

OCaml Scientific Computing

Functional Programming Meets Data Science

Wang, Liang
Zhao, Jianxin

December 9, 2020

To our families, and those who have been supporting us for these years!

DRAFT



Liang Wang is a Technical Lead at Nokia, a Senior Researcher at the University of Cambridge (affiliated with Queens' College), an Intel Software Innovator, a Fellow in the Higher Education Academy in UK. He is currently pursuing the MBA at Aalto University Executive Education.

Liang has broad research interest in operating systems, computer networks, edge computing, machine learning, high performance computing, optimisation theory, big data, and Fin-tech. He leads the R&D of a popular OCaml Scientific Computing System. Liang loves commercialisation and entrepreneurship, he designed the fastest approximate K-NN algorithm and co-founded a Computer Lab spin-off Kvasir Analytics. He also enjoys networking with different people to share ideas and innovate.



Jianxin Zhao is a PhD student in the University of Cambridge, supervised by Prof. Jon Crowcroft. His research interests include numerical computation, high performance computing, machine learning, and their application in the real world.

Table of content

Prologue	7
A Brief History	7
Reductionism vs. Holism	8
Key Features	9
Contact Me	9
Numerical Techniques	10
Introduction	11
What Is Scientific Computing	11
What is Functional Programming	11
Who Is This Book For	13
Structure of the Book	13
Installation	14
Option 1: Install from OPAM	14
Option 2: Pull from Docker Hub	14
Option 3: Pin the Dev-Repo	15
Option 4: Compile from Source	15
CBLAS/LAPACKE Dependency	16
Interacting with Owl	16
Using Toplevel	16
Using Notebook	17
Using Owl-Jupyter	21
Summary	21
Conventions	23
Pure vs. Impure	23
Ndarray vs. Scalar	25
Infix Operators	26
Operator Extension	29
Module Structures	32
Number and Precision	33
Polymorphic Functions	34
Module Shortcuts	34
Type Casting	35
Visualisation	37
Create Plots	37
Specification	38
Subplots	40
Multiple Lines	41
Legend	43
Drawing Patterns	44

Line Plot	46
Scatter Plot	46
Stairs Plot	48
Box Plot	48
Stem Plot	49
Area Plot	51
Histogram & CDF Plot	51
Log Plot	52
3D Plot	52
Advanced Statistical Plot	56
Summary	58
References	59
Mathematical Functions	60
Basic Functions	60
Basic Unary Math Functions	60
Basic Binary Functions	60
Exponential and Logarithmic Functions	61
Trigonometric Functions	62
Other Math Functions	64
Special Functions	64
Airy Functions	65
Bessel Functions	65
Elliptic Functions	68
Gamma Functions	69
Beta Functions	70
Struve Functions	71
Zeta Functions	71
Error Functions	72
Integral Functions	73
Factorials	75
Interpolation and Extrapolation	76
Integration	77
Utility Functions	79
Summary	80
Statistical Functions	81
Random Variables	81
Discrete Random Variables	81
Continuous Random Variables	82
Descriptive Statistics	84
Order Statistics	86
Special Distribution	86
Gamma Distribution	86
Beta Distribution	87
Chi-Square Distribution	87

Student-t Distribution	87
Cauchy Distribution	87
Multiple Variables	87
Sampling	87
Hypothesis Tests	87
Theory	87
Gaussian Distribution in Hypothesis Testing	87
Two-Sample Inferences	89
Goodness-of-fit Tests	89
Non-parametric Statistics	89
Covariance and Correlations	89
Analysis of Variance	90
Summary	90
N-Dimensional Arrays	91
Ndarray Types	91
Creation Functions	92
Properties Functions	93
Fold Functions	95
Scan Functions	96
Comparison Functions	96
Vectorised Functions	98
Iteration Functions	98
Manipulation Functions	100
Serialisation	101
Tensors	102
Summary	105
References	105
Slicing and Broadcasting	106
Slicing	106
Basic Slicing	106
Fancy Slicing	106
Conventions in Definition	107
Extended Operators	111
Advanced Usage	111
Broadcasting	114
What Is Broadcasting?	115
Shape Constraints	115
Supported Operations	117
Slicing in NumPy and Julia	118
Internal Mechanism	120
Summary	121
Linear Algebra	122
Vectors and Matrices	122

Creating Matrices	122
Accessing Elements	123
Iterate, Map, Fold, and Filter	124
Math Operations	126
Gaussian Elimination	127
LU Factorisation	128
Inverse and Transpose	131
Vector Spaces	132
Rank and Basis	132
Orthogonality	134
Solving $Ax = b$	135
Matrix Sensitivity	138
Determinants	139
Eigenvalues and Eigenvectors	140
Solving $Ax = \lambda x$	140
Complex Matrices	141
Similarity Transformation and Diagonalisation	142
Positive Definite Matrices	144
Positive Definiteness	144
Singular Value Decomposition	145
Internal: CBLAS and LAPACKE	149
Low-level Interface to CBLAS & LAPACKE	149
Sparse Matrices	150
Summary	151
References	151
 Ordinary Differential Equations	 152
What Is An ODE	152
Exact Solutions	152
Linear Systems	153
Solving An ODE Numerically	154
Owl-ODE	158
Example: Linear Oscillator System	158
Solver Structure	159
Symplectic Solvers	160
Features and Limits	162
Examples of using Owl-ODE	163
Explicit ODE	163
Two Body Problem	164
Lorenz Attractor	165
Damped Oscillation	170
Stiffness	171
Solve Non-Stiff ODEs	173
Solve Stiff ODEs	173
Summary	175
References	175

Signal Processing	176
Discrete Fourier Transform	176
Fast Fourier Transform	177
Examples	178
Applications of FFT	183
Find period of sunspots	183
Decipher the Tone	185
Image Processing	187
Filtering	188
Example: Smoothing	189
Gaussian Filter	190
Signal Convolution	191
FFT and Image Convolution	193
Summary	195
References	195
Algorithmic Differentiation	196
Chain Rule	196
Differentiation Methods	197
How Algorithmic Differentiation Works	200
Forward Mode	200
Reverse Mode	202
Forward or Reverse?	205
A Strawman AD Engine	205
Simple Forward Implementation	206
Simple Reverse Implementation	208
Unified Implementations	211
Forward and Reverse Propagation API	216
Expressing Computation	217
Example: Forward Mode	218
Example: Reverse Mode	218
High-Level APIs	219
Derivative and Gradient	219
Jacobian	221
Hessian and Laplacian	223
Other APIs	224
Internal of Algorithmic Differentiation	225
Go Beyond Simple Implementation	225
Extend AD module	229
Lazy Evaluation	230
Summary	231
References	232
Optimisation	233
Introduction	233
Root Finding	234

Univariate Function Optimisation	236
Use Derivatives	236
Golden Section Search	238
Multivariate Function Optimisation	238
Nelder-Mead Simplex Method	238
Gradient Descent Methods	239
Conjugate Gradient Method	243
Newton and Quasi-Newton Methods	244
Global Optimisation and Constrained Optimisation	245
Summary	247
References	247
Regression	248
Linear Regression	248
Problem: Where to locate a new McDonald's restaurant?	248
Cost Function	250
Solving Problem with Gradient Descent	251
Multiple Regression	255
Feature Normalisation	256
Analytical Solution	258
Non-linear regressions	259
Regularisation	261
Ols, Ridge, Lasso, and Elastic_net	264
Logistic Regression	264
Sigmoid Function	264
Cost Function	265
Example	266
Multi-class classification	269
Support Vector Machine	270
Kernel and Non-linear Boundary	271
Example	273
Model error and selection	273
Error Metrics	273
Model Selection	275
Summary	276
References	276
Deep Neural Networks	277
Perceptron	277
Yet Another Regression	278
Model Representation	278
Forward Propagation	279
Back propagation	280
Feed Forward Network	281
Layers	281
Activation Functions	282

Initialisation	283
Training	284
Test	285
Neural Network Module	286
Module Structure	286
Neurons	287
Neural Graph	289
Training Parameters	290
Convolutional Neural Network	293
Recurrent Neural Network	294
Long Short Term Memory (LSTM)	296
Generative Adversarial Network	298
Summary	300
References	300
Natural Language Processing	301
Introduction	301
Text Corpus	302
Step-by-step Operation	302
Use the Corpus Module	304
Vector Space Models	306
Bag of Words (BOW)	307
Term Frequency–Inverse Document Frequency (TF-IDF)	308
Latent Dirichlet Allocation (LDA)	311
Models	311
Dirichlet Distribution	313
Gibbs Sampling	313
Topic Modelling Example	315
Latent Semantic Analysis (LSA)	316
Search Relevant Documents	318
Euclidean and Cosine Similarity	318
Linear Searching	319
Summary	320
References	320
Dataframe for Tabular Data	322
Basic Concepts	322
Create Frames	322
Manipulate Frames	323
Query Frames	325
Iterate, Map, and Filter	326
Read/Write CSV Files	328
Infer Type and Separator	330
Summary	331
Symbolic Representation	332

Introduction	332
Design	332
Core abstraction	333
Engines	338
ONNX Engine	339
Example 3: Neural network	342
LaTeX Engine	343
Owl Engine	344
Summary	346
Probabilistic Programming	347
Generative Model vs Discriminative Model	347
Bayesian Networks	347
Sampling Techniques	347
Inference	347
System Architecture	348
Architecture Overview	349
Introduction	349
Architecture Overview	350
Core Implementation	352
N-dimensional Array	352
Interfaced Libraries	353
Advanced Functionality	353
Computation Graph	354
Algorithmic Differentiation	354
Regression	354
Neural Network	355
Parallel Computing	355
Actor Engine	355
GPU Computing	355
OpenMP	356
Community-Driven R&D	356
Summary	358
Core Optimisation	359
Background	359
Numerical Libraries	359
Optimisation of Numerical Computation	360
Interfacing to C Code	361
Ndarray Operations	361
From OCaml to C	362
Optimisation Techniques	367
Map Operations	368

Convolution Operations	371
Reduction Operations	374
Repeat Operations	376
Summary	378
References	378
Automatic Empirical Tuning	379
What is Parameter Tuning	379
Why Parameter Tuning in Owl	379
How to Tune OpenMP Parameters	380
Make a Difference	382
Summary	383
Computation Graph	384
Introduction	384
What is a Computation Graph?	384
From Dynamic to Static	384
Significance in Computing	386
Examples	386
Example 01: Basic CGraph	387
Example 02: CGraph with AD	388
Example 03: CGraph with DNN	390
Design Rationale	391
Optimisation of CGraph	396
Optimising memory with pebbles	397
Allocation Algorithm	398
As Intermediate Representations	399
Summary	400
Scripting and Zoo System	401
Introduction	401
Share Script with Zoo	401
Typical Scenario	402
Create a Script	402
Share via Gist	403
Import in Another Script	403
Select a Specific Version	403
Command Line Tool	404
More Examples	405
System Design	405
Services	406
Type Checking	407
Backend	407
Domain Specific Language	407
Service Discovery	408
Use Case	408

Summary	409
References	410
Compiler Backends	411
Base Library	411
Backend: JavaScript	413
Use Native OCaml	413
Use Facebook Reason	415
Backend: MirageOS	416
MirageOS and Unikernel	416
Example: Gradient Descent	416
Example: Neural Network	418
Evaluation	419
Summary	421
Distributed Computing	423
Actor System	423
Design	423
Actor Engines	423
Map-Reduce Engine	424
Parameter Server Engine	425
Peer-to-Peer Engine	426
Classic Synchronise Parallel	428
Bulk Synchronous Parallel	429
Asynchronous Parallel	429
Stale Synchronous Parallel	430
Probabilistic Synchronise Parallel	430
Basic idea: sampling	431
Compatibility	432
Barrier Trade-off Dimensions	432
Convergence	433
A Distributed Training Example	434
Step Progress	435
Accuracy	437
Summary	439
References	439
Testing Framework	440
Unit Test	440
Example	440
What Could Go Wrong	443
Corner Cases	443
Test Coverage	445
Use Functor	445
Summary	446

Constants and Metric System	447
What Is a Metric System	447
Four Metric Systems	447
SI Prefix	447
Example: Physics and Math constants	448
International System of Units	450
Time	450
Length	451
Area	451
Volume	451
Speed	452
Mass	452
Force	452
Energy	453
Power	453
Pressure	453
Viscosity	453
Luminance	454
Radioactivity	454
Internal Utility Modules	455
Dataset Module	455
MNIST	455
CIFAR-10	456
Graph Module	457
Stack and Heap Modules	460
Count-Min Sketch	460
Summary	464
Case Studies	466
Case - Image Recognition	467
Background	467
LeNet	467
AlexNet	468
VGG	469
ResNet	469
SqueezeNet	470
Capsule Network	471
Building InceptionV3 Network	471
InceptionV1 and InceptionV2	472
Factorisation	473
Grid Size Reduction	476
InceptionV3 Architecture	477
Preparing Weights	477

Processing Image	479
Running Inference	481
Applications	484
Summary	484
References	484
Case - Instance Segmentation	486
Introduction	486
Mask R-CNN Network	488
Building Mask R-CNN	489
Feature Extractor	490
Proposal Generation	491
Classification	492
Run the Code	493
Summary	493
References	494
Case - Neural Style Transfer	495
Content and Style	495
Content Reconstruction	496
Style Recreation	499
Combining Content and Style	501
Running NST	502
Extending NST	502
Fast Style Transfer	504
Building FST Network	504
Running FST	507
Summary	509
References	509
Case - Recommender System	510
Introduction	510
Architecture	511
Build Topic Models	514
Index Text Corpus	515
Random Projection	515
Optimising Vector Storage	517
Optimise Data Structure	518
Optimise Index Algorithm	520
Search Articles	521
Code Implementation	522
Make It Live	527
Summary	528
References	528
Case - Applications in Finance	529

Introduction	529
Bond Pricing	529
Black-Scholes Model	529
Mathematical Model	529
Option Pricing	529
Portfolio Optimisation	529
Mathematical Model	529
Efficient Frontier	529
Maximise Sharpe Ratio	529
Appendix	530
Acknowledgement	531
Theses	532
Probabilistic Synchronous Parallel	532
Supporting Browser-based Machine Learning	532
Adaptable Asynchrony in Distributed Learning	533
Applications of Linear Types	533
Composing Data Analytical Services	533
Computer Vision in OCaml	534
Automatic Parameter Tuning for OpenMP	534
Run Your Owl Computation on TensorFlow	535
Ordinary Differential Equation Solver	535
Resources	537
Epilogue	538

List of Figures

1	Plot example using Owl Notebook	20
2	Plot example using Owl-Jupyter	21
3	Basic function plot	38
4	Plot specification	39
5	Surf plot	40
6	Subplots	41
7	Plot multiple lines	42
8	Mix line plot and histogram	43
9	Plot with legends	44
10	Draw lines	45
11	Fill patterns	46
12	Line plot with customised marker	47
13	Scatter plot	48
14	Stairs plot	49
15	Box plot	49
16	Steam plot	50
17	Stem plot with autocorrelation	50
18	Area plot	51
19	Histogram plot and CDF	52
20	Change plot scale on x- and y-axis to log	53
21	3D plot	54
22	Customised 3D Plot, example 1	54
23	Customised 3D Plot, example 2	55
24	Headmap and contour plot	56
25	Advanced statistical plots with qqplot	57
26	Advanced statistical plots with probplot	58
27	Relationship between different trigonometric functions	63
28	Relationship between different hyperbolic trigonometric functions	64
29	Examples of the two solutions of an Airy equation	66
30	Examples of Bessel function of the first kind, with different order	67
31	Examples of Gamma function along part of the real axis	70
32	Examples of Struve function for different orders.	72
33	Plot of the Error function.	73
34	Plot of the Dawson and Fresnel integral function.	74
35	Plot of interpolation and corresponding Gamma function.	78
36	Probability density functions of two data sets	84
37	Cumulated density functions of two data sets	85
38	Functional relation between x and the other two variables.	90
39	Illustrated Examples of Slicing	109
40	Illustrated Examples of Slicing (Cont.)	110
41	Illustrated example of shape extension in broadcasting	116
42	Illustrated example of shape extension in broadcasting (cont.)	117
43	Illustrated example of shape extension in broadcasting (cont.)	117

44	Comparing the accuracy of Euler method and Midpoint method in approximating solution to ODE	156
45	Visualise the solution of a simple linear system	159
46	Visualise the circle trajectory by solving linear system	161
47	The trajectory of lighter object orbiting the massive object in a simplified two-body problem	166
48	Three components and phase plane plots of Lorenz attractor	168
49	Change the initial states on three dimension by only 0.1%, and the value of Lorenz system changes visibly.	169
50	Step response of a damped harmonic oscillator	172
51	Solving Non-Stiff Van der Pol equations with Sundial CVode solver	174
52	Solving Stiff Van der Pol equations with ODEPACK LSODA solver.	174
53	Using FFT to separate two sine signals from their mixed signal .	181
54	Yearly sunspot data	184
55	Find sunspot cycle with FFT	185
56	Recording of an 11-digit number and its FFT decomposition	186
57	Recording of the first digit and its FFT decomposition	186
58	Noise Moonlanding image	187
59	De-noised Moonlanding image	189
60	Smoothed stock price of Google	190
61	Smoothed stock price of Google with Gaussian filtering	191
62	Smoothed stock price of Google using FFT method	193
63	Image convolution illustration	194
64	Graph expression of function	200
65	Example of forward accumulation with computational graph	202
66	Example of reverse accumulation with computational graph	204
67	Higher order derivatives	221
68	Architecture of the AD module	226
69	The hump function and its derivative function	237
70	Different movement of simplex in Nelder-Mead optimisation method	239
71	Reach the local minimum by iteratively moving downhill	240
72	Optimisation process of gradient descent on multivariate function	242
73	Compare conjugate gradient and gradient descent	243
74	Visualise data for regression problem	249
75	Find possible regression line for given data	250
76	Visualise the cost function in linear regression problem	252
77	Validate regression result with original dataset	253
78	An example of using linear regression to fit data	254
79	Compare gradient descent efficiency with and without data normalisation	258
80	Visualise part of the boston housing dataset	260
81	Polynomial regression based on Boston housing dataset	261
82	Polynomial regression with high order	262
83	Revised polynomial model by applying regularisation in regression	263
84	The logistic function curve	265
85	Visualise the logistic regression dataset	269

86	Simplifying the cost function of logistic regression	270
87	Margins in the Supported Vector Machines	271
88	Using the gaussian kernel to locate non-linear boundary in categorisation	272
89	Visualise the SVM dataset	274
90	Extend logistic regression to neural network with one hidden layer	278
91	Visualise part of MNIST dataset	279
92	Different activation functions in neural network	283
93	Prediction from the model	286
94	Neural network module structure	287
95	Unroll the recurrent neural network	295
96	Basic processing unit in classic recurrent neural network	295
97	Basic processing unit in LSTM	296
98	Basic processing unit in GRU	297
99	Plate notation for LDA with Dirichlet-distributed topic-word distributions	311
100	Two dimensional dirichlet distribution with different alpha parameters	314
101	Applying SVD and then truncating on document-word matrix to retrieve topic model	317
102	Euclidean distance and cosine similarity in a two dimensional space	318
103	Architecture of the symbolic system	332
104	UI of LaTeX engine	345
105	Owl system architecture	351
106	Fork-join model used by OpenMP	368
107	Sin operation performance	370
108	Basic implementation algorithm of convolution: im2col	371
109	Compare the execution time of Conv2D operation of Owl and Eigen	374
110	Sum reduction operation on laptop	375
111	Repeat operation speed	377
112	Repeat operation memory usage comparison	377
113	Compare performance of sin operations	380
114	Observe the cross-points of OpenMP and non-OpenMP operation	380
115	Evaluation of the performance improvement of AEOS	383
116	Computation graph of a simple function: $\sin(x^*y)$	384
117	Computation graph of a simple math function	389
118	Computation graph functor stack in Owl	392
119	Optimisation techniques in computation graph: constant folding	394
120	Optimisation techniques in computation graph: fusing operations	395
121	Optimisation techniques in computation graph: remove zero . . .	395
122	Modelling computation graph memory optimisation problem as a pebble game	398
123	Optimised memory allocation	400
124	Zoo System Architecture	406
125	Core functor stack in owl	411

126	Performance of map and fold operations on ndarray on laptop and RaspberryPi	420
127	Performance of gradient descent on function f	421
128	Barrier control methods used for synchronisation	429
129	Probabilistic Synchronous Parallel example	432
130	Extra trade-off exposed through PSP	433
131	Plot showing the bound on the average of the means and variances of the sampling distribution.	435
132	Progress distribution in steps	436
133	pBSP parameterised by different sample sizes, from 0 to 64.	436
134	MNIST training using Actor	438
135	All tests passes	443
136	Error in tests	444
137	Units of measurement in the SI metric system	448
138	Use Count-Min Sketch method for counting	461
139	Workflow of image classification	468
140	Residual block in the ResNet	470
141	Network Architecture of InceptionV3	472
142	Panda image that is used for image recognition task	483
143	Example: Street view	488
144	Example: Sheep	489
145	Example of applying neural style transfer on a street view picture	495
146	Example content image in neural style transfer	497
147	Contents reconstruction from different layers	498
148	Example style image in neural style transfer	499
149	Style reconstruction from different layers	501
150	Combining content and style reconstruction to perform NST	502
151	System overview of the image transformation network and its training.	504
152	Artistic Styles used in fast style transfer	507
153	Example input image: Willis tower of Chicago	508
154	Fast style transfer examples	508
155	Kvasir architecture with components numbered based on their order in the workflow	511
156	Rank-revealing reduces dimensionality to perform in-memory SVD	515
157	Projection on different random lines	516
158	Construct a binary search tree from the random projection	516
159	Aggregate clustering result from multipel RP-trees	517
160	Use a random seed to generate on the fly	518
161	The number of true nearest neighbours found for different number of trees	519
162	Illustration of parallelising the computation.	520

List of Tables

1	Alias of pure and impure binary math functions	24
2	Infix operators in ndarray and matrix modules	26
3	Operator extensions	29
4	Basic unary math functions	60
5	Binary math functions	61
6	Exponential and logarithmic math functions	61
7	Trigonometric math functions	62
8	Bessel functions	66
9	Elliptic functions	68
10	Gamma functions	69
11	Error functions	73
12	Special integral functions	75
13	Factorial functions	75
14	Permutation and combination functions	76
15	Examples of solutions to certain types of ODE	152
16	Solvers provided by owl-ode and their types.	160
17	FFT functions in Owl	178
18	DTMF keypad frequencies	186
19	A Short Table of Basic Derivatives	197
20	Computation process of forward differentiation	201
21	Forward pass in the reverse differentiation mode	203
22	Computation process of the backward pass in reverse differentiation	203
23	List of other APIs in the AD module of Owl	224
24	Sample of input data: single feature	249
25	Sample of input data: multiple features	255
26	Variable notations in the LDA model	312
27	Tuned results using AEOS on different platforms	382
28	Evaluation of the effect of CGraph memory optimisation using different DNN architectures	399
29	Inference Speed of Deep Neural Networks	421
30	Size of executables generated by backends	421
31	Physical constants	449
32	Math constants	450
33	Time units	450
34	Length units	451
35	Area units	451
36	Volume units	451
37	Speed units	452
38	Mass units	452
39	Force units	452
40	Energy units	453
41	Power units	453
42	Pressure units	453
43	Viscosity units	454

44	Luminance units	454
45	Radioactivity units	454
46	Two data sets are used in Kvasir evaluation	512

DRAFT

Prologue

Owl is a software system for scientific and engineering computing. The library is (mostly) developed in the OCaml language. As a very unique functional programming language, OCaml offers us superb runtime efficiency, flexible module system, static type checking, intelligent garbage collector, and powerful type inference. Owl undoubtedly inherits these great features directly from OCaml. With Owl, you can write succinct type-safe numerical applications in a beautiful and battle-tested functional language without sacrificing performance, significantly speed up the development life-cycle, and reduce the cost from prototype to production use.

A Brief History

Owl originated from a research project which studied the design of synchronous parallel machines for large-scale distributed computing in July 2016. I chose OCaml as the language for developing the system due to its expressiveness and superior runtime efficiency. Another obvious reason is I was working as a PostDoc in OCamlLabs.

Even though OCaml is a very well designed language, the libraries for numerical computing in OCaml ecosystem were very limited and the tooling was fragmented at that time. In order to test various analytical applications, I had to write so many numerical functions myself, from very low level algebra and random number generators to the high level stuff like algorithmic differentiation and deep neural networks. These code snippets started accumulating and eventually grew much bigger than the distributed engine itself. Therefore I decided to take these functions out and wrapped them up as a standalone library – Owl.

Owl’s architecture undertook at least a dozen of iterations in the beginning, and some of the architectural changes are quite drastic. I intentionally avoid looking into the architecture of SciPy, Julia, Matlab to minimise their influence on Owl’s architecture, I really do not want *yet another xyz* When the architecture became stabilised, I started implementing different numerical functions. That was a stressful but fulfilling year in 2017, I worked day and night and added over 6000 functions (over 150,000 LOC). After one-year intensive development, Owl was already capable of doing many complicated numerical tasks. e.g. see our Google Inception V3 demo for image classification. I even held a tutorial in Oxford to demonstrate *Data Science in OCaml*.

Despite the fact that OCaml is a niche language, Owl has been attracting more and more users. I really appreciate their patience with this young but ambitious software. The community has been always supportive and provided useful feedback in these years. I hope Owl can help people to study functional programming and solve real-world problems.

Reductionism vs. Holism

If you are from Python world and familiar with its ecosystem for numerical computing, you may see Owl as a mixture of NumPy, SciPy, Pandas, and many other libraries. You may be curious about why I have packed so much stuff together. As you learn more and more about OCaml, I am almost certain you will start wondering why Owl’s design seems against *minimalist*, a popular design principle adopted by many OCaml libraries.

First of all, I must point out that having many functionalities included in one system does not necessarily indicate a monolithic design. Owl’s functionalities are well defined in various modules, and each module is very self-contained and follows the minimalist design principle.

Second, I would like to argue that these functionalities should be co-designed and implemented together. This is exactly what we have learnt in the past two decades struggling to build a modern numerical system. The current co-design choice avoids a lot of redundant code and duplicated efforts, and makes optimisation a lot easier. NumPy, SciPy, Pandas, and other software spent so many years in order to well define the boundary of their functionality. In the end, NumPy becomes data representation (i.e. N-dimensional array), SciPy builds high-level analytical functions atop of such representation, Pandas evolves into a combination of table manipulation and analytical functions, and PyTorch bridges these functions between heterogeneous devices. However, there is still a significant amount of overlap if you look deep into the implementation code.

Back to the OCaml world, the co-design becomes even more important because of the language’s strict static typing. Especially if every small numerical library wraps its data representation into abstract types, then they will not play together nicely when you try to build a large and complex application. This further indicates that by having a huge number of small libraries in the ecosystem will not effectively improve a programmers’ productivity. Owl is supposed to address this issue with a consistent *holistic* design, with a strong focus on scientific computing.

Different choice of design principle also reveals the difference between system programming and numerical programming. System programming is built atop of a wide range of complicated and heterogeneous hardware, it abstracts out the real-world complexity by providing a (relatively) small set of APIs (recall how many system calls in the Unix operating system). On the other hand, numerical computing is built atop of a small amount of abstract number types (e.g. real and complex), then derives a rich set of advanced numerical operations for various fields (recall how many APIs in a numerical library). As a result, reductionism is preferred in system programming whereas holism is preferred in numerical one.

Key Features

Owl has implemented many advanced numerical functions atop of its solid implementation of n-dimensional arrays. Compared to other numerical libraries, Owl is very unique in many perspectives, e.g. algorithmic differentiation and distributed computing have been included as integral components in the core system to maximise developers' productivity. Owl is young but grows very fast, the current features include:

- N-dimensional array (both dense and sparse)
- Various number types: float32, float64, complex32, complex64, int16, int32, and etc.
- Linear algebra and full interface to CBLAS and LAPACKE
- Algorithmic differentiation (or automatic differentiation)
- Neural network module for deep learning applications
- Dynamic computational graph
- Parallel and Distributed computation engine
- Advanced math and statistics functions (e.g., hypothesis tests, MCMC, etc.)
- Zoo system for efficient scripting and code sharing
- JavaScript and unikernel backends.
- Integration with other frameworks such as TensorFlow and PyTorch.
- GPU and other accelerator frameworks.

The Owl system evolves very fast, and OCaml's numerical ecosystem is booming as well, therefore your feedback is important for me to adjust future direction and the focus. In case you find some important features are missing, you are welcome to submit an issue on the Issue Tracker.

Contact Me

If you want to discuss about the book, the code, or any other related topics, you can reach me in the following ways.

- Email Me
- Slack Channel
- Issue Tracker

Student Project: If you happen to be a student in the Computer Lab and want to do some challenging development and design, here are some Part II Projects.

If you are interested in more researchy topics, I offer Part III Projects and please have a look at :doc:Owl's Sub-Projects page and contact me directly via email.

I am looking forward to hearing from you!

Numerical Techniques

DRAFT

Introduction

This chapter briefly introduces the outline of the whole book, targeted audience, how to use the book, and then the installation of Owl. There are different ways to interact with Owl, including `utop`, notebook, and the Owl-Jupyter. Feel free to choose one as you are exploring the Owl world with us.

What Is Scientific Computing

Scientific Computing is a rapidly evolving multidisciplinary field which uses advanced computing capabilities to understand and solve complex problems. The algorithms used in scientific computing can be generally divided into two types: numerical analysis, and computer algebra (or symbolic computation). The former uses numerical approximation to solve mathematical problems, while the latter requires an exact close-form representation of computation and manipulates symbols that are not assigned specific values.

Both approaches are widely used in various applications fields, such as engineering, physics, biology, finance, etc. Even though these advanced applications are sophisticated, they are all built atop of basic numerical operations in a scientific library, most of which Owl has already provided. For example, you can write a deep neural network with Owl in a few lines of code:

```
open Owl
open Neural.S
open Neural.S.Graph
open Neural.S.Algodiff

let make_network input_shape =
  input input_shape
  |> lambda (fun x -> Maths.(x / F 256.))
  |> conv2d [|5;5;1;32|] [|1;1|] ~act_typ:Activation.Relu
  |> max_pool2d [|2;2|] [|2;2|]
  |> dropout 0.1
  |> fully_connected 1024 ~act_typ:Activation.Relu
  |> linear 10 ~act_typ:Activation.(Softmax 1)
  |> get_network
```

It actually consists of basic operations such as `add`, `div`, `convolution`, `dot`, etc. It's totally OK if you have no idea what this piece of code is doing. We'll cover that later in this book. The point is that how to dissect a complex application into basic building blocks in a numerical library, and that's what we are trying to convey throughout this book.

What is Functional Programming

Most existing numerical or scientific computing software are based on the imperative programming paradigm, which uses statements that change a program's

state. Imperative programs often work by being built from one or more procedures, or functions. This modular style is widely adopted. Later around 1980s the idea of object oriented programming is rapidly developed. It extends the modular programming style to include the idea of “object”. An object can contains both data and procedure codes. The imperative programming is not widely adopted in numerical computing for no reason. Almost all computers’ hardware implementation follows imperative design. Actually, FORTRAN, the first cross-platform programming language and an imperative language, is still heavily used for numerical and scientific computations in various fields after first being developed at the 1950s. There is a good chance that, even if you are using modern popular numerical libraries such as SciPy, Julia, or Matlab, they still rely on FORTRAN in the core part somewhere.

As a contrast, the *Functional Programming* seems to born to perform hight-level tasks. When John McCarthy designed LISP, the first functional programming language, he meant to use it in the artificial intelligence field. The S-expression it uses was meant to be an intermediate representation, but later proved to be powerful and expressive enough. In LISP you can see the clear distinction between functional and imperative programming. Whereas the later uses a sequence of statements to change the state of the program, the former one builds a program that constructs a tree of expressions by using and composing functions.

The fundamental difference between these two programming paradigms lies the underlying model of computation. The imperative one is based on the Alan Turing model. In their book *Alan Turing: His Work and Impact*, S. Barry Cooper and J. Van Leeuwen said that “computability via Turing machines gave rise to imperative programming”. On the other hand, functional programming evolves from the *lambda calculus*, a formal system of computation built from function application. Lambda Calculus was invented by Alonzo Church in the 1930s, and it was meant to be a formal mathematical logic systems, instead of programming language. Actually, it was not until the programming language was invented that the relationship between these two is revealed. Turing himself proved that the lambda calculus is Turing complete. (Fun fact: Turing is the student of Church.) We can say that the Lambda calculus is the basis of all functional programming languages.

Compared to imperative programming, functional programming features immutable data, first-class functions, and optimisations on tail-recursion. By using techniques such as higher oder functions, currying, map & reduce etc., functional programming can often achieves parallelisation of threads, lazy evaluation, and determinism of program execution. But asides from these benefits, we are now talking about numerical computation which requires good performance. The question is, do we want to use a functional programming language to do scientific computing? We hope that by presenting Owl, which built on the functional programming language OCaml, we can give you an satisfactory answer.

Who Is This Book For

We really hope this book can cover as broad an audience as possible. Both scientific computing and functional programming are big areas, therefore it is quite a challenge to write a book that satisfies everyone. If you are reading this book now, we assume you are already interested in analytical tasks and enthusiastic about gaining some hands-on experience with functional programming languages. We also assume you know how to program in OCaml and are familiar with the core concepts of functional programming.

We want this book to be relatively general so we have covered many topics in scientific computing. However, this means we cannot dive very deeply into each topic, and each of them per se is probably worth a book. When designing the chapters, we select those topics which are either classical (e.g. statistics, linear algebra) or popular and proven-to-be-effective in industry (e.g. deep neural network, probabilistic programming, and etc.). We strive to reach a good balance between breadth and depth. For each topic, we will try our best to list sufficient references to guide our readers to study further.

Unlike other data science books, this book can serve as a reference for other software architects building modern numerical software systems. A big part of this book is dedicated to explaining the underlying details of Owl. Not only will we give you a bird's-eye-view of the overall Owl system, but also teach you how to build the system up and optimise each component step-by-step. If you use Owl to build applications, this book can serve as a useful reference manual as well.

Structure of the Book

The book is divided into three parts, and each part focuses on different areas.

Part I first introduces the basics of the Owl system and important conventions to help you in studying how to program with Owl. It then explores various topics in scientific computing, from the classic mathematics, statistics, linear algebra, algorithmic differentiation, optimisation, regression to the popular deep neural network, natural language processing, probabilistic programming, and etc. The chapters are loosely organised based on their dependency, e.g. you need to know optimisation before studying regression and deep neural networks.

Part II is dedicated to presenting the architecture of the Owl system. We will dive into each core component and show how we build and optimise the software. By so doing, you will gain a thorough understanding on how a modern numerical system can be structured and developed, and what are the key components needed in such a complex system. Note that even though Owl is developed in OCaml, the knowledge you learnt in this part can be extrapolated to another language.

Part III is a collection of case studies. This part might be the most interesting one for data scientists and practitioners. We will demonstrate how you can build

a complete numerical application quickly from scratch using Owl. The cases include computer vision, recommender systems, financial technology, and etc.

The book does not enforce any strict order in reading, you can simply jump to the topic that interests you most. If you are completely new to the Owl system, we still strongly recommend you to start with the first two chapters of the book so that you know how to set up a working environment and start programming. All the code snippets included in this book can be compiled with the most recent master branch of Owl, our tooling guarantees the book material stay up-to-date with the software.

Installation

That being said, there is a long way to go from simple math calculation to those large use cases. Now let's start from the very first step: installing Owl. Owl requires OCaml version $\geq 4.10.0$. Please make sure you have a working OCaml environment before you start installing Owl. You can read the guide on how to Install OCaml.

Owl's installation is rather trivial. There are four possible ways as shown below, from the most straightforward one to the least one.

Option 1: Install from OPAM

Thanks to the folks in OCaml Labs, OPAM makes package management in OCaml much easier than before. You can simply type the following command lines to install.

```
| opam install owl
```

There is a known issue when installing Owl on ubuntu-based distribution. The reason is that the binary distribution of BLAS and LAPACK are outdated and failed to provide all the interfaces Owl requires. You will need to compile `openblas` manually, and use the appropriate environment variables to point at your newly compiled library. You can use 'owl's docker file as a reference for this issue.

