The only thing that changes is the number of those blocks. To this end, with more data larger ViT variants can be proved.

After going through the encoders, the output is the context vector $\mathbf{C}$, we are only interested in the context token c_0 for classification purposes. This context token $\mathbf{c}_0$ is passed through an MLP head to give us the final probability vector to help predict the class. The MLP head is implemented with one hidden layer and tanh as non-linearity at the pretraining stage and by a single linear layer at the fine-tuning stage.

The training process, as mentioned in the paper, is divided into pretraining and fine-tuning steps as seen in Figure 19.23. ViT is pretrained on the large dataset and then fine-tuned to small ones. The only modification is to discard the prediction head (MLP head) and attach a new linear layer to classify the pattern of the small dataset. It is interesting that the authors claim that it is better to fine-tune at higher resolutions than pretraining. To fine-tune in higher resolutions, 2D interpolation of the pretrained position embeddings is performed. The reason is that they model positional embeddings with trainable linear layers. Having that said, the key engineering part of this paper is all about feeding an image in the transformer.

19.12 Other works

19.12.1 Comparing Transformers to CNN

Transformer emerges as a powerful tool for visual recognition. In addition to demonstrating competitive performance on a broad range of visual benchmarks, recent works also argue that Transformers are

much more robust than Convolutions Neural Networks (CNNs). Nonetheless, surprisingly, [9] finds these conclusions are drawn from unfair experimental settings, where Transformers and CNNs are compared at different scales and are applied with distinct training frameworks. In their paper, they aim to provide the first fair and in-depth comparisons between Transformers and CNNs, focusing on robustness evaluations. With their unified training setup, they first challenge the previous belief that Transformers outshine CNNs when measuring adversarial robustness. More surprisingly, they find CNNs can easily be as robust as Transformers on defending against adversarial attacks, if Transformers' training recipes are properly adopted. While regarding generalization on out-of-distribution samples, they show pretraining on (external) large-scale datasets is not a fundamental request for enabling Transformers to achieve better performance than CNNs. Moreover, the ablations suggest such stronger generalization is largely benefited by the Transformer's self-attention-like architectures per se, rather than by other training setups.

19.12.2 Transformers compared to brain

Many DNN architectures loosely based on brain networks have recently been shown to replicate neural firing patterns observed in the brain. One of the most exciting and promising novel architectures, the Transformer neural network, was developed without the brain in mind. In this work, we show that transformers, when equipped with recurrent position encodings, replicate the precisely tuned spatial representations of the hippocampal formation; most notably place and grid cells. Furthermore, we show that this result is no surprise since it is closely related to current hippocampal models from neuroscience. We additionally show the transformer version offers dramatic performance gains over the neuroscience version. This work continues to bind computations of artificial and brain networks, offers a novel understanding of the hippocampalcortical interaction, and suggests how wider cortical areas may perform complex tasks beyond current neuroscience models such as language comprehension. [124]

Chapter 20

Explainable AI

Interpretability is the degree to which ML algorithms can be understood by humans. ML models are often referred to as *black box* because their representations of knowledge are not intuitive and, as result, it is often difficult to understand how they work. Interpretability techniques help to reveal how black box ML models make predictions.

By revealing how various features contribute (or do not contribute) to predictions, interpretability techniques can help us validate that a model is using appropriate evidence for predictions, and find biases in our model that were not apparent during training. Some ML models, such as linear regression, decision trees and generative additive models are inherently interpretable. However, interpretability often comes at the expense of power and accuracy, as shown in Figure 20.1.

It is useful to distinguish the difference between *interpretability* and *explainability*. Interpretability is used more often in the context of classic ML, while in the context of DNNs many use *AI explainability*.

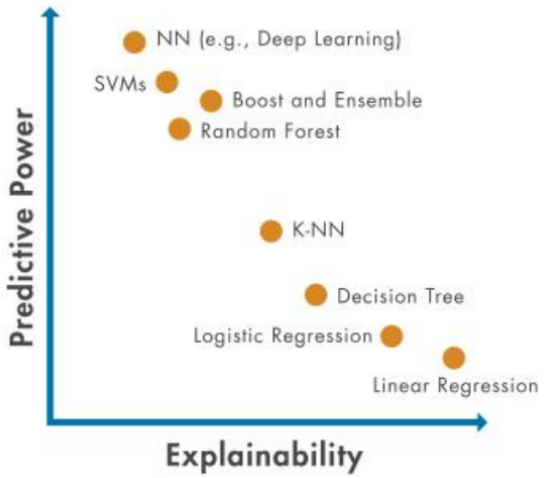


Figure 20.1: Tradeoff between model performance and explainability. Source: [72].

20.1 Reasons for explainable AI

We seek model interpretability for three main reasons:

- *Debugging*: Understanding where or why predictions go wrong and running "what if" scenarios can improve model robustness and eliminate bias.
- *Guidelines*: Black box model may violate corporate technology best practices and personal preferences.
- *Regulations*: Some government regulations require interpretability for sensitive applications such as in finance, public health and transportation.

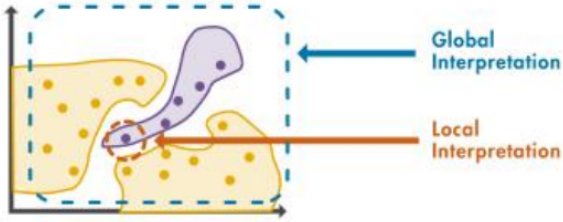


Figure 20.2: Local versus global interpretability. The two classes' instances are represented by purple and orange dots. Source: [72].

Model interpretability addresses these concerns and increases trust in the models in situations where explanations for predictions are important or required by regulation.

20.2 Levels of interpretability

Interpretability is typically applied at two levels:

- *Global methods*: Provide an overview of the most influential variables in the model based on input data and predicted output.
- *Local methods*: Provide an explanation of a single prediction result.

Figure 20.2 illustrates the difference between the local and global scope of interpretability. We can also apply interpretability to groups within our data and arrive at conclusions at the group level. For example, consider the task of classifying products to be faulty or not. It is possible to find why a group of products were classified as faulty by our model.

Popular techniques for local interpretability include *Local Interpretable Model-Agnostic Explanations* (LIME) and *Shapley values*. For global interetability, many employ *feature ranking*.

20.3 LIME

Local Interpretable Model-Agnostic Explanations (LIME) approximates a complex model in the neighborhood of the prediction of interest with a simple interpretable model, such as a linear model or decision tree. You can the use the simpler model as a surrogate to explain how the original complex model works. This idea is shown in Figure 20.3.

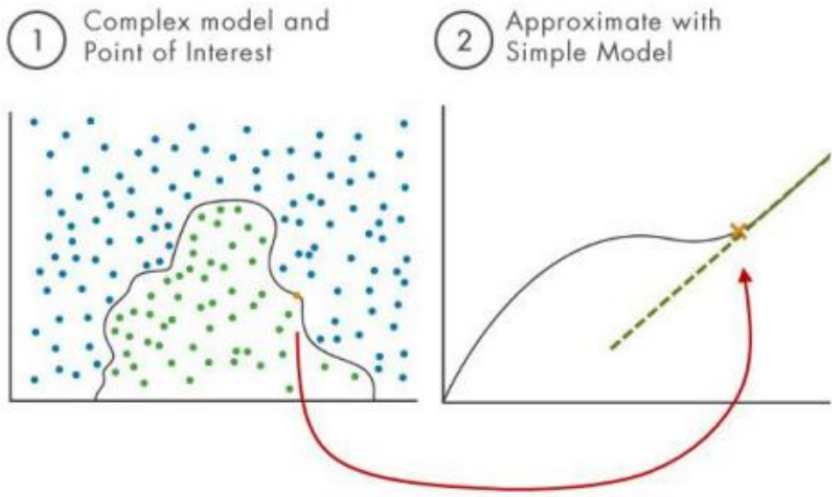


Figure 20.3: Main idea beind LIME. Source: [72].

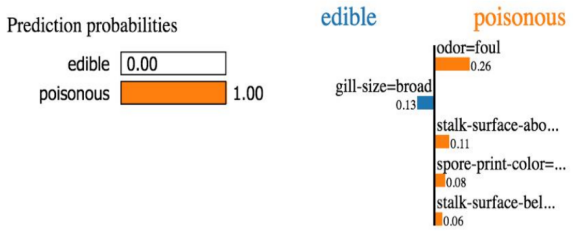


Figure 20.4: Example of LIME's output. Source: [72].

In particular, the output of LIME is a list of explanations, reflecting the contribution of each feature to the prediction of a data sample. This provides local interpretability, and it also allows to determine which feature changes will have most impact on the prediction. Consider the example reported in Figure 20.4, in which we classify a food pattern to be edible or poisonous. The pattern under examination is classified as "poisonous" since odor=foul.

20.3.1 LIME for images

We now provide an example of how LIME works when the examined pattern is an image. In Figure 20.5 the maps shows which area of the image are important to the classification of `golden retriever`. Red areas of the map have higher importance. In fact, when these areas are removed, the score for the `golden retriever` class goed down. The network focuses on the dog's face and ear to make its prediction of golden retriever. This is consistent with other explainability techniques like occlusion sensitivity or Grad-CAM.