This way of installation pulls in the most recent Owl released on OPAM. Owl does not have a fixed release schedule. We usually make a new release whenever there are enough changes accumulated or a significant feature implemented. If you want to try the newest development features, we recommend the other ways to install Owl, as below.

Option 2: Pull from Docker Hub

Owl's docker images are synchronised with the master branch. The image is always automatically built whenever there are new commits. You can check the building history on Docker Hub.

You only need to pull the image then start a container.

```
| docker pull owlbarn/owl  
| docker run -t -i owlbarn/owl
```

Besides the complete Owl system, the docker image also contains an enhanced OCaml toplevel - `utop`. You can start `utop` in the container and try out some examples. The source code of Owl is stored in `/root/owl` directory. You can modify the source code and rebuild the system directly in the started container. There are Owl docker images on various Linux distributions, this can be further specified using tags, e.g. `docker pull owlbarn/owl:alpine`.

Option 3: Pin the Dev-Repo

`opam pin` allows you to pin the local code to Owl's development repository on Github. The first command `opam depext` installs all the dependencies Owl needs.

```
| opam depext owl  
| opam pin add owl --dev-repo
```

Option 4: Compile from Source

Compiling directly from the source is an old-school but a recommended option. First, you need to clone the repository.

```
| git clone git@github.com:owlbarn/owl.git
```

Second, you need to figure out the missing dependencies and install them.

```
| dune external-lib-deps --missing @install @runtest
```

Last, this is perhaps the most classic step.

```
| make && make install
```

If your OPAM is older than v2 beta4, you need one extra step. This is due to a bug in OPAM which copies the compiled library into `/.opam/4.06.0/lib/stublibs` rather than `/.opam/4.06.0/lib/stublibs`. If you don't want to upgrade OPAM, then you need to manually move `dllowl_stubs.so` file from `stublib` to `stublib` folder, then everything should work. However, if you have the most recent OPAM installed, this will not be your concern.

CBLAS/LAPACKE Dependency

The most important dependency of Owl is the OpenBLAS library, which efficiently implement the BLAS and LAPACK linear algebra routines. Linking to the correct OpenBLAS is the key to achieve the best performance. Depending on the specific platform, you can use `yum`, `apt-get`, `brew` to install the binary format. For example on my Mac OSX, the installation looks like this:

```
| brew install homebrew/science/openblas
```

However, installing from OpenBLAS source code give us extra benefits. First, it implements the most recent interfaces comparing to the outdated binary distribution offered by the native package management tool. Second, it leads to way better performance because OpenBLAS tunes many parameters based on your system configuration and architecture to generate the most optimised binary code.

OpenBLAS already contains an implementation of LAPACKE, as long as you have a Fortran complier installed on your computer, the LAPACKE will be compiled and included in the installation automatically.

Interacting with Owl

There are several ways to interact with Owl system. The classic one is to write an OCaml application, compile the code, link to Owl system, then run it natively on a computer. You can also skip the compilation and linking step, and use Zoo system in Owl to run the code as a script.

However, the easiest way for a beginner to try out Owl is using REPL (Read–Eval–Print Loop), namely an interactive toplevel such as that of Python. The toplevel offers a convenient way to play with small code snippets. The code run in the toplevel is compiled into bytecode rather than native code. Bytecode often runs much slower than native code. However, this has very little impact on Owl’s performance because all its performance-critical functions are implemented in C language.

OCaml code can be compiled in either bytecode or native code. The bytecode is executed on OCaml virtual machine which is less performant then platform-optimised native code. Toplevel runs the user code in bytecode mode, but this has little impact on Owl’s performance because its core functions are implemented in C language. It is hard to notice any performance degradation if you run Owl in a script. In the following, we will introduce two options to set up an interactive environment for Owl.

Using Toplevel

OCaml language has bundled with a simple toplevel, but I recommend `utop` as a more advance replacement. Installing `utop` is straightforward using OPAM,

simply run the following command in the system shell.

```
| opam install utop
```

After installation, you can load Owl in *utop* with the following commands. *owl-top* is Owl's toplevel library which will automatically load several related libraries (including *owl-zoo*, *owl-base*, and *owl* core library) to set up a complete numerical environment.

```
| # #require "owl-top"
| # open Owl
```

If you do not want to type these commands every time you start *toplevel*, you can add them to *.ocamlinit* file. The toplevel reads *.ocamlinit* file to initialise the environment during the startup. This file is often stored in the home directory on your computer.

Using Notebook

Jupyter Notebook is a popular way to mix presentation with interactive code execution. It originates from Python world and is widely supported by various languages. One attractive feature of notebook is that it uses client/server architecture and runs in a browser.

If you want to know how to use a notebook and its technical details, please read Jupyter Documentation. Here let me show you how to set up a notebook to run Owl step by step.

Run the following commands in the shell will install all the dependency for you. This includes Jupyter Notebook and its OCaml language extension.

```
| pip install jupyter
| opam install jupyter
| jupyter kernelspec install --name ocaml-jupyter "$(opam config var share)/jupyter"
```

To start a Jupyter notebook, you can run this command. The command starts a local server running on <http://127.0.0.1:8888/>, then opens a tab in your browser as the client.

```
| jupyter notebook
```

If you wish to run a notebook server remotely, please refer to “Running a notebook server” for more information. To set up a server for multiple users, which is especially useful for educational purpose, please consult to JupyterHub system.

When everything is up and running, you can start a new notebook in the web interface. In the new notebook, you must run the following OCaml code in the first input field to load Owl environment.

```
# #use "topfind"
# #require "owl-top, jupyter.notebook"
```

At this point, a complete Owl environment is set up in the Jupyter Notebook, and you are free to go with any experiments you like. For example, you can simply copy & paste the whole `lazy_mnist.ml` to train a convolutional neural network in the notebook. But here, let us just use the following code.

```
# #use "topfind"
# #require "owl-top, jupyter.notebook"

# open Owl
# open Neural.S
# open Neural.S.Graph
# open Neural.S.Algodiff

# let make_network input_shape =
  input input_shape
  |> lambda (fun x -> Maths.(x / F 256.))
  |> conv2d [|5;5;1;32|] [|1;1|] ~act_typ:Activation.Relu
  |> max_pool2d [|2;2|] [|2;2|]
  |> dropout 0.1
  |> fully_connected 1024 ~act_typ:Activation.Relu
  |> linear 10 ~act_typ:Activation.(Softmax 1)
  |> get_network
val make_network : int array -> network = <fun>
```

The `make_network` function defines the structure of a convolution neural network. By passing the shape of input data, Owl automatically infers the shape of whole network, and prints out the summary of network structure nicely on the screen.

```
# make_network [|28;28;1|]
- : network =
18839

[ Node input_0 ]:
  Input : in/out:[*,28,28,1]
  prev:[] next:[lambda_1]

[ Node lambda_1 ]:
  Lambda : in:[*,28,28,1] out:[*,28,28,1]
  customised f : t -> t
  prev:[input_0] next:[conv2d_2]

[ Node conv2d_2 ]:
  Conv2D : tensor in:[*;28,28,1] out:[*,28,28,32]
```

```

init : tanh
params : 832
kernel : 5 x 5 x 1 x 32
b : 32
stride : [1; 1]
prev:[lambda_1] next:[activation_3]

[ Node activation_3 ]:
  Activation : relu in/out:[*,28,28,32]
  prev:[conv2d_2] next:[maxpool2d_4]

[ Node maxpool2d_4 ]:
  MaxPool2D : tensor in:[*,28,28,32] out:[*,14,14,32]
  padding : SAME
  kernel : [2; 2]
  stride : [2; 2]
  prev:[activation_3] next:[dropout_5]

[ Node dropout_5 ]:
  Dropout : in:[*,14,14,32] out:[*,14,14,32]
  rate : 0.1
  prev:[maxpool2d_4] next:[fullyconnected_6]

[ Node fullyconnected_6 ]:
  FullyConnected : tensor in:[*,14,14,32] matrix out:(*,1024)
  init : standard
  params : 6423552
  w : 6272 x 1024
  b : 1 x 1024
  prev:[dropout_5] next:[activation_7]

[ Node activation_7 ]:
  Activation : relu in/out:[*,1024]
  prev:[fullyconnected_6] next:[linear_8]

[ Node linear_8 ]:
  Linear : matrix in:(*,1024) out:(*,10)
  init : standard
  params : 10250
  w : 1024 x 10
  b : 1 x 10
  prev:[activation_7] next:[activation_9]

[ Node activation_9 ]:
  Activation : softmax 1 in/out:[*,10]
  prev:[linear_8] next:[]
```

The Second example demonstrates how to plot figures in notebook. Because Owl's Plot module does not support in-memory plotting, the figure needs to be written into a file first before passing to `Jupyter_notebook.display_file` to render.

```

# #use "topfind"
# #require "owl-top, owl-plplot, jupyter.notebook"
# open Owl
# open Owl_plplot
```

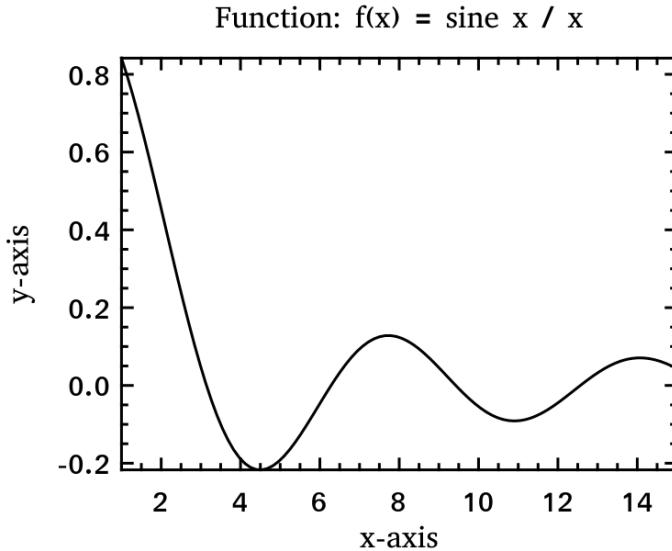


Figure 1: Plot example using Owl Notebook

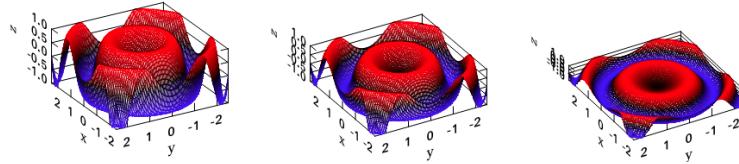
```
# let f x = Maths.sin x /. x in
let h = Plot.create "plot_00.png" in
Plot.set_title h "Function:  $f(x) = \sin x / x$ ";
Plot.set_xlabel h "x-axis";
Plot.set_ylabel h "y-axis";
Plot.set_font_size h 8.;
Plot.set_pen_size h 3.;
Plot.plot_fun ~h f 1. 15.;
Plot.output h
- : unit = ()
```

To load the image into browser, we need to call the `Jupyter_notebook.display_file` function. Then we can see the plot fig. 1 is correctly rendered in the notebook running in your browser. Plotting capability greatly enriches the content of an interactive presentation.

```
Jupyter_notebook.display_file ~base64:true "image/png" "plot_00.png"
```

Even though the extra call to `display_file` is not ideal, it is obvious that the tooling in OCaml ecosystem has been moving forward quickly. I believe we will soon have even better and more convenient tools for interactive data analytical applications.

```
[4]: let x, y = Mat.meshgrid (-2.5) 2.5 (-2.5) 2.5 100 100 in
let z = Mat.(sin ((x * x) + (y * y))) in
let h = Plot.create ~m:1 ~n:3 "plot_015.png" in
Plot.subplot h 0 0;
Plot.(mesh ~h ~spec:[ Altitude 50.; Azimuth 120.] x y z);
Plot.subplot h 0 1;
Plot.(mesh ~h ~spec:[ Altitude 65.; Azimuth 120.] x y z);
Plot.subplot h 0 2;
Plot.(mesh ~h ~spec:[ Altitude 80.; Azimuth 120.] x y z);
Plot.output h
```



```
[4]: - : unit = ()
```

Figure 2: Plot example using Owl-Jupyter

Using Owl-Jupyter

For the time being, if you want to save that extra line to display a image in Jupyter. There is a convenient module called `owl-jupyter`. Owl-jupyter module overloads the original `Plot.output` function so that a plotted figure can be directly shown on the page.

```
# #use "topfind"
# #require "owl-jupyter"
# open Owl_jupyter

# let f x = Maths.sin x /. x in
let h = Plot.create "plot_01.png" in
Plot.set_title h "Function: f(x) = sine x / x";
Plot.set_xlabel h "x-axis";
Plot.set_ylabel h "y-axis";
Plot.set_font_size h 8.;
Plot.set_pen_size h 3.;
Plot.plot_fun ~h f 1. 15.;
Plot.output h
- : unit = ()
```

From the example above, you can see Owl users' experience can be significantly improved by using the notebook.

Summary

In this chapter we give an brief introduction to the background of Owl, including scientific computing, functional programming, and target audience, and the layout of this book. Then we start introduces how Owl can be installed and used, as a first step to start this journey. You can feel free to browse any part of this book as you want.

At this point you have installed a working environment of Owl on your computer, you should feel really proud of yourself. To be honest, this can be the most challenging part for a new user, even though Owl team has spent tons of time in improving its compilation and installation. Now, let's roll out and start the exploration of more interesting topics.

DRAFT

Conventions

Every software system has its own rules and conventions which require the developers to comply with. Owl is not an exception, for example the rules on broadcasting operation and the conventions on slice definition. In this chapter, I will cover the function naming and various conventions in Owl.

Pure vs. Impure

Ndarray module contains many functions to manipulate and perform mathematical operations over multi-dimensional arrays. The **pure functions** (a.k.a immutable functions) refer to those which do not modify the passed in variables but always return a new one as result. In contrast, **impure functions** (a.k.a mutable functions) refer to those which modifies the passed-in variables in place.

The arguments between pure and impure functions will never end. Functional programming in general promotes the use of immutable data structures. Using impure functions makes it difficult to reason the correctness of the code, therefore you need to think carefully when you decide to use them. On the other hand, generating a fresh 1000×1000 matrix every time simply because you modify one element does not seem very practical either.

The introduction of impure functions into Owl is under many careful and practical considerations. One primary motivation of using in-place modification is to avoid expensive memory allocation and deallocation operations, this can significantly improve the runtime performance of a numerical application especially when large ndarrays and matrices involved.

Can we have the best parts of both world, i.e. writing functional code and being memory efficient at the same time? As you learn more about Owl, you will realise that this can be achieved by lazily evaluating a mathematical expression using computation graph. The programmer focusses on the functional code, Owl's computation graph module takes care of the “dangerous task” – allocating and managing the memory efficiently.

Many pure functions in Ndarray module have their corresponding impure version, the difference is that impure version has an extra underscore “_” at the end of function names. For example, the following functions are the pure functions in `Arr` module.

```
Arr.sin;;
Arr.cos;;
Arr.log;;
Arr.abs;;
Arr.add;;
Arr.mul;;
```

Their corresponding impure functions are as follows.

```
Arr.sin_;;
Arr.cos_;;
Arr.log_;;
Arr.abs_;;
Arr.add_;;
Arr.mul_;;
```

For unary operators such as `Arr.sin x`, the situation is rather straightforward, `x` will be modified in place. However, for binary operates such as `Arr.add_scalar_ x a` and `Arr.add_ x y`, the situation needs some clarifications. For `Arr.add_scalar_ x a`, `x` will be modified in place and stores the final result, this is trivial because `a` is a scalar.

For `Arr.add_ x y`, the question is where to store the final result when both inputs are `ndarray`. Let's look at the type of `Arr.add_` function.

```
val Arr.add_ : ?out:Arr.arr -> Arr.arr -> Arr.arr -> unit
```

As we can see from the function type, the output can be specified by an optional `out` parameter. If `out` is missing in the inputs, then Owl will try to use first operand (i.e. `x`) to store the final result. Because the binary operators in Owl support broadcasting operations by default, this further indicates when using impure functions every dimension of the first argument `x` must not be smaller than that of the second argument `y`. In other words, impure function only allows broadcasting smaller `y` onto `x` which is big enough to accommodate the result.

Most binary math functions in Owl are associated with a shorthand operator, such as `+`, `-`, `*`, and `/`. The impure versions also have their own operators. For example, corresponding to `Arr.(x + y)` which returns the result in a new `ndarray`, you can write `Arr.(x += y)` which adds up `x` and `y` and saves the result into `x`.

Table 1: Alias of pure and impure binary math functions

Function Name	Pure	Impure
add	<code>+</code>	<code>+=</code>
sub	<code>-</code>	<code>-=</code>
mul	<code>*</code>	<code>*=</code>
div	<code>/</code>	<code>/=</code>
add_scalar	<code>+\$</code>	<code>+\$=</code>
sub_scalar	<code>-\$</code>	<code>-\$=</code>
mul_scalar	<code>*\$</code>	<code>*\$=</code>
div_scalar	<code>/\$</code>	<code>/\$=</code>

Ndarray vs. Scalar

There are three types of ndarray operations: *map*, *scan*, and *reduce*. Many functions can be categorised as reduction operations, such as `Arr.sum`, `Arr.prod`, `Arr.min`, `Arr.mean`, `Arr.std`, and etc. All the reduction functions in Owl has a name parameter called `axis`. When you apply these reduction operations on a multi-dimensional array, there are two possible cases:

- if axis is explicitly specified, then Owl reduces along the specified axis;
- if axis is not specified, then Owl flattens the ndarray into a vector first and reduce all the elements along the axis 0.

If the passed in ndarray is already one-dimensional, then two cases are equivalent. In the following code snippet, `a` has shape `[|3;1;3|]` whereas `b` has shape `[|1|]` since it only contains one element.

```
let x = Arr.sequential [|3;3;3|];;
let a = Arr.sum ~axis:1 x;;
let b = Arr.sum x;;
```

If you want to add the result in `b` with another float number, you need to retrieve the value by calling `get` function.

```
let c = Arr.get b [|0|] in
c +. 10.;;
```

This does not look very convenient if we always need to extract a scalar value from the return of reduction operations. This is not a problem for the languages like Python and Julia since the return type is dynamically determined. However, for OCaml, this turns out to be challenging: we either use a unified type; or we implement another set of functions. In the end, we picked the latter in Owl's design. Every reduction operation has two versions:

- one allows you to reduce along the specified axis, or reduce all the elements, but always returns an ndarray;
- one only reduces all the elements and always returns a scalar value.

The difference between the two is that the functions returning a scalar ends up with an extra prime “`'`” character in their names. For example, for the first type of functions that return an ndarray, their function names look like these.

```
Arr.sum;;
Arr.min;;
Arr.prod;;
Arr.mean;;
Arr.std;;
```

For the second type of functions that return a scalar, their name looks like these.

```
Arr.sum';;
Arr.min';;
Arr.prod';;
Arr.mean';;
Arr.std';;
```

Technically, `Arr.sum'` is equivalent to the following code.

```
let sum' x =
  let y = Arr.sum x in
  Arr.get y [|0|]
```

Let's extend the previous code snippet, and test it in OCaml's toplevel. Then you will understand the difference immediately.

```
let x = Arr.sequential [|3;3;3|];;
let a = Arr.sum ~axis:1 x;;
let b = Arr.sum x;;
let c = Arr.sum' x;;
```

Rules and conventions often reveals the tradeoffs in a design. By clarifying the restrictions, we hope the programmers can choose the right functions to use in a specific scenario.

Infix Operators

The operators in Owl are implemented in the functors defined in the `owl_operator` module. These operators are categorised into `Basic`, `Extend`, `Matrix`, and `Ndarray` four module type signatures, because some operations are only meaningful for certain data structures. E.g., matrix multiplication `*@` is only defined in `Matrix` signature.

As long as a module implements all the functions defined in the module signature, you can use these functors to generate corresponding operators. In most cases, you do not need to work with these functors directly in Owl since I have done the generation part for you already.

The operators have been included in each `Ndarray` and `Matrix` module. The following table summarises the operators currently implemented. In the table, both `x` and `y` represent either a matrix or an ndarray while `a` represents a scalar value.

Table 2: Infix operators in `ndarray` and `matrix` modules

Operator	Example	Operation	Dense/Sparse	Ndarray/Matrix
<code>+</code>	<code>x + y</code>	element-wise add	both	both

Operator	Example	Operation	Dense	Sparse	Ndarray/Matrix
-	x - y	element-wise sub	both	both	
*	x * y	element-wise mul	both	both	
/	x / y	element-wise div	both	both	
+\$	x +\$ a	add scalar	both	both	
-\$	x -\$ a	sub scalar	both	both	
*\$	x *\$ a	mul scalar	both	both	
/\$	x /\$ a	div scalar	both	both	
\$+	a \$+ x	scalar add	both	both	
\$-	a \$- x	scalar sub	both	both	
\$*	a \$* x	scalar mul	both	both	
\$/	a \$/ x	scalar div	both	both	
=	x = y	comparison	both	both	
!=	x != y	comparison	both	both	
<>	x <> y	same as !=	both	both	
>	x > y	comparison	both	both	
<	x < y	comparison	both	both	
>=	x >= y	comparison	both	both	
<=	x <= y	comparison	both	both	
=.	x =. y	element-wise cmp	Dense	both	
!=.	x !=. y	element-wise cmp	Dense	both	
<>.	x <>. y	same as !=.	Dense	both	
>.	x >. y	element-wise cmp	Dense	both	
<.	x <. y	element-wise cmp	Dense	both	
>=.	x >=. y	element-wise cmp	Dense	both	
<=.	x <=. y	element-wise cmp	Dense	both	
=\$	x =\$ y	comp to scalar	Dense	both	
!=\$	x !=\$ y	comp to scalar	Dense	both	
<>\$	x <>\$ y	same as !=	Dense	both	
>\$	x >\$ y	compare to scalar	Dense	both	
<\$	x <\$ y	compare to scalar	Dense	both	
>=\$	x >=\$ y	compare to scalar	Dense	both	
<=\$	x <=\$ y	compare to scalar	Dense	both	
=.	x =. y	element-wise cmp	Dense	both	
!=.	x !=. y	element-wise cmp	Dense	both	
<>.	x <>. y	same as !=.	Dense	both	
>.	x >. y	element-wise cmp	Dense	both	
<.	x <. y	element-wise cmp	Dense	both	
>=.	x >=. y	element-wise cmp	Dense	both	
<=.	x <=. y	element-wise cmp	Dense	both	
=~	x =~ y	approx =	Dense	both	
=~\$	x =~\$ y	approx =\$	Dense	both	
=~.	x =~. y	approx =.	Dense	both	
=~.	x =~. y	approx =.	Dense	both	
%	x % y	mod divide	Dense	both	

Operator	Example	Operation	Dense	Sparse	Ndarray/Matrix
<code>%\$</code>	<code>x %\$ a</code>	mod divide scalar	Dense	both	
<code>**</code>	<code>x ** y</code>	power function	Dense	both	
<code>*@</code>	<code>x *@ y</code>	matrix multiply	both		Matrix
<code>/@</code>	<code>x /@ y</code>	solve linear system	both		Matrix
<code>**@</code>	<code>x **@ a</code>	matrix power	both		Matrix
<code>min2</code>	<code>min2 x y</code>	element-wise min	both	both	
<code>max2</code>	<code>max2 x y</code>	element-wise max	both	both	
<code>@=</code>	<code>x @= y</code>	concatenate vertically	Dense	both	
<code>@ </code>	<code>x @ y</code>	concatenate horizontally	Dense	both	

There is a list of things worth your attention as below.

- `*` is for element-wise multiplication; `*@` is for matrix multiplication. You can easily understand the reason if you read the source code of Algodiff module. Using `*` for element-wise multiplication (for matrices) leads to the consistent implementation of algorithmic differentiation.
- `+$` has its corresponding operator `$+` if we flip the order of parameters. However, be very careful about the operator precedence since OCaml determines the precedence based on the first character of an infix. `+$` preserves the precedence whereas `$+` does not. Therefore, I recommend using `$+` with great care. Please always use parentheses to explicitly specify the precedence. The same also applies to `$-`, `$*`, and `$/`.
- For comparison operators, e.g. `=` and `=.` compare all the elements in two variables `x` and `y`. The difference is that `=` returns a boolean value whereas `=.` returns a matrix or ndarray of the same shape and same type as `x` and `y`. In the returned result, the value in a given position is `1` if the values of the corresponding position in `x` and `y` satisfy the predicate, otherwise it is `0`.
- For the comparison operators ended with `$`, they are used to compare a matrix/ndarray to a scalar value.

Operators are easy to use, here are some examples.

```

let x = Mat.uniform 5 5;;
let y = Mat.uniform 5 5;;

Mat.(x + y);;
Mat.(x * y);;
Mat.(x ** y);;
Mat.(x *@ y);;
```

```
(* compare the returns of the following two *)
```

```
Mat.(x > y);;
Mat.(x >. y);;
```

Here is the return of the first example.

```
# Mat.(x > y)
- : bool = false
```

Here is the return of the second example.

```
# Mat.(x >. y)
- : (float, float64_elt) Owl_dense_matrix_generic.t =
  C0 C1 C2 C3 C4
R0 0 1 1 0 1
R1 0 1 1 0 1
R2 0 0 0 0 0
R3 1 0 0 0 1
R4 0 1 0 0 1
```

Now I am sure you can understand the difference between `>` and `>.`, and the same applies to other binary comparison operators.

Note that the extending indexing and slicing operators are not included in the table above, but you can find the detailed explanation in Indexing and Slicing Chapter.

Operator Extension

As you can see, the operators above do not allow interoperation on different number types (which may not be bad thing in my opinion actually). E.g., you cannot add a `float32` matrix to `float64` matrix unless you explicitly call the `cast` functions in `Generic` module.

Some people just like Pythonic way of working, `Owl.Ext` module is specifically designed for this purpose, to make prototyping faster and easier. Once you open the module, `Ext` immediately provides a set of operators to allow you to interoperate on different number types, as below. It automatically casts types for you if necessary.

Table 3: Operator extensions

Operator	Example	Operation
<code>+</code>	<code>x + y</code>	add
<code>-</code>	<code>x - y</code>	sub

Operator	Example	Operation
*	$x * y$	mul
/	x / y	div
=	$x = y$	comparison, return bool
!=	$x != y$	comparison, return bool
<>	$x <> y$	same as !=
>	$x > y$	comparison, return bool
<	$x < y$	comparison, return bool
>=	$x >= y$	comparison, return bool
<=	$x <= y$	comparison, return bool
=.	$x =. y$	element_wise comparison
!=.	$x !=. y$	element_wise comparison
<>.	$x <>. y$	same as !=.
>.	$x >. y$	element_wise comparison
<.	$x <. y$	element_wise comparison
>=.	$x >=. y$	element_wise comparison
<=.	$x <=. y$	element_wise comparison
%	$x \% y$	element_wise mod divide
**	$x ** y$	power function
*@	$x *@ y$	matrix multiply
min2	$\text{min2 } x \ y$	element-wise min
max2	$\text{max2 } x \ y$	element-wise max

You may have noticed, the operators ended with \$ (e.g., +\$, -\$...) disappeared from the table, which is simply because we can add/sub/mul/div a scalar with a matrix directly and we do not need these operators any more. Similar for comparison operators, because we can use the same > operator to compare a matrix to another matrix, or compare a matrix to a scalar, we do not need >\$ any longer. Allowing interoperation makes the operator table much shorter.

Currently, the operators in `Ext` only support interoperation on dense structures. Besides binary operators, `Ext` also implements most of the common math functions which can be applied to float numbers, complex numbers, matrices, and ndarray. These functions are:

```
im; re; conj, abs, abs2, neg, reci, signum, sqr, sqrt, cbrt, exp, exp2, expm1, log, log10, log2,
log1p, sin, cos, tan, atan, sinh, cosh, tanh, asinh, acosh, atanh, floor, ceil,
round, trunc, erf, erfc, logistic, relu, softplus, softsign, softmax, sigmoid, log_sum_exp,
l1norm, l2norm, l2norm_sqr, inv, trace, sum, prod, min, max, minmax, min_i, max_i, minmax_i.
```

Note that `Ext` contains its own `Ext.Dense` module which further contains the following submodules.

- `Ext.Dense.Ndarray.S`
- `Ext.Dense.Ndarray.D`
- `Ext.Dense.Ndarray.C`

- Ext.Dense.Ndarray.Z
- Ext.Dense.Matrix.S
- Ext.Dense.Matrix.D
- Ext.Dense.Matrix.C
- Ext.Dense.Matrix.Z

These modules are simply the wrappers of the original modules in `Owl.Dense` module so they provide most of the APIs already implemented. The extra thing these wrapper modules does is to pack and unpack the raw number types for you automatically. However, you can certainly use the raw data types then use the constructors defined in `owl_ext_types` to wrap them up by yourself. The constructors are defined as below.

```
type ext_typ =
  F of float
  C of Complex.t
  DMS of dms
  DMD of dmd
  DMC of dmc
  DMZ of dmz
  DAS of das
  DAD of dad
  DAC of dac
  DAZ of daz
  SMS of sms
  SMD of smd
  SMC of sms
  SMZ of smd
  SAS of sas
  SAD of sad
  SAC of sac
  SAZ of saz
```

There are also corresponding packing and unpacking functions you can use, please read `owl_ext_types.ml` <https://github.com/owlbarn/owl/blob/master/src/owl/ext/owl_ext_types.ml> for more details.

Let's see some examples to understand how convenient it is to use `Ext` module.

```
open Owl.Ext;;
let x = Dense.Matrix.S.uniform 5 5;;
let y = Dense.Matrix.C.uniform 5 5;;
let z = Dense.Matrix.D.uniform 5 5;;
x + F 5;;
x * C Complex.(re = 2.; im = 3.);;
x - y;;
x / y;;
x *@ y;;
```

```
(** ... *)

x > z;;
x >. z;;
(x >. z) * x;;
(x >. F 0.5) * x;;
(F 10. * x) + y *@ z;;;

(** ... *)

round (F 10. * (x *@ z));;
sin (F 5.) * cos (x + z);;
tanh (x * F 10. - z);;

(** ... *)
```

Before we finish this chapter, I want to point out the caveat. `Ext` tries to mimic the dynamic languages like Python by with unified types. This prevents OCaml compiler from doing type checking in compilation phase and introduces extra overhead in calling functions. Therefore, besides fast experimenting in toplevel, I do not recommend to use `Ext` module in the production code.

Module Structures

In Owl, `Dense` module contains the modules of dense data structures. For example, `Dense.Matrix` supports the operations of dense matrices. Similarly, `sparse` module contains the modules of sparse data structures.

```
Dense.Ndarray;; (* dense ndarray *)
Dense.Matrix;; (* dense matrix *)

Sparse.Ndarray;; (* sparse ndarray *)
Sparse.Matrix;; (* sparse ndarray *)
```

All these four modules consists of five submodules to handle different types of numbers.

- `s` module supports single precision float numbers `float32`;
- `d` module supports double precision float numbers `float64`;
- `c` module supports single precision complex numbers `complex32`;
- `z` module supports double precision complex numbers `complex64`;
- `Generic` module supports all aforementioned number types via GADT.

With `Dense.Ndarray`, you can create a dense n-dimensional array of no more than 16 dimensions. This constraint originates from the underlying `Bigarray.Genarray` module. In practice, this constraint makes sense since the space requirement will explode as the dimension increases. If you need anything higher than 16 dimensions, you need to use `Sparse.Ndarray` to create a sparse data structure.

Number and Precision

After deciding the suitable data structure (either dense or sparse), you can create a ndarray/matrix using creation function in the modules, using e.g., `empty`, `create`, `zeros`, `ones` ... The type of numbers (real or complex) and its precision (single or double) needs to be passed to the creations functions as the parameters.

```
# Dense.Ndarray.Generic.zeros Float64 [|5;5|]
- : (float, float64_elt) Dense.Ndarray.Generic.t =
  C0 C1 C2 C3 C4
R0 0 0 0 0 0
R1 0 0 0 0 0
R2 0 0 0 0 0
R3 0 0 0 0 0
R4 0 0 0 0 0
```

With `zeros` function, all the elements in the created data structure will be initialised to zeros.

Technically, `s`, `d`, `c`, and `z` are the wrappers of `Generic` module with explicit type information provided. Therefore you can save the type constructor which was passed into the `Generic` module if you use these submodules directly.

```
Dense.Ndarray.S.zeros [|5;5|];; (* single precision real ndarray *)
Dense.Ndarray.D.zeros [|5;5|];; (* double precision real ndarray *)
Dense.Ndarray.C.zeros [|5;5|];; (* single precision complex ndarray *)
Dense.Ndarray.Z.zeros [|5;5|];; (* double precision complex ndarray *)
```

The following examples are for dense matrices.

```
Dense.Matrix.S.zeros 5 5;; (* single precision real matrix *)
Dense.Matrix.D.zeros 5 5;; (* double precision real matrix *)
Dense.Matrix.C.zeros 5 5;; (* single precision complex matrix *)
Dense.Matrix.Z.zeros 5 5;; (* double precision complex matrix *)
```

The following examples are for sparse ndarrays.

```
Sparse.Ndarray.S.zeros [|5;5|];; (* single precision real ndarray *)
Sparse.Ndarray.D.zeros [|5;5|];; (* double precision real ndarray *)
Sparse.Ndarray.C.zeros [|5;5|];; (* single precision complex ndarray *)
Sparse.Ndarray.Z.zeros [|5;5|];; (* double precision complex ndarray *)
```

The following examples are for sparse matrices.

```
Sparse.Matrix.S.zeros 5 5;; (* single precision real matrix *)
Sparse.Matrix.D.zeros 5 5;; (* double precision real matrix *)
Sparse.Matrix.C.zeros 5 5;; (* single precision complex matrix *)
Sparse.Matrix.Z.zeros 5 5;; (* double precision complex matrix *)
```

In short, `Generic` module can do everything that submodules can, but for some functions (e.g. creation functions) you need to explicitly pass in the type information.

Polymorphic Functions

Polymorphism is achieved by pattern matching and GADT in `Generic` module. This means many functions in `Generic` module can handle aforementioned four different number types.

In the following, I use the `sum` function in `Dense.Matrix.Generic` module as an example. `sum` function returns the summation of all the elements in a matrix.

```
open Owl;;

let x = Dense.Matrix.S.eye 5 in
  Dense.Matrix.Generic.sum x;;
let x = Dense.Matrix.D.eye 5 in
  Dense.Matrix.Generic.sum x;;
let x = Dense.Matrix.C.eye 5 in
  Dense.Matrix.Generic.sum x;;
let x = Dense.Matrix.Z.eye 5 in
  Dense.Matrix.Generic.sum x;;
```

As we can see, no matter what kind of numbers are held in an identity matrix, we always pass it to `Dense.Matrix.Generic.sum` function. Similarly, we can do the same thing for other modules (`Dense.Ndarray`, `Sparse.Matrix`, and etc.) and other functions (`add`, `mul`, `neg`, and etc.).

Meanwhile, each submodule also contains the same set of functions, e.g., as below,

```
Dense.Matrix.S.(eye 5 |> sum);;
```

Module Shortcuts

In reality, we often work with double precision numbers, therefore Owl provides some shortcuts to the data structures of double precision float numbers:

- `Arr` is equivalent to double precision real `Dense.Ndarray.D`;
- `Mat` is equivalent to double precision real `Dense.Matrix.D`;

With these shortcut modules, you are no longer required to pass in type information. Here are some examples.

```
Arr.zeros [|5|];; (* same as Dense.Ndarray.D.zeros [|5|] *)
Mat.zeros 5 5;; (* same as Dense.Matrix.D.zeros 5 5 *)
```

More examples besides creation functions are as follows.

```
Mat.load "data.mat";; (* same as Dense.Matrix.D.load "data.mat" *)
Mat.of_array 5 5 x;; (* same as Dense.Matrix.D.of_array 5 5 x *)
Mat.linspace 0. 9. 10;; (* same as Dense.Matrix.D.linspace 0. 9. 10 *)
```

If you actually work more often with other number types like Complex, you can certainly make your own alias to corresponding S, D, C, and Z module if you like.

Type Casting

As I mentioned before, there are four basic number types. You can therefore cast one value from one type to another one by using the `cast_*` functions in `Generic` module.