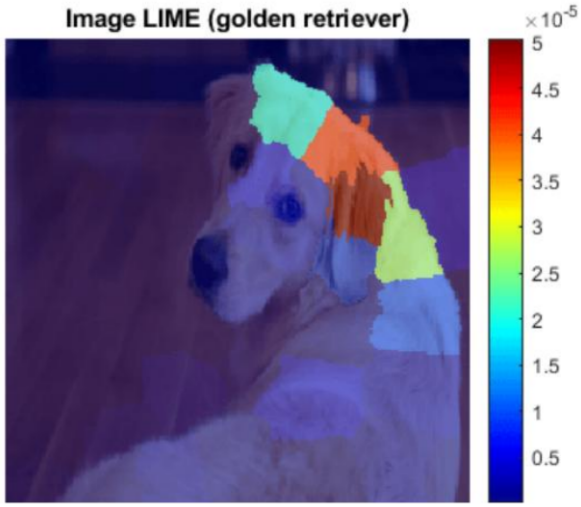


Figure 20.5: Source: [110].

For the `labrador retriever` class, the network is more focused on the dog's nose and eyes, rather than the ear, as shown in Figure 20.6. While both maps highlight the dog's forehead, the network has decided that the dog's ear and neck indicate the `golden retriever` class, while the dog's eye and nose indicate the `labrador retriever` class.

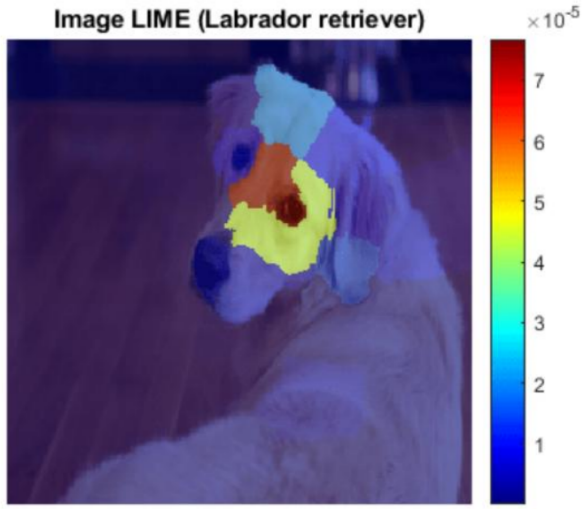


Figure 20.6: Source: [110].

20.4 Shapley values

This technique explains how much each predictor contributes to a prediction by calculating the deviation of a prediction of interest from the average. We interpret the Shapley value for the feature value j as follows: the value of the j-th feature contributed ϕ_j to the prediction of this particular instance compared to the average prediction for the dataset.

20.5 Grad-CAM

Gradient-weighted class activation mapping (Grad-CAM) is an explainability technique that can be used to help understand the predictions made by a DNN. Grad-CAM determines the importance of each neuron by considering the gradients of the target flowing through the deep network.

In particular, Grad-CAM computes the gradient of a differentiable output, e.g. the class score, with respect to the convolutional feature maps in the chosen layer. The gradients are spatially pooled to find the neuron importance weights. These weights are then used to linearly combine the activations maps and determine which features are most important to the prediction.

Suppose we have an image classification network with output y^c representing the score for class c, and want to compute the Grad-CAM map for a convolutional layer with k feature maps $A^1_{i,j}, A^2_{i,j}, \cdots, A^k_{i,j}$, where i, j indexes the pixels. The *neuron importance weight* of the k-th feature map is:

$$\alpha^c_k = \frac{1}{N} \sum_i \sum_j \frac{\partial y^c}{\partial A^k_{i,j}}, \tag{20.1}$$

where N is the total number of pixels in the feature map. The Grad-CAM map is then a weighted combination of the feature maps with an applied ReLU:

$$M = \text{ReLU}\left(\sum_k \alpha^c_k A^k\right). \tag{20.2}$$

The ReLU activation ensures we only get the features that have a positive contribution to the class of interest. The output is therefore a heatmap for the specified class, which is the same size as the feature map. The Grad-CAM map is then upsampled to the size of the input data. An overview of this procedure is shown in Figure 20.7.

Although Grad-CAM is commonly used for image classification tasks, we can compute a Grad-CAM map for any differentiable activation. For example, for semantic segmentation tasks, we can

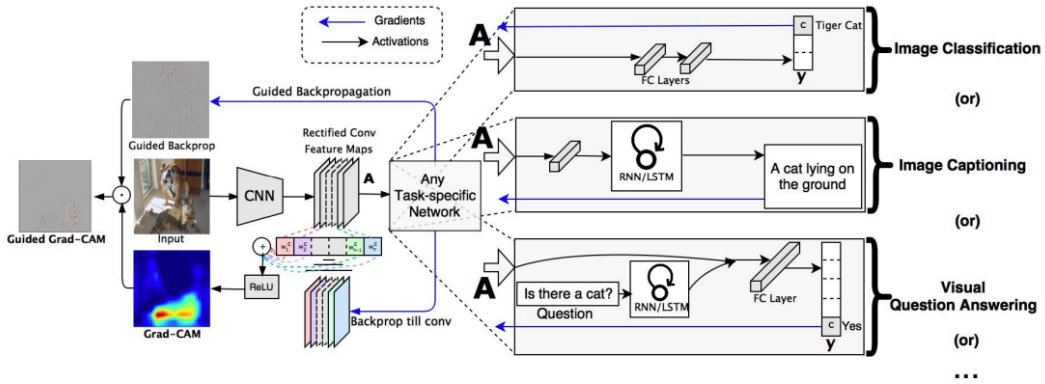


Figure 20.7: Grad-CAM overview. Source: unavailable.

calculate the Grad-CAM map by replacing y^c with

$$\sum_{(i,j)\in S} y^c_{i,j}, \tag{20.3}$$

where S is the set of pixels of interest, and

$$y^c_{i,j} = \begin{cases} 1 & \text{if pixel } (i,j) \text{ is predicted to be class } c, \\ 0 & \text{otherwise.} \end{cases} \tag{20.4}$$

20.6 Occlusion sensitivity

Occlusion sensitivity is a simple technique for understanding which parts of an image are most important for a DNN's classification. We can measure a network's sensitivity to occlusion in different regions of the data using *small perturbations*. This technique allows to gain a high-level understanding of what image features a network uses to make a particular classification and to provide insight into the reasons why a network can misclassify an image.

For instance, consider the MATLAB's function `occlusionSensitivity`. This function perturbs small areas of the input by replacing it with an occluding mask, typically a grey square. The mask moves across the image, and the change in probability score for a given class is measured as a function of mask position. We can use this method to highlight which parts of the image are most important to the classification: when that part of the image is occluded, the probability score for the predicted class will fall sharply.

Chapter 21

Quantum Deep Learning

Quantum computers systems are governed by *quantum physics*. Within quantum physics, we have a variety of tools that are used to describe the interaction between different atoms. In the case of quantum computers, these atoms are called *qubits*. A qubit acts as both a particle and a wave. A wave distribution stores a lot of data, as compared to a particle (or bit).

Back to deep learning, in the previous sections we discussed that loss functions are used to quantify how accurate a machine learning solution is. While training a machine learning model and getting its predictions, we often observe that predictions are not correct. The loss function is represented by some mathematical expression, the result of which shows by how much the algorithm has missed the target. Quantum computers also aims to reduce the loss function. It has a property called *quantum tunneling* which searches through the entire loss function space and finds the value where the loss is lowest, and hence, where the algorithm will perform the best and at a very fast rate.

21.1 Bra-ket notation

In quantum mechanics and quantum physics, the *Bra-ket notation* or *Dirac notation* is used to write equations. The notation uses angle brackets, $\langle$ and $\rangle$, and a vertical bar, $|$, to construct *bras* and *kets*.

- A ket is of the form $|\mathbf{v}\rangle$. Mathematically it denotes a vector, $\boldsymbol{v}$, in a complex vector space V, and physically it represents a state of some quantum system.
- A bra is of the form $\langle f|$. Mathematically it denotes a linear form $f : V \to \mathbb{C}$, i.e. a linear map that maps each vector in V to a number in the complex plane $\mathbb{C}$.

Letting the linear function $\langle f|$ act on a vector $|\mathbf{v}\rangle$ is written as

$$\langle f|\mathbf{v}\rangle \in \mathbb{C}. \tag{21.1}$$

Wave functions and other quantum states can be represented as vectors in a complex state using the bra-ket notation. Quantum Superposition can also be denoted by this notation.

21.2 Qubits

Quantum Computing uses *qubits* instead of bits, which are used by classical computers. A bit refers to a binary digit, and it forms the basis of classical computing. The term qubit stands for quantum binary digit. While bits have only two states, 0 and 1, qubits can have multiple states at the same time. The value ranges between 0 and 1.

21.3 Superposition

To better understand this concept, take the analogy of a coin toss. A coin has two sides, heads (1) or tails (0). While the coin is being tossed, we do not know which side it has until we stop it or it falls on

the ground. Look at the coin toss. Can you tell which side it has? It shows both 0 and 1 depending on your perspective; only if you stop it to look does it show just one side. The case with qubits is similar.

This is called the *superposition* of two states. Superposition is the ability of a quantum system to be in multiple states at the same time until it is measured [101]. This means that the probabilities of measuring a 0 or 1 are generally neither 0.0 nor 1.0. In other words, there is a likelihood that the qubit is in various states at once. In the case of the coin toss, when we get our result (heads or tails) the superposition *collapses*.

21.4 Qubit representation

Unlike a classical bit which has a value of either 0 or 1, superposition allows for a qubit to exists in a combination of the two states. In general, a qubit is represented as

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,. \tag{21.2}$$

$|0\rangle$ and $|1\rangle$ represents the two computational *basis states*, α and β are complex amplitudes corresponding to each, satisfying

$$|\alpha|^2 + |\beta|^2 = 1. \tag{21.3}$$

Observing a qubit causes a collapse into one of the basis states. These vectors are usually denoted as

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \tag{21.4}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{21.5}$$

The probability of each state being observed is proportional to the square of the amplitude of its coefficient, i.e. the probabilities of observing $|0\rangle$ and $|1\rangle$ are $|\alpha|^2$ and $|\beta|^2$ respectively. A qubit is physically realizable as a quantum system, for example the two basis states may correspond to the horizontal and vertical polarization of a photon. Superposition allows quantum computing systems to potentially achieve exponential speedups over their classical counterparts, due to the parallel computations on the probabilistic combinations of states.

Qubit basis states can also be combined to form product basis states. A set of qubits taken together is called a *quantum register*. For example, two qubits could be represented in a four-dimensional linear vector space spanned by the following product basis states:

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{21.6}$$

$$|01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \tag{21.7}$$

$$|10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \tag{21.8}$$

$$|11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}. \tag{21.9}$$

In general, n qubits are represented by a superposition state vector in 2^n dimensional space.

21.5 Entanglement

Entanglement refers to the phenomenon by which qubits exhibit correlation with one another. In general, a set of n entangled qubits exists as a superposition of 2^n basis states. Observing one or more qubits among them causes a collapse of their states, and alters the original superposition to account for the observed values of the qubits. For example, consider the 2-qubits system in the following initial state:

$$|\psi\rangle = \frac{1}{\sqrt{3}}|00\rangle + \frac{1}{\sqrt{3}}|01\rangle + \frac{1}{\sqrt{6}}|10\rangle + \frac{1}{\sqrt{6}}|11\rangle . \tag{21.10}$$

The probability that a measure of the first qubit yields a value of 0 is the probability to measure $|00\rangle$ or $|01\rangle$, that are $(1/\sqrt{3})^2$ and $(1/\sqrt{3})^2$, respectively. So the probability of measuring one of the two is the sum of the two probabilities, i.e. $(1/\sqrt{3})^2 + (1/\sqrt{3})^2 = 2/3$.

Suppose a measurement of the first qubit yields a value of 0. Then ψ collapses into ψ':

$$|\psi'\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle . \tag{21.11}$$

In this case, the probability that the second bit is 0 or 1 is 1/2. Note that the relative probabilities of the possible states are conserved, after accounting for the state collapse of the observed qubits.

21.5.1 General formula for $n = 2$ qubits

Suppose

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle , \tag{21.12}$$

and that the most significant bit (leftmost) collapses to $|0\rangle$ after a measurement. We may think that the state after the measurement is:

$$|\psi'\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \cancel{\alpha_{10}|10\rangle} + \cancel{\alpha_{11}|11\rangle}. \tag{21.13}$$

However, this state is not normalized, since the sum of squares does not add up to 1, and so we have to normalize it:

$$|\psi'\rangle = \frac{\alpha_{00}|00\rangle + \alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}. \tag{21.14}$$

21.6 Bit vs Qubit

A qubit is the basic unit of quantum information — the quantum version of the classical binary bit physically realised with a two-state device. A qubit is a two-level quantum-mechanical system. Figure 21.1 shows the main differences between a bit and a qubit. Imf with normal bits it is simply the number of representations we can represent that increases exponentially, with qubits it is the size of the vector space that increases exponentially. This means that:

- In an n-bit system a value can be in *only one* of the 2^n states.
- In an n-qubit system a value can be with 2^n different probabilities in all and 2^n different states (one probability for each different state).

This is why a quantum computer cannot be simulated on an ordinary computer. Only systems of very few qubits could be simulated.

register of n bits	register of n qubits
2^n possible states **One at a time**	2^n possible states **Simultaneously**
evaluable	partially evaluable
independent copies	Incopiables independently
individually erasable	indelible individually
lecture non destructive	lecture modify the present value
Deterministic	Probabilistic

101

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

Figure 21.2: Main differences between a regiter of n bits and a register of n qubits. Source: [72].

	Bits : 0 or 1	Qubits : 0 and 1
State	two possible exclusive states	two simultaneous possible states
Initialization	0 or 1	0
Internal representation	0 or 1	vector of 2 dimensions
Internal dimensionality	1 binary digit	two floating numbers
Modifications	logical gates	quantum gates
Read	0 or 1, deterministic	0 or 1, probabilistic

Figure 21.1: Main differences between a bit and a qubit. Source: [72].

21.7 Register of n qubits

One qubit can do nothing alone, we need more to be able to represent more data. Consider a register of 4 qubits:

$$|\psi\rangle = \alpha_0 |0000\rangle + \alpha_1 |0001\rangle + \alpha_2 |0010\rangle + \alpha_3 |0011\rangle + \cdots + \alpha_{15} |1111\rangle . \tag{21.15}$$

This register represents 16 states that are in superposition, so we have 16 states *at the same time*. In this way we can develop 16 calculations simultaneously. In general, with n qubits we can perform 2^n calculations simultaneously. The main differences between a classical register of n bits and a register of n qubits are listen in Figure 21.2.

21.7.1 Quantum error correction codes

In principle, a quantum computer with 300 qubits could perform more calculations in an instant than there are atoms in the visible universe, but qubit devices can *lose state* due to decoherence, and the operations on qubits can also experience errors. Qubits can be protected against errors using specialized codes, called *Quantum error correction codes* (QEC). Unfortunately, QEC requires significant overheads.

21.8 Basic principles

We now summarize the most important basic principles that are used with qubits:

- *Superposition*. It allows to have qubits that are both in state 0 and 1.
- *Entanglement*, It makes it possible to connect the qubits to each other to synchronize them and make copies of them. However, we are not able to read their contents or modify them independently.
- *Wave-particle duality*. It makes it possible to interact with the qubits in some cases or to make the qubits interact with each other by interference in the context of quantum algorithms.

21.9 Quantum decoherence

The superposition of qubits causes issues like *quantum decoherence*. These are unwanted collapses that happen randomly and naturally because of noise in the system. Ultimately, this leads to errors in computation. If we think that a qubit is in a superposition when it is not, and we do an operation on it, it is going to give you a different answer than you might have expected. This is why we run the same program over and over again many times, similar to training a machine learning model.

21.9.1 Causes of quantum decoherence

Quantum systems need to be isolated from the environment, because contact with the environment is what causes quantum decoherence, even a single molecule can be an issue. Qubits are chilled to near absolute zero. When qubits interact with the environment, information from the environment leaks into them, and information from within the qubits leaks out. The information that leaks out is most likely needed for a future or current computation, and the information that leaks in is random noise.

This concept is exactly like the *second law of thermodynamics*.

Theorem 21.9.1 *The total entropy of an isolated system can never decrease over time, and is constant if and only if all processes are reversible. Isolated systems spontaneously evolve towards thermodynamic equilibrium, the state with maximum entropy.*

Quantum gates have to be *reversible* because quantum mechanics are reversible and *unitary* (Section 21.11). Thus, quantum systems need to be in a state of coherence.

Quantum decoherence is more visible in minute particles as compared to bigger objects, like a book or a table. It is a fact that all materials have a particular wavelength associated with them (wave–particle duality of matter), but the bigger the item, the lesser its wavelength.

21.9.2 Environment

Qubits are created using specific atomic particles, or even photons, which are light waves. An electron of phosphorus is an example. To begin the process of creating a qubit, the atoms are placed in a superconducting magnet. This forces the electron into what we call a *spin-down position*. In an environment with a normal temperature, the electron's spin direction is not stable, as heat produces energy, forcing the electron to move. The environment needs to be contained in a supercooled refrigerator to create stability. For now, the need for large cooling systems ensures that quantum computers will be quite big for some time. Now to get the electron to spin up, a pulse of microwaves must be fired at it. Stopping the microwaves at a point anywhere between spin-down and spin-up creates a superposition. It is now a stable, coherent qubit. Any changes to temperature, light, sound, vibration, and other external factors will impact the qubit state. The big technical challenge today is making those quantum states of matter in the laboratory, and ultimately in quantum hardware, so that we have intrinsically, inherently, much more coherent and more stable qubits. So, the art of having stable qubits needs to be mastered. Pairs of entangled photons in polarization can be created when a laser beam passes through a nonlinear crystal such as beta barium borate.

21.9.3 Hydrogen atom system

Another option is considering the system consisting of the hydrogen atom. In this system, the state $|0\rangle$ of the qubit can be represented by the first energy level ($n = 0$), corresponding to the base state of the electron, and the state $|1\rangle$ by the second energy level ($n = 1$), corresponding to the excited state of the electron. The transition of the electron from one state to the other can be accomplished by subjecting the electron to a laser pulse of appropriate intensity, duration and wavelength. In an atom, the energy levels of the various electrons are discrete. Two of them can be selected to represent logical values 0 and 1. These levels correspond to particular excited states of the electrons in the atom.

21.10 Three main components of quantum computers

The quantum computer has three main components:

- *Qubit area* that houses the qubits.
- A *signal transfer method* for transferring signals to the qubits.
- Classical compute to execute a program that will send instructions.

21.11 Unitarity

In quantum physics, *unitarity* is the condition that the time evolution of a quantum state according to the Schrödinger equation is mathematically represented by a unitary operator.

Definition 21.11.1 *In linear algebra, a complex square matrix U is* unitary *if its conjugate transpose $U^\star$ is also its inverse, that is, if*

$$U^\star U = UU^\star = UU^{-1} = I. \tag{21.16}$$

In physics, especially in quantum mechanics, the conjugate transpose is referred to as the *Hermitian adjoint* of a matrix and is denoted by a dagger †, so the equation above is written as

$$U^\dagger U = UU^\dagger = UU^{-1} = I. \tag{21.17}$$

From Equation (21.17) it is possible to prove that [121]:

$$U = U^\dagger. \tag{21.18}$$

In mathematics, each linear operator A on a Euclidean vector space defines a Hermitian adjoint operator $A^\dagger$ on that space according to the rule

$$\langle Ax, y\rangle = \langle x, A^\dagger y\rangle, \tag{21.19}$$

where $\langle \cdot, \cdot \rangle$ is the inner product on the vector space.