- `Generic.cast_s2d`: cast from `float32` to `float64`;
- `Generic.cast_d2s`: cast from `float64` to `float32`;
- `Generic.cast_c2z`: cast from `complex32` to `complex64`;
- `Generic.cast_z2c`: cast from `complex64` to `complex32`;
- `Generic.cast_s2c`: cast from `float32` to `complex32`;
- `Generic.cast_d2z`: cast from `float64` to `complex64`;
- `Generic.cast_s2z`: cast from `float32` to `complex64`;
- `Generic.cast_d2c`: cast from `float64` to `complex32`;

In fact, all these function rely on the following `cast` function.

```
val cast : ('a, 'b) kind -> ('c, 'd) t -> ('a, 'b) t
```

The first parameter specifies the cast type. If the source type and the cast type are the same, `cast` function simply makes a copy of the passed in value.

```
# let x = Arr.uniform [|5;5|] (* created in float64 *)
val x : Arr.arr =
  C0 C1 C2 C3 C4
R0 0.648406 0.616945 0.828173 0.579604 0.212017
R1 0.960002 0.0563993 0.219521 0.855164 0.414024
R2 0.526179 0.532062 0.0640247 0.786426 0.956565
R3 0.810557 0.476031 0.516506 0.11439 0.964041
R4 0.981665 0.446936 0.276383 0.414747 0.174775
```

Now let's cast `x` from `float64` to `complex32`.

```
# let y = Dense.Ndarray.Generic.cast Complex32 x (* cast to complex32 *)
val y : (Complex.t, complex32_elt) Dense.Ndarray.Generic.t =
  C0 C1 C2 C3 C4
  R0 (0.648406, 0i) (0.616945, 0i) (0.828173, 0i) (0.579604, 0i) (0.212017, 0i)
  R1 (0.960002, 0i) (0.0563993, 0i) (0.219521, 0i) (0.855164, 0i) (0.414024, 0i)
  R2 (0.526179, 0i) (0.532062, 0i) (0.0640247, 0i) (0.786426, 0i) (0.956565, 0i)
  R3 (0.810557, 0i) (0.476031, 0i) (0.516506, 0i) (0.11439, 0i) (0.964041, 0i)
  R4 (0.981664, 0i) (0.446936, 0i) (0.276383, 0i) (0.414747, 0i) (0.174775, 0i)
```

To know more about the functions provided in each module, please read the corresponding interface file of `Generic` module. The `Generic` module contains the documentation.

- `Dense.Ndarray.Generic`
- `Dense.Matrix.Generic`
- `Sparse.Ndarray.Generic`
- `Sparse.Matrix.Generic`

Visualisation

Owl is an OCaml numerical library. Besides its extensive supports to matrix operations, it also has a flexible plotting module. Owl's `Plot` module is designed to help you in making fairly complicated plots with minimal coding efforts. It is built atop of [Pplot but hides its complexity from users.

The module is cross-platform since `Pplot` calls the underlying graphics device driver to plot. However, based on our experience, the Cairo Package provides the best quality and most accurate figure so we recommend installing Cairo. In fact, the examples in this tutorial are all generated by using the `Cairo PNG Driver`, but as you will see, you can have a full control over the figures and configure them in multiple ways.

In the sections below, we will demonstrate, with examples, how the plotting module is used in Owl to assist you with multiple tasks in data visualisation and analysis.

Create Plots

Let's start with the standard way of creating a plot using `Plot.create` function. Here is its type definition in `Owl_plot.mli`:

```
val create : ?m:int -> ?n:int -> string -> handle
```

Here is an example.

```
# let f x = Maths.sin x /. x in
let h = Plot.create "plot_001.png" in
  Plot.set_title h "Function: f(x) = sine x / x";
  Plot.set_xlabel h "x-axis";
  Plot.set_ylabel h "y-axis";
  Plot.set_font_size h 8.;
  Plot.set_pen_size h 3.;
  Plot.plot_fun ~h f 1. 15.;

Plot.output h
- : unit = ()
```

For any graph, we need to have a handle `h` by calling the `create` function. They type of images can be automatically inferred from the figure file name. In this case, we want to create a PNG image that plots the sine function. We can do that with the `plot_fun` function. Besides, we also set the x- and y-axis labels and figure title, together with the font and line size. You will find that these commands are similar to those in other plotting tools such as Matplotlib. Once you call `Plot.output`, the plot will be “sealed” and written into the final file. The generated figure is shown below.

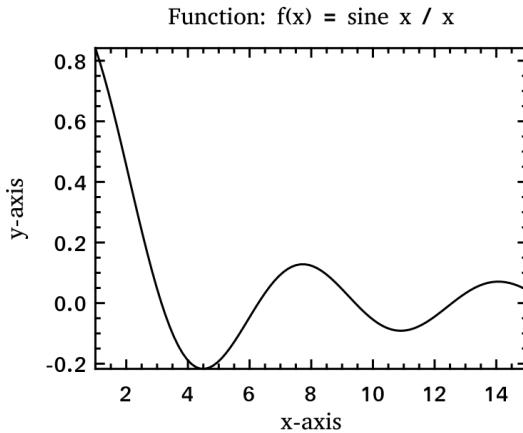


Figure 3: Basic function plot

If this looks too basic for you, then we have some fancy 3-D mesh graphs in the next part.

Specification

For most high-level plotting functions in Owl, there is an optional parameter called `spec`. The `spec` parameter take a list of specifications to let you finer control the appearance of the plot. Every function has a set of slightly different parameters. In case you pass in some parameters that a function cannot understand, they will be simply ignored. If you pass in the same parameter for multiple times, only the last one will take effects.

In the following, we will provide some examples to show how to use the `spec` parameter to finer tune Owl's plots. The first example shows how to configure the `mesh` plot using `ZLine`, `Contour`, and other `spec` parameters.

```
# let x, y = Mat.meshgrid (-2.5) 2.5 (-2.5) 2.5 50 50 in
let z = Mat.(sin ((x * x) + (y * y))) in
let h = Plot.create ~m:2 ~n:3 "plot_020.png" in

Plot.subplot h 0 0;
Plot.(mesh ~h ~spec:[ ZLine XY ] x y z);

Plot.subplot h 0 1;
Plot.(mesh ~h ~spec:[ ZLine X ] x y z);

Plot.subplot h 0 2;
Plot.(mesh ~h ~spec:[ ZLine Y ] x y z);
```

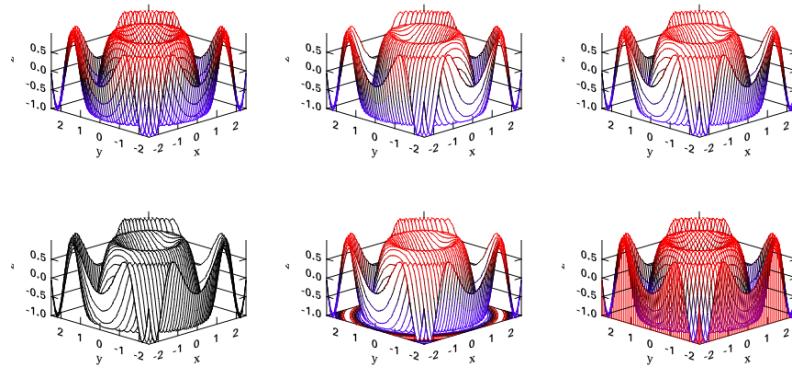


Figure 4: Plot specification

```

Plot.subplot h 1 0;
Plot.(mesh ~h ~spec:[ ZLine Y; NoMagColor ] x y z);

Plot.subplot h 1 1;
Plot.(mesh ~h ~spec:[ ZLine Y; Contour ] x y z);

Plot.subplot h 1 2;
Plot.(mesh ~h ~spec:[ ZLine XY; Curtain ] x y z);

Plot.output h
- : unit = ()

```

The second example shows how to tune the `surf` plotting function in drawing a 3D surface.

```

# let x, y = Mat.meshgrid (-1.) 1. (-1.) 1. 50 50 in
let z = Mat.(tanh ((x * x) + (y * y))) in
let h = Plot.create ~m:2 ~n:3 "plot_021.png" in

Plot.subplot h 0 0;
Plot.(surf ~h ~spec:[ ] x y z);

Plot.subplot h 0 1;
Plot.(surf ~h ~spec:[ Faceted ] x y z);

Plot.subplot h 0 2;
Plot.(surf ~h ~spec:[ NoMagColor ] x y z);

Plot.subplot h 1 0;
Plot.(surf ~h ~spec:[ Contour ] x y z);

Plot.subplot h 1 1;
Plot.(surf ~h ~spec:[ Curtain ] x y z);

```

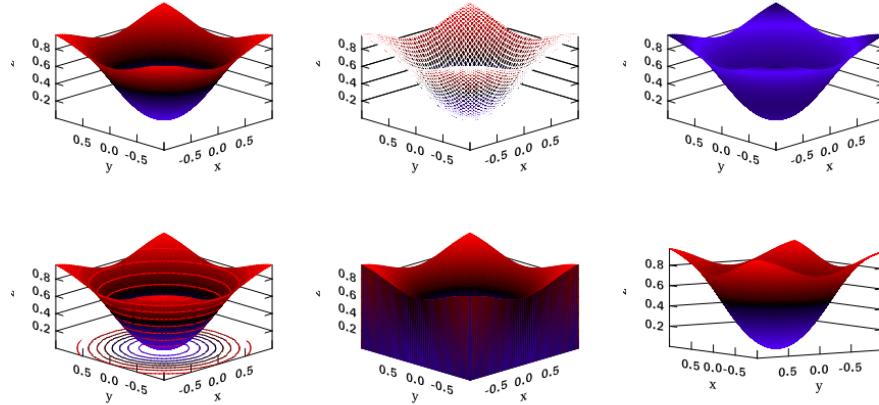


Figure 5: Surf plot

```

Plot.subplot h 1 2;
Plot.(surf ~h ~spec:[ Altitude 10.; Azimuth 125. ] x y z);

Plot.output h
- : unit = ()

```

Subplots

You might already have spotted another feature in the previous example: subplots. Indeed, you can change the number of rows and columns of the subplot layout by varying the number of the `m` and `n` parameters in the `create` function.

```

# let f p i = match i with
| 0 -> Stats.gaussian_rvs ~mu:0. ~sigma:0.5 +. p.(1)
| _ -> Stats.gaussian_rvs ~mu:0. ~sigma:0.1 *. p.(0)
in
let y = Stats.gibbs_sampling f [|0.1;0.1|] 5_000 |> Mat.of_arrays in
let h = Plot.create ~m:2 ~n:2 "plot_002.png" in
Plot.set_background_color h 255 255 255;

(* focus on the subplot at 0,0 *)
Plot.subplot h 0 0;
Plot.set_title h "Bivariate model";
Plot.scatter ~h (Mat.col y 0) (Mat.col y 1);

(* focus on the subplot at 0,1 *)
Plot.subplot h 0 1;
Plot.set_title h "Distribution of y";
Plot.set_xlabel h "y";
Plot.set_ylabel h "Frequency";
Plot.histogram ~h ~bin:50 (Mat.col y 1);

```

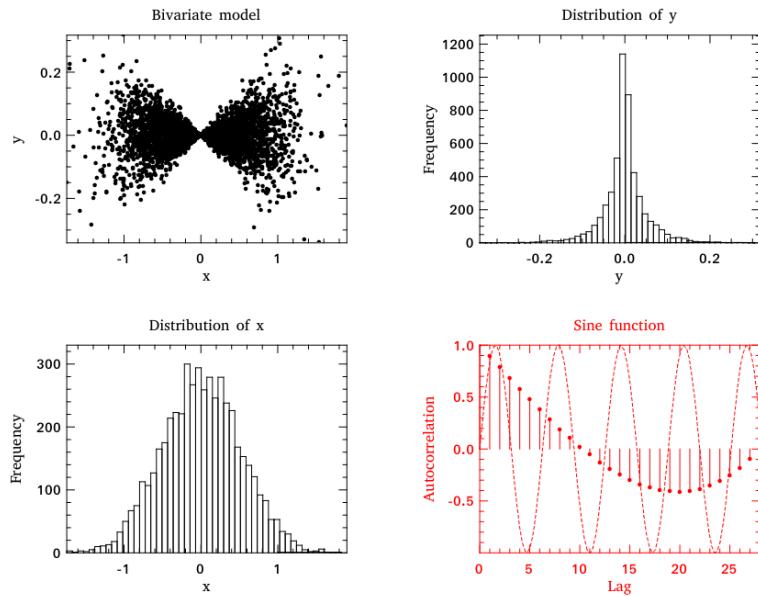


Figure 6: Subplots

```
(* focus on the subplot at 1,0 *)
Plot.subplot h 1 0;
Plot.set_title h "Distribution of x";
Plot.set_ylabel h "Frequency";
Plot.histogram ~h ~bin:50 (Mat.col y 0);

(* focus on the subplot at 1,1 *)
Plot.subplot h 1 1;
Plot.setForeground_color h 255 0 0;
Plot.set_title h "Sine function";
Plot.(plot_fun ~h ~spec:[ LineStyle 2 ] Maths.sin 0. 28.);
Plot.autocorr ~h (Mat.sequential 1 28);

(* output your final plot *)
Plot.output h
- : unit = ()
```

Multiple Lines

You can certainly plot multiple lines (or other types of plots) on the same page. Here is one example with both sine and cosine lines in one plot.

```
# let h = Plot.create "plot_024.png" in
```

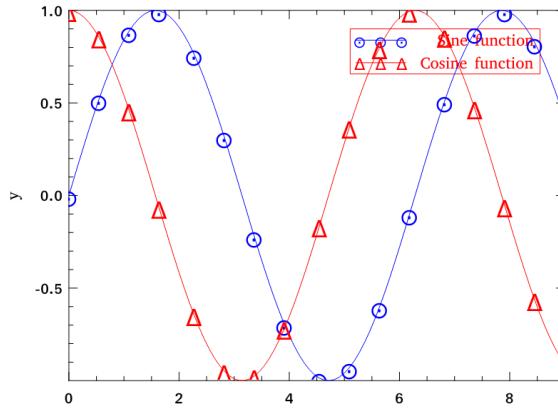


Figure 7: Plot multiple lines

```

Plot.(plot_fun ~h ~spec:[ RGB (0,0,255); Marker "#[0x2299]"; MarkerSize 8. ]
      Maths.sin 0. 9.);
Plot.(plot_fun ~h ~spec:[ RGB (255,0,0); Marker "#[0x0394]"; MarkerSize 8. ]
      Maths.cos 0. 9.);
Plot.legend_on h [|"Sine function"; "Cosine function"|];
Plot.output h
- : unit = ()
  
```

Here is another example which has both histogram and line plot in one figure.

```

# (* generate data *)
let g x = (Stats.gaussian_pdf x ~mu:0. ~sigma:1.) *. 100. in
let y = Mat.gaussian ~mu:0. ~sigma:1. 1 1000 in

(* plot multiple data sets *)
let h = Plot.create "plot_025.png" in
Plot.set_background_color h 255 255 255;
Plot.histogram ~h ~spec:[ RGB (255,0,50) ] ~bin:100 y;
Plot.(plot_fun ~h ~spec:[ RGB (0,0,255); LineWidth 2. ] g (-4.) 4.);
Plot.legend_on h [|"data"; "model"|];

Plot.output h
- : unit = ()
  
```

So as long as you “hold” the plot without calling `Plot.output`, you can plot many data sets in one figure.

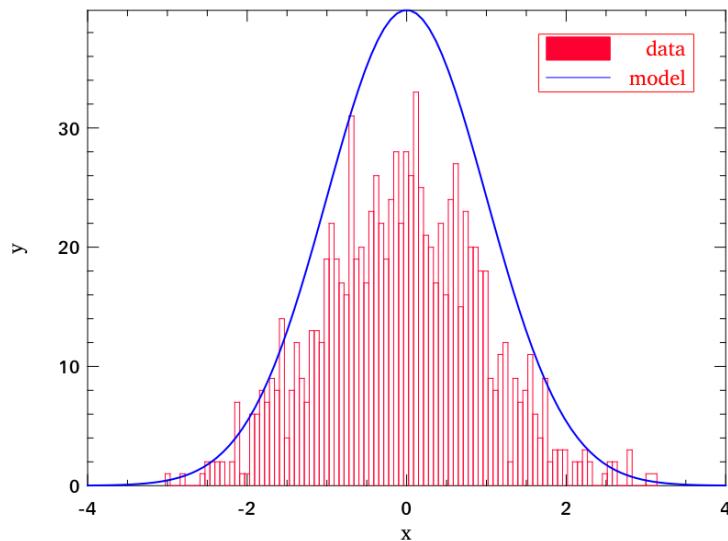


Figure 8: Mix line plot and histogram

Legend

Legend can be turned on and off by calling `Plot.legend_on` and `Plot.legend_off` respectively. When you call `Plot.legend_on`, you also need to provide an array of legend names and the position of legend. There are eight default positions in `Plot`:

```
type legend_position =
  North | South | West | East | NorthWest | NorthEast | SouthWest | SouthEast
```

Despite of its messy looking, the following example shows how to use legend in Owl's plot module.

```
# (* generate data *)
let x = Mat.(uniform 1 20 *$ 10.) in
let y = Mat.(uniform 1 20) in
let z = Mat.gaussian 1 20 in

(* plot multiple data sets *)
let h = Plot.create "plot_026.png" in
Plot.(plot_fun ~h ~spec:[ RGB (0,0,255); LineStyle 1; Marker "*" ] Maths.sin 1.
8.);
Plot.(plot_fun ~h ~spec:[ RGB (0,255,0); LineStyle 2; Marker "+" ] Maths.cos 1.
8.);
```

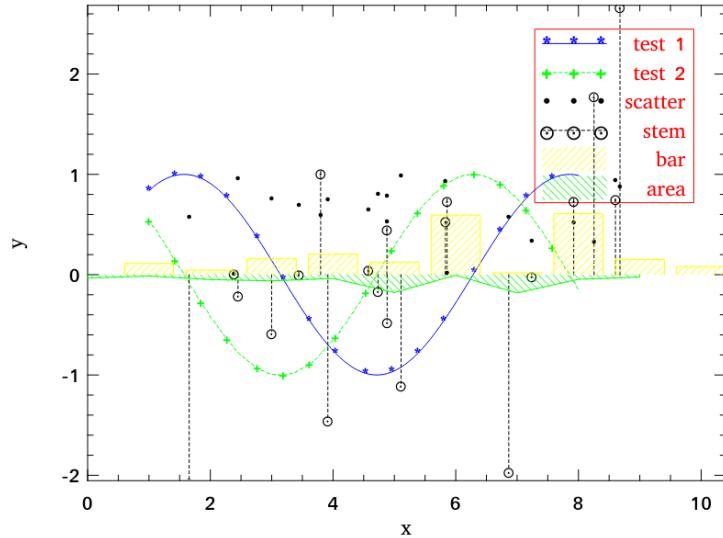


Figure 9: Plot with legends

```

Plot.scatter ~h x y;
Plot.stem ~h x z;

let u = Mat.(abs(gaussian 1 10 ** 0.3)) in
Plot.(bar ~h ~spec:[ RGB (255,255,0); FillPattern 3 ] u);

let v = Mat.(neg u ** 0.3) in
let u = Mat.sequential 1 10 in
Plot.(area ~h ~spec:[ RGB (0,255,0); FillPattern 4 ] u v);

(* set up legend *)
Plot.(legend_on h ~position:NorthEast [|"test 1"; "test 2"; "scatter"; "stem";
"bar"; "area"|]);
Plot.output h
- : unit = ()

```

Drawing Patterns

The plotting module supports multiple pattern of lines, as shown below:

```

# let h = Plot.create "plot_004.png" in
Plot.set_background_color h 255 255 255;
Plot.set_pen_size h 2.;
Plot.(draw_line ~h ~spec:[ LineStyle 1 ] 1. 1. 9. 1.);
Plot.(draw_line ~h ~spec:[ LineStyle 2 ] 1. 2. 9. 2.);

```

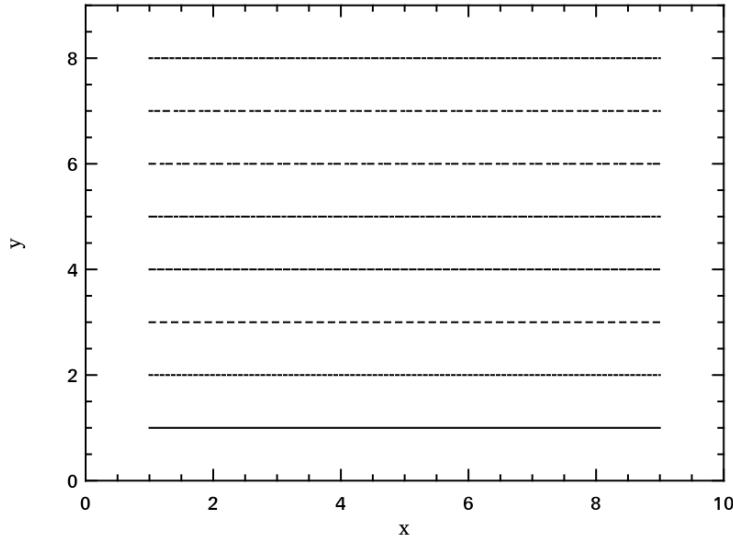


Figure 10: Draw lines

```

Plot.(draw_line ~h ~spec:[ LineStyle 3 ] 1. 3. 9. 3.);
Plot.(draw_line ~h ~spec:[ LineStyle 4 ] 1. 4. 9. 4.);
Plot.(draw_line ~h ~spec:[ LineStyle 5 ] 1. 5. 9. 5.);
Plot.(draw_line ~h ~spec:[ LineStyle 6 ] 1. 6. 9. 6.);
Plot.(draw_line ~h ~spec:[ LineStyle 7 ] 1. 7. 9. 7.);
Plot.(draw_line ~h ~spec:[ LineStyle 8 ] 1. 8. 9. 8.);
Plot.set_xrange h 0. 10.;
Plot.set_yrange h 0. 9.;
Plot.output h
- : unit = ()

```

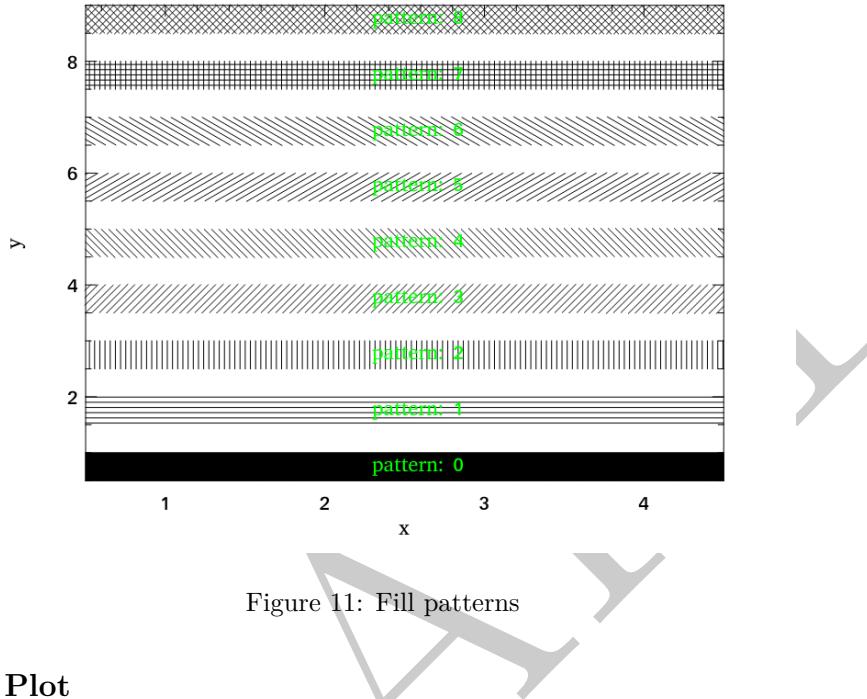
Similarly, we can also fill rectangles with different patterns, as shown in the example below.

```

# let h = Plot.create "plot_005.png" in
  Array.init 9 (fun i =>
    let x0, y0 = 0.5, float_of_int i +. 1.0 in
    let x1, y1 = 4.5, float_of_int i +. 0.5 in
    Plot.(draw_rect ~h ~spec:[ FillPattern i ] x0 y0 x1 y1);
    Plot.(text ~h ~spec:[ RGB (0,255,0) ] 2.3 (y0-.0.2) ("pattern: "
      (string_of_int i)))
  ) |> ignore;

Plot.output h
- : unit = ()

```



Line Plot

After getting to know these plotting elements, in the rest of chapter, we will demonstrate some different types of plots that are supported. Line plot is the most basic function. You can specify the colour, marker, and line style in the function.

```
# let x = Mat.linspace 0. 2. 100 in
let y0 = Mat.sigmoid x in
let y1 = Mat.map Maths.sin x in
let h = Plot.create "plot_022.png" in
Plot.(plot ~h ~spec:[ RGB (255,0,0); LineStyle 1; Marker "#[0x2299]"; MarkerSize
8. ] x y0);
Plot.(plot ~h ~spec:[ RGB (0,255,0); LineStyle 2; Marker "#[0x0394]"; MarkerSize
8. ] x y1);
Plot.(legend_on h ~position:SouthEast [|"sigmoid"; "sine"|]);
Plot.output h
- : unit = ()
```

Scatter Plot

Next is the scatter plot. Similar to line plot, you can specify the marker type and marker size. The example below actually shows the various patterns of markers. They are referenced by different ids.

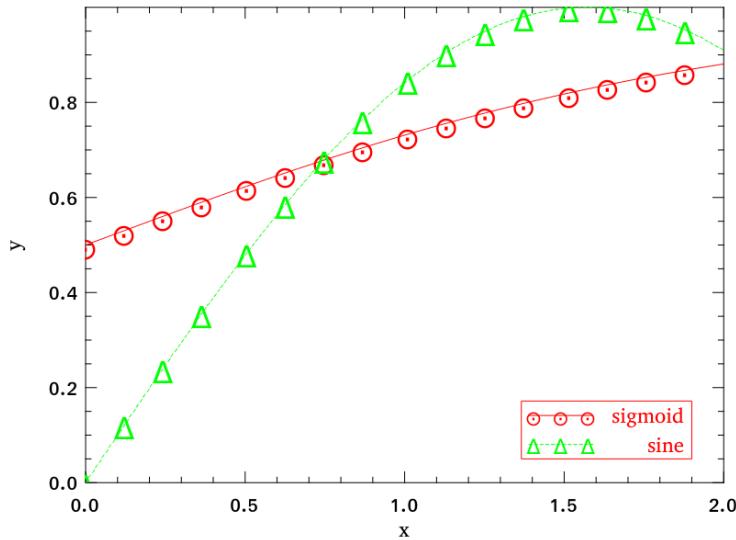


Figure 12: Line plot with customised marker

```

# let x = Mat.uniform 1 30 in
let y = Mat.uniform 1 30 in
let h = Plot.create ~m:3 ~n:3 "plot_006.png" in
Plot.set_background_color h 255 255 255;
Plot.subplot h 0 0;
Plot.(scatter ~h ~spec:[ Marker "#[0x2295]"; MarkerSize 5. ] x y);
Plot.subplot h 0 1;
Plot.(scatter ~h ~spec:[ Marker "#[0x229a]"; MarkerSize 5. ] x y);
Plot.subplot h 0 2;
Plot.(scatter ~h ~spec:[ Marker "#[0x2206]"; MarkerSize 5. ] x y);
Plot.subplot h 1 0;
Plot.(scatter ~h ~spec:[ Marker "#[0x229e]"; MarkerSize 5. ] x y);
Plot.subplot h 1 1;
Plot.(scatter ~h ~spec:[ Marker "#[0x2217]"; MarkerSize 5. ] x y);
Plot.subplot h 1 2;
Plot.(scatter ~h ~spec:[ Marker "#[0x2296]"; MarkerSize 5. ] x y);
Plot.subplot h 2 0;
Plot.(scatter ~h ~spec:[ Marker "#[0x2666]"; MarkerSize 5. ] x y);
Plot.subplot h 2 1;
Plot.(scatter ~h ~spec:[ Marker "#[0x22a1]"; MarkerSize 5. ] x y);
Plot.subplot h 2 2;
Plot.(scatter ~h ~spec:[ Marker "#[0x22b9]"; MarkerSize 5. ] x y);
Plot.output h
- : unit = ()

```

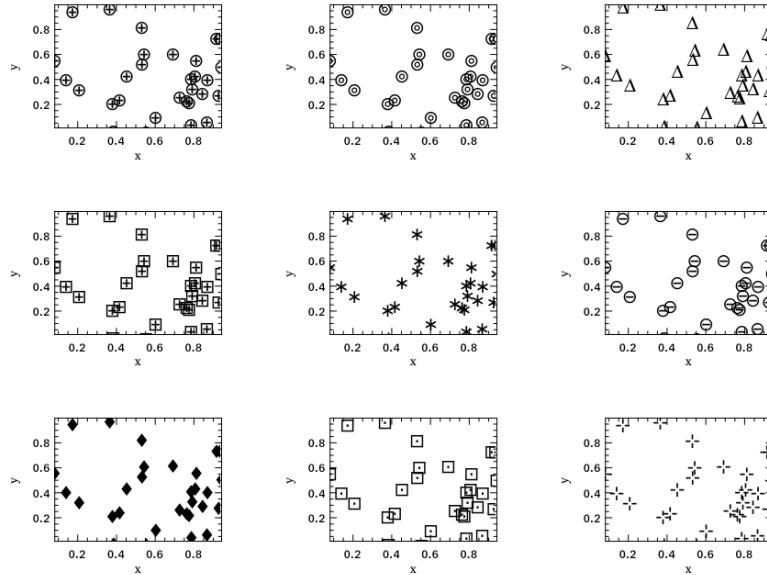


Figure 13: Scatter plot

Stairs Plot

The step plot is also called “stairstep plot”, since it draws the elements in a given ndarray in a stairstep-like curve.

```
# let x = Mat.linspace 0. 6.5 20 in
let y = Mat.map Maths.sin x in
let h = Plot.create ~m:1 ~n:2 "plot_007.png" in
Plot.set_background_color h 255 255 255;
Plot.subplot h 0 0;
Plot.plot_fun ~h Maths.sin 0. 6.5;
Plot.(stairs ~h ~spec:[ RGB (0,128,255) ] x y);
Plot.subplot h 0 1;
Plot.(plot ~h ~spec:[ RGB (0,0,0) ] x y);
Plot.(stairs ~h ~spec:[ RGB (0,128,255) ] x y);
Plot.output h
- : unit = ()
```

Box Plot

A box plot graphically shows groups of numerical data through their quartiles. It is often used in descriptive statistics.

```
# let y1 = Mat.uniform 1 10 in
```

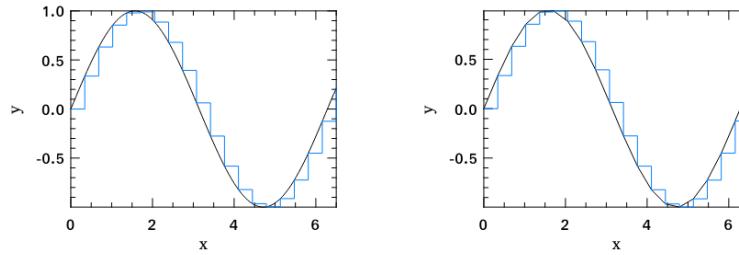


Figure 14: Stairs plot

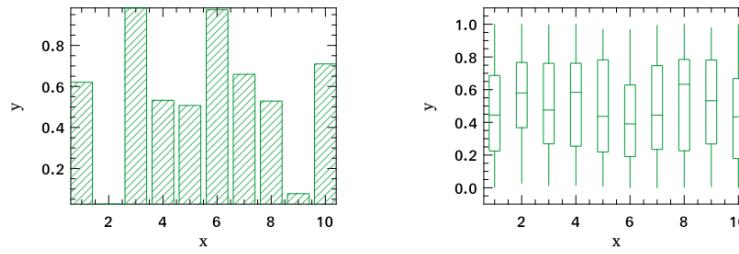


Figure 15: Box plot

```

let y2 = Mat.uniform 10 100 in
let h = Plot.create ~m:1 ~n:2 "plot_008.png" in
Plot.subplot h 0 0;
Plot.(bar ~h ~spec:[ RGB (0,153,51); FillPattern 3 ] y1);
Plot.subplot h 0 1;
Plot.(boxplot ~h ~spec:[ RGB (0,153,51) ] y2);
Plot.output h
- : unit = ()
  
```

Stem Plot

Stem plot is simple, as the following code shows.

```

# let x = Mat.linspace 0.5 2.5 25 in
let y = Mat.map (Stats.exponential_pdf ~lambda:0.1) x in
let h = Plot.create ~m:1 ~n:2 "plot_009.png" in
Plot.set_background_color h 255 255 255;
Plot.subplot h 0 0;
Plot.set_foreground_color h 0 0 0;
Plot.stem ~h x y;
Plot.subplot h 0 1;
Plot.(stem ~h ~spec:[ Marker "#[0x2295]"; MarkerSize 5.; LineStyle 1 ] x y);
  
```

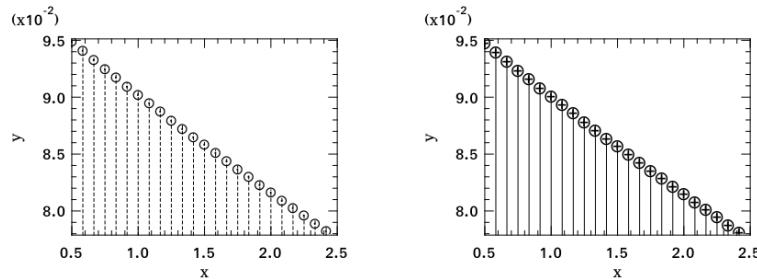


Figure 16: Stem plot

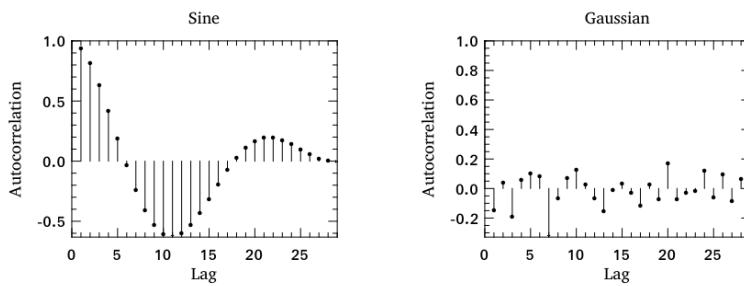


Figure 17: Stem plot with autocorrelation

```
Plot.output h
- : unit = ()
```

Stem plot is often used to show the autocorrelation of a variable, therefore Plot module already includes `autocorr` for your convenience.

```
# let x = Mat.linspace 0. 8. 30 in
let y0 = Mat.map Maths.sin x in
let y1 = Mat.uniform 1 30 in
let h = Plot.create ~m:1 ~n:2 "plot_010.png" in
Plot.subplot h 0 0;
Plot.set_title h "Sine";
Plot.autocorr ~h y0;
Plot.subplot h 0 1;
Plot.set_title h "Gaussian";
Plot.autocorr ~h y1;
Plot.output h
- : unit = ()
```

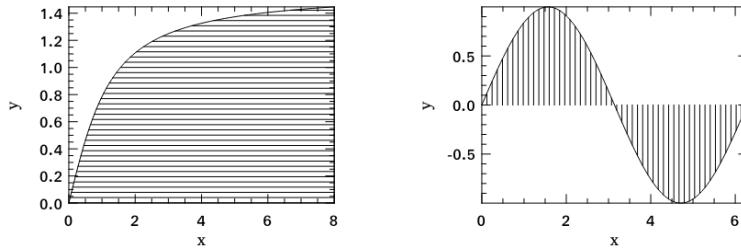


Figure 18: Area plot

Area Plot

Area plot is similar to the line plot, but it fills the space between the line and x-axis, as shown below.

```
# let x = Mat.linspace 0. 8. 100 in
let y = Mat.map Maths.atan x in
let h = Plot.create ~m:1 ~n:2 "plot_011.png" in
Plot.subplot h 0 0;
Plot.(area ~h ~spec:[ FillPattern 1 ] x y);
let x = Mat.linspace 0. (2. *. 3.1416) 100 in
let y = Mat.map Maths.sin x in
Plot.subplot h 0 1;
Plot.(area ~h ~spec:[ FillPattern 2 ] x y);
Plot.output h
- : unit = ()
```

Histogram & CDF Plot

Histogram is one of the most commonly used plot in data visualisation. Given a series of measurements, you can easily plot the histogram and empirical cumulative distribution of the data by using the `histogram` and `ecdf` plotting functions.

```
# let x = Mat.gaussian 200 1 in
let h = Plot.create ~m:1 ~n:2 "plot_012.png" in
Plot.subplot h 0 0;
Plot.set_title h "histogram";
Plot.histogram ~h ~bin:25 x;
Plot.subplot h 0 1;
Plot.set_title h "empirical cdf";
Plot.ecdf ~h x;
Plot.output h
- : unit = ()
```

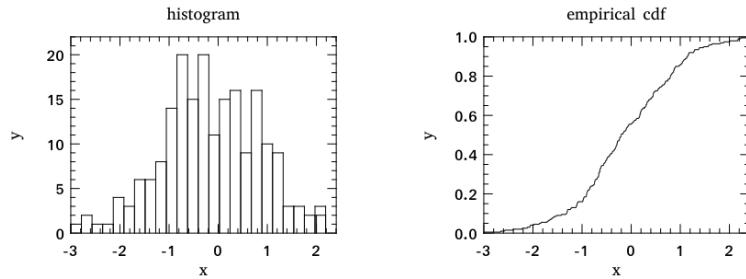


Figure 19: Histogram plot and CDF

Log Plot

In the Owl plots, you can choose to use the log-scale on either or both x and y axis.

```
# let x = Mat.logspace (-1.5) 2. 50 in
let y = Mat.map Maths.exp x in
let h = Plot.create ~m:2 ~n:2 "plot_013.png" in

Plot.subplot h 0 0;
Plot.xlabel h "Input Data X";
Plot.ylabel h "Input Data Y";
Plot.(loglog ~h ~spec:[ RGB (0,255,0); LineStyle 2; Marker "+" ] ~x:x y);

Plot.subplot h 0 1;
Plot.xlabel h "Index of Input Data Y";
Plot.ylabel h "Input Data Y";
Plot.(loglog ~h ~spec:[ RGB (0,0,255); LineStyle 1; Marker "*" ] y);

Plot.subplot h 1 0;
Plot.xlabel h "Input Data X";
Plot.ylabel h "Input Data Y";
Plot.semilogx ~h ~x:x y;

Plot.subplot h 1 1;
Plot.xlabel h "Index of Input Data Y";
Plot.ylabel h "Input Data Y";
Plot.semilogy ~h y;

Plot.output h
- : unit = ()
```

3D Plot

We have seen examples of 3D plots above. There are four functions in `Plot` module related to 3D plot. They are `surf`, `mesh`, `heatmap`, and `contour` functions. First, let's look at `mesh` and `surf` functions.