21.12 Wave-particle duality of matter

Einstein initially assumed that photons had zero mass, which made them a peculiar sort of particle indeed. In 1905, however, he published his special theory of relativity, which related energy and mass according to the famous equation:

$$E = hu = h\frac{c}{\lambda} = mc^2, \tag{21.20}$$

where h is Planck's constant, m is the mass of the particle, u is the frequency, λ is the wavelength and c is the speed of light. In fact, for a sinusoidal of speed v and wavelength λ it holds:

$$u = \frac{v}{\lambda}. \tag{21.21}$$

According to this theory, a photon of wavelength λ and frequency u has nonzero mass, which is given as follows:

$$m = \frac{E}{c^2} = \frac{hu}{c^2} = \frac{h}{\lambda c}. \tag{21.22}$$

That is, light, which had always been regarded as wave, also has properties typical of particles, a condition known as *wave-particle duality*, where matter and energy have properties typical of both waves and particles. Depending on conditions, light could be viewed as either a wave or a particle.

In 1922, Compton reported the results of experiments involving the collision of x-rays and electrons that supported the particle nature of light. At about the same time, De Broglie began to wonder whether the converse was true: could particles exhibit the properties of waves? He proposed that a particle such as an electron could be described by a wave whose wavelength is given by

$$\lambda = \frac{h}{mv}. \tag{21.23}$$

As an example, we calculate the wavelength of a baseball, which has a mass of 149 g and a speed of 44.7 m/s. By Equation (21.23):

$$\lambda = \frac{6.626 \times 10^{-34} J \cdot s}{(0.149kg)(44.7m \cdot s)} = 9.95 \times 10^{-35} m. \tag{21.24}$$

Given that the diameter of the nucleus of an atom is approximately $10^{-14}m$, the wavelength of the baseball is almost unimaginably small.

21.13 Bloch's sphere

The *Bloch's sphere* is a mathematical representation of the qubit. This model is linked to the representation of the state of a qubit or of any quantum with two states by a two-dimensional vector whose "norm" length is always 1. This vector has the particularity of having two elements: a real number α and a complex number β. The *spherical coordinates system* is a system defined by the radius R and two angles θ and ϕ. In the Bloch's sphere, $R = 1$, and only ϕ and θ are variable. The qubit state function in the Bloch's Sphere is defined as follows.

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle, \tag{21.25}$$

where

$$|\alpha|^2 + |\beta|^2 = 1. \tag{21.26}$$

We impose

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \cos\frac{\theta}{2} \\ e^{i\phi}\sin\frac{\theta}{2}, \end{bmatrix} \tag{21.27}$$

where $0 \leq \theta \leq \pi$ and $0 \leq \phi \leq 2\pi$. Therefore we express Equation (21.25) as:

$$|\psi\rangle = \cos\frac{\theta}{2} |0\rangle + e^{i\phi}\sin\frac{\theta}{2} |1\rangle. \tag{21.28}$$

The Bloch's sphere is shown in Figure 21.3.

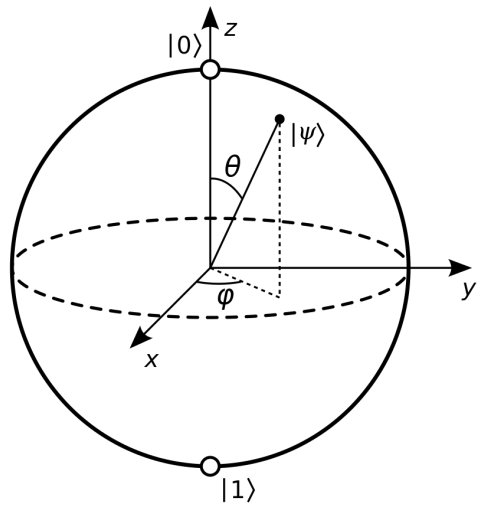


Figure 21.3: Bloch's sphere. Source: [72].

21.13.1 Bloch's sphere examples

The real values θ and ψ are variable. We now show different configurations of these values and the represented qubit.

If $\theta = \pi$ and $\psi = 0$ we have:

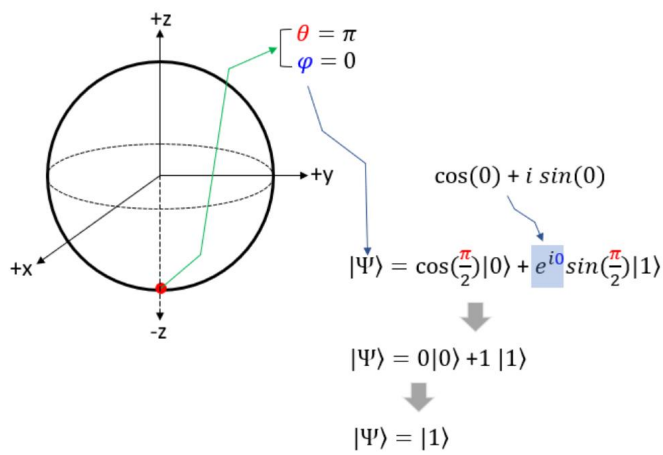


Figure 21.4: Source: [72].

If $\theta = 0$ and $\psi = 0$ we have:

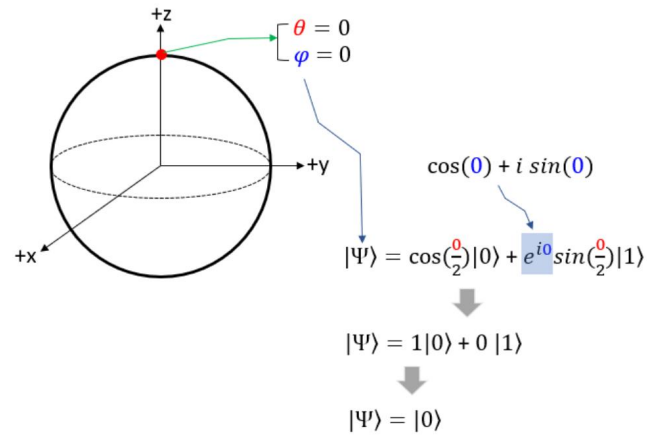


Figure 21.5: Source: [72].

If $\theta = \pi/2$ and $\phi = 3\pi/2$ we have:

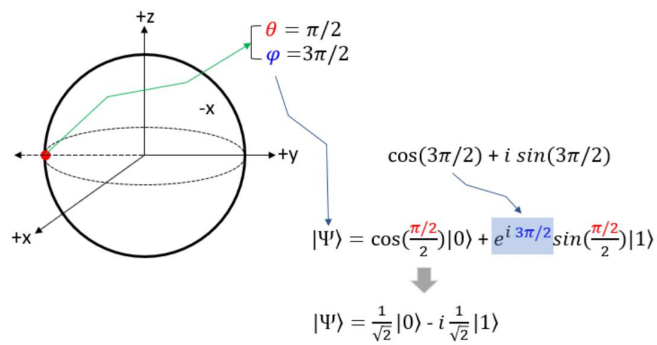


Figure 21.6: Source: [72].

21.14 Other basis

The basis of $|1\rangle$ and $|0\rangle$ is the most common in quantum computing. The $|+\rangle$ state and the $|-\rangle$ state are defined as following in terms of the $|0\rangle$, $|1\rangle$ basis:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \tag{21.29}$$

$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \tag{21.30}$$

Trivially:

$$|0\rangle = \frac{1}{\sqrt{2}}(|+\rangle + |-\rangle), \tag{21.31}$$

$$|1\rangle = \frac{1}{\sqrt{2}}(|+\rangle - |-\rangle). \tag{21.32}$$

They are both in an equal superposition, with equal probabilities of 1/2 of measuring 0 and 1. The difference is solely a phase difference.

In quantum mechanics, when we take a measurement the state collapses onto one of the states of the system. Therefore for these basis there are two different types of measurement. The first is called a *Z measurement* and in taking this measurement we collapse the state onto one of the $|0\rangle$, $|1\rangle$ states

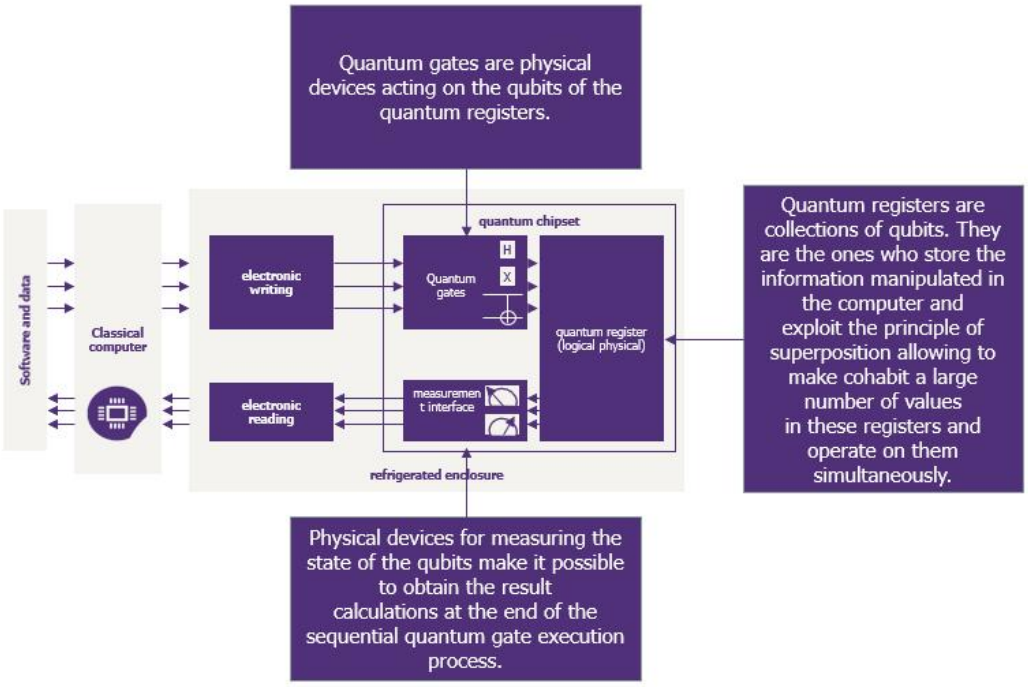


Figure 21.7: Global architecture of a quantum computer. Source: [72].

(Z *basis*). The second is called an X *measurement* and this collapses the state onto one of the $|+\rangle$, $|-\rangle$ states (X *basis*). Given a qubit as in Equation (21.2), we have

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \frac{\alpha}{\sqrt{2}}(|+\rangle + |-\rangle) + \frac{\beta}{\sqrt{2}}(|+\rangle - |-\rangle) = \frac{\alpha+\beta}{\sqrt{2}}\,|+\rangle + \frac{\alpha-\beta}{\sqrt{2}}\,|-\rangle\,. \tag{21.33}$$

21.15 Global architecture of a quantum computer

Figure 21.7 shows the global architecture of a quantum computer.

21.16 Basic operations

Suppose we represent classical bits in vector form as $|0\rangle = [1,0]^T$ for 0 and $|1\rangle = [0,1]^T$ for 1. We represent an operation defined by the matrix A as:

$$A\mathbf{x} = \mathbf{b}. \tag{21.34}$$

21.16.1 Identity

The *identity* operation does not change the bit and is defined as:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{21.35}$$

Therefore:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} |0\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle\,, \tag{21.36}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} |1\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle\,. \tag{21.37}$$

21.16.2 Negation

The *negation* operation switches the bit and is defined as:

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{21.38}$$

Therefore:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} |0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle, \tag{21.39}$$

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} |1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle, \tag{21.40}$$

21.16.3 Constant-0

The *constant-0* operation sets the bit to zero and is defined as:

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}. \tag{21.41}$$

Therefore:

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} |0\rangle = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle, \tag{21.42}$$

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} |1\rangle = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle, \tag{21.43}$$

21.16.4 Constant-1

The *constant-1* operation sets the bit to one and is defined as:

$$A = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}. \tag{21.44}$$

Therefore:

$$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} |0\rangle = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle, \tag{21.45}$$

$$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} |1\rangle = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle, \tag{21.46}$$

21.16.5 Tensor product of vectors

We now review the *tensor product of vectors*. Given two vectors $[x_0, x_1]^T$ and $[y_0, y_1]^T$, the tensor product of vectors is:

$$\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \otimes \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0 \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \\ x_1 \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} x_0 y_0 \\ x_0 y_1 \\ x_1 y_0 \\ x_1 y_1 \end{bmatrix} \tag{21.47}$$

The tensor product of three vectors is:

$$\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \otimes \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \otimes \begin{bmatrix} z_0 \\ z_1 \end{bmatrix} = \begin{bmatrix} x_0 y_0 z_0 \\ x_0 y_0 z_1 \\ x_0 y_1 z_0 \\ x_0 y_1 z_1 \\ x_1 y_0 z_0 \\ x_1 y_0 z_1 \\ x_1 y_1 z_0 \\ x_1 y_1 z_1 \end{bmatrix} \tag{21.48}$$

As an example, we compute the following tensor products of vectors.

$$\begin{bmatrix}1\\2\end{bmatrix} \otimes \begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}3\\4\\6\\8\end{bmatrix}. \tag{21.49}$$

$$\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{bmatrix}. \tag{21.50}$$

21.16.6 Representing multiple bits

We can use the tensor product of vectors to represent a registry of multiple bits. For example:

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \tag{21.51}$$

$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \tag{21.52}$$

$$|10\rangle = |1\rangle \otimes |0\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \tag{21.53}$$

$$|11\rangle = |1\rangle \otimes |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}, \tag{21.54}$$

$$|100\rangle = |1\rangle \otimes |0\rangle \otimes |0\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\0\end{bmatrix}. \tag{21.55}$$

We call this tensored representation the *product state*. It is easy to see that we can factor the product state back into the individual state representation. The product state of n bits is a vector of size 2^n.

21.16.7 CNOT

The *controlled NOT* gate (CNOT) is a quantum logic gate that is an essential component in the construction of a gate-based quantum computer [18]. It operates on pairs of bits, one of which is the *control bit* and the other is the *target bit*.

- If the control bit is 1, then the target bit is flipped.
- If the control bit is 0, then the target bit is unchanged.

The control bit is always unchanged. We consider the most-significant bit (left) as control and the least significant bit (right) as target. The we have:

$$A = C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{21.56}$$

Therefore:

$$C\left|00\right\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \left|00\right\rangle, \tag{21.57}$$

$$C\left|01\right\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \left|01\right\rangle, \tag{21.58}$$

$$C\left|10\right\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \left|11\right\rangle, \tag{21.59}$$

$$C\left|11\right\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \left|10\right\rangle. \tag{21.60}$$

The C matrix is invertible. In fact, quantum computers only use reversible operations.

21.16.8 Qubits and superposition

We represented operations on bits by matrix multiplication on bit vectors. In addition, registry of multiple bits are written as the tensor product of single-bit vectors. This notion can be generalized to qubits. By Equation (21.2), we can represent a qubit by the vector

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \tag{21.61}$$

where α and β are complex numbers, and

$$|\alpha|^2 + |\beta|^2 = 1. \tag{21.62}$$

Note that the classical bit vectors $[1, 0]^T$ and $[0, 1]^T$ fit within this definition.

Multiple qubits (registers) are represented by the tensor product of vectors, i.e.:

$$\begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix}. \tag{21.63}$$

Since we are dealing with probabilities, it holds:

$$|ac|^2 + |ad|^2 + |bc|^2 + |bd|^2 = 1. \tag{21.64}$$

For example, consider the system:

$$\begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} \otimes \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{bmatrix} = \sqrt{\begin{bmatrix} \Pr(\text{collapsing to } \left|00\right\rangle) \\ \Pr(\text{collapsing to } \left|01\right\rangle) \\ \Pr(\text{collapsing to } \left|10\right\rangle) \\ \Pr(\text{collapsing to } \left|11\right\rangle) \end{bmatrix}}, \tag{21.65}$$

where the square root is applied element wise. This means that there is a 1/4 chance each of collapsing to $\left|00\right\rangle$, $\left|01\right\rangle$, $\left|10\right\rangle$ or $\left|11\right\rangle$.

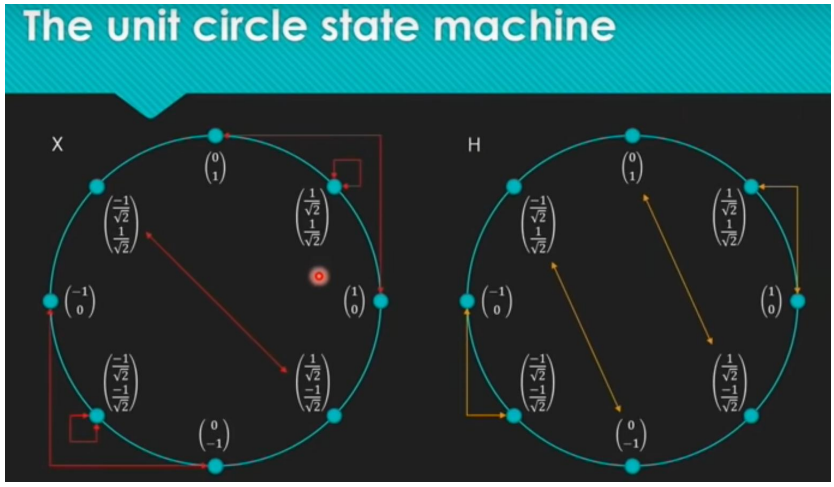


Figure 21.8: Visualization of reversibility of quantum operators. X and the red arrows represent the flip bit operator, while H and the yellow arrows represent the Hadamard operator. Source: [90].

21.16.9 Representing superposition

Informally, superposition means that the qubit is both 0 and 1 at the same time. When we measure the qubit, it collapses to 0 or 1. We usually do this at the end of a quantum computation to get a result. We operate on qubits using the matrices already seen for classical bits (bit flip, CNOT, etc). Matrix operators model the effect of some device which manipulates qubit spin or polarization without measuring and collapsing it. For example, by applying the bit flip to the qubit $[1/2, \sqrt{3}/2]^T$ we obtain:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1/2 \\ \sqrt{3}/2 \end{bmatrix} = \begin{bmatrix} \sqrt{3}/2 \\ 1/2 \end{bmatrix}. \tag{21.66}$$

However, several matrix operators only make sense in a quantum context. We now present some of them.