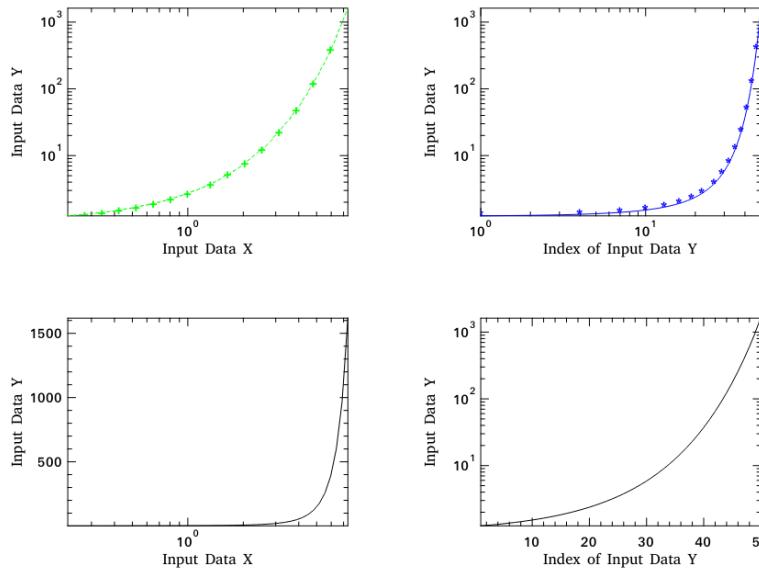


Figure 20: Change plot scale on x- and y-axis to log

```
# let x, y = Mat.meshgrid (-2.5) 2.5 (-2.5) 2.5 100 100 in
let z0 = Mat.(sin ((x **$ 2.) + (y **$ 2.))) in
let z1 = Mat.(cos ((x **$ 2.) + (y **$ 2.))) in
let h = Plot.create ~m:2 ~n:2 "plot_014.png" in
Plot.subplot h 0 0;
Plot.surf ~h x y z0;
Plot.subplot h 0 1;
Plot.mesh ~h x y z0;
Plot.subplot h 1 0;
Plot.surf ~h x y z1;
Plot.subplot h 1 1;
Plot.mesh ~h x y z1;
Plot.output h
- : unit = ()
```

It is easy to control the viewpoint with `altitude` and `azimuth` parameters. Here is an example.

```
# let x, y = Mat.meshgrid (-2.5) 2.5 (-2.5) 2.5 100 100 in
let z = Mat.(sin ((x * x) + (y * y))) in
let h = Plot.create ~m:1 ~n:3 "plot_015.png" in
Plot.subplot h 0 0;
Plot.(mesh ~h ~spec:[ Altitude 50.; Azimuth 120. ] x y z);
Plot.subplot h 0 1;
Plot.(mesh ~h ~spec:[ Altitude 65.; Azimuth 120. ] x y z);
```

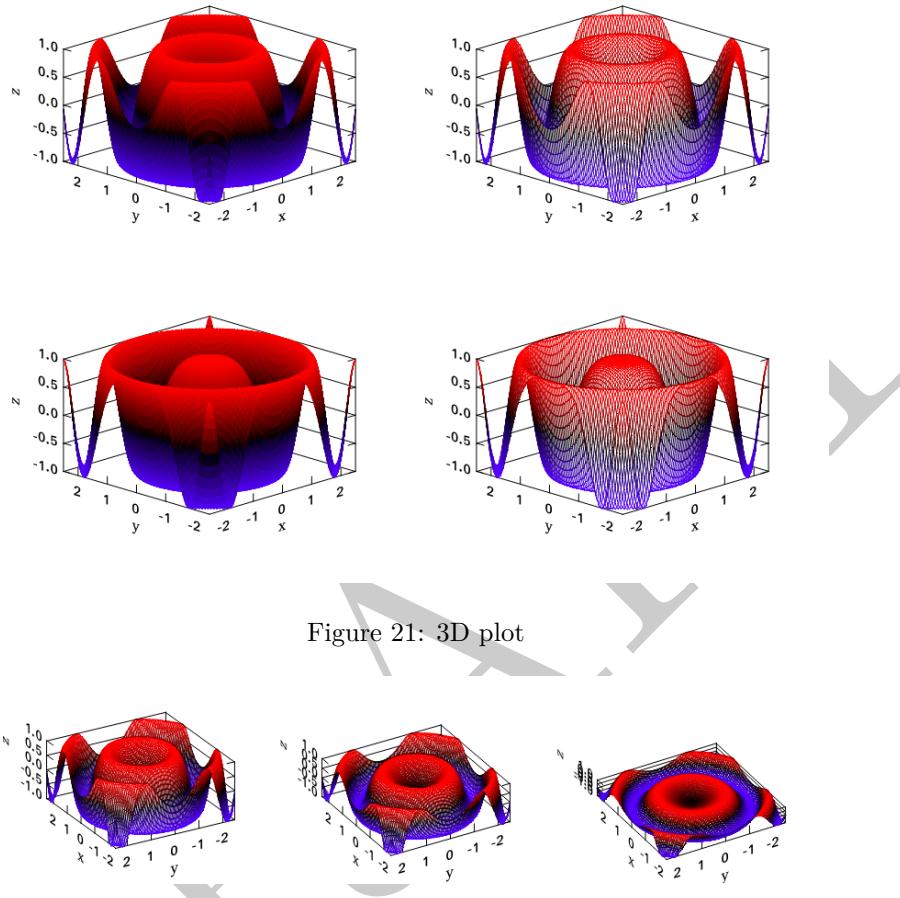


Figure 21: 3D plot

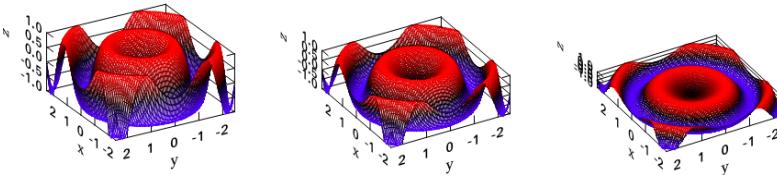


Figure 22: Customised 3D Plot, example 1

```
Plot.subplot h 0 2;
Plot.(mesh ~h ~spec:[ Altitude 80.; Azimuth 120. ] x y z);
Plot.output h
- : unit = ()
```

The generated figure is shown as below.

Here is another similar example with different functions.

```
# let x, y = Mat.meshgrid (-3.) 3. (-3.) 3. 50 50 in
let z = Mat.(
  3. $$ ((1. $- x) **$ 2.) * exp (neg (x **$ 2.) - ((y +$ 1.) **$ 2.)) -
  (10. $$ (x /$ 5. - (x **$ 3.) - (y **$ 5.)) * (exp (neg (x **$ 2.) - (y **$ 2.))) -
  ((1./.3.) $$ exp (neg ((x +$ 1.) **$ 2.) - (y **$ 2.)))
```

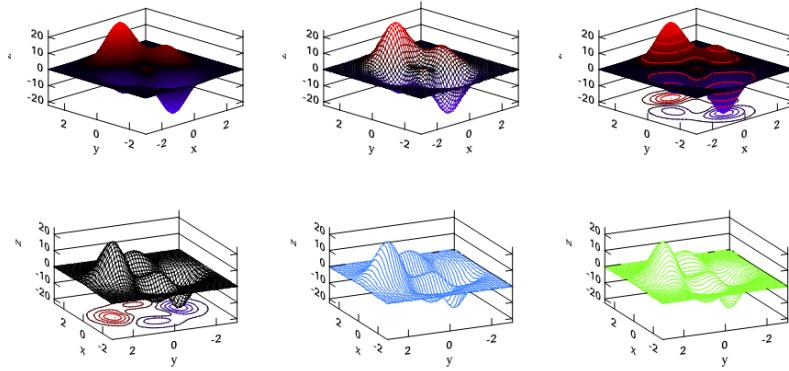


Figure 23: Customised 3D Plot, example 2

```

)
in

let h = Plot.create ~m:2 ~n:3 "plot_016.png" in
Plot.subplot h 0 0;
Plot.surf ~h x y z;
Plot.subplot h 0 1;
Plot.mesh ~h x y z;
Plot.subplot h 0 2;
Plot.(surf ~h ~spec:[ Contour ] x y z);
Plot.subplot h 1 0;
Plot.(mesh ~h ~spec:[ Contour; Azimuth 115.; NoMagColor ] x y z);
Plot.subplot h 1 1;
Plot.(mesh ~h ~spec:[ Azimuth 115.; ZLine X; NoMagColor; RGB (61,129,255) ] x y
z);
Plot.subplot h 1 2;
Plot.(mesh ~h ~spec:[ Azimuth 115.; ZLine Y; NoMagColor; RGB (130,255,40) ] x y
z);
Plot.output h
- : unit = ()

```

Finally, let's look at how heatmap and contour plot look like, using the same function as before.

```

# let x, y = Mat.meshgrid (-3.) 3. (-3.) 3. 100 100 in
let z = Mat.(
  3. $$ ((1. $- x) **$ 2.) * exp (neg (x **$ 2.) - ((y +$ 1.) **$ 2.)) -
  (10. $$ (x /$ 5. - (x **$ 3.) - (y **$ 5.)) * (exp (neg (x **$ 2.) - (y **$ 2.))) -
  ((1./3.) $$ exp (neg ((x +$ 1.) **$ 2.) - (y **$ 2.)))
)
in

let h = Plot.create ~m:2 ~n:2 "plot_017.png" in

```

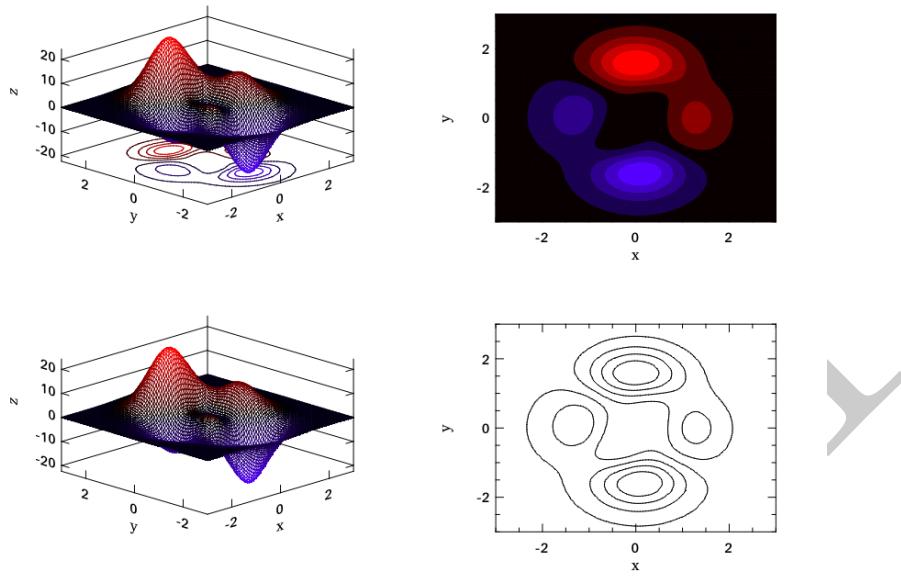


Figure 24: Headmap and contour plot

```

Plot.subplot h 0 0;
Plot.(mesh ~h ~spec:[ Contour ] x y z);
Plot.subplot h 0 1;
Plot.heatmap ~h x y z;
Plot.subplot h 1 0;
Plot.mesh ~h x y z;
Plot.subplot h 1 1;
Plot.contour ~h x y z;
Plot.output h
- : unit = ()

```

Advanced Statistical Plot

Besides these commonly used basic plot types, we also support several advanced statistical plots. For example, both the `qqplot` and `probplot` are simple graphical tests for determining if a data set comes from a certain distribution.

A `qqplot` displays a quantile-quantile plot of the quantiles of the sample data `y` versus the theoretical quantiles values from a given distribution, or the quantiles of the sample data `x`. Here is an example.

```

# let y = Mat.(gaussian 100 1 *$ 10.) in
let x = Mat.gaussian 200 1 in
let h = Plot.create ~m:2 ~n:2 "plot_018.png" in

```

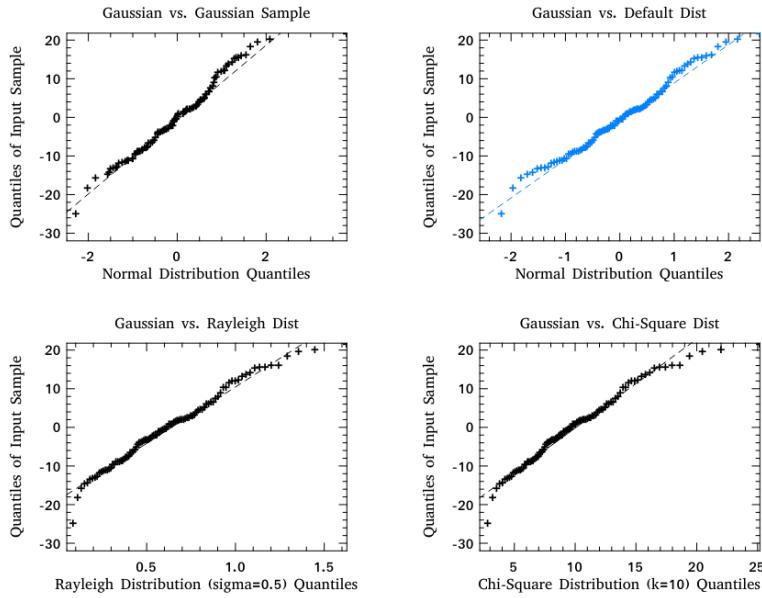


Figure 25: Advanced statistical plots with qqplot

```

Plot.subplot h 0 0;
Plot.set_title h "Gaussian vs. Gaussian Sample";
Plot.set_ylabel h "Quantiles of Input Sample";
Plot.set_xlabel h "Normal Distribution Quantiles";
Plot.qqplot ~h y ~x:x;

Plot.subplot h 0 1;
Plot.set_title h "Gaussian vs. Default Dist";
Plot.set_ylabel h "Quantiles of Input Sample";
Plot.set_xlabel h "Normal Distribution Quantiles";
Plot.(qqplot ~h y ~spec:[RGB (0,128,255)]);

Plot.subplot h 1 0;
Plot.set_title h "Gaussian vs. Rayleigh Dist";
Plot.set_ylabel h "Quantiles of Input Sample";
Plot.set_xlabel h "Rayleigh Distribution (sigma=0.5) Quantiles";
Plot.qqplot ~h y ~pd:(fun p -> Stats.rayleigh_ppf p 0.5);

Plot.subplot h 1 1;
Plot.set_title h "Gaussian vs. Chi-Square Dist";
Plot.set_ylabel h "Quantiles of Input Sample";
Plot.set_xlabel h "Chi-Square Distribution (k=10) Quantiles";
Plot.qqplot ~h y ~pd:(fun p -> Stats.chi2_ppf p 10.);

Plot.output h
- : unit = ()

```

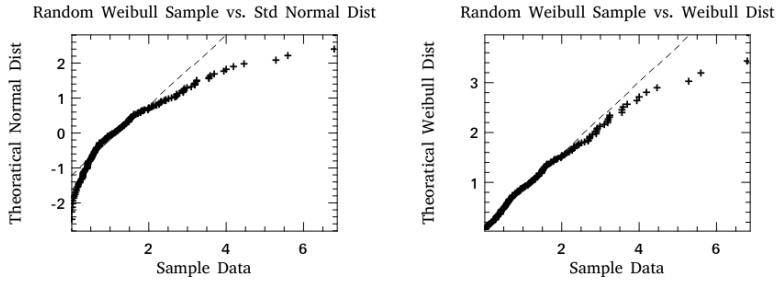


Figure 26: Advanced statistical plots with probplot

The probplot is similar to qqplot. It contains two special cases: `normplot` for when the given theoretical distribution is Normal distribution, and `wblplot` for Weibull Distribution. Here is an example of them.

```
# let x = Mat.empty 200 1 |> Mat.map (fun _ -> Stats.weibull_rvs 1.2 1.5) in
let h = Plot.create ~m:1 ~n:2 "plot_019.png" in
  Plot.subplot h 0 0;
  Plot.set_title h "Random Weibull Sample vs. Std Normal Dist";
  Plot.set_xlabel h "Sample Data";
  Plot.set_ylabel h "Theoretical Normal Dist";
  Plot.normplot ~h x;

  Plot.subplot h 0 1;
  Plot.set_title h "Random Weibull Sample vs. Weibull Dist";
  Plot.set_xlabel h "Sample Data";
  Plot.set_ylabel h "Theoretical Weibull Dist";
  Plot.wblplot ~h ~lambda:1.2 ~k:1.5 x;
  Plot.output h
- : unit = ()
```

Summary

This chapter demonstrate the plotting module in Owl, including how to create, manipulate plots, to the different types of plots that are supported. This chapter provides a lot of examples. It aims to be a manual that you can keep going back to check when you need visualisation in your task. In fact, most of the plots in this book are based on this module.

One pitfall in this chapter is that, we may not have given too much consideration about the “color harmony”. The default choice of colour may not always be pleasing to your eyes. The good news is that, you can easily change the colors in the plot. Try google “colour theory” and you can find a lot of guidelines. For example, the analogous colours can be a good choice in your line plots. These colours are any sequential three colors on a 12-part color wheel, such as yellow-

green, yellow, and yellow-orange. We find this artistic aspect of visualisation is often enjoyable.

References

DRAFT

Mathematical Functions

Starting from this chapter, we begin our journey to explore the world of numerical computing with Owl. But let's not be hasty. Before the main dishes such as algorithmic differentiation, optimisation, computation graph, etc., let's taste some starters first. In this chapter, we will introduce how the familiar mathematical operations are supported in Owl. This chapter is organised according to different types of functions, and you can feel free to browse any of them. Note that functions in this chapter work on scalar values. The N-dimensional array module introduced in later chapters contains these basic functions that work on n-dimensional arrays, including vectors and matrices.

Basic Functions

Basic Unary Math Functions

Many basic mathematical functions take one float number as input and return one float number. We call them *unary* functions. You can use these unary functions easily from the `Maths` module. For example:

```
# Maths.sqrt 2.
- : float = 1.41421356237309515
```

The tbl. 4 lists these unary functions supported in this module.

Table 4: Basic unary math functions

Function	Explanation
<code>abs</code>	$ x $
<code>neg</code>	$-x$
<code>reci</code>	$1/x$
<code>floor</code>	the largest integer that is smaller than x
<code>ceil</code>	the smallest integer that is larger than x
<code>round</code>	rounds x towards the bigger integer when on the fence
<code>trunc</code>	integer part of x
<code>sqr</code>	x^2
<code>sqrt</code>	\sqrt{x}

Basic Binary Functions

Unlike the unary ones, the *binary functions* take two floats as inputs and return one float as output. Most common arithmetic functions belong to this category, as shown in tbl. 5.

Table 5: Binary math functions

Function	Explanation
add	$x + y$
sub	$x - y$
mul	$x * y$
div	x / y
fmod	$x \% y$
pow	x^y
hypot	$\sqrt{x^2 + y^2}$
atan2	returns $\arctan(y/x)$, accounting for the sign of the arguments; this is the angle to the vector (x, y) counting from the x-axis.

Exponential and Logarithmic Functions

The constant $e = \sum_{n=0}^{\infty} \frac{1}{n!}$ is what we call the *natural constant*. It is named this way because the exponential function and its inverse function logarithm are so frequently used in nature and our daily life: logarithmic spiral, population growth, carbon date ancient artifacts, computing bank investments, etc. As an example, in a scientific experiment about bacteria, we can assume the number of bacterial at time t follows an exponential function $n(t) = Ce^{rt}$ where C is the initial population and r is the daily increase rate. With this model, we can predict how the population of bacterial grows within certain time.

We also have this beautiful Euler's formula that connects the two most frequently used constants and the base of complex numbers and natural numbers:

$$e^{i\pi} + 1 = 0.$$

The full list of exponential and logarithmic functions, together with some handy variants, are presented in tbl. 6.

Table 6: Exponential and logarithmic math functions

Function	Explanation
exp	exponential e^x
exp2	2^x
exp10	10^x
expm1	returns $\exp(x) - 1$ but more accurate for $x \sim 0$
log	$\log_e x$
log2	$\log_2 x$
log10	$\log_{10} x$
logn	$\log_n x$

Function	Explanation
log1p	inverse of expm1
logabs	$\log(x)$
xlogy	$x \log(y)$
xlog1py	$x \log(y + 1)$
logit	$\log(p/(1 - p))$
expit	$1/(1 + \exp(-x))$
log1mexp	$\log(1 - \exp(x))$
log1pexp	$\log(1 + \exp(x))$

Trigonometric Functions

The sine, cosine, and tangent functions belong to the *trigonometric functions*, which relate ratios of two side lengths to an angle of a right-angled triangle. They include the commonly used `sin` and `cos` etc., and their reciprocals such as the cosecant, secant, etc. These functions are widely used in the numerical computation applications of different fields, such as mechanics and geometry. The triangular functions are all unary functions, for example:

```
# Maths.sin (Owl_const.pi /. 2.)
- : float = 1.
```

They are all included in the math module in Owl, as shown in tbl. 7.

Table 7: Trigonometric math functions

Function	Explanation	Derivatives	Taylor Expansion
sin	$\sin(x)$	$\cos(x)$	$\sum_{n=1} (-1)^{n+1} \frac{x^{2n+1}}{(2n+1)!}$
cos	$\cos(x)$	$-\sin(x)$	$\sum_{n=1} (-1)^n \frac{x^{2n}}{(2n)!}$
tan	$\tan(x)$	$1 + \tan^2(x)$	$\sum_{n=1} \frac{4^n (4^n - 1) B_n}{(2n)!} x^{2n-1}$
cot	$1/\tan(x)$	$-(1 + \cot^2(x))$	$\sum_{n=0} \frac{E_n x^{2n}}{(2n)!}$
sec	$1/\cos(x)$	$\sec(x) \tan(x)$	$\sum_{n=0} \frac{2(2^{2n-1}) B_n}{(2n)!} x^{2n-1}$
csc	$1/\sin(x)$	$-\csc(x) \cot(x)$	$\frac{1}{x} - \sum_{n=1} \frac{4^n B_n}{(2n)!} x^{2n-1}$

Here the B_n is the n th Bernoulli number, and E_n is the n th Euler number. The fig. 27 shows the relationship between these trigonometric functions. This figure is inspired by a wiki post. These functions also have corresponding inverse functions: `asin`, `acos`, `atan`, `acot`, `asec`, `acsc`. For example, if $\sin(a) = b$, then $\sin(b) = a$.

Another related idea is the *Hyperbolic functions* such as `sinh` and `cosh`. These func-

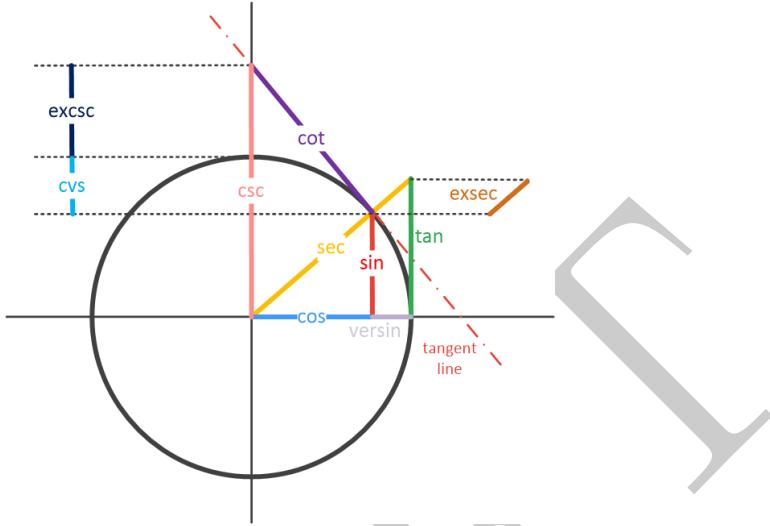


Figure 27: Relationship between different trigonometric functions

tions are defined using exponential functions. We have seen in [fig:algodiff:trio] that the trigonometric functions are related to a circle. Similarly, the hyperbolic functions are related to a hyperbola. For example, the points $(\cosh(x), \sinh(x))$ form the right half of the equilateral hyperbola, just like $(\cos(x), \sin(x))$ on a circle. The hyperbolic functions is applied widely in numerical computing, such as in the differential equation solutions, hyperbolic geometry, etc.

These functions in Owl are shown below:

- $\sinh: \frac{e^x - e^{-x}}{2}$, derivative is $\cosh(x)$, and taylor expansion is $\sum_{n=0} \frac{x^{2n+1}}{(2n+1)!}$.
- $\cosh: \frac{e^x + e^{-x}}{2}$, derivative is $\sinh(x)$, and taylor expansion is $\sum_{n=0} \frac{x^{2n+1}}{(2n+1)!}$.
- $\tanh: \frac{\sinh x}{\cosh x}$, derivative is $1 - \tanh^2(x)$, and taylor expansion is $\sum_{n=1} \frac{4^n(4^n-1)B_{2n} x^{2n-1}}{(2n)!}$.
- $\coth: \frac{\cosh x}{\sinh x}$, derivative is $1 - \coth^2(x)$, and taylor expansion is $\frac{1}{x} - \sum_{n=1} \frac{4^n B_{2n} x^{2n-1}}{(2n)!}$.
- $\operatorname{sech}: 1/\cosh(x)$, derivative is $-\tanh(x)/\cosh(x)$, and taylor expansion is $\sum_{n=0} \frac{E_{2n} x^{2n}}{(2n)!}$.
- $\operatorname{csch}: 1/\sinh(x)$, derivative is $-\coth(x)/\sinh(x)$, and taylor expansion is $\frac{1}{x} + \sum_{n=1} \frac{2(-2^{2n-1})B_{2n} x^{2n-1}}{(2n)!}$.

Similarly, each of these functions has a corresponding inverse functions: `asinh`, `acosh`, `atanh`, `acoth`, `asech`, `acsch`. The relationship between these hyperbolic trigonometric functions are clearly depicted in fig. 28.

Besides these functions, there are also some related functions. `sinc` returns $\sin(x)/x$ and 1 for $x = 0$. `logsinh` returns $\log(\sinh(x))$ but handles large $|x|$.

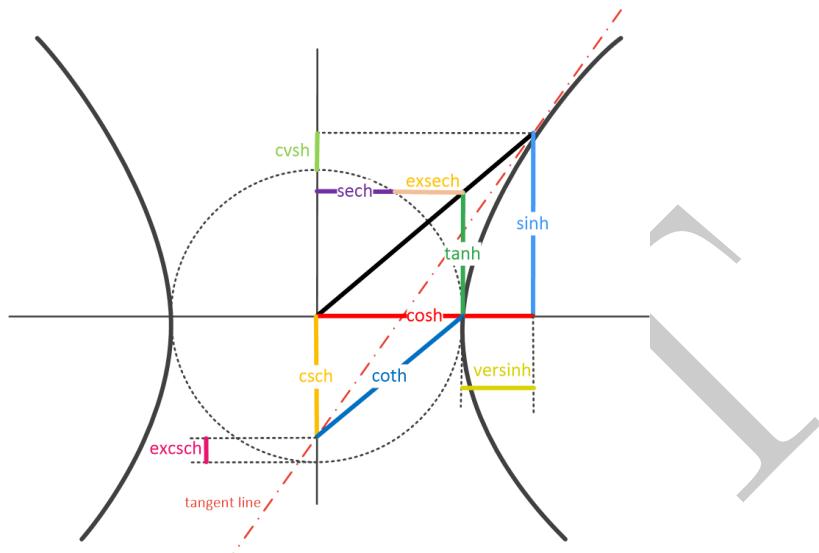


Figure 28: Relationship between different hyperbolic trigonometric functions

`logcosh` returns $\log(\cosh(x))$ but handles large $|x|$. `sindg/cosdg/tandg/cotdg` are the sine/cosine/tangent/cotangent of the angle given in degrees.

Other Math Functions

There are some other functions that may not be very commonly used in traditional mathematics. Functions such as `sigmoid` and `relu` are frequently used in the Deep Learning as the activation functions in a neural network. The activation functions are crucial to the neural network regarding various aspects, including output result, accuracy, convergence speed, etc. We will talk about them in detail in the Neural Network chapter later in this book.

- `sigmoid x`: $1/(1 + \exp(-x))$
- `signum x`: returns the sign of x : -1, 0, or 1
- `softsign x`: smooths `sign` function
- `relu x`: $\max(0, x)$

Special Functions

Besides what we have just listed, there are also a lot special functions. You may not heard of them before, but they have established names and are important in different fields such as mathematical analysis, physics, etc. In Owl, the implementations of these functions rely on the Cephes Mathematical Functions Library, a C language library with special functions of interest to scientists and engineers. They are list in the rest of this section. Perhaps we cannot dig deep

into the mathematical or physical implication of all these functions, but you may find them handy when you need one.

Airy Functions

The Airy function $\text{Ai}(x)$ is named after the British astronomer George B. Airy. It is the solution of the second order linear differential equation:

$$y''(x) = xy(x).$$

This differential equation has two linearly independent solutions Ai and Bi . Owl provides the `airy` function to do that:

```
val airy : float -> float * float * float * float
```

The four returned numbers are Ai , its derivative Ai' , Bi , and its derivative Bi' . Let's look at an example. It plots the two solutions Ai and Bi in fig. 29.

```
let x = Mat.linspace (-15.) 5. 200
let y0 = Mat.map (fun x ->
  let ai, _, _, _ = Maths.airy x in ai
) x
let y1 = Mat.map (fun x ->
  let _, _, bi, _ = Maths.airy x in bi
) x
let _ =
  let h = Plot.create "special_airy.png" in
  Plot.(plot ~h ~spec:[ RGB (66, 133, 244); LineStyle 1; LineWidth 2. ] x y0);
  Plot.(plot ~h ~spec:[ RGB (219, 68, 55); LineStyle 2; LineWidth 2. ] x y1);
  Plot.(set_yrange h (-0.5) 1.);
  Plot.(legend_on h ~position:SouthEast [| "Ai"; "Bi" |]);
  Plot.output h
```

Bessel Functions

Bessel functions, first defined by the mathematician Daniel Bernoulli and then generalized by Friedrich Bessel, are canonical solutions of Bessel's differential equation:

$$x^2y'' + xy' + (x^2 - \alpha^2)y = 0.$$

The complex number α is called the “order” of the bessel function. Bessel functions are important for many problems in studying the wave propagation and static potentials, such as electromagnetic waves in a cylindrical waveguide.

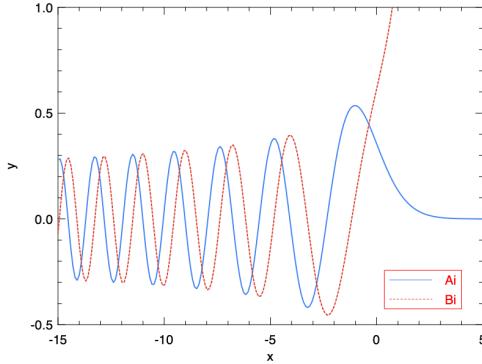


Figure 29: Examples of the two solutions of an Airy equation

In solving cylindrical coordinate systems, Bessel functions of integer order or half integer order are often used.

The Bessel functions can be divided into two kinds. Both kinds are solutions to the Bessel's differential equations, but the first kind is non-singular at the origin, while the second kind is singular at the origin ($x = 0$). A special case is when x is purely imaginary. In this case, the solutions are called the modified Bessel functions, which can also be categorised into first kind and second kind. Based on these category, Owl provides these functions.

Table 8: Bessel functions

Function	Explanation
<code>j0 x</code>	Bessel function of the first kind of order 0
<code>j1 x</code>	Bessel function of the first kind of order 1
<code>jv x y</code>	Bessel function of the first kind of real order
<code>y0 x</code>	Bessel function of the second kind of order 0
<code>y1 x</code>	Bessel function of the second kind of order 1
<code>yv x y</code>	Bessel function of the second kind of real order
<code>yn a x</code>	Bessel function of the second kind of integer order
<code>i0 x</code>	Modified Bessel function of order 0
<code>i1 x</code>	Modified Bessel function of order 1
<code>iv x y</code>	Modified Bessel function of real order
<code>i0e x</code>	Exponentially scaled modified Bessel function of order 0
<code>i1e x</code>	Exponentially scaled modified Bessel function of order 1
<code>k0 x</code>	Modified Bessel function of the second kind of order 0

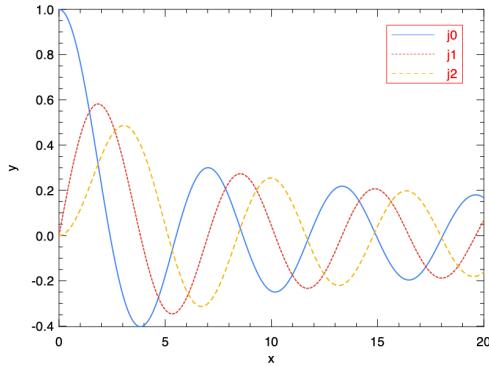


Figure 30: Examples of Bessel function of the first kind, with different order

Function	Explanation
k1	Modified Bessel function of the second kind of order 1
k0e	Exponentially scaled modified Bessel function of the second kind of order 0
k1e	Exponentially scaled modified Bessel function of the second kind of order 1

In the example below, we plot the first kind Bessel function of order 0, 1, and 2, as shown in fig. 30.

```

let x = Mat.linspace (0.) 20. 200
let y0 = Mat.map Maths.j0 x
let y1 = Mat.map Maths.j1 x
let y2 = Mat.map (Maths.jv 2.) x

let _ =
  let h = Plot.create "example_bessel.png" in
  Plot.(plot ~h ~spec:[ RGB (66, 133, 244); LineStyle 1; LineWidth 2. ] x y0);
  Plot.(plot ~h ~spec:[ RGB (219, 68, 55); LineStyle 2; LineWidth 2. ] x y1);
  Plot.(plot ~h ~spec:[ RGB (244, 180, 0); LineStyle 3; LineWidth 2. ] x y2);
  Plot.legend_on h ~position:NorthEast [|"j0"; "j1"; "j2"|]);
  Plot.output h

```

Elliptic Functions

The *Jacobian elliptic functions* are used in studying the pendulum motion. There are twelve Jacobi elliptic functions and `ellipj` we include here in Owl returns three of them: `sn`, `cn`, and `dn`. The fourth output `phi` of this function is called the amplitude of input `u`.