21.16.10 Hadamard gate

The *Hadamard gate* is defined as:

$$A = H = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{21.67}$$

The Hadamard gate has two properties. Firstly, it takes a 0 or 1 bit and puts it into exactly equal superposition:

$$H\left|0\right\rangle = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}, \tag{21.68}$$

$$H\left|1\right\rangle = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix}. \tag{21.69}$$

Secondly, it takes a qubit in exactly-equal superposition and transforms it into a 0 or 1 bit. Infact, in this context, operations are their own inverse, since H is unitary.

$$H \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \left|0\right\rangle, \tag{21.70}$$

$$H \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \left|1\right\rangle. \tag{21.71}$$

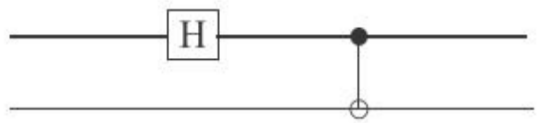
The application of the bit flip and Hadamard are shown in Figure 21.8. The Hadamard gate can be generalized for multiple qubits. The generalized operator transforms each of the qubits according to

the Hadamard gate. We now show the Hadamard operator for $n = 2$ qubits, which gives the idea for an arbitrary value of n. Let $H^{\otimes 2}$ be such operator. It holds:

$$H^{\otimes 2} |00\rangle = H |0\rangle \otimes H |0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle). \tag{21.72}$$

21.16.11 Bell's states generation

Bell's states are specific quantum states of two qubits that represent the simplest examples of quantum entanglement [10]. Bell's states can be generated with a Hadamard gate and a CNOT gate. Consider the following diagram:



The first qubit q_0 represents the first horizontal line, while the second qubit q_1 represents the second horizonal line. The first qubit is passed through a Hadamard gate and then both qubits are entangled by a CNOT gate. For instance, suppose the input to the system is $|00\rangle = |0\rangle \otimes |0\rangle$. The the Hadamard gate changes the state to

$$H |0\rangle \otimes |0\rangle = \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |10\rangle . \tag{21.73}$$

Note that H is applied only to the first qubit, but the state of the system is given by the tensor product of vectors. After the CNOT gate:

$$C\left(\frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |10\rangle\right) = C\left(\frac{1}{\sqrt{2}} |00\rangle\right) + C\left(\frac{1}{\sqrt{2}} |10\rangle\right) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \tag{21.74}$$

We are use the Hadamard gates to apply superposition into qubits and CNOT gates to generate entanglement.

21.16.12 Pauli operators

The Pauli matrices are a set of three 2×2 complex matrices that are Hermitian and unitary [83]:

$$\sigma_x = X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \tag{21.75}$$

$$\sigma_y = Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \tag{21.76}$$

$$\sigma_z = Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{21.77}$$

21.16.13 Rotation operators

Given Equation (21.28), we reinterpret θ and ϕ in spherical coordinates. We define the *Bloch vector* $\mathbf{a} \in \mathbb{R}$ as:

$$\mathbf{a} = \begin{bmatrix} \sin\theta\cos\phi & \sin\theta\sin\phi & \cos\theta \end{bmatrix}^T . \tag{21.78}$$

When the Pauli matrices are exponentiated, they give rise to the rotation operators, which rotate the Bloch vector around the x, y and z axis by a given angle θ (unrelated from Equation (21.78)):

$$R_x(\theta) = e^{-i\frac{\theta}{2}X}, \tag{21.79}$$

$$R_y(\theta) = e^{-i\frac{\theta}{2}Y}, \tag{21.80}$$

$$R_z(\theta) = e^{-i\frac{\theta}{2}Z}. \tag{21.81}$$

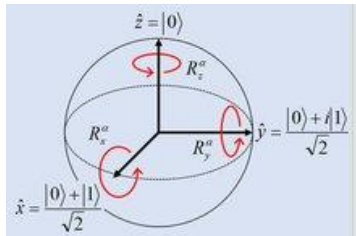


Figure 21.9: Rotation produced by the rotation operators. Source: [72].

Now, if operator A satisfies $A^2 = I$, it can be shown that:

$$e^{i\theta A} = \cos(\theta)I + i\sin(\theta)A. \tag{21.82}$$

Since all the Pauli operators are unitary, Equation (21.82) can be employed to compute $R_x(\theta)$, $R_y(\theta)$ and $R_z(\theta)$, thus obtaining:

$$R_x(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}, \tag{21.83}$$

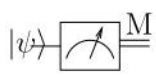
$$R_y(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}, \tag{21.84}$$

$$R_z(\theta) = \begin{bmatrix} e^{-i\theta/2} & -0 \\ 0 & e^{i\theta/2} \end{bmatrix}. \tag{21.85}$$

The rotations produced by these operators are shown in Figure 21.9.

21.17 Measurements

The circuit symbol for a measurement M is the following:



A quantum measurement is described by a set of operators acting on the state space of the system. The aim of these operations is to obtain the probability p of a measurement result m: the result is a classical bit that will be 0 or 1 with probability $|\alpha|^2$ and $|\beta|^2$, respectively.

21.18 Hybrid quantum network

"Classical" layers can be used to represent the images as features, then the feature vector is used to feed a quantum processor. The *quantum layers* have set of weights that are tuned for minimizing a given loss, while the classification phase is performed using classical methods. The measurement outcomes are fed into a fully connected layer with the softmax activation function to generate the predicted label. The loss function between the predicted label and the true label is computed, and a classical optimizer is employed to update parameters. Essentially, in this network the quantum processor is a *feature extractor* for the fully connected layers.

The *hybrid quantum network* in Figure 21.10 can be employed for classification of 10 classes of 28×28 images. We have a register of n qubits all initialized to 0.

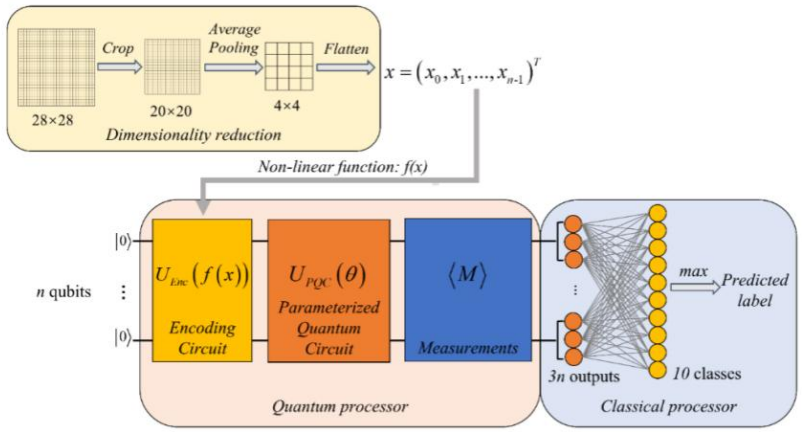


Figure 21.10: Hybrid quantum network. Source: [72].

21.18.1 Encoding circuit

Figure 21.11 shows the details of the quantum processor. The input data $\mathbf{x} = \begin{bmatrix} x_0 & x_1 & \cdots & x_{n-1} \end{bmatrix}$ is encoded in a quantum state $|\phi_{\text{in}}\rangle$ by applying the *encoding circuit* U_{Enc}:

$$U_{\text{Enc}} = \otimes_{i=0}^{n-1} R_y(f(x_i)) \tag{21.86}$$

on an initial state $\otimes_{i=0}^{n-1} |0\rangle$, where

$$f(x_i) = \arcsin x_i \tag{21.87}$$

is a nonlinear function that maps the data to angle space. Since

$$R_y(\theta)\,|0\rangle = \cos(\frac{\theta}{2})\,|0\rangle + \sin(\frac{\theta}{2})\,|1\rangle\,, \tag{21.88}$$

it is easy to see that:

$$|\phi_{\text{in}}\rangle = \otimes_{i=0}^{n-1} \begin{bmatrix} \cos(f(x_i)/2) \\ \sin(f(x_i)/2) \end{bmatrix} = \begin{bmatrix} \cos(f(x_0)/2) \\ \sin(f(x_0)/2) \end{bmatrix} \otimes \cdots \otimes \begin{bmatrix} \cos(f(x_{n-1})/2) \\ \sin(f(x_{n-1})/2) \end{bmatrix}. \tag{21.89}$$

This nonlinear quantum encoding circuit can map the input data into a higher dimensional space, which will facilitate the subsequent classification process.

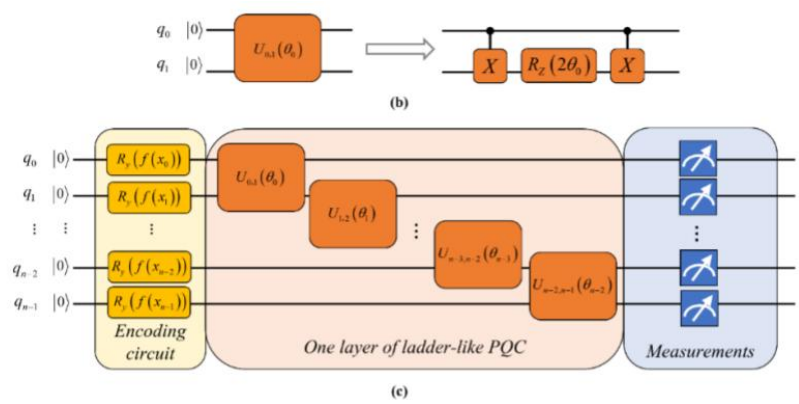


Figure 21.11: Quantum processor details. X represents the CNOT gate, $R_z(2\theta_i)$ represents rotation-z gate with an angle of $2\theta_i$. Source: [72].