On the other hand, the *Elliptic integrals* are initially used to find the perimeters of ellipses. A Elliptic integral function can be expressed in the form of:

$$f(x) = \int_c^x R(t, \sqrt(P(t)))dt,$$

where R is a rational function of its two arguments, P is a polynomial of degree 3 or 4 with no repeated roots, and c is a constant. An elliptic integral can be categorised as “complete” or “incomplete”. The former one is function of a single argument, while the latter contains two arguments. Each elliptic integral can be transformed so that it contains integrals of rational functions and the three Legendre canonical forms, according to which the elliptic can be categorised into the first, second, and third kind. The elliptic functions in Owl are listed in tbl. 9.

Table 9: Elliptic functions

Function	Explanation
<code>ellipj u m</code>	Jacobian elliptic functions of parameter m between 0 and 1, and real argument u
<code>ellipk m</code>	Complete elliptic integral of the first kind
<code>ellipkm1 p</code>	Complete elliptic integral of the first kind around $m = 1$
<code>ellipkinc phi m</code>	Incomplete elliptic integral of the first kind
<code>ellipe m</code>	Complete elliptic integral of the second kind
<code>ellipeinc phi m</code>	Incomplete elliptic integral of the second kind

We can use `ellipe` to compute the circumference of an ellipse. To compute that normally requires calculus, but the elliptic functions provides a simple solution. Suppose an ellipse has semi-major axis $a = 4$ and semi-minor axis $b = 3$. We can compute its circumference simply using $4a\text{ellipe}(1 - \frac{b^2}{a^2})$.

```
# let a = 4.
val a : float = 4.
# let b = 3.
val b : float = 3.
# let c = 4. *. a *. Maths.(ellipe (1. -. pow (b /. a) 2.))
val c : float = 22.1034921607095072
```

Gamma Functions

For a positive integer n, the *Gamma function* is the factorial function:

$$\Gamma(n) = (n - 1)!$$

For a complex numbers z with a positive real part, the Gamma function is defined as:

$$\Gamma(z) = \int_0^{\infty} x^{z-1} e^{-x} dx.$$

Here the gamma function is an integral from zero to infinity. If we change it to an integral from zero to a certain upper limit, it is called the “lower incomplete gamma function”. Similarly, if it is an integral from a certain lower limit to infinity, the integral is called the “upper incomplete gamma function”.

The Gamma function is widely used in a range of areas such as fluid dynamics, geometry, astrophysics, etc. It is especially suitable for describing a common pattern of processes that decay exponentially in time or space. The Gamma function and related function provided in Owl are listed in tbl. 10.

Table 10: Gamma functions

Function	Explanation
gamma z	Returns the value of the Gamma function
rgamma z	Reciprocal of the Gamma function
loggamma z	Principal branch of the logarithm of the Gamma function
gammainc a x	Regularized lower incomplete gamma function
gammaincinv a y	Inverse function of gammainc
gammaincc a x	Complemented incomplete gamma integral
gammainccinv a y	Inverse function of gammaincc
psi z	The digamma function

Here is an example of using `gamma`.

```
let x = Mat.linspace (-3.5) 5. 2000
let y = Mat.map Maths.gamma x
let _ =
  let h = Plot.create "example_gamma.png" in
  Plot.(plot ~h ~spec:[ RGB (66, 133, 244); LineStyle 1; LineWidth 2. ] x y);
  Plot.(set_yrange h (-10.) 20.);
  Plot.output h
```

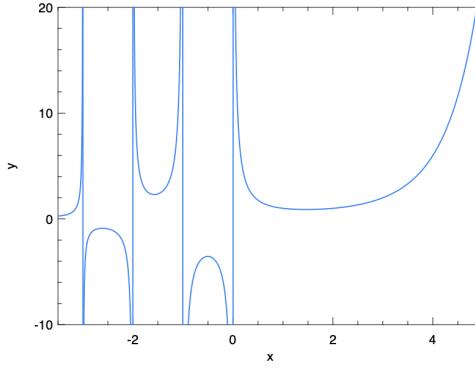


Figure 31: Examples of Gamma function along part of the real axis

Beta Functions

Beta function is defined as:

$$B(x, y) = \int_0^1 t^{x-1} (1-t)^{y-1} dt = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

Here Γ is the Gamma function, and similar to it, the Beta function has its “incomplete” version. The incomplete Beta function extends this definition to:

$$B(x, a, b) = \int_0^x t^{a-1} (1-t)^{b-1} dt.$$

In Owl the Beta function is called using the `beta x y` function from the `Maths` module, and its incomplete version is `betainc a b x`. We also provide the `betaincinv` function, which is the inverse of `betainc`.

The Beta function has several properties. For example, the code below shows the relationship between beta function and gamma function.

```
# let x = Maths.beta 3. 4.
val x : float = 0.016666666666666664
# let y = Maths.((gamma 3.) *. (gamma 4.) /. (gamma 7.))
val y : float = 0.016666666666666664
```

Another property is its symmetry, which means $B(x, y) = B(y, x)$.

```
# let x = Maths.beta 3. 4.
val x : float = 0.016666666666666664
```

```
# let y = Maths.beta 4. 3.
val y : float = 0.016666666666666664
```

Beta function is the first known scattering amplitude in the String theory in physics. It is also used in the analysis of the preferential attachment process, a type of stochastic urn process that describes how the resource can be distributed among a group of individuals based on the existing resource each one has acquired.

Struve Functions

The *Struve function* is defined as:

$$H_v(x) = (z/2)^{v+1} \sum_{n=0}^{\infty} \frac{(-1)^n (z/2)^{2n}}{\Gamma(n + \frac{3}{2}) \Gamma(n + v + \frac{3}{2})},$$

where Γ is the Gamma function. x must be positive unless v is an integer. Struve functions are used across a wide variety of physics applications, such as water-wave problems and calculations in unsteady aerodynamics.

The Owl function `struve v x` returns the value of Struve function. The parameter v is called the *order* of this function. Here is an example that shows the curves of Struve functions with order from 0 to 4.

```
let _ =
  let h = Plot.create "example_struve.png" in
  Plot.(plot_fun ~h ~spec:[ RGB (66, 133, 244); LineStyle 1; LineWidth 2.]
    (Maths.struve 0.) (-12.) 12.);
  Plot.(plot_fun ~h ~spec:[ RGB (219, 68, 55); LineStyle 2; LineWidth 2.]
    (Maths.struve 1.) (-12.) 12.);
  Plot.(plot_fun ~h ~spec:[ RGB (244, 180, 0); LineStyle 3; LineWidth 2.]
    (Maths.struve 2.) (-12.) 12.);
  Plot.(plot_fun ~h ~spec:[ RGB (77, 81, 57); LineStyle 1; LineWidth 2.]
    (Maths.struve 3.) (-12.) 12.);
  Plot.(plot_fun ~h ~spec:[ RGB (111, 51, 129); LineStyle 2; LineWidth 2.]
    (Maths.struve 4.) (-12.) 12.);
  Plot.(set_yrange h (-3.) 5.);
  Plot.(legend_on h ~position:SouthEast [| "H0"; "H1"; "H2"; "H3"; "H4" |]);
  Plot.output h
```

Zeta Functions

The *Hurwitz zeta function* `zeta x q` is defined as:

$$\zeta(x, q) = \sum_{k=0}^{\infty} \frac{1}{(k + q)^x}.$$

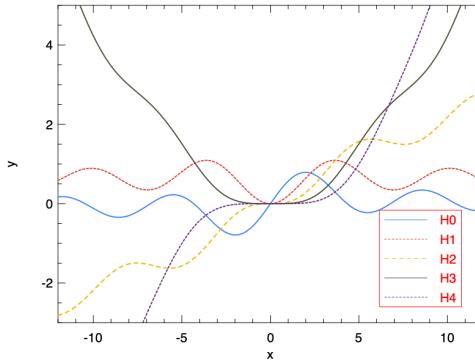


Figure 32: Examples of Struve function for different orders.

When q is set to 1, this function is reduced to the *Riemann zeta function*. The function `zetac x` returns Riemann zeta function minus 1. The zeta function is often used to analyse the dynamic systems. Besides, the Riemann zeta function plays an important role in number theory and is widely applied in quantum physics, probability theory, and applied statistics, etc.

We can evaluate the zeta function at certain points, for example:

```
# Maths.zeta 4. 1.
- : float = 1.08232323371113837

# (Maths.pow Owl_const.pi 4.) /. 90.
- : float = 1.08232323371113792
```

Error Functions

The error functions here are not about error processing in programming, but yet another family of special functions. In mathematics, an error function is defined as:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

The error function occurs often in probability and statistics. Actually, in statistics, for a non-negative value x , `erf x` is the probability that a random variable y falls in the range $[-x, x]$. Here y follows a normal distribution with mean 0 and variance 0.5. Since it represents certain probability, the *complementary* error function $1 - \text{erf}(x)$ is also frequently used. The error function and related variants in Owl are listed in tbl. 11.

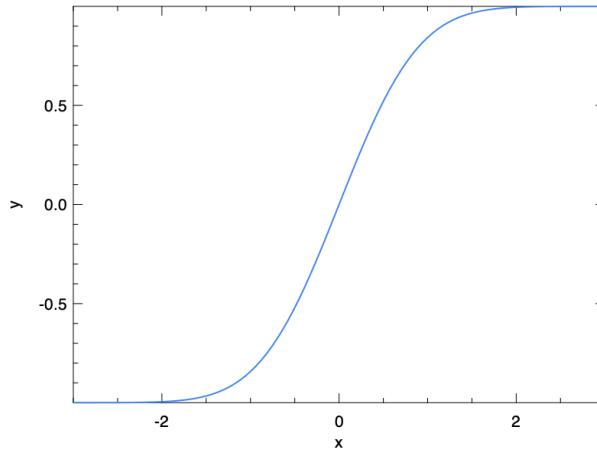


Figure 33: Plot of the Error function.

Table 11: Error functions

Function	Explanation
<code>erf x</code>	Error function
<code>erfc x</code>	Complementary error function: $1 - \text{erf}(x)$
<code>erfcx x</code>	Scaled complementary error function: $\exp(x^2)\text{erfc}(x)$
<code>erfinv x</code>	Inverse function of <code>erf</code>
<code>erfcinv x</code>	Inverse function of <code>erfc</code>

The error function is a sigmoid function. We can observe its shape by the code below.

```
let _ =
  let h = Plot.create "example_erf.png" in
  Plot.(plot_fun ~h ~spec:[ RGB (66, 133, 244); LineStyle 1; LineWidth 2.]
    Maths.erf (-3.) 3.);
  Plot.output h
```

Integral Functions

Besides what we have mentioned so far, Owl also provides several special integral functions.

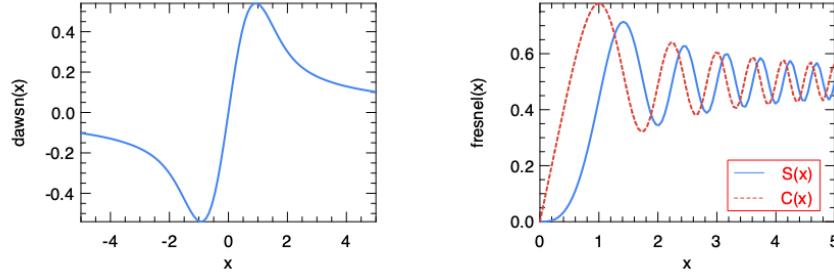


Figure 34: Plot of the Dawson and Fresnel integral function.

For example, the *Dawson function* is defined as:

$$D(x) = e^{-x^2} \int_0^x e^{t^2} dt$$

And the *Fresnel trigonometric integral* returns a tuple that contains two parts:

$$S(x) = \int_0^x \sin(t^2) dt, C(x) = \int_0^x \cos(t^2) dt.$$

Just like many other special functions, these two types of integrals are motivated by research in physics, such as the electromagnetic problems. They are provided by the `dawson` and `fresnel` functions in the `Maths` module respectively. We can observe the functions of these integrals with plots.

```
let _ =
  let h = Plot.create ~m:1 ~n:2 "example_integrals.png" in
  Plot.subplot h 0 0;
  Plot.(plot_fun ~h ~spec:[ RGB (66, 133, 244); LineStyle 1; LineWidth 2. ]
        Maths.dawson (-5.) 5.);
  Plot.set_ylabel h "dawson(x)";
  Plot.subplot h 0 1;
  Plot.(plot_fun ~h ~spec:[ RGB (66, 133, 244); LineStyle 1; LineWidth 2. ] (fun x
    -> let s, _ = Maths.fresnel x in s) 0. 5.);
  Plot.(plot_fun ~h ~spec:[ RGB (219, 68, 55); LineStyle 2; LineWidth 2. ] (fun x
    -> let _, c = Maths.fresnel x in c) 0. 5.);
  Plot.(legend_on h ~position:SouthEast [| "S(x)" ; "C(x)" |]);
  Plot.set_ylabel h "fresnel(x)";
  Plot.output h
```

Besides these two, several other type of special integral functions are also provided. The full list is shown in tbl. 12.

Table 12: Special integral functions

Function	Explanation
<code>expn n x</code>	Generalized exponential integral $E_n(x) = x^{n-1} \int_x^\infty \frac{e^{-t}}{t^n} dt$
<code>sh x</code>	Hyperbolic sine integral: $\int_0^x \frac{\sinh t}{t} dt$
<code>chi x</code>	Hyperbolic cosine integral: $\gamma + \log(x) + \int_0^x \frac{\cosh t - 1}{t} dt$
<code>shichi x</code>	(<code>sh x</code> , <code>chi x</code>)
<code>si x</code>	Sine integral: $\int_0^x \frac{\sin t}{t} dt$
<code>ci x</code>	Cosine integral: $\gamma + \log(x) + \int_0^x \frac{\cos t - 1}{t} dt$
<code>sici x</code>	(<code>si x</code> , <code>ci x</code>)

Factorials

After the functions, let's turn to a concept that we are familiar with: the *factorials*. The definition of factorials is simple:

$$F(n) = n! = n \times (n - 1) \times (n - 2) \dots \times 1$$

The factorial function, together with several of its variants, are contained in the `Math` module.

Table 13: Factorial functions

Function	Explanation
<code>fact n</code>	Factorial function $!n$
<code>log_fact n</code>	Logarithm of factorial function
<code>doublefact n</code>	Double factorial function calculates $n!! = n(n - 2)(n - 4) \dots 2$ (or 1)
<code>log_doublefact n</code>	Logarithm of double factorial function

The factorial functions accepts integer as input, for example:

```
# Maths.fact 5
- : float = 120.
```

The factorials are applied in many areas of mathematics, most notably the combinatorics. The permutation and combination are both defined in factorials. The permutation function returns the number $n!/(n - k)!$ of ordered subsets of length k , taken from a set of n elements. The combination function returns the number $\binom{n}{k} = n!/(k!(n - k)!)$ of subsets of k elements of a set of n elements.tbl. 14 provides the combinatorics functions you can use in the `Math` module.

Table 14: Permutation and combination functions

Function	Explanation
<code>permutation n k</code>	Permutation number
<code>permutation_float n k</code>	Similar to <code>permutation</code> but deals with larger range and returns float
<code>combination n k</code>	Combination number
<code>combination_float n k</code>	Similar to <code>combination</code> but deals with larger range and returns float
<code>log_combination n k</code>	Returns the logarithm of $\binom{n}{k}$

Let's take a look at a simple example.

```
# let x = Maths.combination 10 2
val x : int = 45
# let y = Maths.combination_float 10 2
val y : float = 45.
```

Interpolation and Extrapolation

Sometimes we don't know the full description of a function f , but only some points on it, and therefore we cannot calculate its value at an arbitrary point. The target is to estimate the $f(x)$ for an arbitrary x by drawing a smooth curve through the given data. If x is within the range of the given data, this task is called *interpolation*, otherwise it's called *extrapolation*, which is much more difficult to do.

The `owl_maths_interpolate` module provides an `polint` function for interpolation and extrapolation:

```
val polint : float array -> float array -> float -> float * float
```

The function `polint xs ys x` performs polynomial interpolation of the given arrays `xs` and `ys`. Given arrays $xs[0 \dots (n-1)]$ and $ys[0 \dots (n-1)]$, and a value x , this function returns a value y , and an error estimate dy .

As its name suggests, the `polint` approximates complicated curves with polynomial of the lowest possible degree that passes the given points. We show how this interpolation method works with an example. In the previous section we have said that the Gamma function is actually an interpolation solution to the integer function $y(x) = (n-1)!$. So we can specify five nodes on a plane that are generated from this factorial functions, and see how the interpolation function works compared with the Gamma function itself.

```
# let x = [|2; 3; 4; 5; 6|]
val x : int array = [|2; 3; 4; 5; 6|]
# let y = Array.map (fun x -> Maths факт (x - 1)) x
val y : float array = [|1.; 2.; 6.; 24.; 120.|]
# let x = Array.map float_of_int x
val x : float array = [|2.; 3.; 4.; 5.; 6.|]
```

Now we can define the interpolation function f that accepts one float number and returns another float number. Also we convert the given data x and y into matrix format for plotting purpose.

```
let f a =
  let v, _ = Owl_maths_interpolate.polint x y a in
  v

let xm = Mat.of_array x 1 5
let ym = Mat.of_array y 1 5
```

Now we can plot the interpolation function and compare it to the Gamma function. As can be seen in fig. 35, both lines cross the given nodes. The interpolated line fits well with the “true interpolation”, i.e. the Gamma function, within a certain range. However, the extrapolation fitting where the x-value falls out of given data, is less than ideal.

```
let _ =
  let h = Plot.create "interp.png" in
  Plot.(plot_fun ~h ~spec:[ RGB (66, 133, 244); LineStyle 1; LineWidth 2.] f 2.
    6.5);
  Plot.(plot_fun ~h ~spec:[ RGB (219, 68, 55); LineStyle 2; LineWidth 2.] Maths.gamma 2. 6.5);
  Plot.(scatter ~h ~spec:[ Marker "#[0x229a]"; MarkerSize 5. ] xm ym);
  Plot.(legend_on h ~position:NorthWest [| "Interpolation"; "Gamma function";
    "Given values" |]);
  Plot.output h
```

Integration

We have introduced some special integral functions, but we still need general integration methods that work for any input functions. Given a function f that accepts a real variable and an interval $[a, b]$ of the real line, the integral of this function

$$\int_a^b f(x) dx$$

can be thought of as the sum of signed area of the region in the cartesian plane that is bounded by the curve of f , the x-axis within the x-axis range $[a, b]$. The

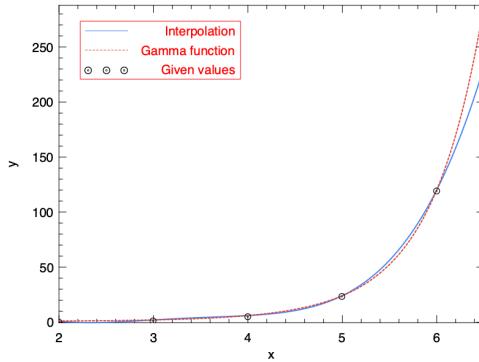


Figure 35: Plot of interpolation and corresponding Gamma function.

area above the x-axis adds to the sum and that below the x-axis subtracts from the area sum.

In the `owl_maths_quadrature` module, Owl provides several numerical routines to help you to do integrations. For example, we can compute $\int_1^4 x^2$ with the code below:

```
# Owl_maths_quadrature.trapz (fun x -> x ** 2.) 1. 4.
- : float = 21.0001344681758439
```

We can verify this result using the fundamental theorem of calculus:

$$\int_1^4 x^2 = (4^3 - 1^3)/3 = 21$$

So you might be thinking, what is this `trapz`? Why the result is not exactly 21? Using numerical methods (or *quadrature*) to do integration dates back to the invention of calculus or even earlier. The basic idea is to use summation of small areas to approximate that of an integration. There exist a lot of algorithms to do numerical integration, and using the trapezoidal rule is one of them.

This classical method divides a to b into N equally spaced abscissas: x_0, x_1, \dots, x_N . Each area between x_i and x_j is seen as an “Trapezoid” and the area formula is computed as:

$$\int_{x_0}^{x_1} f(x)dx = h\left(\frac{f(x_0)}{2} + \frac{f(x_1)}{2}\right) + O(h^3 f'').$$

Here the error term $O(h^3 f'')$ indicated that the error of approximation is related with that of abscissas size h and second order derivative of the original function. The function `trapz` implements this method. It's interface is:

```
val trapz : ?n:int -> ?eps:float -> (float -> float) -> float -> float
```

`trapz ~n ~eps f a b` computes the integral of f on the interval $[a,b]$ using the trapezoidal rule. It works by iterating for several stages, each stage improving the accuracy by adding more interior points. The argument n specifies the maximum step which defaults to 20, and eps is the desired fractional accuracy threshold, which defaults to $1e-6$.

The other methods are similar to `trapz` in interface, only different in implementation. For example, the `simpson` uses the Simpson formula:

$$\int_{x_0}^{x_2} f(x)dx = h\left(\frac{f(x_0)}{3} + \frac{4f(x_1)}{3} + \frac{f(x_2)}{3}\right) + O(h^5 f(4)).$$

Then there is the *Romberg integration* (`romberg`) that can choose methods of different orders to give good accuracy, and the algorithms are normally much faster than the `trapz` and `simpson` methods. Moreover, if the abscissas can be varied, we have the adaptive Gaussian quadrature of fixed tolerance (`gaussian`) and Gaussian quadrature of fixed order (`gaussian_fixed`).

As an example, we can compute the special sine integral function $Si(x) = \int_0^x \frac{\sin(t)}{t} dt$ from previous section using the numerical integration method. Let's set $x = 4$. We can see the numerical method `gaussian` works well to approximate this special integral function.

```
# let f t = Maths.(div (sin t) t)
val f : float -> float = <fun>
# Owl_maths_quadrature.gaussian f 0. 4.
- : float = 1.75820313914469306
# Owl_maths.si 4.
- : float = 1.75820313894905289
```

Utility Functions

Besides what we have mentioned so far, there are also some utility math functions that worth mentioning.

We know that a prime number is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers. It is a key idea in information technology and widely used in applications such as the public-key cryptography. The `is_prime` function checks if an integer is a prime number.

This function is deterministic for all numbers representable by an int. It is implemented using the Miller-Rabin primality test method.

```
# Maths.is_prime 997
- : bool = true
```

Another number theory related idea is the *Fermat's factorization*, which represents an odd integer as the difference of two squares: $N = a^2 - b^2$, and therefore n can be factorised as $(a + b)(a - b)$. The function `fermat_fact` performs Fermat factorisation over odd number n , i.e. into two roughly equal factors x and y so that $N = x \times y$.

```
# Maths.fermat_fact 6557
- : int * int = (83, 79)
# 83 * 79
- : int = 6557
```

Next two functions concerns the precision of float numbers in computer.

The `nextafter from to` returns the next representable double precision value of `from` in the direction of `to`. The other one, `nextafterf from to` returns the next representable single precision value of `from` in the direction of `to`. In both cases, if `from` equals `to`, this value itself is returned. For example:

```
# Maths.nextafterf 1. 2.;;
- : float = 1.00000011920928955
# Maths.nextafter 1. 2.;;
- : float = 1.00000000000000022
# Maths.nextafter 1. 0.;;
- : float = 0.99999999999999889
```

Summary

We start the topic of numeric computation from the mathematics functions we are familiar with. This chapter introduces the math functions supported by Owl, including the basic and commonly used functions, some less frequently used but nonetheless important special function, factorial, interpolation, extrapolation, integration, etc. Feel free to jump into any part you are interested in and use these functions to solve a mathematical problem at hand. You may find Owl as good a calculator as any other numerical library.

Statistical Functions

Statistics is an indispensable tool for data analysis, it helps us to gain the insights from data. The statistical functions in Owl can be categorised into three groups: descriptive statistics, distributions, and hypothesis tests.

Random Variables

We start from assigning probabilities to *events*. A event may comprise of finite or infinite number of possible outcomes. All possible output make up the *sample space*. To better capture this assigning processes, we need the idea of *Random Variables*.

A random variable is a function that associate sample output of events with some numbers of interests. Imagine the classic tossing coin game, we toss the coin four times, and the result is “head”, “head”, “tail”, “head”. We are interested in the number of “head” in this outcome. So we make a Random Variable “X” to denote this number, and $X([“head”, “head”, “tail”, “head”]) = 3$. You can see that using random variables can greatly reduce the event sample space.

Depending on the number of values it can be, a random variable can be broadly categorised into *Discrete Random Variable* (with finite number of possible output), and *Continuous Random Variable* (with infinite number of possible output).

Discrete Random Variables

Back to the coin tossing example. Suppose that the coin is specially minted so that the probability of tossing head is p . In this scenario, we toss for three times. Use the number of heads as a random variable X , and it contains four possible outcomes: 0, 1, 2, or 3.

We can calculate the possibility of each output result. Since each toss is a individual trial, the possibility of three heads $P(X=2)$ is p^3 . Two heads includes three cases: HHT, HTH, THH, each has a probability of $p^2(1-p)$, and together $P(X = 2) = 3p^2(1-p)$. Similarly $P(X = 1) = 3p(1-p)^2$, and $P(X = 0) = (1-p)^3$.

Formally, consider a series of n independent trials, each trial containing two possible results, and the result of interest happens at a possibility of p , then the possibility distribution of random variable X is (X being the number of result of interests):

$$P(X = k) = \binom{N}{k} p^k (1-p)^{n-k}. \quad (1)$$

This type of distribution is called the *Binomial Probability Distribution*. We can simulate this process of tossing coins with the `stats.binomial_rvs` function.

Suppose the probability of tossing head is 0.4, and for 10 times.

```
# let _ =
  let toss = Array.make 10 0 in
  Array.map (fun _ -> Stats.binomial_rvs 0.3 1) toss
- : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 1; 0|]
```

The equation eq. 1 is called the *probability density function* (PDF) of this binomial distribution. Formally the PDF of random variable X is denoted with $p_X(k)$ and is defined as:

$$p_X(k) = P(s \in S | X(s) = k),$$

where S is the sample space. This can also be expressed with the code:

```
# let x = [|0; 1; 2; 3|]
val x : int array = [|0; 1; 2; 3|]
# let p = Array.map (Stats.binomial_pdf ~p:0.3 ~n:3) x
val p : float array =
  [|0.34299999999999916; 0.44099999999999837; 0.18899999999999918;
   0.02699999999999823|]
# Array.fold_left (+.) 0. p
- : float = 0.99999999999999778
```

Aside from the PDF, another related and frequently used idea is to see the probability of random variable X being within a certain range: $P(a \leq X \leq b)$. It can be rewritten as $P(X \leq b) - P(X \leq a - 1)$. Here the term $P(X \leq t)$ is called the *Cumulative Distribution Function* of random variable X . For the binomial distribution, it CDF is:

$$P(X \leq k) = \sum_{i=0}^k \binom{N}{i} p^k (1-p)^{n-i}.$$

We can calculate the CDF in the 3-tossing problem with code again.

```
# let x = [|0; 1; 2; 3|]
val x : int array = [|0; 1; 2; 3|]
# let p = Array.map (Stats.binomial_cdf ~p:0.3 ~n:3) x
val p : float array = [|0.3429999999999972; 0.784; 0.973; 1.|]
```

Continuous Random Variables

Unlike discrete random variable, a continuous random variable has infinite number of possible outcomes. For example, in uniform distribution, we can pick

a random real number between 0 and 1. Apparently there can be infinite number of outputs.

One of the most widely used continuous distribution is no doubt the *Gaussian distribution*. Its probability function is a continuous one:

$$p(x) = \frac{1}{\sqrt{2\pi}\delta} s \exp -\frac{1}{2} \left(\frac{t-\mu}{\sigma} \right)^2 \quad (2)$$

Here the μ and σ are parameters. Depending on them, the $p(x)$ can take different shapes. Let's look at an example.

We generate two data sets in this example, and both contain 999 points drawn from different Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$. For the first one, the configuration is $(\mu = 1, \sigma = 1)$; whilst for the second one, the configuration is $(\mu = 12, \sigma = 3)$.

```
let noise sigma = Stats.gaussian_rvs ~mu:0. ~sigma;;
let x = Array.init 999 (fun _ -> Stats.gaussian_rvs ~mu:1. ~sigma:1.);;
let y = Array.init 999 (fun _ -> Stats.gaussian_rvs ~mu:12. ~sigma:3.);;
```

We can visualise the data sets using histogram plot as below. When calling `histogram`, we also specify 30 bins explicitly. You can also fine tune the figure using `spec` named parameter to specify the colour, x range, y range, and etc. We will discuss in details on how to use Owl to plot in a separate chapter.

```
(* convert arrays to matrices *)
let x' = Mat.of_array x 1 999;;
let y' = Mat.of_array y 1 999;;

(* plot the figures *)
let h = Plot.create ~m:1 ~n:2 "plot_02.png" in
  Plot.subplot h 0 0;
  Plot.set_ylabel h "frequency";
  Plot.histogram ~bin:30 ~h x';
  Plot.histogram ~bin:30 ~h y';

  Plot.subplot h 0 1;
  Plot.set_ylabel h "PDF p(x)";
  Plot.plot_fun ~h (fun x -> Stats.gaussian_pdf ~mu:1. ~sigma:1. x) (-2.) 6.;
  Plot.plot_fun ~h (fun x -> Stats.gaussian_pdf ~mu:12. ~sigma:3. x) 0. 25.;

Plot.output h;;
```

In subplot 1, we can see the second data set has much wider spread. In subplot 2, we also plot corresponding the probability density functions of the two data sets.

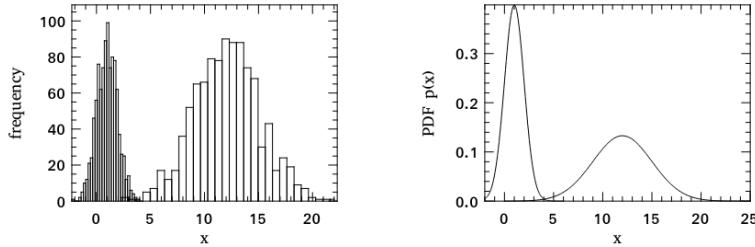


Figure 36: Probability density functions of two data sets

The CDF of Gaussian can be calculated with infinite summation, i.e. integration:

$$p(x \leq k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^k e^{-t^2/2} dt.$$

We can observe this function with `gaussian_cdf`.

```
let h = Plot.create "plot_gaussian_cdf.png" in
Plot.set_ylabel h "CDF";
Plot.plot_fun ~h ~spec:[ RGB (66,133,244); LineStyle 1; LineWidth 2.; Marker "*" ]
  (fun x -> Stats.gaussian_cdf ~mu:1. ~sigma:1. x) (-2.) 6.;
Plot.plot_fun ~h ~spec:[ RGB (219,68,55); LineStyle 2; LineWidth 2.; Marker "+" ]
  (fun x -> Stats.gaussian_cdf ~mu:12. ~sigma:3. x) 0. 25.;
Plot.(legend_on h ~position:SouthEast [|"mu=1,sigma=1"; "mu=12, sigma=3"|]);
Plot.output h
```

Descriptive Statistics

Mean, Variance: Definition, and math derivation of Gaussian as examples.

The definition of moments, and higher moments.

Descriptive statistics are used to summarise the characteristics of data. The commonly used ones are mean, variance, standard deviation, skewness, kurtosis, and etc.

We first draw one hundred random numbers which are uniformly distributed between 0 and 10. Here we use `Stats.uniform_rvs` function to generate numbers following uniform distribution.

```
| let data = Array.init 100 (fun _ -> Stats.uniform_rvs 0. 10.);;
```

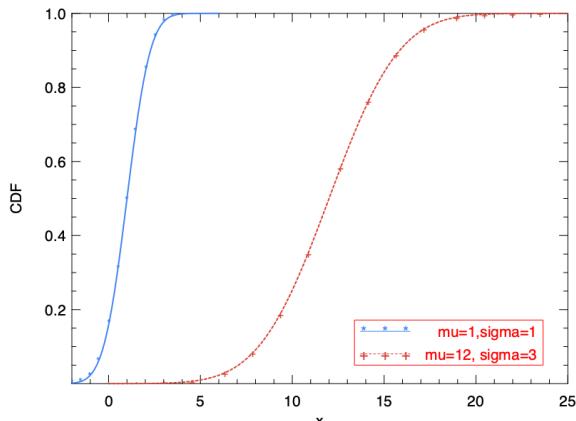


Figure 37: Cumulated density functions of two data sets

Then We use `mean` function calculate sample average. As we can see, it is around 5. We can also calculate higher moments such as variance and skewness easily with corresponding functions.

```
# Stats.mean data
- : float = 5.18160409659184573
# Stats.std data
- : float = 2.92844832850280135
# Stats.var data
- : float = 8.57580961271085229
# Stats.skew data
- : float = -0.109699186612116223
# Stats.kurtosis data
- : float = 1.75165078829330856
```

The following code calculates different central moments of `data`. A central moment is a moment of a probability distribution of a random variable about the random variable's mean. The zero-th central moment is always 1, and the first is close to zero, and the second is close to the variance.

```
# Stats.central_moment 0 data
- : float = 1.
# Stats.central_moment 1 data
- : float = -3.13082892944294137e-15
# Stats.central_moment 2 data
- : float = 8.49005151658374224
# Stats.central_moment 3 data
- : float = -2.75496511397836663
```

Order Statistics

Order statistics and rank statistics are among the most fundamental tools in non-parametric statistics and inference. The k^{th} order statistic of a statistical sample is equal to its k th-smallest value. The example functions of ordered statistics are as follows.

```
Stats.min;; (* the minimum of the samples *)
Stats.max;; (* the maximum of the samples *)
Stats.median;; (* the median value of the samples *)
Stats.quartile;; (* quartile of the samples *)
Stats.first_quartile;; (* the first quartile of the samples *)
Stats.third_quartile;; (* the third quartile of the samples *)
Stats.interquartile;; (* the interquartile of the samples *)
Stats.percentile;; (* percentile of the samples *)
```

In addition to the aforementioned ones, there are many other ordered statistical functions in Owl for you to explore.

IMAGE to show the difference of mean, median, mode, etc.

Special Distribution

illustrate the naming convention.

- gaussian_rvs : random number generator.
- gaussian_pdf : probability density function.
- gaussian_cdf : cumulative distribution function.
- gaussian_ppf : percent point function (inverse of CDF).
- gaussian_sf : survival function ($1 - \text{CDF}$).
- gaussian_isf : inverse survival function (inverse of SF).
- gaussian_logpdf : logarithmic probability density function.
- gaussian_logcdf : logarithmic cumulative distribution function.
- gaussian_logsf : logarithmic survival function.

Stats module supports many distributions. For each distribution, there is a set of related functions using the distribution name as their common prefix.

TODO: Add Poisson Distribution Implementation

Gamma Distribution

Definition

PDF, CDF

Application

Beta Distribution**Chi-Square Distribution****Student-t Distribution****Cauchy Distribution****Multiple Variables**

Joint Density

Independence of random variables

Mean and Variance

Multinomial distribution

Sampling

So far we have talked about the whole population, now we turn to a sample of it.

Sample median/variance

Infer population parameters from sample

Z test, t test, chi-square test

Hypothesis Tests**Theory**

While descriptive statistics solely concern properties of the observed data, statistical inference focusses on studying whether the data set is sampled from a larger population. In other words, statistical inference make propositions about a population. Hypothesis test is an important method in inferential statistical analysis. There are two hypotheses proposed with regard to the statistical relationship between data sets.