21.18.2 Ladder-like PQC

The *ladder-like Parametrized Quandum Circuit* (PQC) is the main part of the PQC quantum circuit to perform the calculation. This architecture is capable of disentangling the input state as much as possible, even with a small number of quantum gates. In this ladder-like circuit, two qubit quantum gates in steps of one are applied. In other words, a family of quantum gates with different parameters is performed on each pair of nearest-neighbor qubits, which allows to capture the quantum correlations of a specific scale on the same layer of the network. In the terms of the input state $|\phi_{\text{in}}\rangle$, the effective information (discrete labels in the classification task) is embedded in the quantum subsystem. Therefore, the PQC is used for performing quantum computations to extract the information hidden in this quantum subsystem. Specifically, the purpose of implementing unitary gates is to remove the superposition in quantum data, leaving the information containing the label. Then, with an appropriate measurement strategy, the classical information is extracted, but it may not be the direct representation of the label, so further classical classification is required.

$U_{PQC}(\boldsymbol{\theta})$ consists of unary gates $U_i(\theta_i)$ with parameters

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 & \theta_1 & \cdots & \theta_{m-1} \end{bmatrix}, \tag{21.90}$$

where m is the number of quantum gates. In the learning phase the parameters are optimized by the Adam optimizer, so that the initial state can evolve the desired state through the operation of the PQC.

Bibliography

[1] *3D volumes of neurons*. URL: https://compphysics.github.io/MachineLearning/doc/pub/week42/html/._week42-bs057.html.

[2] *A Gentle Introduction to Positional Encoding in Transformer Models, Part 1*. URL: https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/.

[3] *Adversarial example using FGSM*. URL: https://www.tensorflow.org/tutorials/generative/adversarial_fgsm#:~:text=Adversarial%20examples%20are%20specialised%20inputs,the%20contents%20of%20the%20image..

[4] Forest Agostinelli et al. "Learning activation functions to improve deep neural networks". In: *arXiv preprint arXiv:1412.6830* (2014).

[5] Afshine Amidi et al. "EnzyNet: enzyme classification using 3D convolutional neural networks on spatial representation". In: *PeerJ* 6 (2018), e4750.

[6] *Automatic differentiation*. URL: https://en.wikipedia.org/wiki/Automatic_differentiation.

[7] *Bagging in Machine Learning*. URL: https://deepchecks.com/glossary/bagging-in-machine-learning/.

[8] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling". In: *CoRR* abs/1803.01271 (2018). arXiv: 1803.01271. URL: http://arxiv.org/abs/1803.01271.

[9] Yutong Bai et al. *Are Transformers More Robust Than CNNs?* 2021. arXiv: 2111.05464 [cs.CV].

[10] *Bell state*. URL: https://en.wikipedia.org/wiki/Bell_state.

[11] *Bias–variance tradeoff*. URL: https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff.

[12] *capsule networks*. URL: https://github.com/gram-ai/capsule-networks/blob/master/capsule_network.py.

[13] Liang-Chieh Chen et al. "Encoder-decoder with atrous separable convolution for semantic image segmentation". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 801–818.

[14] Liang-Chieh Chen et al. "Rethinking atrous convolution for semantic image segmentation". In: *arXiv preprint arXiv:1706.05587* (2017).

[15] *Co-adaptation*. URL: https://machinelearning.wtf/terms/co-adaptation/.

[16] *Comprehensive Introduction to Graph Neural Networks (GNNs)*. URL: https://www.datacamp.com/tutorial/comprehensive-introduction-graph-neural-networks-gnns-tutorial.

[17] *Contrastive Loss Explained*. URL: https://towardsdatascience.com/contrastive-loss-explaned-159f2d4a87ec.

[18] *Controlled NOT gate*. URL: https://en.wikipedia.org/wiki/Controlled_NOT_gate.

[19] *CONV2D*. URL: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html.

[20] *Conv2d certain values for groups and out channels don't work*. URL: https://discuss.pytorch.org/t/conv2d-certain-values-for-groups-and-out-channels-dont-work/14228/2.

[21] *Convolution and cross-correlation in neural networks*. URL: https://pyimagesearch.com/2021/05/14/convolution-and-cross-correlation-in-neural-networks/.

[22] *Cosa significa pattern?* URL: https://www.andreaminini.com/ai/machine-learning/pattern.

[23] *Covariate Shift*. URL: https://www.doc.ic.ac.uk/~bkainz/teaching/DL/T04_covariateShift.pdf.

[24] Antonia Creswell et al. "Generative adversarial networks: An overview". In: *IEEE signal processing magazine* 35.1 (2018), pp. 53–65.

[25] Ekin D. Cubuk et al. *AutoAugment: Learning Augmentation Policies from Data*. 2019. arXiv: 1805.09501 [cs.CV].

[26] Mengyu Dai and Haibin Hang. "Manifold matching via deep metric learning for generative modeling". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 6587–6597.

[27] George Dasoulas, Johannes Lutzeyer, and Michalis Vazirgiannis. *Learning Parametrised Graph Shift Operators*. 2022. arXiv: 2101.10050 [cs.LG].

[28] *Data Augmentation Library*. URL: https://github.com/aleju/imgaug.

[29] *DeepLabV3*. URL: https://medium.com/@itberrios6/deeplabv3-c0c8c93d25a4.

[30] *Delta rule*. URL: https://en.wikipedia.org/wiki/Delta_rule.

[31] Janez Demšar. "Statistical comparisons of classifiers over multiple data sets". In: *The Journal of Machine learning research* 7 (2006), pp. 1–30.

[32] Xinyang Deng et al. "An improved method to construct basic probability assignment based on the confusion matrix for classification problem". In: *Information Sciences* 340 (2016), pp. 250–261.

[33] *DepthwiseConv2D layer*. URL: https://keras.io/api/layers/convolution_layers/depthwise_convolution2d/#:~:text=Depthwise%20convolution%20is%20a%20type,in%20a%20depthwise%20separable%20convolution..

[34] *Detecting Concept Shift: Impact on Machine Learning Performance*. URL: https://towardsdatascience.com/detecting-concept-shift-impact-on-machine-learning-performance-16923261cda8#:~:text=A%20Concept%20shift%20is%20a,which%20the%20features'%20distributions%20change..

[35] Laurent Dinh et al. "Sharp minima can generalize for deep nets". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 1019–1028.

[36] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV].

[37] *Dropout layer*. URL: https://keras.io/api/layers/regularization_layers/dropout/#:~:text=The%20Dropout%20layer%20randomly%20sets,over%20all%20inputs%20is%20unchanged..

[38] Shiv Ram Dubey et al. "diffGrad: an optimization method for convolutional neural networks". In: *IEEE transactions on neural networks and learning systems* 31.11 (2019), pp. 4500–4511.

[39] *Ensemble learning*. URL: http://www.scholarpedia.org/article/Ensemble_learning.

[40] *Fast R-CNN for object detection*. URL: https://towardsdatascience.com/fast-r-cnn-for-object-detection-a-technical-summary-a0ff94faa022.

[41] *Faster RCNN Explained for Object Detection Tasks*. URL: https://blog.paperspace.com/faster-r-cnn-explained-object-detection/.

[42] *Figure of image segmentation*. URL: https://www.researchgate.net/figure/A-55documentclass12ptminimal-usepackageamsmath-usepackagewasysym_fig6_342157198.

[43] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[44] Ian Goodfellow et al. "Generative adversarial networks". In: *Communications of the ACM* 63.11 (2020), pp. 139–144.

[45] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. *Explaining and Harnessing Adversarial Examples*. 2015. arXiv: 1412.6572 [stat.ML].

[46] *Graph Convolutional Networks —Deep Learning on Graphs*. URL: https://towardsdatascience.com/graph-convolutional-networks-deep-99d7fee5706f.

[47] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. 2018. arXiv: 1706.02216 [cs.SI].

[48] Dan Hendrycks and Kevin Gimpel. *Gaussian Error Linear Units (GELUs)*. 2023. arXiv: 1606.08415 [cs.LG].

[49] *How to compute number of weights of CNN?* URL: https://stackoverflow.com/questions/59343843/how-to-compute-number-of-weights-of-cnn.

[50] *How to Conduct the Wilcoxon Sign Test*. URL: https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/how-to-conduct-the-wilcox-sign-test/.

[51] *How to define specific number of convolutional kernels/filters in pytorch?* URL: https://stackoverflow.com/questions/56030884/how-to-define-specific-number-of-convolutional-kernels-filters-in-pytorch.

[52] *How Transformers work in deep learning and NLP: an intuitive introduction*. URL: https://theaisummer.com/transformer/.

[53] Gao Huang et al. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[54] *Intersection over Union (IoU) in Object Detection and Segmentation*. URL: https://learnopencv.com/intersection-over-union-iou-in-object-detection-and-segmentation/.

[55] Bin Jiang, Guo-Sheng Yang, and Huan-Long Zhang. "Comparative study of dimension reduction and recognition algorithms of DCT and 2DPCA". In: *2008 international conference on machine learning and cybernetics*. Vol. 1. IEEE. 2008, pp. 407–410.

[56] Honghua Jiang et al. "CNN feature based graph convolutional network for weed and crop recognition in smart farming". In: *Computers and Electronics in Agriculture* 174 (2020), p. 105450. ISSN: 0168-1699. DOI: https://doi.org/10.1016/j.compag.2020.105450. URL: https://www.sciencedirect.com/science/article/pii/S0168169919321349.

[57] Xiaojie Jin et al. "Deep Learning with S-Shaped Rectified Linear Activation Units". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30 (Dec. 2015). DOI: 10.1609/aaai.v30i1.10287.

[58] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[59] Teerath Kumar et al. *Image Data Augmentation Approaches: A Comprehensive Survey and Future directions*. 2023. arXiv: 2301.02830 [cs.CV].

[60] Ludmila I Kuncheva and Christopher J Whitaker. "Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy". In: *Machine learning* 51 (2003), pp. 181–207.

[61] *Laplacian matrix*. URL: https://en.wikipedia.org/wiki/Laplacian_matrix.

[62] *Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning*. URL: https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1.