- Null hypothesis H_0 : there is no relationship between two data sets.
- Alternative hypothesis H_1 : there is statistically significant relationship between two data sets.

Type I and Type II errors: the 2x2 matrix.

Gaussian Distribution in Hypothesis Testing

The stats module in Owl supports many different kinds of hypothesis tests.

- Z-Test
- Student's T-Test
- Paired Sample T-Test
- Unpaired Sample T-Test

- Kolmogorov-Smirnov Test
- Chi-Square Variance Test
- Jarque-Bera Test
- Fisher's Exact Test
- Wald-Wolfowitz Runs Test
- Mann-Whitney Rank Test
- Wilcoxon Signed-rank Test

Now let's see how to perform a z-test in Owl. We first generate two data sets, both are drawn from Gaussian distribution but with different parameterisation. The first one `data_0` is drawn from $\mathcal{N}(0, 1)$, while the second one `data_1` is drawn from $\mathcal{N}(3, 1)$.

```
let data_0 = Array.init 10 (fun _ -> Stats.gaussian_rvs ~mu:0. ~sigma:1.);;
let data_1 = Array.init 10 (fun _ -> Stats.gaussian_rvs ~mu:3. ~sigma:1.);;
```

Our hypothesis is that the data set is drawn from Gaussian distribution $\mathcal{N}(0, 1)$. From the way we generated the synthetic data, it is obvious that `data_0` will pass the test, but let's see what Owl will tell us using its `Stats.z_test` function.

```
# Stats.z_test ~mu:0. ~sigma:1. data_0
- : Owl_stats.hypothesis =
{Owl.Stats.reject = false; p_value = 0.289340080583773251;
 score = -1.05957041132113083}
```

The returned result is a record with the following type definition. The fields are self-explained: `reject` field tells whether the null hypothesis is rejected, along with the `p_value` and `score` calculated with the given data set.

```
type hypothesis = {
  reject : bool;
  p_value : float;
  score : float;
}
```

From the previous result, we can see `reject = false`, indicating null hypothesis is rejected, therefore the data set `data_0` is drawn from $\mathcal{N}(0, 1)$. How about the second data set then?

```
# Stats.z_test ~mu:0. ~sigma:1. data_1
- : Owl_stats.hypothesis =
{Owl.Stats.reject = true; p_value = 5.06534675819424548e-23;
 score = 9.88035435799393547}
```

As we expected, the null hypothesis is accepted with a very small p value. This indicates that `data_1` is drawn from a different distribution rather than assumed $\mathcal{N}(0, 1)$.

Two-Sample Inferences

Goodness-of-fit Tests

Non-parametric Statistics

Wilcoxon Tests

Covariance and Correlations

Correlation studies how strongly two variables are related. There are different ways of calculating correlation. For the first example, let's look at Pearson correlation.

`x` is our explanatory variable and we draw 50 random values uniformly from an interval between 0 and 10. Both `y` and `z` are response variables with a linear relation to `x`. The only difference is that we add different level of noise to the response variables. The noise values are generated from Gaussian distribution.

```
let noise sigma = Stats.gaussian_rvs ~mu:0. ~sigma;;
let x = Array.init 50 (fun _ -> Stats.uniform_rvs 0. 10.);;
let y = Array.map (fun a -> 2.5 *. a ++ noise 1.) x;;
let z = Array.map (fun a -> 2.5 *. a ++ noise 8.) x;;
```

It is easier to see the relation between two variables from a figure. Herein we use Owl's `Plplot` module to make two scatter plots.

```
(* convert arrays to matrices *)
let x' = Mat.of_array x 1 50;;
let y' = Mat.of_array y 1 50;;
let z' = Mat.of_array z 1 50;;

(* plot the figures *)

let h = Plot.create ~m:1 ~n:2 "plot_01.png" in

Plot.subplot h 0 0;
Plot.set_xlabel h "x";
Plot.set_ylabel h "y (sigma = 1)";
Plot.scatter ~h x' y';

Plot.subplot h 0 1;
Plot.set_xlabel h "x";
Plot.set_ylabel h "z (sigma = 8)";
Plot.scatter ~h x' z';
```

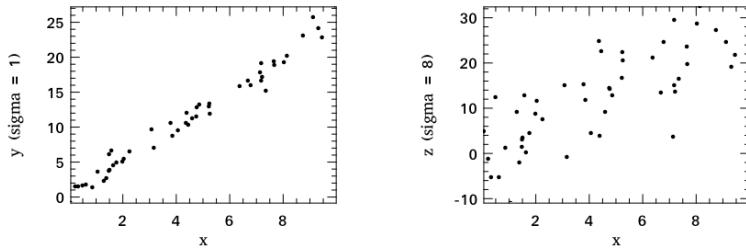


Figure 38: Functional relation between x and the other two variables.

```
Plot.output h;;
```

The subfigure 1 shows the functional relation between x and y whilst the subfigure 2 shows the relation between x and z . Because we have added higher-level noise to z , the points in the second figure are more diffused.

Intuitively, we can easily see there is stronger relation between x and y from the figures. But how about numerically? In many cases, numbers are preferred because they are easier to compare with by a computer. The following snippet calculates the Pearson correlation between x and y , as well as the correlation between x and z . As we see, the smaller correlation value indicates weaker linear relation between x and z comparing to that between x and y .

```
# Stats.corrcoef x y
- : float = 0.991145445979576656
# Stats.corrcoef x z
- : float = 0.692163016204755288
```

Analysis of Variance

So far we have talked about compare two variables. how about comparing more? one by one solution increases error.

Introduce the simplest of ANOVA...

Summary

N-Dimensional Arrays

N-dimensional array (a.k.a ndarray) is the building block of Owl library. Ndarray to Owl is like NumPy to SciPy. It serves as the core dense data structure and many advanced numerical functions are built atop of it. For example, `Algodiff`, `Optimise`, `Neural`, and `Lazy`... all these functors take Ndarray module as the module input.

Due to its importance, Owl has implemented a comprehensive set of operations on Ndarray, all of which are defined in the file `owl_dense_ndarray_generic.mli`. Many of these functions (especially the critical ones) in Owl's core library have corresponding C-stub code to guarantee the best performance. If you take a look at the Ndarray's `mli` file, you probably can see hundreds of them. But do not get scared by the number, since many of them are similar and can be grouped together. In this chapter, we will explain these functions in details regarding these several groups.

Ndarray Types

The very first thing to understand is the types used in Ndarray. Owl's Ndarray module is built directly on top of OCaml's native `Bigarray`. More specifically, it is `Bigarray.Genarray`. Ndarray has the same type as that of `Genarray`, therefore exchanging data between Owl and other libraries relying on `Bigarray` is trivial.

OCaml's `Bigarray` uses kind GADT to specify the number type, precision, and memory layout. Owl only keeps the first two but fixes the last one because Owl only uses `C-layout`, or `Row-based layout` in its implementation. The same design decisions can also be seen in ONNX. See the type definition in Ndarray module.

```
| type ('a, 'b) t = ('a, 'b, c_layout) Genarray.t
```

Technically, `c-layout` indicates the memory address is continuous at the highest dimensions, comparing to the `Fortran-layout` whose continuous memory address is at the lowest dimensions. The reasons why we made this decision are as follows.

- Mixing two layouts together opens a can of worms and is the source of bugs. Especially, indexing in FORTRAN starts from 1 whereas indexing in C starts from 0. Many native OCaml data structures such as `Array` and `List` all start indexing from 0, so using `c-layout` avoids many potential troubles in using the library.
- Supporting both layouts adds a significant amount of complexity in implementing underlying Ndarray functions. Due to the difference in memory layout, code performs well on one layout may not do well on another. Many functions may require different implementations given different layout. This will add too much complexity and increase the code base significantly with marginal benefits.

- Owl has rather different design principles comparing to OCaml's Bigarray. The Bigarray serves as a basic tool to operate on a chunk of memory living outside OCaml's heap, facilitating exchanging data between different libraries (including FORTRAN ones). Owl focuses on providing high-level numerical functions allowing programmers to write concise analytical code. The simple design and small code base outweighs the benefits of supporting both layouts.

Because of Bigarray's mechanism, Owl's Ndarray is also subject to maximum 16 dimensions limits. Moreover, matrix is just a special case of n-dimensional array, and in fact many functions in the `Matrix` module simply calls the same functions in Ndarray. But the module does provide more matrix-specific functions such as iterating rows or columns, and etc.

Creation Functions

The first group of functions we would like to introduce is the ndarray creation functions. They generate dense data structures for you to work on further. The most frequently used ones are probably these four:

```
open Owl.Dense.Ndarray.Generic

val empty : ('a, 'b) kind -> int array -> ('a, 'b) t
val create : ('a, 'b) kind -> int array -> 'a -> ('a, 'b) t
val zeros : ('a, 'b) kind -> int array -> ('a, 'b) t
val ones : ('a, 'b) kind -> int array -> ('a, 'b) t
```

These functions return ndarrays of specified shape, number type, and precision. The `empty` function is different from the other three. It does not really allocate any memory until you access it. Therefore, calling `empty` function is very fast. The other three functions are self-explained. The `zeros` and `ones` fill the allocated memory with zeros and one respectively, whereas `create` function fills the memory with the specified value.

If you need random numbers, you can use another three creation functions that return an ndarray where the elements follow certain distributions.

```
open Owl.Dense.Ndarray.Generic

val uniform : ('a, 'b) kind -> ?a:'a -> ?b:'a -> int array -> ('a, 'b) t
val gaussian : ('a, 'b) kind -> ?mu:'a -> ?sigma:'a -> int array -> ('a, 'b) t
val bernoulli : ('a, 'b) kind -> ?p:float -> int array -> ('a, 'b) t
```

Sometimes, we want to generate numbers with equal distance between two consecutive elements. These ndarrays are useful in generating intervals and plotting figures.

```
open Owl.Dense.Ndarray.Generic

val sequential : ('a, 'b) kind -> ?a:'a -> ?step:'a -> int array -> ('a, 'b) t
val linspace : ('a, 'b) kind -> 'a -> 'a -> int -> ('a, 'b) t
val logspace : ('a, 'b) kind -> ?base:float -> 'a -> 'a -> int -> ('a, 'b) t
```

If these functions cannot satisfy your need, `Ndarray` provides a more flexible mechanism allowing you to have more control over the initialisation of an ndarray.

```
open Owl.Dense.Ndarray.Generic

val init : ('a, 'b) kind -> int array -> (int -> 'a) -> ('a, 'b) t
val init_nd : ('a, 'b) kind -> int array -> (int array -> 'a) -> ('a, 'b) t
```

The difference between the two group is: `init` passes 1-d indices to the user-defined function, whereas `init_nd` passes n-dimensional indices. As a result, `init` is much faster than `init_nd`. As an example, the following code creates an ndarray where all the elements are even numbers.

```
# let x = Arr.init [|6;8|] (fun i -> 2. *. (float_of_int i))
val x : Arr.arr =
  C0 C1 C2 C3 C4 C5 C6 C7
  R0 0 2 4 6 8 10 12 14
  R1 16 18 20 22 24 26 28 30
  R2 32 34 36 38 40 42 44 46
  R3 48 50 52 54 56 58 60 62
  R4 64 66 68 70 72 74 76 78
  R5 80 82 84 86 88 90 92 94
```

Properties Functions

After an ndarray is created, you can use various functions in the module to obtain its properties. For example, the following functions are commonly used ones.

```
open Owl.Dense.Ndarray.Generic

val shape : ('a, 'b) t -> int array
```

```
(** [shape x] returns the shape of ndarray [x]. *)
val num_dims : ('a, 'b) t -> int
(** [num_dims x] returns the number of dimensions of ndarray [x]. *)

val nth_dim : ('a, 'b) t -> int -> int
(** [nth_dim x] returns the size of the nth dimension of [x]. *)

val numel : ('a, 'b) t -> int
(** [numel x] returns the number of elements in [x]. *)

val nnz : ('a, 'b) t -> int
(** [nnz x] returns the number of non-zero elements in [x]. *)

val density : ('a, 'b) t -> float
(** [density x] returns the percentage of non-zero elements in [x]. *)

val size_in_bytes : ('a, 'b) t -> int
(** [size_in_bytes x] returns the size of [x] in bytes in memory. *)

val same_shape : ('a, 'b) t -> ('a, 'b) t -> bool
(** [same_shape x y] checks whether [x] and [y] has the same shape or not. *)
```

val kind : ('a, 'b) t -> ('a, 'b) kind
 (** [kind x] returns the type of ndarray [x]. *)
 ``Property functions are easy to understand.

Note that `nnz` and `density` need to traverse through all the elements in an ndarray, but because the implementation is in C so even for a very large ndarray the performance is still good.

In the following, we focus on three typical operations on n-dimensional array worth your special attention : the `map`, `fold`, and `scan`.

Map Functions

The `map` function transforms one ndarray to another according to a given function, which is often done by applying the transformation function to every element in the original ndarray.

The `map` function in Owl is pure and always generates a fresh new data structure rather than modifying the original one.

For example, the following code creates a three-dimensional ndarray, and then adds 1 to every element in `x`.

```
``ocaml
# let x = Arr.uniform [|3;4;5|]
val x : Arr.arr =
  C0 C1 C2 C3 C4
R[0,0] 0.378545 0.861025 0.712662 0.563556 0.964339
R[0,1] 0.582878 0.834786 0.722758 0.265025 0.712912
R[0,2] 0.0894476 0.13984 0.475555 0.616536 0.202631
R[0,3] 0.983487 0.0167333 0.25018 0.483741 0.736418
R[1,0] 0.0757294 0.662478 0.460645 0.203446 0.725446
  ...
R[1,3] 0.83694 0.897979 0.912516 0.833211 0.4145
R[2,0] 0.903692 0.883623 0.809134 0.859235 0.188514
R[2,1] 0.236758 0.566636 0.613932 0.215875 0.00911335
```

```
R[2,2] 0.859797 0.708086 0.518328 0.974299 0.472426
R[2,3] 0.126273 0.946126 0.42223 0.955181 0.422184
```

```
# let y = Arr.map (fun a -> a +. 1.) x
val y : Arr.arr =
  C0 C1 C2 C3 C4
  R[0,0] 1.37854 1.86103 1.71266 1.56356 1.96434
  R[0,1] 1.58288 1.83479 1.72276 1.26503 1.71291
  R[0,2] 1.08945 1.13984 1.47556 1.61654 1.20263
  R[0,3] 1.98349 1.01673 1.25018 1.48374 1.73642
  R[1,0] 1.07573 1.66248 1.46065 1.20345 1.72545
  ...
  R[1,3] 1.83694 1.89798 1.91252 1.83321 1.4145
  R[2,0] 1.90369 1.88362 1.80913 1.85923 1.18851
  R[2,1] 1.23676 1.56664 1.61393 1.21588 1.00911
  R[2,2] 1.8598 1.70809 1.51833 1.9743 1.47243
  R[2,3] 1.12627 1.94613 1.42223 1.95518 1.42218
```

The `map` function can be very useful in implementing vectorised math functions. Many functions in `Ndarray` can be categorised into this group, such as `sin`, `cos`, `neg`, and etc. Here are some examples to show how to make your own vectorised functions.

```
let vec_sin x = Arr.map sin x;;
let vec_cos x = Arr.map cos x;;
let vec_log x = Arr.map log x;;
```

If you need indices in the transformation function, you can use the `mapi` function which accepts the 1-d index of the element being accessed.

```
val mapi : (int -> 'a -> 'a) -> ('a, 'b) t -> ('a, 'b) t
```

Fold Functions

The `fold` function is often referred to as “reduction” in other programming languages. It has a named parameter called `axis`, with which you can specify along what axis you want to fold a given `ndarray`.

```
val fold : ?axis:int -> ('a -> 'a -> 'a) -> 'a -> ('a, 'b) t -> ('a, 'b) t
```

The `axis` parameter is optional. If you do not specify one, the `ndarray` will be flattened first folding happens along the zero dimension. In other words, all the elements will be folded into a one-element one-dimensional `ndarray`. The

`fold` function in Ndarray is actually folding from left, and you can also specify an initial value of the folding. The code below demonstrates how to implement your own `sum'` function.

```
| let sum' ?axis x = Arr.fold ?axis ( +. ) 0. x;;
```

The functions `sum`, `sum'`, `prod`, `prod'`, `min`, `min'`, `mean`, and `mean'` all belong to this group. The difference between the functions with and without prime ending is that the former one returns an ndarray, while the latter one returns a number.

Similarly, if you need indices in folding function, you can use `foldi` which passes in 1-d indices.

```
| val foldi : ?axis:int -> (int -> 'a -> 'a -> 'a) -> 'a -> ('a, 'b) t -> ('a, 'b) t
```

Scan Functions

To some extent, the `scan` function is like the combination of `map` and `fold`. It accumulates the value along the specified axis but it does not change the shape of the input. Think about how we generate a cumulative distribution function (CDF) from a probability density/mass function (PDF/PMF). The type signature of `scan` looks like this in Ndarray.

```
| val scan : ?axis:int -> ('a -> 'a -> 'a) -> ('a, 'b) t -> ('a, 'b) t
```

Several functions belong to this group, such as `cumsum`, `cumprod`, `cummin`, `cummax`, and etc. To implement one `cumsum` for yourself, you can write in the following way.

```
| let cumsum ?axis x = Arr.scan ?axis ( +. ) x;;
```

Again, you can use the `scani` to obtain the indices in the passed in cumulative functions.

Comparison Functions

The comparison functions themselves can be divided into several groups. The first group compares two ndarrays then returns a boolean value.

```
| val equal : ('a, 'b) t -> ('a, 'b) t -> bool
| val not_equal : ('a, 'b) t -> ('a, 'b) t -> bool
| val less : ('a, 'b) t -> ('a, 'b) t -> bool
```

```
val greater : ('a, 'b) t -> ('a, 'b) t -> bool
...
...
```

The second group compares two ndarrays but returns an 0-1 ndarray of the same shape. The elements where the predicate is satisfied have value 1 otherwise 0.

```
val elt_equal : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
val elt_not_equal : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
val elt_less : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
val elt_greater : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
...
...
```

The third group is similar to the first one but compares an ndarray with a scalar value, the return is a Boolean value.

```
val equal_scalar : ('a, 'b) t -> 'a -> bool
val not_equal_scalar : ('a, 'b) t -> 'a -> bool
val less_scalar : ('a, 'b) t -> 'a -> bool
val greater_scalar : ('a, 'b) t -> 'a -> bool
...
...
```

The fourth group is similar to the second one but compares an ndarray with a scalar value, and the returned value is a 0-1 ndarray.

```
val elt_equal_scalar : ('a, 'b) t -> 'a -> ('a, 'b) t
val elt_not_equal_scalar : ('a, 'b) t -> 'a -> ('a, 'b) t
val elt_less_scalar : ('a, 'b) t -> 'a -> ('a, 'b) t
val elt_greater_scalar : ('a, 'b) t -> 'a -> ('a, 'b) t
...
...
```

You probably have noticed the pattern in naming these functions. In general, we recommend using operators rather than calling these function name directly, since it leads to more concise code. Please refer to the chapter about Conventions.

The comparison functions can do a lot of useful things for us. As an example, the following code shows how to keep the elements greater than 0.5 as they are but set the rest to zeros in an ndarray.

```
let x = Arr.uniform [|10; 10|];;

(* the first solution *)
let y = Arr.map (fun a -> if a > 0.5 then a else 0.) x;;

(* the second solution *)
let z = Arr.((x >.$ 0.5) * x);;
```

As you can see, comparison function combined with operators can lead to more concise code. Moreover, it sometimes outperforms the first solution at the price of higher memory consumption, because the loop is done in C rather than in OCaml.

Vectorised Functions

Many common operations on ndarrays can be decomposed as a series of `map`, `fold`, and `scan` operations. There is even a specific programming paradigm built atop of this which is called **Map-Reduce**. It was hyped several years ago in many data processing frameworks. Nowadays, map-reduce is one dominant data-parallel processing paradigm.

The ndarray module has included a very comprehensive set of mathematical functions and all have been vectorised. This means you can apply them directly on an ndarray and the function will be automatically applied to every element in the ndarray.

For binary math operators, there are `add`, `sub`, `mul`, and etc. For unary operators, there are `sin`, `cos`, `abs`, and etc. You can obtain the complete list of functions in `owl_dense_ndarray_generic.mli`.

Conceptually, Owl can implement all these functions using the aforementioned `map`, `fold`, and `scan`. In reality, these vectorised math is done in C code to guarantee the best performance. Accessing the elements in a bigarray is way faster in C than in OCaml.

Iteration Functions

Like native OCaml array, Owl also provides `iter` and `iteri` functions with which you can iterate over all the elements in an ndarray.

```
val iteri :(int -> 'a -> unit) -> ('a, 'b) t -> unit

val iter : ('a -> unit) -> ('a, 'b) t -> unit
```

One common use case is iterating all the elements and checking if one (or several) predicate is satisfied. There is a special set of iteration functions to help you finish this task.

```
val is_zero : ('a, 'b) t -> bool
val is_positive : ('a, 'b) t -> bool
val is_negative : ('a, 'b) t -> bool
val is_nonpositive : ('a, 'b) t -> bool
val is_nonnegative : ('a, 'b) t -> bool
val is_normal : ('a, 'b) t -> bool
```

The predicates can be very complicated sometimes. In that case you can use the following three functions to pass in arbitrarily complicated functions to check them.

```
val exists : ('a -> bool) -> ('a, 'b) t -> bool
val not_exists : ('a -> bool) -> ('a, 'b) t -> bool
val for_all : ('a -> bool) -> ('a, 'b) t -> bool
```

All aforementioned functions only tell us whether the predicates are met or not. They cannot tell which elements satisfy the predicate. The following `filter` function can return the 1-d indices of those elements satisfying the predicates.

```
val filteri : (int -> 'a -> bool) -> ('a, 'b) t -> int array
val filter : ('a -> bool) -> ('a, 'b) t -> int array
```

We have mentioned that 1-d indices are passed in. The reason is passing in 1-d indices is way faster than passing in n-d indices. However, if you do need n-dimensional indices, you can use the following two functions to convert between 1-d and 2-d indices, both are defined in the `owl.Utils` module.

```
val ind : ('a, 'b) t -> int -> int array
(* 1-d to n-d index conversion *)
val i1d : ('a, 'b) t -> int array -> int
(* n-d to 1-d index conversion *)
```

Note that you need to pass in the original ndarray because the shape information is required for calculating index conversion.

Manipulation Functions

Ndarray module contains many useful functions to manipulate ndarrays. For example, you can tile and repeat an ndarray along a specified axis. Let's first create a sequential ndarray.

```
# let x = Arr.sequential [|3;4|]
val x : Arr.arr =
  C0 C1 C2 C3
  R0 0 1 2 3
  R1 4 5 6 7
  R2 8 9 10 11
```

The code below tiles `x` once on both dimensions.

```
# let y = Arr.tile x [|2;2|]
val y : Arr.arr =
  C0 C1 C2 C3 C4 C5 C6 C7
  R0 0 1 2 3 0 1 2 3
  R1 4 5 6 7 4 5 6 7
  R2 8 9 10 11 8 9 10 11
  R3 0 1 2 3 0 1 2 3
  R4 4 5 6 7 4 5 6 7
  R5 8 9 10 11 8 9 10 11
```

Comparing to `tile`, the `repeat` function replicates each element in their adjacent places along specified dimension.

```
# let z = Arr.repeat x [|2;1|]
val z : Arr.arr =
  C0 C1 C2 C3
  R0 0 1 2 3
  R1 0 1 2 3
  R2 4 5 6 7
  R3 4 5 6 7
  R4 8 9 10 11
  R5 8 9 10 11
```

You can also expand the dimensionality of an ndarray, or squeeze out those dimensions having only one element, or even padding elements to an existing ndarray.

```
val expand : ('a, 'b) t -> int -> ('a, 'b) t
val squeeze : ?axis:int array -> ('a, 'b) t -> ('a, 'b) t
val pad : ?v:'a -> int list list -> ('a, 'b) t -> ('a, 'b) t
```

Another two useful functions are `concatenate` and `split`. The `concatenate` allows us to concatenate an array of ndarrays along the specified axis. The constraint on the shapes is that, except for the dimension of concatenation, the rest dimensions must be equal. For matrices, there are two operators associated with concatenation: `@||` for concatenating horizontally (i.e. along axis 1); `@=` for concatenating vertically (i.e. along axis 0). The `split` is simply the inverse operation of concatenation.

```
val concatenate : ?axis:int -> ('a, 'b) t array -> ('a, 'b) t
val split : ?axis:int -> int array -> ('a, 'b) t -> ('a, 'b) t array
```

You can also sort an ndarray but note that modification will happen in place.

```
val sort : ('a, 'b) t -> unit
```

Converting between ndarrays and OCaml native arrays can be efficiently done with these conversion functions:

```
val of_array : ('a, 'b) kind -> 'a array -> int array -> ('a, 'b) t
val to_array : ('a, 'b) t -> 'a array
```

Again, there also exist the `to_arrays` and `of_arrays` two functions for the special case of matrix module.

Serialisation

Serialisation and de-serialisation are simply done with the `save` and `load` functions.

```
val save : out:string -> ('a, 'b) t -> unit
val load : ('a, 'b) kind -> string -> ('a, 'b) t
```

Note that you need to pass in type information in the `load` function, otherwise Owl cannot figure out what is contained in the chunk of binary file. Alternatively, you can use the corresponding `load` functions in the `s/d/c/z` modules to save the type information.

```
# let x = Mat.uniform 8 8 in
Mat.save "data.mat" x;
let y = Mat.load "data.mat" in
Mat.(x = y)
- : bool = true
```

The `save` and `load` currently use the `Marshall` module which is brittle since it depends on specific OCaml versions. In the future, these two functions will be improved.

With the help of `npy-ocaml`, we can save and load files in the format of `npy` file. Proposed by NumPy, NPY is a standard binary file format for persisting a single arbitrary ndarray on disk. The format stores all of the shape and data type information necessary to reconstruct the array correctly even on another machine with a different architecture. NPY is a widely used serialisation format. Owl can thus easily interact with the Python-world data by using this format.

Using NPY files are the same as that of normal serialisation methods. Here is a simple example:

```
# let x = Arr.uniform [|3; 3|] in
  Arr.save_npy ~out:"data.npy" x;
let y = Arr.load_npy "data.npy" in
  Arr.(x = y)
- : bool = true
```

There are way more functions contained in the `Ndarray` module than the ones we have introduced here. Please refer to the API documentation for the full list.

Tensors

In the last part of this chapter, we will briefly introduce the idea of *tensor*. If you look at some articles online the tensor is often defined as an n-dimensional array. However, mathematically, there are differences between these two. In a n-dimension space, a tensor that contains m indices is a mathematical object that obeys certain transformation rules. For example, in a three dimension space, we have a value $A = [0, 1, 2]$ that indicate a vector in this space. We can find each element in this vector by a single index i , e.g. $A_1 = 1$. This vector is an object in this space, and it stays the same even if we change the standard cartesian coordinate system into other systems. But if we do so, then the content in A needs to be updated accordingly. Therefore we say that, a tensor can normally be expressed in the form of an ndarray, but it is not an ndarray. That's why we keep using the term "ndarray" in this chapter and through out the book.

The basic idea about tensor is that, since the object stays the same, if we change the coordinate towards one direction, the component of the vector needs to be changed to another direction. Considering a single vector v in a coordinate system with basis e . We can change the coordinate base to \tilde{e} with linear transformation: $\tilde{e} = Ae$ where A is a matrix. For any vector in this space using e as base, its content will be transformed as: $\tilde{v} = A^{-1}v$, or we can write it as:

$$\tilde{v}^i = \sum_j B_j^i v^j.$$

Here $B = A^{-1}$. We call a vector *contravector* because it changes in the opposite way to the basis. Note we use the superscript to denote the element in contravectors.

As a comparison, think about a matrix multiplication αv . The α itself forms a different vector space, the basis of which is related to the basis of v 's vector space. It turns out that the direction of change of α is the same as that of e . When v uses new $\tilde{e} = Ae$, its component changes in the same way:

$$\tilde{\alpha}_j = \sum_i A_j^i \alpha_i.$$

It is called a *covector*, denoted with subscript. We can further extend it to matrix. Think about a linear mapping L . It can be represented as a matrix so that we can apply it to any vector using matrix dot multiplication. With the change of the coordinate system, it can be proved that the content of the linear map L itself is updated to:

$$\tilde{L}_j^i = \sum_{kl} B_k^i L_l^k A_j^l.$$

Again, note we use both superscript and subscript for the linear map L , since it contains one covariant component and one contravariant component. Furthermore, we can extend this process and define the tensor. A tensor T is an object that is invariant under a change of coordinates, and with a change of coordinates its component changes in a special way. The way is that:

$$T_{xyz\dots}^{abc\dots} = \sum_{ijk\dots rst\dots} B_i^a B_j^b B_k^c \dots T_{rst\dots}^{ijk\dots} A_x^r A_y^s A_z^t \dots \quad (3)$$

Here the $ijk\dots$ are indices of the contravariant part of the tensor and the $rst\dots$ are that of the covariant part.

One of the important operations of tensor is the *tensor contraction*. We are familiar with the matrix multiplication:

$$C_j^i = \sum_k A_k^i B_j^k. \quad (4)$$

The *contraction* operations extends this process to multiple dimension space. It sums the products of the two ndarrays' elements over specified axes. For example, we can perform the matrix multiplication with contraction:

```
let x = Mat.uniform 3 4
let y = Mat.uniform 4 5

let z1 = Mat.dot x y
let z2 = Arr.contract2 [(1,0)] x y
```

We can see that the matrix multiplication is a special case of contraction operation and can be implemented with it.

Next, let's extend the two dimension case to multiple dimensions. Let's say we have two three-dimensional array A and B. We hope to compute the matrix C so that:

$$C_j^i = \sum_{hk} A_{hk}^i B_j^{kh} \quad (5)$$

We can use the `contract2` function in the `Ndarray` module. It takes an array of `int * int` tuples to specifies the pair of indices in the two input ndarrays. Here is the code:

```
let x = Arr.sequential [|3;4;5|]
let y = Arr.sequential [|4;3;2|]

let z1 = Arr.contract2 [(0, 1); (1, 0)] x y
```

The indices mean that, in the contraction, the 0th dimension of `x` corresponds with the 1st dimension of `y`, and the 1st dimension of `x` corresponds with the 0th dimension of `y`, as shown in eq. 5. We can verify the result with the naive way of implementation:

```
let z2 = Arr.zeros [|5;2|]

let _ =
  for h = 0 to 2 do
    for k = 0 to 3 do
      for i = 0 to 4 do
        for j = 0 to 1 do
          let r = (Arr.get x [|h;k;i|]) *. (Arr.get y [|k;h;j|]) in
            Arr.set z2 [|i;j|] ((Arr.get z2 [|i;j|]) +. r)
          done
        done
      done
    done
```

Then we can check if the two results agree:

```
# Arr.equal z1 z2
- : bool = true
```

The contraction can also be applied on one single ndarray to perform the reduction operation using the `contract1` function.

```
# let x = Arr.sequential [|2;2;3|]
val x : Arr.arr =
```

	C0	C1	C2
R[0,0]	0	1	2
R[0,1]	3	4	5
R[1,0]	6	7	8
R[1,1]	9	10	11

```
# let y = Arr.contract1 [||(0,1)|] x
val y : Arr.arr =
  C0 C1 C2
  R 9 11 13
```

We can surely perform the matrix multiplication with contraction. High-performance implementation of the contraction operation has been a research topic. Actually, many tensor operations involve summation over particular indices. Therefore in using tensors in applications such as linear algebra and physics, the *Einstein notation* is used to simplified notations. It removes the common summation notation, and also, any twice-repeated index in a term is summed up (no index is allowed to occur three times or more in a term). For example, the matrix multiplication notation $C_{ij} = \sum_k A_{ik}B_{kj}$ can be simplified as $C = A_{ik}B_{kj}$. The eq. 3 can also be greatly simplified in this way.

The tensor calculus is of important use in disciplines such as geometry and physics. More details about the tensor calculation is beyond the scope of this book. We refer readers to work such as (Dullemond and Peeters 1991) for deeper understanding about this topic.

Summary

N-dimensional array is the fundamental data type in Owl, as well as in many other numerical libraries such as NumPy. This chapter explain in detail the Ndarray module, including its creation, properties, manipulation, serialisation, etc. Besides, we also discuss the subtle difference between tensor and ndarray in this chapter. This chapter is easy to follow, and can serve as a reference whenever users need a quick check of functions they need.

References

Dullemond, Kees, and Kasper Peeters. 1991. “Introduction to Tensor Calculus.” *Kees Dullemond and Kasper Peeters*, 42–44.

Slicing and Broadcasting

Indexing, slicing, and broadcasting are three fundamental functions to manipulate multidimensional arrays. They are used in practically every numerical application. Therefore understanding their nuts and bolts is very important. In this chapter we will introduce how to use these functions in Owl.

Slicing

Indexing and slicing is arguably the most important function in any numerical library. A flexible design is able to significantly simplify the code and allow us to write concise algorithms. Before we start, let's clarify some things:

- slicing refers to the operation that extracts part of the data from an ndarray or a matrix according to a well-defined *slice definition*;
- slicing can be applied to all dense data structures, i.e. both ndarrays and matrices;
- slice definition is an `index` list which clarifies *what indices* should be accessed and in *what order* for each dimension of the passed in variable;
- there are two types of slicing in Owl: *basic slicing* and *fancy slicing*. The difference between the two lies in how the slice is defined.

Basic Slicing

For basic slicing, each dimension in the slice definition must be defined in the format of `[start:stop:step]`. Owl provides two functions `get_slice` and `set_slice` to retrieve and assign slice values respectively.

```
val get_slice : int list list -> ('a, 'b) t -> ('a, 'b) t
val set_slice : int list list -> ('a, 'b) t -> ('a, 'b) t -> unit
```

Both functions accept `int list list` as its slice definition. Every `list` element in the `int list list` is assumed to be a range. For example, `[[]; [2]; [-1;3]]` is equivalent to its full slice definition `[R []; R [2]; R [-1;3]]`, as we will introduce below in the fancy slicing.

Fancy Slicing

Fancy slicing is more powerful than the basic one thanks to its slice definition. With fancy slicing, we can pass in a list of arbitrarily ordered indices which may not be possible to specify with aforementioned simple `[start;stop;step]` format.

```
type index =
| I of int (* single index *)
| L of int list (* list of indices *)
| R of int list (* index range *)
```

As shown above, fancy slice is defined by an `index list` where you can use three type constructors to specify:

- an individual index (using `I` constructor);
- a list of indices (using `L` constructor);
- a range of indices (using `R` constructor).

Similar to the basic slicing, there are two functions to handle fancy slicing operations.

```
val get_fancy : index list -> ('a, 'b) t -> ('a, 'b) t
val set_fancy : index list -> ('a, 'b) t -> ('a, 'b) t -> unit
```

The `get_fancy s x` retrieves a slice of `x` defined by `s`; whereas `set_fancy s x y` assigns the slice of `x` defined by `s` according to values in `y`. Note that `y` must have the same shape as that defined by `s`.

Basic slicing is a special case of fancy slicing where only type constructor `R` is used in the definition. For example, the following two definitions are equivalent.

```
let x = Arr.sequential [|10; 10; 10|];
Arr.get_slice [ []; [0;8]; [3;9;2] ] x;;
Arr.get_fancy [ R[]; R[0;8]; R[3;9;2] ] x;;
```

Note that both `get_basic` and `get_fancy` return a copy rather than a view as that in NumPy; whilst `set_basic` and `set_fancy` modifies the original data in place.

Conventions in Definition

Essentially, Owl's slicing functions are very similar to those in NumPy. So if you already know how to slice n-dimensional arrays in NumPy, you should find this chapter quite easy to follow.

The core building block is the slice definition. Slice definition is an `index list`. Each element within the `index list` corresponds to one dimension in the passed in data, and it defines how the indices along this dimension should be accessed. Owl provides three constructors `I`, `L`, and `R` to let you specify single index, a list of indices, or a range of indices. Constructor `I` is trivial, it specifies an index. E.g., `[I 2; I 5]` returns the element at position $(2, 5)$ in a matrix. Constructor `L` is used to specify a list of indices. E.g., `[I 2; L [5;3]]` returns a 1×2 matrix consists of the elements at $(2, 5)$ and $(2, 3)$ in the original matrix.

Constructor `R` is for specifying a range of indices. It has more conventions but by no means complicated. The following text is dedicated for range conventions. These conventions or rules require our attentions in order to write correct slice definition. These conventions can be equally applied to both basic and fancy slicing.

Rule #1: The format of the range definition follows `R [start; stop; step]`. Obviously, `start` specifies the starting index; `stop` specifies the stopping index (inclusive); and `step` specifies the step size. You do not have to specify all three variables in the definition; please see the following rules.

Rule #2: All three variables `start`, `stop`, and `step` can take both positive and negative values, but `step` is not allowed to take `0` value. Positive step indicates that indices will be visited in increasing order from `start` to `stop`; and vice versa.

Rule #3: For `start` and `stop` variables, positive value refers to a specific index; whereas negative value `a` will be translated into `n + a` where `n` is the total number of indices. E.g., `[-1; 0]` means from the last index to the first one.

Rule #4: If you pass in an empty list `R []`, this will be expanded into `[0; n - 1; 1]` which means all the indices will be visited in increasing order with step size 1.

Rule #5: If you only specify one variable such as `[start]`, `get_slice` function assumes that you will take one specific index by automatically extending it into `[start; start; 1]`. As we can see, `start` and `stop` are the same, with step size 1.

Rule #6: If you only specify two variables then `slice` function assumes they are `[start; stop]` which defines the range of indices. However, how `get_slice` will expand this slice definition depends. As we can see below, `slice` will visit the indices in different orders:

- if `start <= stop`, it will be expanded to `[start; stop; 1]`;
- if `start > stop`, it will be expanded to `[start; stop; -1]`;

Rule #7: It is not necessary to specify all the definitions for all the dimensions, `get_slice` function will also expand it by assuming you will take all the data in higher dimensions. E.g., `x` has the shape `[2; 3; 4]`, if we define the slice as `[0]` then `get_slice` will expand the definition into `[[0]; []; []]`

OK, that's all. Please make sure you understand it well before you start, but it is also fine you just learn by doing. Now here are some illustrated examples that can get you started with some of these rules. These examples are based on a 8×8 matrix.

```
| let x = Arr.sequential [|8; 8|]
```

The first example as shown in fig. 39(a) is to take one column of this matrix. It can be achieved by using both basic and fancy slicing:

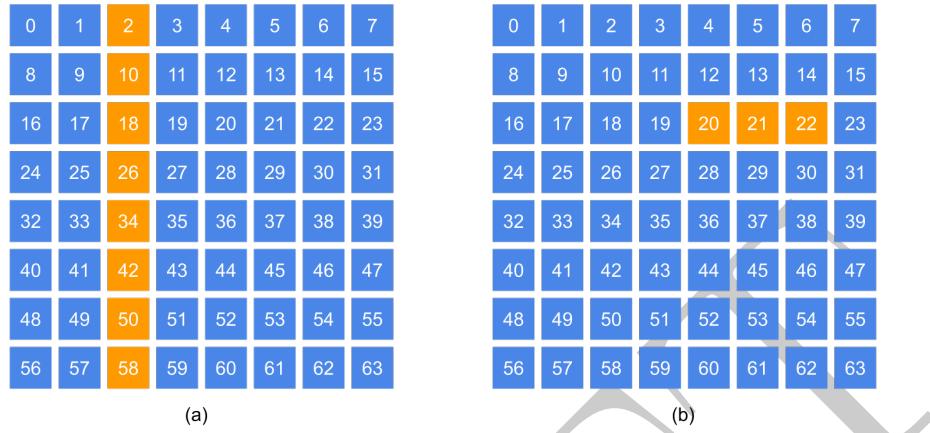


Figure 39: Illustrated Examples of Slicing

```
# Arr.get_fancy [ R[]; I 2 ] x;;
- : Arr.arr =
```

```
C0
R0 2
R1 10
R2 18
R3 26
R4 34
R5 42
R6 50
R7 58
```

```
# Arr.get_slice [ []; [2] ] x;;
- : Arr.arr =
```

```
C0
R0 2
R1 10
R2 18
R3 26
R4 34
R5 42
R6 50
R7 58
```

The second example in in fig. 39(b) is similar, but about retrieving part of a row. Still, this can be gotten using both methods.

```
# Arr.get_fancy [ I 2; R [4; 6] ] x
```

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(a)

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(b)

Figure 40: Illustrated Examples of Slicing (Cont.)

```
- : Arr.arr =
  C0 C1 C2
R0 20 21 22
```

```
# Arr.get_slice [ 2]; [4; 6 ] x
- : Arr.arr =
  C0 C1 C2
R0 20 21 22
```

The next example in fig. 40(a) is a bit more complex. It chooses certain rows, and then choose the columns by a fixed step 2. We can use the fancy slicing in this way:

```
# Arr.get_fancy [ L [3; 5]; R [1; 7; 2] ] x
- : Arr.arr =
  C0 C1 C2 C3
R0 25 27 29 31
R1 41 43 45 47
```

Finally, the last example concerns taking a sub matrix. We can do it in the similar way as the example 1 and 2. Or, since this sub matrix is close to the end of both dimension, we can use the negative integers as indices.

```
# Arr.get_fancy [ L [-2; -1]; R [-3; -2] ] x
- : Arr.arr =
  C0 C1
R0 53 54
R1 61 62
```

Extended Operators

The operators for indexing and slicing are built on the extended indexing operators introduced in OCaml 4.06. All of them are defined in the functors in `owl_operator` module. They are used in Owl as follows.

- `.%{ }` : get
- `.%{ }<-` : set
- `.${ }` : get_slice
- `.${ }<-` : set_slice
- `.!{ }` : get_fancy
- `.!{ }<-` : set_fancy

Here are some examples that show how to use them.

`.%{ }` for indexing:

```
open Arr;;  
  
let x = sequential [|10; 10; 10|];;  
let a = x.%{2; 3; 4};; (* i.e. Arr.get *)  
x.%{2; 3; 4} <- 111.;; (* i.e. Arr.set *)
```

`.${ }` for basic slicing:

```
open Arr;;  
  
let x = sequential [|10; 10; 10|] in  
let a = x.${[0;4]; [6;-1]; [-1;0]} in (* i.e. Arr.get_slice *)  
let b = zeros (shape a) in  
x.${[0;4]; [6;-1]; [-1;0]} <- b;; (* i.e. Arr.set_slice *)
```

`.!{ }` for fancy slicing:

```
open Arr;;  
  
let x = sequential [|10; 10; 10|] in  
let a = x.!{L [2;2;1]; R [6;-1]; I 5} in (* i.e. Arr.get_fancy *)  
let b = zeros (shape a) in  
x.!{L [2;2;1]; R [6;-1]; I 5} <- b;; (* i.e. Arr.set_fancy *)
```

Advanced Usage

There are more advanced usages of slicing besides what we have just seen. We will first use the basic slicing to demonstrate some examples in this section. Note that all the following examples can be equally applied to ndarray.

Let's first define a sequential matrix as the input data for the following examples.

```
# let x = Mat.sequential 5 7;;
val x : Mat.mat =
  C0 C1 C2 C3 C4 C5 C6
R0 0 1 2 3 4 5 6
R1 7 8 9 10 11 12 13
R2 14 15 16 17 18 19 20
R3 21 22 23 24 25 26 27
R4 28 29 30 31 32 33 34
```

Now, we can start our experiment. One benefit of running code in utop is that you can observe the output immediately to understand better how slice function works.

```
let x = Arr.sequential [|10; 10; 10|];;
(* simply take all the elements *)
let s = [ ] in
Mat.get_slice s x;;

(* take row 2 *)
let s = [ [2]; [] ] in
Mat.get_slice s x;;

(* same as above, take row 2, but only specify low dimension slice definition *)
let s = [ [2] ] in
Mat.get_slice s x;;

(* take from row 1 to 3 *)
let s = [ [1;3] ] in
Mat.get_slice s x;;

(* take from row 3 to 1, same as the example above but in reverse order *)
let s = [ [3;1] ] in
Mat.get_slice s x;;
```

Let's see some more complicated examples.

```
(* take from row 1 to 3 and column 3 to 5, so a sub-matrix of x *)
let s = [ [1;3]; [3;5] ] in
Mat.get_slice s x;;

(* take from row 1 to the last row *)
let s = [ [1;-1]; [] ] in
Mat.get_slice s x;;

(* take the rows of even number indices, i.e., 0;2;4 *)
let s = [ [0;-1;2] ] in
Mat.get_slice s x;;

(* take the columns of odd number indices, i.e.,1;3;5 ... *)
let s = [ []; [1;-1;2] ] in
Mat.get_slice s x;;
```

```
(* reverse all the rows of x *)
let s = [ [-1;0] ] in
Mat.get_slice s x;;

(* reverse all the elements of x, same as applying reverse function *)
let s = [ [-1;0]; [-1;0] ] in
Mat.get_slice s x;;

(* take the second last row, from the first column to the last, with step size 3 *)
let s = [ [-2]; [0;-1;3] ] in
Mat.get_slice s x;;
```

The following are some more advanced examples to show how to use slicing to achieve quite complicated operations. Let's use a 5×5 sequential matrix for illustration.

```
# let x = Mat.sequential 5 5
val x : Mat.mat =
  C0 C1 C2 C3 C4
  R0 0 1 2 3 4
  R1 5 6 7 8 9
  R2 10 11 12 13 14
  R3 15 16 17 18 19
  R4 20 21 22 23 24
```

The first function `flip` a matrix upside down, i.e. flip vertically.

```
# let flip x = Mat.get_slice [ [-1; 0]; [ ] ] x in
  flip x
- : Mat.mat =
  C0 C1 C2 C3 C4
  R0 20 21 22 23 24
  R1 15 16 17 18 19
  R2 10 11 12 13 14
  R3 5 6 7 8 9
  R4 0 1 2 3 4
```

The second `reverse` function treats a matrix as one-dimensional vector and reverse its elements. This operation is equivalent to flipping in both vertical and horizontal directions.

```
# let reverse x = Mat.get_slice [ [-1; 0]; [-1; 0] ] x in
  reverse x
- : Mat.mat =
  C0 C1 C2 C3 C4
  R0 24 23 22 21 20
  R1 19 18 17 16 15
```

```
R2 14 13 12 11 10
R3 9 8 7 6 5
R4 4 3 2 1 0
```

The third function rotates a matrix 90 degrees in clockwise direction. As we can see, slicing function leads to very concise code.

```
# let rotate90 x = Mat.(transpose x |> get_slice [ []; [-1;0] ]) in
  rotate90 x
- : Mat.mat =
C0 C1 C2 C3 C4
R0 20 15 10 5 0
R1 21 16 11 6 1
R2 22 17 12 7 2
R3 23 18 13 8 3
R4 24 19 14 9 4
```

The last function `cshift` performs right circular shift along the columns of a matrix.

```
let cshift x n =
let c = Mat.col_num x in
let h = Utils.Array.(range (c - n) (c - 1)) |> Array.to_list in
let t = Utils.Array.(range 0 (c - n - 1)) |> Array.to_list in
Mat.get_fancy [ R []; L (h @ t) ] x
```

Applying to the previous `x`, the outcome should look like this.

```
# cshift x 2
- : Mat.mat =
C0 C1 C2 C3 C4
R0 3 4 0 1 2
R1 8 9 5 6 7
R2 13 14 10 11 12
R3 18 19 15 16 17
R4 23 24 20 21 22
```

Slicing and indexing is an important topic in Owl, so make sure you understand it well before proceeding to other chapters.

Broadcasting

Following indexing and slicing introduced in the previous section, this section introduces the broadcasting operation in Owl. In contrast to indexing and slicing which are explicitly used, broadcasting are often implicitly called when certain conditions are met. This automatic behaviour on one hand is able to simplify the

code, on the other it can also potentially introduce bugs and make the debugging really difficult.

What Is Broadcasting?

There are many binary (mathematical) operators that take two ndarrays as inputs, e.g. add, sub, and etc. In a trivial case, the inputs have exactly the same shape. However, in many real-world applications, we need to operate on two ndarrays whose shapes do not match. How to apply the smaller one to the bigger one is referred to as *broadcasting*.

Broadcasting can save unnecessary memory allocation. For example, assume we have a 1000×500 matrix x containing 1000 samples, and each sample has 500 features. Now we want to add a bias value for each feature, i.e. a bias vector v of shape 1×500 . Because the shape of x and v do not match, we need to tile v so that it has the same shape as that of x .

```
let x = Mat.uniform 1000 500;; (* generate random samples *)
let v = Mat.uniform 1 500;; (* generate random bias *)
let u = Mat.tile v [|1000;1|];; (* align the shape by tiling *)
Mat.(x + u);;
```

The code above certainly works, but it is obvious that the solution uses too much memory. High memory consumption is not desirable for many applications, especially for those running on resource-constrained devices. Therefore we need the broadcasting.

```
let x = Mat.uniform 1000 500;; (* generate random samples *)
let v = Mat.uniform 1 500;; (* generate random bias *)
Mat.(x + u);; (* returns 1000 x 500 ndarray *)
```

Shape Constraints

In broadcasting, the shapes of two inputs cannot be arbitrarily different, they must be subject to some constraints. The convention used in broadcasting operation is much simpler than slicing. Given two matrices/ndarrays of the same dimensionality, for each dimension, one of the following two conditions must be met: 1) both are equal; 2) either is one.

Here are some valid shapes where broadcasting can be applied between x and y .

```
x : [| 2; 1; 3 |] y : [| 1; 1; 1 |]
x : [| 2; 1; 3 |] y : [| 2; 1; 1 |]
x : [| 2; 1; 3 |] y : [| 2; 3; 1 |]
x : [| 2; 1; 3 |] y : [| 2; 3; 3 |]
x : [| 2; 1; 3 |] y : [| 1; 1; 3 |]
...
...
```



Figure 41: Illustrated example of shape extension in broadcasting

Here are some invalid shapes that violate the aforementioned constraints so that the broadcasting cannot be applied.

```
x : [| 2; 1; 3 |] y : [| 1; 1; 2 |]
x : [| 2; 1; 3 |] y : [| 3; 1; 1 |]
x : [| 2; 1; 3 |] y : [| 3; 1; 1 |]
...
...
```

What if y has less dimensionality than x ? E.g., x has the shape $[|2;3;4;5|]$ whereas y has the shape $[|4;5|]$. In this case, Owl first calls `Ndarray.expand` function to increase y 's dimensionality to the same number as x 's. Technically, two ndarrays are aligned along the highest dimension. In other words, this is done by appending 1s to lower dimension of y , so the new shape of y becomes $[|1;1;4;5|]$. You can try `expand` by yourself, as shown below.

```
let y = Arr.sequential [|4;5|];
let y' = Arr.expand y 4;
```

```
# Arr.shape y'
- : int array = [|1; 1; 4; 5|]
```

If these seem too abstract, here are three concrete 2D examples for you to better understand how the shapes are extended in the broadcasting. The first example is vector multiplied by scalar.

```
let a = Arr.sequential [|1;3|]
```

```
# Arr.add_scalar a 3.
- : Arr.arr =
  C0 C1 C2
R0 3 4 5
```

The second example is matrix plus vector.

```
let b0 = Arr.sequential [|3;3|]
let b1 = Arr.sequential ~a:1. [|1;3|]
```

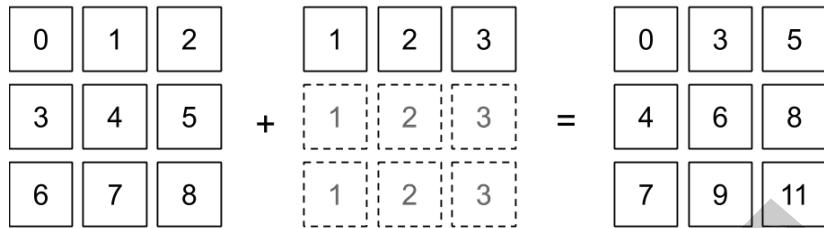


Figure 42: Illustrated example of shape extension in broadcasting (cont.)

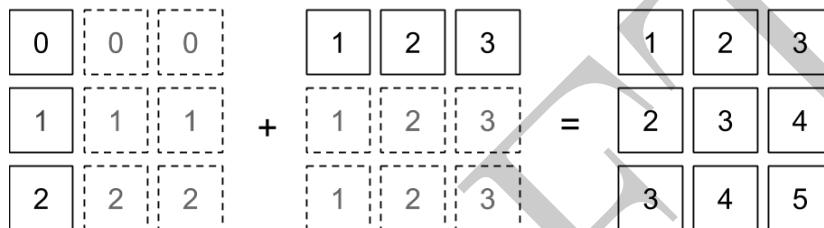


Figure 43: Illustrated example of shape extension in broadcasting (cont.)

```
# Arr.mul b0 b1
- : Arr.arr =
  C0 C1 C2
R0 0 2 6
R1 3 8 15
R2 6 14 24
```

The third example is column vector plus row vector.

```
let c0 = Arr.sequential [|3;1|]
let c1 = Arr.copy b1

# Arr.mul c0 c1
- : Arr.arr =
  C0 C1 C2
R0 0 0 0
R1 1 2 3
R2 2 4 6
```

Supported Operations

The broadcasting operation is transparent to programmers, which means it will be automatically applied if the shapes of two operators do not match (given the

constraints are met of course). Currently, the operations in Owl that support broadcasting are listed below:

- basic computation: `add`, `sub`, `mul`, `div`, `pow`
- comparison operations: `elt_equal`, `elt_not_equal`, `elt_less`, `elt_greater`, `elt_less_equal`, `elt_greater_equal`
- other operations: `min2`, `max2`, `atan2`, `hypot`, `fmod`

Slicing in NumPy and Julia

The indexing and slicing functions are fundamental in all the multi-dimensional array implementations in various other libraries. For example, the examples in fig. 39 and fig. 40 can be implemented using NumPy with code below.

```
>> import numpy as np
>> x = np.arange(64).reshape([8,8])

>> x[:, 2]
array([ 2, 10, 18, 26, 34, 42, 50, 58])

>> x[2, 4:7]
array([20, 21, 22])

>> x[[3,5], 1:8:2]
array([[25, 27, 29, 31],
       [41, 43, 45, 47]])

>> x[[-2,-1], -3:-1]
array([[53, 54],
       [61, 62]])
```

You can see that the basic indexing syntax are similar, only that Python is not strong-typed, so the users can mix single index, list of indices, or index range at will. Note that index range in NumPy is different from that in Owl.

Also, in Julia it can be done with:

```
> x = transpose(reshape([0:1:63], 8 ,8))

> x[:, 3]
8-element Array{Int64,1}:
 2
 10
 18
 26
 34
 42
 50
 58
```

```
> x[3, 5:7]
3-element Array{Int64,1}:
 20
 21
 22

> x[[4,6], 2:2:8]
2x4 Array{Int64,2}:
 25 27 29 31
 41 43 45 47

> x[7:8, [6,7]]
2x2 Array{Int64,2}:
 53 54
 61 62
```

The Julia interface is also similar to that of NumPy. However, there are some crucial differences as shown in these examples. First, the array in Julia uses column-major order, so we need to use the `transpose` function to build the same example. The other obvious difference is that, the indexing of Julia array starts from 1, not 0. Besides, the negative indexing is not supported in Julia.

One important thing to notice in slicing is the difference between `copy` and `view`. For example, in Owl we can make a slice, change the slice, and check what the original ndarray looks like.

```
# let x = Arr.sequential [|3;3|]
val x : Arr.arr =
  C0 C1 C2
R0 0 1 2
R1 3 4 5
R2 6 7 8

# let y = Arr.get_slice [[0]; []] x
val y : Arr.arr =
  C0 C1 C2
R0 0 1 2

# Arr.set y [|0; 2|] 200.;;
- : unit = ()
```

```
# y
- : Arr.arr =
  C0 C1 C2
R0 0 1 200
```

```
# x
- : Arr.arr =
  C0 C1 C2
R0 0 1 2
R1 3 4 5
R2 6 7 8
```

We can see that in Owl, changing the local content of the slice `y` does not change that of the original ndarray `x`. That's because in Owl each slice makes a different copy.

As a contrast, in NumPy the default indexing only makes a “view” of the original array, and any change on the view will also change the original array.

```
>> x = np.arange(9).reshape([3,3])
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>> y = x[0, :]
array([0, 1, 2])

>> y[2] = 200
>> y
array([0, 1, 200])

>> x
array([[0, 1, 200],
       [3, 4, 5],
       [6, 7, 8]])
```

Internal Mechanism

To achieve good performance, slicing is implemented with C in Owl. The basic algorithm of slicing is simple. We need to copy part of the source array `x` to the target array `y`. So you can imagine two cursors move one step at a time, only that for each dimension, the cursor start at different position and offset compared to the starting point, and at different increments. At each step, we simply copy the content from `x` to `y`. To do this, we define a structure `slice_pair` for slicing operations.

```
struct slice_pair {
    int64_t dim; // number of dimensions, x and y must be the same
    int64_t dep; // the depth of current recursion.
    intnat *n; // number of iteration in each dimension, i.e. y's shape
    void **x; // x, source if operation is get, destination if set.
    int64_t posx; // current offset of x.
    int64_t *ofsx; // offset of x in each dimension.
    int64_t *incx; // stride size of x in each dimension.
    void **y; // y, destination if operation is get, source if set.
```

```

int64_t posy; // current offset of y.
int64_t *ofsy; // offset of y in each dimension.
int64_t *incy; // stride size of y in each dimension.
};

```

Taking a 2-dimensional slicing as example, here is the core step:

```

for (int64_t i0 = 0; i0 < n0; i0++) {
    posx0 = posx0 + ofsx0;
    posy0 = posy0 + ofsy0;

    for (int64_t i1 = 0; i1 < n1; i1++) {
        MAPFUN (*(x + posx1), *(y + posy1));
        posx1 += incx1;
        posy1 += incy1;
    }

    posx0 += incx0;
    posy0 += incy0;
}

```

So this algorithm basically says that for each row, we calculate its starting points in x and y ; for each column, copy the element; and then move the cursors forward until the current row is finished. And then move the rows forward. If it becomes multiple dimension, we implement it with recursive algorithm.

Summary

In this chapter we introduce the fundamental operation for ndarrays in a numerical library: the slicing, indexing, and broadcasting. These topics are not difficult, but mastering them takes practice. This chapter provides a lot examples and illustrations. Hope they can be of help to the readers. We also briefly list how the slicing is done in NumPy and Julia, so that the users can see the similarities and differences between Owl and them. In the last of this chapter, we briefly introduce the implementation mechanism of slicing in Owl, for those who are interested in the low level details. More will be discussed in the Part II of this book.

Linear Algebra

Linear Algebra is a key mathematics field behind computer science and numerical computating. A thorough coverage of this topic is apparently beyond the scope of this book. Please refer to (Strang 2006) for this subject. In this chapter we will follow the basic structure of this book, first giving you a overall picture, then focussing on how to use the functions provided in Owl to solve problems and better understand some basic linear algebra concepts.

The high level APIs of Linear Algebra are provided in the `Linalg` module. The module provides four types of number types: single precision, double precision, complex single precision, and complex double precision. They are included in `Linalg.S`, `Linalg.D`, `Linalg.C` and `Linalg.Z` modules respectively. Besides, the `Linalg.Generic` can do everything that `S/D/C/Z` can but needs some extra type information.

Vectors and Matrices

The fundamental problem of linear algebra: solving linear equations. This is more efficiently expressed with vectors and matrices. Therefore, we need to first get familiar with these basic structures in Owl.

Similar to the `Linalg` module, all the matrix functions can be accessed from the `Dense.Matrix` module, and support four different type of modules. The `Mat` module is an alias of `Dense.Matrix.D`. Except for some functions such as `re`, most functions are shared by these four submodules. Note that that matrix module is actually built on the `Ndarray` module, and thus the supported functions are quite similar, and matrices and ndarrays can interoperate with each other. The vectors are expressed using Matrix in Owl.

Creating Matrices

There are multiple functions to help you in creating an initial matrix to start with.

```
Mat.empty 5 5;; (* create a 5 x 5 matrix with initialising elements *)
Mat.create 5 5 2.;; (* create a 5 x 5 matrix and initialise all to 2. *)
Mat.zeros 5 5;; (* create a 5 x 5 matrix of all zeros *)
Mat.ones 5 5;; (* create a 5 x 5 matrix of all ones *)
Mat.eye 5;; (* create a 5 x 5 identity matrix *)
Mat.uniform 5 5; (* create a 5 x 5 random matrix of uniform distribution *)
Mat.uniform_int 5 5;; (* create a 5 x 5 random integer matrix *)
Mat.sequential 5 5;; (* create a 5 x 5 matrix of sequential integers *)
Mat.semidef 5;; (* create a 5 x 5 random semi-definite matrix *)
Mat.gaussian 5 5;; (* create a 5 x 5 random Gaussian matrix *)
Mat.bernoulli 5 5 (* create a 5 x 5 random Bernoulli matrix *)
```

Owl can create some special matrices with specific properties. For example, a *magic square* is a $n \times n$ matrix (where n is the number of cells on each side) filled

with distinct positive integers in the range $1, 2, \dots, n^2$ such that each cell contains a different integer and the sum of the integers in each row, column and diagonal is equal.

```
# let x = Mat.magic 5
val x : Mat.mat =
  C0 C1 C2 C3 C4
  R0 17 24 1 8 15
  R1 23 5 7 14 16
  R2 4 6 13 20 22
  R3 10 12 19 21 3
  R4 11 18 25 2 9
```

We can validate this property with the following code. The summation of all the elements on each column is 65.

```
# Mat.sum_rows x
- : Mat.mat =
  C0 C1 C2 C3 C4
  R0 65 65 65 65 65
```

You can try the similar `sum_cols`. The summation of all the diagonal elements is also 65.

```
# Mat.trace x
- : float = 65.
```

Accessing Elements

Similar to `ndarray`, the matrix module support `set` and `get` to access and modify matrix elements. The only difference is that instead of accessing according to an array, an element in matrix is accessed using two integers.

```
let x = Mat.uniform 5 5;;
Mat.set x 1 2 0.;; (* set the element at (1,2) to 0. *)
Mat.get x 0 3;; (* get the value of the element at (0,3) *)
```

For dense matrices, i.e., `Dense.Matrix.*`, you can also use shorthand `.%(i; j)` to access elements.

```
# open Mat
# x.%(1;2) <- 0.;; (* set the element at (1,2) to 0. *)
- : unit = ()
# let a = x.%(0;3);; (* get the value of the element at (0,3) *)
val a : float = 0.563556290231645107
```

The modifications to a matrix using `set` are in-place. This is always true for dense matrices. For sparse matrices, the thing can be complicated because of performance issues.

We can take some rows out of `x` by calling `rows` function. The selected rows will be used to assemble a new matrix. Similarly, we can also select some columns using `cols`.

Iterate, Map, Fold, and Filter

In reality, a matrix usually represents a collections of measurements (or points). We often need to go through these data over and over again for various reasons. Owl provides very convenient functions to help you to iterate these elements. There is one thing I want to emphasise: Owl uses row-major matrix for storage format in the memory, which means accessing rows are much faster than those column operations.

Let's first create a 4×6 matrix of sequential numbers as below.

```
| let x = Mat.sequential 4 6;;
```

You should be able to see the following output in your `utop`.

C0	C1	C2	C3	C4	C5
1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24

Iterating all the elements can be done by using `iteri` function. The following example prints out all the elements on the screen.

```
| Mat.iteri_2d (fun i j a -> Printf.printf "(%i,%i) %.1f\n" i j a) x;;
```

If you want to create a new matrix out of the existing one, you need `mapi` and `map` function. E.g., we create a new matrix by adding one to each element in `x`.

```
# Mat.map ((+.) 1.) x
- : Mat.mat =

```

C0	C1	C2	C3	C4	C5
R0	1	2	3	4	5
R1	7	8	9	10	11
R2	13	14	15	16	17
R3	19	20	21	22	23
	24				

Iterating rows and columns are similar to iterating elements, by using `iteri_rows`, `mapi_rows`, and etc. The following example prints the sum of each row.

```
Mat.iteri_rows (fun i r ->
  Printf.printf "row %i: %.1f\n" i (Mat.sum' r)
) x;;
```

You can also fold elements, rows, and columns. We can calculate the summation of all column vectors by using `fold_cols` function.

```
let v = Mat.(zeros (row_num x) 1) in
Mat.(fold_cols add v x);;
```

It is also possible to change a specific row or column. E.g., we make a new matrix out of `x` by setting row 2 to zero vector.

```
Mat.map_at_row (fun _ -> 0.) x 2;;
```

The filter functions is also commonly used in manipulating matrix. Here are some examples. The first one is to filter out the elements in `x` greater than 20.

```
# Mat.filter ((<) 20.) x
- : int array = [|21; 22; 23|]
```

You can compare the next example which filters out the two-dimensional indices.

```
# Mat.filteri_2d (fun i j a -> a > 20.) x
- : (int * int) array = [(3, 3); (3, 4); (3, 5)]
```

The second example is to filter out the rows whose summation is less than 22.

```
# Mat.filter_rows (fun r -> Mat.sum' r < 22.) x
- : int array = [|0|]
```

If we want to check whether there is one or (or all) element in `x` satisfying some condition, then

```
Mat.exists ((>) 5.) x;; (* is there someone smaller than 5. *)
Mat.not_exists ((>) 5.) x;; (* is no one smaller than 5. *)
Mat.for_all ((>) 5.) x;; (* is everyone smaller than 5. *)
```

Math Operations

The math operations can be generally categorised into several groups.

Comparison Suppose we have two matrices:

```
let x = Mat.uniform 2 2;;
let y = Mat.uniform 2 2;;
```

We can compare the relationship of x and y element-wisely as below.

```
Mat.(x = y);; (* is x equal to y *)
Mat.(x <> y);; (* is x unequal to y *)
Mat.(x > y);; (* is x greater to y *)
Mat.(x < y);; (* is x smaller to y *)
Mat.(x >= y);; (* is x not smaller to y *)
Mat.(x <= y);; (* is x not greater to y *)
```

All aforementioned infix have their corresponding functions in the module, e.g., $=@$ has `Mat.is_equal`.

Matrix Arithmetic

The arithmetic operation also heavily uses infix. Similar to matrix comparison, each infix has its corresponding function in the module.

```
Mat.(x + y);; (* add two matrices *)
Mat.(x - y);; (* subtract y from x *)
Mat.(x * y);; (* element-wise multiplication *)
Mat.(x / y);; (* element-wise division *)
Mat.(x *@ y);; (* dot product of x and y *)
```

If you do match between a matrix and a scalar value, you need to be careful about their order. Please see the examples below. In the following examples, x is a matrix as we used before, and a is a float scalar value.

```
let a = 2.5;;
Mat.(x +$ a);; (* add a to every element in x *)
Mat.(a $+ x);; (* add a to every element in x *)
```

Similarly, we have the following examples for other math operations.

```
Mat.(x -$ a);; (* sub a from every element in x *)
Mat.(a $- x);;
Mat.(x **$ a);; (* mul a with every element in x *)
Mat.(a $$* x);;
```

```
Mat.(x /$ a);; (* div a to every element in x *)
Mat.(a $/ x);;
Mat.(x **$ a);; (* power of every element in x *)
```

There are some ready-made math functions such as `Mat.log` and `Mat.abs` etc. to ease your life when operating matrices. These math functions apply to every element in the matrix.

There are other functions such as concatenation:

```
Mat.(x @= y);; (* concatenate x and y vertically *)
Mat.(x @|| y);; (* concatenate x and y horizontally *)
```

Gaussian Elimination

Solving linear equations systems is the core problem in Linear Algebra and is frequently used in scientific computation. *Gaussian Elimination* is a classic method to do that. With a bit of techniques, elimination works surprisingly well in modern numerical libraries as one way of implementation. Here is a simple example.

$$\begin{aligned} 2x_1 + 2x_2 + 2x_3 &= 4 \\ 2x_1 + 2x_2 + 3x_3 &= 5 \\ 3x_1 + 4x_2 + 5x_3 &= 7 \end{aligned} \tag{6}$$

Divide the first equation by 2:

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ 2x_1 + 2x_2 + 3x_3 &= 5 \\ 3x_1 + 4x_2 + 5x_3 &= 7 \end{aligned} \tag{7}$$

Multiply the first equation by -2, then add it to the second one. Also, multiply the first equation by -3, then add it to the third one. We have:

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ x_3 &= 1 \\ x_2 + 2x_3 &= 1 \end{aligned} \tag{8}$$

Finally, swap the second and third line:

$$x_1 + x_2 + x_3 = 2$$

$$x_2 + 2x_3 = 1 \quad (9)$$

$$x_3 = 1$$

Here $x_3 = 1$, and we can put it back in the second equation and get $x_2 = -1$. Put both back to the first equation and we have $x_1 = 2$

This process demonstrate the basic process of elimination: eliminate unknown variables until this group of linear equations is easy to solve, and then do the back-substitution. There are three kinds of basic operations we can use: multiplication, adding one line to another, and swap two lines.

The starting eq. 6 can be more concisely expressed with vector:

$$x_1 \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix} + x_2 \begin{bmatrix} 2 \\ 2 \\ 4 \end{bmatrix} + x_3 \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 7 \end{bmatrix}$$

or it can be expressed as $Ax = b$ using matrix notation.

$$\begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 3 \\ 3 & 4 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 7 \end{bmatrix} \Rightarrow \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}$$

Here A is a matrix, b is a column vector, and x is the unknown vector. The matrix notation is often used to describe the linear equation systems as a concise way.

LU Factorisation

Let's check the gaussian elimination example again. The final form in eq. 9 can be expressed with the matrix notation as:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

Here all the elements below the diagonal of this square matrix is zero. Such matrix is called an *upper triangular matrix*, usually denoted by U . Similarly, a square matrix that all the elements below the diagonal of this square matrix is zero is called *lower triangular matrix*, denoted by L . We can use the `is_triu` and `is_tril` to verify if a matrix is triangular.

The diagonal elements of U are called pivots. The i-th pivot is the coefficient of the i-th variable in the i-th equation at the i-th step during the elimination.

In general, a square matrix can often be factorised into the dot product of a lower and a upper triangular matrices: $A = LU$. It is called the *LU factorisation*. It

embodies the process of Gauss elimination. Back to the initial problem of solving the linear equation $Ax = b$. One reason the LU Factorisation is important is that if the matrix A in $Ax = b$ is triangular, then solving it would be straightforward, as we have seen in the previous example. Actually, we can use `triangular_solve` to efficiently solve the linear equations if we already know that the matrix is triangular.

For a normal square matrix that can be factorised into LU , we can change $Ax = b$ to $LUX = b$. First we can find column vector c so that $Lc = b$, then we can find x so that $Ux = c$. Both triangular equations are easy to solve.