[63] Fan Liang et al. "Survey of Graph Neural Networks and Applications". In: *Wireless Communications and Mobile Computing* 2022 (July 2022). DOI: 10.1155/2022/9261537.

[64] *Local Binary Patterns*. URL: http://www.scholarpedia.org/article/Local_Binary_Patterns.

[65] *mAP (mean Average Precision) for Object Detection*. URL: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173.

[66] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. "Split Computing and Early Exiting for Deep Learning Applications: Survey and Research Challenges". In: *ACM Comput. Surv.* 55.5 (Dec. 2022). ISSN: 0360-0300. DOI: 10.1145/3527155. URL: https://doi.org/10.1145/3527155.

[67] *McCulloch-Pitts Neuron — Mankind's First Mathematical Model Of A Biological Neuron*. URL: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1.

[68] *McNemar's test*. URL: https://en.wikipedia.org/wiki/McNemar%27s_test.

[69] *Metric Learning Tips and Tricks*. URL: https://towardsdatascience.com/metric-learning-tips-n-tricks-2e4cfee6b75b.

[70] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[71] *MNIST dataset*. URL: https://en.wikipedia.org/wiki/MNIST_database.

[72] Loris Nanni. "Deep Learning".

[73] Loris Nanni and Alessandra Lumini. "An experimental comparison of ensemble of classifiers for bankruptcy prediction and credit scoring". In: *Expert systems with applications* 36.2 (2009), pp. 3028–3033.

[74] Loris Nanni et al. "Closing the performance gap between siamese networks for dissimilarity image classification and convolutional neural networks". In: *Sensors* 21.17 (2021), p. 5809.

[75] Loris Nanni et al. "Experiments of image classification using dissimilarity spaces built with siamese networks". In: *Sensors* 21.5 (2021), p. 1573.

[76] Loris Nanni et al. "Feature transforms for image data augmentation". In: *Neural Computing and Applications* 34.24 (Aug. 2022), pp. 22345–22356. ISSN: 1433-3058. DOI: 10.1007/s00521-022-07645-z. URL: http://dx.doi.org/10.1007/s00521-022-07645-z.

[77] Loris Nanni et al. "Stochastic selection of activation layers for convolutional neural networks". In: *Sensors* 20.6 (2020), p. 1626.

[78] *Neural Networks Intuitions: 2. Dot product, Gram Matrix and Neural Style Transfer*. URL: https://towardsdatascience.com/neural-networks-intuitions-2-dot-product-gram-matrix-and-neural-style-transfer-5d39653e7916.

[79] *Neural style transfer*. URL: https://en.wikipedia.org/wiki/Neural_style_transfer.

[80] *One Cycle Policy — A Deep Understanding*. URL: https://medium.com/@varunsivamani/one-cycle-policy-a-deep-understanding-6d4d352ec7b1.

[81] *One stop for object detectors*. URL: https://medium.com/swlh/one-stop-for-object-detectors-2c99daa08c50.

[82] *Overview of GAN structure*. URL: https://developers.google.com/machine-learning/gan/gan_structure.

[83] *Pauli matrices*. URL: https://en.wikipedia.org/wiki/Pauli_matrices.

[84] *Pointwise Convolution*. URL: https://paperswithcode.com/method/pointwise-convolution.

[85] *Pre Training*. URL: https://www.larksuite.com/en_us/topics/ai-glossary/pre-training.

[86] *Precision and recall*. URL: https://en.wikipedia.org/wiki/Precision_and_recall.

[87] *Preparing Text Data for Transformers: Tokenization, Mapping and Padding*. URL: https://medium.com/@lokaregns/preparing-text-data-for-transformers-tokenization-mapping-and-padding-9fbfbce28028.

[88] *Q learning*. URL: https://en.wikipedia.org/wiki/Q-learning.

[89] *QLearning Algorithm: From Explanation to Implementation*. URL: https://towardsdatascience.com/q-learning-algorithm-from-explanation-to-implementation-cdbeda2ea187.

[90] *Quantum Computing for Computer Scientists*. URL: https://www.microsoft.com/en-us/research/video/quantum-computing-computer-scientists/.

[91] *RCNN in depth*. URL: https://www.youtube.com/watch?v=eJ7now8Q-w4.

[92] *Receiver operating characteristic*. URL: https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

[93] *Recurrent neural network*. URL: https://en.wikipedia.org/wiki/Recurrent_neural_network.

[94] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[95] *ResNet (actually) explained in under 10 minutes*. URL: https://www.youtube.com/watch?v=o_3mboe1jYI.

[96] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. "Dynamic routing between capsules". In: *Advances in neural information processing systems* 30 (2017).

[97] *Self-Supervised Learning*. URL: https://cs229.stanford.edu/notes2021spring/notes2021spring/cs229_lecture_selfsupervision_final.pdf.

[98] *Siamese Neural Networks in a nutshell*. URL: https://imbiswas.medium.com/siamese-neural-networks-in-a-nutshell-12b1f82e56f9.

[99] Leslie N Smith. "Cyclical learning rates for training neural networks". In: *2017 IEEE winter conference on applications of computer vision (WACV)*. IEEE. 2017, pp. 464–472.

[100] Leslie N Smith and Nicholay Topin. "Super-convergence: Very fast training of neural networks using large learning rates". In: *Artificial intelligence and machine learning for multi-domain operations applications*. Vol. 11006. SPIE. 2019, pp. 369–386.

[101] *Superposition*. URL: https://www.techtarget.com/whatis/definition/superposition#:~:text=Superposition%20is%20the%20ability%20of,the%20English%20physicist%2C%20Thomas%20Young..

[102] Christian Szegedy et al. "Intriguing properties of neural networks". In: *arXiv preprint arXiv:1312.6199* (2013).

[103] *Tensor Flow Models*. URL: https://github.com/tensorflow/models/blob/master/research/deeplab/model.py.

[104] *The Chi Square Test*. URL: https://www.westga.edu/academics/research/vrc/assets/docs/ChiSquareTest_LectureNotes.pdf.

[105] *The Rise And Fall Of The Perceptron*. URL: https://www.jailsogh.shop/ProductDetail.aspx?iid=149423063&pr=40.88.

[106] *Transformer*. URL: https://theaisummer.com/transformer/.

[107] *Transformers Encoder-Decoder*. URL: https://kikaben.com/transformers-encoder-decoder/.

[108] *Triplet loss*. URL: https://en.wikipedia.org/wiki/Triplet_loss.

[109] *Turing test*. URL: https://en.wikipedia.org/wiki/Turing_test.

[110] *Understand Network Predictions Using LIME*. URL: https://se.mathworks.com/help/deeplearning/ug/understand-network-predictions-using-lime.html.

[111] *Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names*. URL: https://gombru.github.io/2018/05/23/cross_entropy_loss/.

[112] *Understanding LSTM Networks*. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[113] *Understanding the Local Binary Pattern (LBP): A Powerful Method for Texture Analysis in Computer Vision*. URL: https://aihalapathirana.medium.com/understanding-the-local-binary-pattern-lbp-a-powerful-method-for-texture-analysis-in-computer-4fb55b3ed8b8.

[114] *Understanding Transformers, the Data Science Way*. URL: https://resources.experfy.com/bigdata-cloud/understanding-transformers-data-science-way/.

[115] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[116] Jinxian Wang et al. "DeepDDS: deep graph neural network with attention mechanism to predict synergistic drug combinations". In: *bioRxiv* (2021). DOI: 10.1101/2021.04.06.438723. eprint: https://www.biorxiv.org/content/early/2021/07/06/2021.04.06.438723.full.pdf. URL: https://www.biorxiv.org/content/early/2021/07/06/2021.04.06.438723.

[117] *What is a transformer model*. URL: https://blogs.nvidia.com/blog/what-is-a-transformer-model/.

[118] *What is Hyperparameter Tuning?* URL: https://aws.amazon.com/what-is/hyperparameter-tuning/#:~:text=Hyperparameters%20are%20external%20configuration%20variables,set%20before%20training%20a%20model..

[119] *What Is Inductive Bias in Machine Learning?* URL: https://www.baeldung.com/cs/ml-inductive-bias.

[120] *What is RCNN?* URL: https://blog.roboflow.com/what-is-r-cnn/.

[121] *What is the name for quantum gates that can be reversed?* URL: https://quantumcomputing.stackexchange.com/questions/7189/what-is-the-name-for-quantum-gates-that-can-be-reversed.

[122] *What Is the Purpose of a Feature Map in a Convolutional Neural Network*. URL: https://www.baeldung.com/cs/cnn-feature-map.

[123] *Whitening transformation*. URL: https://en.wikipedia.org/wiki/Whitening_transformation.

[124] James C. R. Whittington, Joseph Warren, and Timothy E. J. Behrens. *Relating transformers to models and neural representations of the hippocampal formation*. 2022. arXiv: 2112.04035 [cs.NE].

[125] *Word2vec*. URL: https://it.wikipedia.org/wiki/Word2vec.

[126] Xi-Zhu Wu and Zhi-Hua Zhou. "A unified view of multi-label performance measures". In: *international conference on machine learning*. PMLR. 2017, pp. 3780–3788.

[127] Zonghan Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (Jan. 2021), pp. 4–24. ISSN: 2162-2388. DOI: 10.1109/tnnls.2020.2978386. URL: http://dx.doi.org/10.1109/TNNLS.2020.2978386.

[128] Ryota Yoshihashi et al. "Classification-reconstruction learning for open-set recognition". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4016–4025.

[129] Aston Zhang et al. *Dive into Deep Learning*. 2023. arXiv: 2106.11342 [cs.LG]. URL: https://arxiv.org/abs/2106.11342.