We use the `lu` function to perform the LU factorisation. Let's use the previous example.

```
# let a = [|2.;2.;2.;2.;2.;3.;3.;4.;5.|]
val a : float array = [|2.; 2.; 2.; 2.; 2.; 3.; 3.; 4.; 5.|]
# let a = Arr.of_array a [|3; 3|]
val a : Arr.arr =
  C0 C1 C2
  R0 2 2 2
  R1 2 2 3
  R2 3 4 5

# let l, u, p = Linalg.D.lu a
val l : Owl_dense_matrix_d.mat =
  C0 C1 C2
  R0 1 0 0
  R1 0.666667 1 0
  R2 0.666667 1 1

val u : Owl_dense_matrix_d.mat =
  C0 C1 C2
  R0 3 4 5
  R1 0 -0.666667 -0.333333
  R2 0 0 -1

val p : Linalg.D.int32_mat =
  C0 C1 C2
  R0 3 2 3
```

The first two returned matrix are the lower and upper triangular matrices. However, if we try to check the correctness of this factorisation with dot product, the result does not fit:

```
# let a' = Mat.dot l u
val a' : Mat.mat =
  C0 C1 C2
  R0 3 4 5
```

R1	2	2	3
R2	2	2	2

```
# a' = a
- : bool = false
```

It turns out that we need to some extra row exchange to get the right answer. That's because the row exchange is required in certain cases, such as when the number we want to use as the pivot could be zero. This process is called *pivoting*. It is closely related to the numerical computation stability. Choosing the improper pivots can lead to wrong linear system solution. It can be expressed with a permutation matrix that has the same rows as the identity matrix, each row and column has exactly one “1” element. The full LU factorisation can be expressed as:

$$PA = LU.$$

```
# let p = Mat.of_array [|0.;0.;1.;0.;1.;0.;1.;0.;0.|] 3 3
val p : Mat.mat =
  C0 C1 C2
R0 0 0 1
R1 0 1 0
R2 1 0 0

# Mat.dot p a = Mat.dot l u
- : bool = true
```

How do we translate the third output, the permutation vector, to the required permutation matrix? Each element p_i in the vector represents a updated identity matrix. On this identity matrix, we set (i, i) and (p_i, p_i) to zero, and then (i, p_i) and (p_i, i) to one. Multiply these n matrices, we can get the permutation matrix P . Here is a brief implementation of this process in OCaml:

```
let perm_vec_to_mat vec =
  let n = Array.length vec in
  let mat = ref (Mat.eye n) in
  for i = n - 1 downto 0 do
    let j = vec.(i) in
    let a = Mat.eye n in
    Arr.set a [| i; i |] 0.;
    Arr.set a [| j; j |] 0.;
    Arr.set a [| i; j |] 1.;
    Arr.set a [| j; i |] 1.;
    mat := Arr.dot a !mat
  done;
  !mat
```

Note that there is more than one way to do the LU factorisation. For example, for the same matrix, we can have:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 3 \\ 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1.5 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 2 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

Inverse and Transpose

The concept of inverse matrix is related with the identity matrix, which can be built with *Mat.eyen*, where n is the size of the square matrix. The identity matrix is a special form of *Diagonal Matrix*, which is a square matrix that only contains non-zero element along its diagonal. You can check if a matrix is diagonal with *is_diag* function.

```
| Mat.eyen 5 |> Linalg.D.is_diag
```

The inverse of a n by n square matrix A is denoted by A^{-1} , so that $AA^{-1} = I_n$. Note that not all square matrix has inverse.

There are many sufficient and necessary conditions to decide if A is invertible, one of them is that A has n pivots.

We use function *inv* to do the inverse operation. It's straightforward and easy to verify according to the definition. Here we use the *semidef* function to produce a matrix that is certainly invertible.

```
# let x = Mat.semidef 5
val x : Mat.mat =
  C0 C1 C2 C3 C4
  R0 2.56816 1.0088 1.57793 2.6335 2.06612
  R1 1.0088 0.441613 0.574465 1.02067 0.751004
  R2 1.57793 0.574465 2.32838 2.41251 2.13926
  R3 2.6335 1.02067 2.41251 3.30477 2.64877
  R4 2.06612 0.751004 2.13926 2.64877 2.31124

# let y = Linalg.D.inv x
val y : Owl_dense_matrix_d.mat =
  C0 C1 C2 C3 C4
  R0 12.229 -15.606 6.12229 -1.90254 -9.34742
  R1 -15.606 33.2823 -4.01361 -4.96414 12.5403
  R2 6.12229 -4.01361 7.06372 -2.62899 -7.69399
  R3 -1.90254 -4.96414 -2.62899 8.1607 -3.60533
  R4 -9.34742 12.5403 -7.69399 -3.60533 15.9673
```

```
# Mat.(x *@ y =~ eye 5)
- : bool = true
```

The next frequently used special matrix is the *Transpose Matrix*. Denoted by A^T , its i th row is taken from the i -th column of the original matrix A. It has properties such as $(AB)^T = B^T A^T$. We can check this property using the matrix function `Mat.transpose`. Note that this function is deemed basic ndarray operations and is not included in the `Linalg` module.

```
# let flag =
  let a = Mat.uniform 4 4 in
  let b = Mat.uniform 4 4 in
  let m1 = Mat.(dot a b |> transpose) in
  let m2 = Mat.(dot (transpose b) (transpose a)) in
  Mat.(m1 =~ m2)
val flag : bool = true
```

A related special matrix is the *Symmetric Matrix*, which equals to its own transpose. This simple test can be done with the `is_symmetric` function.

Vector Spaces

We have talked about solving the $Ax = b$ linear equations with elimination, and A is a square matrix. Now we need to further discuss, how do we know if there exists one or maybe more than one solution. To answer such question, we need to be familiar with the concepts of *vector space*.

A vector space, denoted by R^n , contains all the vectors that has n elements. In this vector space we have the `add` and `multiplication` operation. Applying them to the vectors is called *linear combination*. Then a *subspace* in a vector space is a non-empty set that linear combination of the vectors in this subspace still stays in the same subspace.

There are four fundamental subspaces concerning solving linear systems $Ax = b$, where A is a m by n matrix. The *column space* consists of all the linear combinations of the columns of A. It is a subspace of R^m . Similarly, the *row space* consists of all the linear combinations of the rows of A. The *nullspace* contains all the vectors x so that $Ax = 0$, denoted by $N(A)$. It is a subspace of R^n . The *left nullspace* is similar. It is the nullspace of A^T .

Rank and Basis

In the Gaussian Elimination section, we assume an ideal situation: the matrix A is $n \times n$ square, and we assume that there exists one solution. But that does not happen every time. In many cases A is not an square matrix. It is possible that these m equations are not enough to solve a n -variable linear system when $m < n$. Or there might not exist a solution when $m > n$. Besides, even it is a

square matrix, the information provided by two of the equations are actually repeated. For example, one equation is simply a multiplication of the other.

For example, if we try to apply LU factorisation to such a matrix:

```
# let x = Mat.of_array [|1.; 2.; 3.; 0.; 0.; 1.; 0.; 0.; 2.|] 3 3
val x : Mat.mat =
  C0 C1 C2
R0 1 2 3
R1 0 0 1
R2 0 0 2

# Linalg.D.lu x
Exception: Failure "LAPACKE: 2".
```

Obviously, we cannot have pivot in the second column, and therefore this matrix is singular and cannot be factorised into LU . As can be seen in this example, we cannot expect the linear algebra functions to be a magic lamb and do our bidding every time. Understanding the theory of linear algebra helps to better understand how these functions work.

To decide the general solutions to $Ax = b$, we need to understand the concept of *rank*. The rank of a matrix is the number of pivots in the elimination process. To get a more intuitive understanding of rank, we need to know the concept of *linear independent. In a linear combination $\sum_{i=1}^n c_i v_i$ where v_i are vectors and c_i are numbers, if $\sum_{i=1}^n c_i v_i = 0$ only happens when $c_i = 0$ for all the i 's, then the vectors v_1, v_2, \dots, v_n are linearly independent. Then the rank of a matrix is the number of independent rows in the matrix. We can understand rank as the number of “effective” rows in the matrix.

As an example, we can check the rank of the previous matrix.

```
Linalg.D.rank x
```

As can be example, the rank is 2, which means only two effective rows, and thus cannot be factorised to find the only solution.

One application of rank is in a crucial linear algebra idea: basis. A sequence of vectors is the *basis* of a space or subspace if: 1) these vectors are linear independent and 2) all the the vectors in the space can be represented as the linear combination of vectors in the basis.

A space can have infinitely different bases, but the number of vectors in these bases are the same. This number is called the *dimension* of this vector space. For example, a m by n matrix A has rank of r , then the dimension of its null space is $n - r$, and the dimension of its column space is r . Think about a full-rank matrix where $r = n$, then the dimension of column matrix is n , which means all its

columns can be a basis of the column space, and that the null space dimension is zero so that the only solution of $Ax = 0$ is a zero vector.

Orthogonality

We can think of the basis of a vector space as the Cartesian coordinate system in a three-dimensional space, where every vector in the space can be represented with the three vectors in the space: the x, y and z axis. Actually, we can use many three vectors system as the coordinate bases, but the x, y, z axis is used because they are orthogonal to each other. An orthogonal basis can greatly reduce the complexity of problems. The same can be applied in the basis of vector spaces.

Orthogonality is not limited to vectors. Two vectors a and b are orthogonal if $a^T b = 0$. Two subspaces A and B are orthogonal if every vector in A is orthogonal to every vector in B. For example, the nullspace and row space of a matrix are perpendicular to each other.

Among the bases of a subspace, if every vector is perpendicular to each other, it is called an orthogonal matrix. Moreover, if the length of each vector is normalised to one unit, it becomes the *orthonormal basis*.

For example, we can use the `null` function to find an orthonormal basis vector x or the null space of a matrix, i.e. $Ax = 0$.

```
# let a = Mat.magic 4
val a : Mat.mat =
  C0 C1 C2 C3
  R0 1 15 14 4
  R1 12 6 7 9
  R2 8 10 11 5
  R3 13 3 2 16

# let x = Linalg.D.null a
val x : Owl_dense_matrix_d.mat =
  C0
  R0 -0.223607
  R1 -0.67082
  R2 0.67082
  R3 0.223607

# Mat.dot a x |> Mat.l2norm'
- : float = 2.87802701599908967e-15
```

Now that we know what is orthogonal basis, the next question is, how to build one? The *QR Factorisation* is used to construct orthogonal basis in a subspace.

Specifically, it decomposes a matrix A into the product of an orthogonal matrix Q and an upper triangular matrix R , i.e. $A = QR$. It is provided in the linear algebra module.

```
val qr : ?thin:bool -> ?pivot:bool -> ('a, 'b) t -> ('a, 'b) t * ('a, 'b) t *
(int32, int32_elt) t
```

The `qr x` function calculates QR decomposition for an m by n matrix x . The function returns a 3-tuple, the first two are q and r , and the third is the permutation vector of columns. The default value of parameter `pivot` is `false`, setting `pivot` to `true` lets `qr` performs pivoted factorisation. Note that the returned indices are not adjusted to 0-based C layout. By default, `qr` performs a reduced QR factorisation, full factorisation can be enabled by setting `thin` parameter to `false`.

```
# let a = Mat.of_array [|12.; -51.; 4.; 6.; 167.; -68.; -4.; 24.; -41.|] 3 3
```

```
val a : Mat.mat =
```

	C0	C1	C2
R0	12	-51	4
R1	6	167	-68
R2	-4	24	-41

```
# let q, r, _ = Linalg.D.qr a
```

```
val q : Owl_dense_matrix_d.mat =
```

	C0	C1	C2
R0	-0.857143	0.394286	0.331429
R1	-0.428571	-0.902857	-0.0342857
R2	0.285714	-0.171429	0.942857

```
val r : Owl_dense_matrix_d.mat =
```

	C0	C1	C2
R0	-14	-21	14
R1	0	-175	70
R2	0	0	-35

Solving $Ax = b$

We can now discuss the general solution structure to $Ax = 0$ and $Ax = b$. Again, here A is a $m \times n$ matrix. The theorems declare that, there exists non-zero solution(s) to $Ax = 0$ if and only if $\text{rank}(a) \leq n$. If $r(A) < n$, then the nullspace of A is of dimension $n - r$ and the $n - r$ orthogonal basis can be found with `null` function. Here is an example.

These two vectors are called the *fundamental system of solutions* of $Ax = 0$. All the solutions of $Ax = 0$ can then be expressed using the fundamental system:

$$c_1 \begin{bmatrix} -0.85 \\ 0.27 \\ 0.07 \\ 0.44 \end{bmatrix} + c_2 \begin{bmatrix} 0.013 \\ 0.14 \\ 0.95 \\ -0.23 \end{bmatrix}$$

Here c_1 and c_2 can be any constant numbers.

For solving the general form $Ax = b$ where b is $m \times 1$ vector, there exist only one solution if and only if $\text{rank}(A) = \text{rank}([A, b]) = n$. Here $[A, b]$ means concatenating A and b along the column. If $\text{rank}(A) = \text{rank}([A, b]) < n$, $Ax = b$ has infinite number of solutions. These solutions has a general form:

$$x_0 + c_1 x_1 + c_2 x_2 + \dots + c_k x_k$$

Here x_0 is a particular solution to $Ax = b$, and x_1, x_2, \dots, x_k are the fundamental solution system of $Ax = 0$.

We can use `linsolve` function to find one particular solution. In the Linear Algebra, the function `linsolve a b -> x` solves a linear system of equations $a * x = b$. By default, the function uses LU factorisation with partial pivoting when a is square and QR factorisation with column pivoting otherwise. The number of rows of a must be equal to the number of rows of b . If a is a upper or lower triangular matrix, the function calls the `solve_triangular` function.

Here is an example.

```
# let a = Mat.of_array [|2.;3.;1.;1.;;-2.;4.;3.;8.;;-2.;4.;-1.;9.|] 4 3
val a : Mat.mat =
```

```
C0 C1 C2
R0 2 3 1
R1 1 -2 4
R2 3 8 -2
R3 4 -1 9
```

```
# let b = Mat.of_array [|4.;-5.;13.;-6.|] 4 1
val b : Mat.mat =
```

```
C0
R0 4
R1 -5
R2 13
R3 -6
```

```
# let x0 = Linalg.D.linsolve a b
val x0 : Owl_dense_matrix_d.mat =
```

```
C0
R0 -5
R1 4
R2 2
```

Then we use `null` to find the fundamental solution system. You can verify that matrix `a` is of rank 2, so that the solution system for $ax = 0$ should contain only $3 - 2 = 1$ vector.

```
# let x1 = Linalg.D.null a
val x1 : Owl_dense_matrix_d.mat =
```

```
C0
R0 -0.816497
R1 0.408248
R2 0.408248
```

So the solutions to $Ax = b$ can be expressed as:

$$\begin{bmatrix} -1 \\ 2 \\ 0 \end{bmatrix} + c_1 \begin{bmatrix} -0.8 \\ 0.4 \\ 0.4 \end{bmatrix}$$

So the takeaway from this chapter is that the using these linear algebra functions often requires solid background knowledge. Blindly using them could leads to wrong or misleading answers.

Matrix Sensitivity

The *sensitivity* of a matrix is perhaps not the most important issue in the traditional linear algebra, but is crucial in the numerical computation related problems. It answers this question: in $Ax = b$, if we change the A and b slightly, how much will the x be affected? The *Condition Number* is a measurement of the sensitivity of a square matrix.

First, we need to understand the *Norm* of a matrix. The norm, or 2-norm of a matrix $\|A\|$ is calculated as square root of the maximum eigenvalue of $A^H A$. The norm of a matrix is a upper limit so that for any x we can be certain that $\|Ax\| \leq \|A\| \|x\|$. Here $\|Ax\|$ and $\|x\|$ are the L2-Norm for vectors. The $\|A\|$ bounds the how large the A can amplify the input x . We can calculate the norm with `norm` in the linear algebra module.

The most frequently used condition number is that represent the sensitivity of inverse matrix. With the definition of norm, the *condition number for inversion* of a matrix can be expressed as $\|A\| \|A^{-1}\|$. We can calculate it using the `cond` function.

Let's look at an example:

```
# let a = Mat.of_array [|4.1; 2.8; 9.7; 6.6 |] 2 2;;
val a : Mat.mat =
  C0 C1
R0 4.1 2.8
R1 9.7 6.6

# let c = Linalg.D.cond a
val c : float = 1622.99938385651058
```

Its condition number for inversion is much larger than one. Therefore, a small change in A should leads to a large change of A^{-1} .

```
# let a' = Linalg.D.inv a
val a' : Owl_dense_matrix_d.mat =
  C0 C1
R0 -66.28
R1 97. -41.

# let a2 = Mat.of_array [|4.1; 2.8; 9.67; 6.607 |] 2 2
val a2 : Mat.mat =
  C0 C1
R0 4.1 2.8
R1 9.67 6.607
```

```
# let a2' = Linalg.D.inv a2
val a2' : Owl_dense_matrix_d.mat =
  C0 C1
  R0 520.236 -220.472
  R1 -761.417 322.835
```

We can see that by changing the matrix by only a tiny bit, the inverse of A changes dramatically, and so is the resulting solution vector x .

Determinants

Other than pivots, another basic quantity in linear algebra is the *determinants*. For a square matrix A :

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

its determinants $\det(A)$ is defined as:

$$\sum_{j_1 j_2 \dots j_n} (-1)^{\tau(j_1 j_2 \dots j_n)} a_{1j_1} a_{2j_2} \dots a_{nj_n}.$$

Here $\tau(j_1 j_2 \dots j_n) = i_1 + i_2 + \dots + i_{n-1}$, where i_k is the number of j_p that is smaller than j_k for $p \in [k+1, n]$.

Mathematically, there are many techniques that can be used to simplify this calculation. But as far as this book is concerned, it is sufficient for us to use the `det` function to calculate the determinants of a matrix.

Why is the concept of determinant important? Its most important application is to using determinant to decide if a square matrix A is invertible or singular. The determinant $\det(A) \neq 0$ if and only if $A = n$. Also it can be expressed as $\det(A) \neq 0$ if and only if matrix A is invertible.

We can also use it to understand the solution of $Ax = b$: if $\det(A) \neq 0$, then $Ax = b$ has one and only one solution. This theorem is part of the *Cramer's rule*. These properties are widely used in finding *eigenvalues*. As will be shown in the next section.

Since sometimes we only care about if the determinant is zero or not, instead of the value itself, we can also use a similar function `logdet`. It computes the logarithm of the determinant, but it avoids the possible overflow or underflow problems in computing determinant of large matrices.

```
# let x = Mat.magic 5
val x : Mat.mat =
  C0 C1 C2 C3 C4
R0 17 24 1 8 15
R1 23 5 7 14 16
R2 4 6 13 20 22
R3 10 12 19 21 3
R4 11 18 25 2 9

# Linalg.D.det x
- : float = 5070000.00000000093

# Linalg.D.logdet x
- : float = 15.4388513755673653
```

Eigenvalues and Eigenvectors

Solving $Ax = \lambda x$

Now we need to change the topic from $Ax = b$ to $Ax = \lambda x$. For an $n \times n$ square matrix, if there exist number λ and non-zero column vector x to satisfy:

$$(\lambda I - A)x = 0, \quad (10)$$

then λ is called *eigenvalue*, and x is called the *eigenvector* of A .

To find the eigenvalues of A , we need to find the roots of the determinant of $\lambda I - A$. $\det(\lambda I - A) = 0$ is called the *characteristic equation*. For example, for

$$A = \begin{bmatrix} 3 & 1 & 0 \\ -4 & -1 & 0 \\ 4 & -8 & 2 \end{bmatrix}$$

Its characteristic matrix $\lambda I - A$ is:

$$\begin{bmatrix} \lambda - 3 & 1 & 0 \\ -4 & \lambda + 1 & 0 \\ 4 & -8 & \lambda - 2 \end{bmatrix}$$

According to the definition of determinant,

$$\det(\lambda I - A) = (\lambda - 1)^2(\lambda - 2) = 0.$$

According to the theory of polynomials, this characteristic polynomials has and only has n roots in the complex space. Specifically, here we have three eigenvalues: $\lambda_1 = 1, \lambda_2 = 1, \lambda = 2$.

Put λ_1 back to characteristic equation, we have: $(I - A)x = 0$. Therefore, we can find the fundamental solution system of $I - A$ with:

```
# let basis =
  let ia = Mat.((eye 3) - (of_array [|3.;1.;0.;-4.;-1.;0.;4.;-8.;2.|] 3 3)) in
  Linalg.D.null ia
val basis : Owl_dense_matrix_d.mat =
  C0
  R0 -0.0496904
  R1 0.0993808
  R2 0.993808
```

We have a fundamental solution $x_0 = [-0.05, 0.1, 1]^T$. Therefore all the $k_0 x_0$ are the corresponding eigenvector of the eigenvalue 1. Similarly, we can calculate that eigenvectors for the eigenvalue 2 are $k_1 [0, 0, 1]^T$.

We can use `eig` to find the eigenvectors and eigenvalues of a matrix. `eig x -> v, w` computes the right eigenvectors `v` and eigenvalues `w` of an arbitrary square matrix `x`. The eigenvectors are column vectors in `v`, their corresponding eigenvalues have the same order in `w` as that in `v`.

```
# let eigvec, eigval =
  let a = Mat.of_array [|3.;1.;0.;-4.;-1.;0.;4.;-8.;2.|] 3 3 in
  Linalg.D.eig a
val eigvec : Owl_dense_matrix_z.mat =
  C0 C1 C2
  R0 (0, 0i) (0.0496904, 0i) (0.0496904, 0i)
  R1 (0, 0i) (-0.0993808, 0i) (-0.0993808, 0i)
  R2 (1, 0i) (-0.993808, 0i) (-0.993808, 0i)

val eigval : Owl_dense_matrix_z.mat =
  C0 C1 C2
  R0 (2, 0i) (1, 0i) (1, 0i)
```

Note that the result are expressed as complex numbers. If we only want the eigenvalues, we can use the `eigvals` function. Both functions provide the boolean `permute` and `scale` arguments to indicate whether the input matrix should be permuted and/or diagonally scaled. One reason that eigenvalue and eigenvector are important is that the pattern $Ax = \lambda x$ frequently appears in scientific and engineering analysis to describe the change of dynamic system over time.

Complex Matrices

As can be seen in the previous example, complex matrices are frequently used in eigenvalues in eigenvectors. In this section we re-introduce some previous concepts in the complex space.

We have seen the Symmetric Matrix. It can be extended to the complex numbers, called *Hermitian Matrix*, denoted by A^H . Instead of requiring it to be the same as its transpose, a hermitian matrix equals to its conjugate transpose. A conjugate transpose means that during transposing, each element $a + bi$ changes to its conjugate $a - bi$. Hermitian is thus a generalisation of the symmetric matrix. We can use the `is_hermitian` function to check if a matrix is hermitian, as can be shown in the next example.

```
# let a = Dense.Matrix.Z.of_array [|{re=1.; im=0.}; {re=2.; im=(-1.)}; {re=2.; im=1.}; {re=3.; im=0.}|] 2 2
val a : Dense.Matrix.Z.mat =
  C0 C1
  R0 (1, 0i) (2, -1i)
  R1 (2, 1i) (3, 0i)

# Linalg.Generic.is_hermitian a
- : bool = true
```

We can use the `conj` function of a complex matrix to perform the conjugate transpose:

```
# Dense.Matrix.Z.(conj a |> transpose)
- : Dense.Matrix.Z.mat =
  C0 C1
  R0 (1, -0i) (2, -1i)
  R1 (2, 1i) (3, -0i)
```

A theorem declares that if a matrix is hermitian, then for all complex vectors x , $x^H Ax$ is real, and every eigenvalue is real.

```
# Linalg.Z.eigvals a
- : Owl_dense_matrix_z.mat =
  C0 C1
  R0 (-0.44949, 1.50231E-17i) (4.44949, 2.07021E-16i)
```

A related concept is the *Unitary Matrix*. A matrix U is unitary if $U^H U = I$. The inverse and conjugate transpose of U are the same. It can be compared to the orthogonal vectors in the real space.

Similarity Transformation and Diagonalisation

For a $n \times n$ matrix A , and any invertible $n \times n$ matrix M , the matrix $B = M^{-1}AM$ is *similar* to A . One important property is that similar matrices share the same

eigenvalues. The intuition is that, think of M as the change of basis matrix, and A itself is a linear transformation, so $M^{-1}AM$ means changing the basis first, applying the linear transformation, and then change the basis back. Therefore, changing from A to B actually changes the linear transformation using one set of basis to another.

In a three dimensional space, if we can change using three random vectors as the basis of linear transformation to using the standard basis $[1, 0, 0]$, $[0, 1, 0]$, $[0, 0, 1]$, the related problem can be greatly simplified. Finding the suitable similar matrix is thus important in simplifying the calculation in many scientific and engineering problems.

One possible kind of simplification is to find a triangular matrix as similar. The *Schur's Lemma* declares that A can be decomposed into UTU^{-1} where U is a unitary function, and T is an upper triangular matrix.

```
# let a = Dense.Matrix.Z.of_array [|{re=1.; im=0.}; {re=1.; im=0.}; {re=(-2.); im=0.}; {re=3.; im=0.}|] 2 2
val a : Dense.Matrix.Z.mat =
  C0 C1
  R0 (1, 0i) (1, 0i)
  R1 (-2, 0i) (3, 0i)

# let t, u, eigvals = Linalg.Z.schur a
val t : Owl_dense_matrix_z.mat =
  C0 C1
  R0 (2, 1i) (2.10381, -0.757614i)
  R1 (0, 0i) (2, -1i)

val u : Owl_dense_matrix_z.mat =
  C0 C1
  R0 (-0.408248, 0.408248i) (0.563384, -0.590987i)
  R1 (-0.816497, 0i) (-0.577185, 0.0138014i)

val eigvals : Owl_dense_matrix_z.mat =
  C0 C1
  R0 (2, 1i) (2, -1i)
```

The returned result t is apparent a upper triangular matrix, and the u can be verified to be a unitary matrix:

```
# Dense.Matrix.Z.(dot u (conj u |> transpose))
- : Dense.Matrix.Z.mat =
```

```
C0 C1
```

```
R0 (1, 0i) (7.97973E-17, 5.81132E-17i)
R1 (7.97973E-17, -5.81132E-17i) (1, 0i)
```

Another very important similar transformation is *diagonalisation*. Suppose A has n linear-independent eigenvectors, and make them the columns of a matrix Q, then $Q^{-1}AQ$ is a diagonal matrix Λ , and the eigenvalues of A are the diagonal elements of Λ . Its inverse $A = Q\Lambda Q^{-1}$ is called *Eigendecomposition*. Analysing A's diagonal similar matrix Λ instead of A itself can greatly simplify the problem.

Not every matrix can be diagonalised. If any two of the n eigenvalues of A are not the same, then its n eigenvectors are linear-independent and thus A can be diagonalised. Specifically, every real symmetric matrix can be diagonalised by an orthogonal matrix. Or put into the complex space, every hermitian matrix can be diagonalised by a unitary matrix.

Positive Definite Matrices

Positive Definiteness

In this section we introduce the concept of *Positive Definite Matrix*, which unifies the three most basic ideas in linear algebra: pivots, determinants, and eigenvalues.

A matrix is called *Positive Definite* if it is symmetric and that $x^T Ax > 0$ for all non-zero vectors x . There are several necessary and sufficient condition for testing if a symmetric matrix A is positive definite:

1. $x^T Ax > 0$ for all non-zero real vectors x
2. $\lambda_i > 0$ for all eigenvalues λ_i of A
3. all the upper left matrices have positive determinants
4. all the pivots without row exchange satisfy $d > 0$
5. there exists invertible matrix B so that $A=B^T B$

For the last condition, we can use the *Cholesky Decomposition* to find the matrix B. It decompose a Hermitian positive definite matrix into the product of a lower triangular matrix and its conjugate transpose LL^H :

```
# let a = Mat.of_array [|4.;12.;-16.;12.;37.;-43.;-16.;-43.;98.|] 3 3
val a : Mat.mat =
  C0 C1 C2
R0 4 12 -16
R1 12 37 -43
R2 -16 -43 98
```

```
# let l = Linalg.D.chol a
val l : Owl_dense_matrix_d.mat =
```

	C0	C1	C2
R0	2	6	-8
R1	0	1	5
R2	0	0	3

```
# Mat.(dot (transpose 1) 1)
- : Mat.mat =


|    | C0  | C1  | C2  |
|----|-----|-----|-----|
| R0 | 4   | 12  | -16 |
| R1 | 12  | 37  | -43 |
| R2 | -16 | -43 | 98  |


```

If in $Ax = b$ we know that A is hermitian and positive definite, then we can instead solve $L^T x = b$. As we have seen previously, solving linear system that expressed with triangular matrices is easy. The Cholesky decomposition is more efficient than the LU decomposition.

In the Linear Algebra module, we use `is_posdef` function to do this test. If you look at the code in Owl, it is implemented by checking if the Cholesky decomposition can be performed on the input matrix.

```
# let is_pos =
  let a = Mat.of_array [|4.;12.;-16.;12.;37.;-43.;-16.;-43.;98.|] 3 3 in
  Linalg.D.is_posdef a
val is_pos : bool = true
```

The definition of *semi-positive definite* is similar, only that it allows the “equals to zero” part. For example, $x^T Ax \leq 0$ for all non-zero real vectors x .

The pattern $Ax = \lambda Mx$ exists in many engineering analysis problems. If A and M are positive definite, this pattern is parallel to the $Ax = \lambda x$ where $\lambda > 0$. For example, a linear system $y' = Ax$ where $x = [x_1, x_2, \dots, x_n]$ and $y' = [\frac{dx_1}{dt}, \frac{dx_2}{dt}, \dots, \frac{dx_n}{dt}]$. We will see such an example in the Ordinary Differential Equation chapter. In a linearised differential equations the matrix A is the Jacobian matrix. The eigenvalues decides if the system is stable or not. A theorem declares that this system is stable if and only if there exists positive and definite matrix V so that $-(VA + A^T V)$ is semi-positive definite.

Singular Value Decomposition

The singular value decomposition (SVD) is among the most important matrix factorizations of the computational era. The SVD provides a numerically stable matrix decomposition that can be used for a variety of purposes and is guaranteed to exist.

Any m by n matrix can be factorised in the form:

$$A = U \Sigma V^T \quad (11)$$

Here U is a $m \times m$ matrix. Its columns are the eigenvectors of AA^T . Similarly, V is a $n \times n$ matrix, and the columns of V are eigenvectors of A^TA . The r (rank of A) singular value on the diagonal of the $m \times n$ diagonal matrix Σ are the square roots of the nonzero eigenvalues of both AA^T and A^TA . It's close related with eigenvector factorisation of a positive definite matrix. For a positive definite matrix, the SVD factorisation is the same as the $Q\Lambda Q^T$.

The SVD has a profound intuition. A matrix A represents a linear transformation. SVD states that, any such linear transformation, can be decomposed into three simple transformation: a rotation (V), a scaling transformation (Σ), and another rotation (U). These three transformations are much easier to analyse than a random transformation A . After applying A to a domain, the columns of V is and a set of orthonormal basis in the original domain, and columns of U is the new set of orthonormal basis of the domain that is transferred after applying A . The Σ diagonal matrix contains the scaling factors on different dimensions, and a singular value in Σ thus represents the *significance* of that certain dimension in the linear space. A small singular value indicates that the information contained in a matrix is somehow redundant and can be compressed/removed without affecting the information carried in this matrix. This is why SVD can be used for *Principal Component Analysis* (PCA), as we will show in the NLP chapter later in this book.

We can use the `svd` function to perform this factorisation. Let's use the positive definite matrix as an example:

```
# let a = Mat.of_array [|4.;12.;-16.;12.;37.;-43.;-16.;-43.;98.|] 3 3
val a : Mat.mat =
  C0 C1 C2
  R0 4 12 -16
  R1 12 37 -43
  R2 -16 -43 98

# let u, s, vt = Linalg.D.svd ~thin:false a
val u : Owl_dense_matrix_d.mat =
  C0 C1 C2
  R0 -0.163007 -0.212727 0.963419
  R1 -0.457324 -0.848952 -0.26483
  R2 0.874233 -0.483764 0.0410998

val s : Owl_dense_matrix_d.mat =
  C0 C1 C2
  R0 123.477 15.504 0.018805
```

```

val vt : Owl_dense_matrix_d.mat =
  C0 C1 C2
R0 -0.163007 -0.457324 0.874233
R1 -0.212727 -0.848952 -0.483764
R2 0.963419 -0.26483 0.0410998

```Note that the diagonal matrix `s` is represented as a vector. We can extend it
with

```ocaml
# let s = Mat.diagm s
val s : Mat.mat =
  C0 C1 C2
R0 123.477 0 0
R1 0 15.504 0
R2 0 0 0.018805

```However, it is only possible when we know that the original diagonal matrix is
square, otherwise the vector contains the $\min(m, n)$ diagonal elements.

Also, we can find to the eigenvectors of AA^T to verify that it equals to the
eigenvector factorisation.

```ocaml
# Linalg.D.eig Mat.(dot a (transpose a))
- : Owl_dense_matrix_z.mat * Owl_dense_matrix_z.mat =
(
  C0 C1 C2
R0 (0.163007, 0i) (0.963419, 0i) (0.212727, 0i)
R1 (0.457324, 0i) (-0.26483, 0i) (0.848952, 0i)
R2 (-0.874233, 0i) (0.0410998, 0i) (0.483764, 0i)
,
  C0 C1 C2
R0 (15246.6, 0i) (0.000353627, 0i) (240.373, 0i)
)

```

In this example we uses the `thin` parameter. By default, the `svd` function performs a reduced SVD, where Σ is a $m \times m$ matrix and V^T is a m by n matrix.

Besides, `svd`, we also provide `svdvals` that only returns the singular values, i.e. the vector of diagonal elements. The function `gsvd` performs a generalised SVD. `gsvd x y -> (u, v, q, d1, d2, r)` computes the generalised SVD of a pair of general rectangular matrices x and y . $d1$ and $d2$ contain the generalised singular value pairs of x and y . The shape of x is $m \times n$ and the shape of y is $p \times n$. Here is an example:

```

# let x = Mat.uniform 5 5
val x : Mat.mat =
  C0 C1 C2 C3 C4

```

```
R0 0.548998 0.623231 0.95821 0.440292 0.551542
R1 0.406659 0.631188 0.434482 0.519169 0.0841121
R2 0.439047 0.459974 0.767078 0.148038 0.445326
R3 0.307424 0.129056 0.998469 0.163971 0.718515
R4 0.474817 0.176199 0.316661 0.476701 0.138534
```

```
# let y = Mat.uniform 2 5
val y : Mat.mat =
  C0 C1 C2 C3 C4
R0 0.523882 0.150938 0.718397 0.1573 0.00542669
R1 0.714052 0.874704 0.436799 0.198898 0.406196

# let u, v, q, d1, d2, r = Linalg.D.gsvd x y
val u : Owl_dense_matrix_d.mat =
  C0 C1 C2 C3 C4
R0 -0.385416 -0.294725 -0.398047 0.0383079 -0.777614
R1 0.18222 -0.404037 -0.754063 -0.206208 0.438653
R2 -0.380469 0.0913876 -0.199462 0.847599 0.297795
R3 -0.807427 -0.147819 0.194202 -0.418909 0.336172
R4 0.146816 -0.848345 0.442095 0.249201 0.0347409

val v : Owl_dense_matrix_d.mat =
  C0 C1
R0 0.558969 0.829189
R1 0.829189 -0.558969

val q : Owl_dense_matrix_d.mat =
  C0 C1 C2 C3 C4
R0 -0.436432 -0.169817 0.642272 -0.603428 0.0636394
R1 -0.124923 0.407939 -0.376937 -0.494889 -0.656494
R2 0.400859 0.207482 -0.268507 -0.567199 0.634391
R3 -0.283012 -0.758558 -0.559553 -0.173745 0.0347457
R4 0.743733 -0.431612 0.245375 -0.197629 -0.40163

val d1 : Owl_dense_matrix_d.mat =
  C0 C1 C2 C3 C4
R0 1 0 0 0 0
R1 0 1 0 0 0
R2 0 0 1 0 0
R3 0 0 0 0.319964 0
R4 0 0 0 0 0.0583879

val d2 : Owl_dense_matrix_d.mat =
  C0 C1 C2 C3 C4
R0 0 0 0 0.94743 0
R1 0 0 0 0 0.998294
```

```

val r : Owl_dense_matrix_d.mat =
  C0 C1 C2 C3 C4
  R0 -0.91393 0.196148 0.0738038 1.45659 -0.268024
  R1 0 0 0.463548 0.286501 1.38499 -0.0595374
  R2 0 0 0.346057 0.954629 0.167467
  R3 0 0 0 -1.56104 -0.124984
  R4 0 0 0 0.555067

# Mat.(u *@ d1 *@ r *@ transpose q =~ x)
- : bool = true
# Mat.(v *@ d2 *@ r *@ transpose q =~ y)
- : bool = true

```

The SVD is not only important linear algebra concept, but also has a wide and growing applications. For example, the Moore-Penrose pseudo-inverse that works for non-invertible matrix can be implemented efficiently using SVD (we provide `pinv` function in the linear algebra module for the pseudo inverse). It can also be used for information compression such as in image processing. As we have said, in the Natural Language Processing chapter we will see how SVD plays a crucial role in the language processing field to perform principal component analysis.

Internal: CBLAS and LAPACKE

This section is for those of you who are eager for more low level information. The BLAS (Basic Linear Algebra Subprogramms) is a specification that describes a set of low-level routines for common linear algebra operation. The LAPACKE contains more linear algebra routines, such as solving linear systems and matrix factorisations, etc. Efficient implementation of these function has been practices for a long time in many softwares. Interfacing to them can provide easy access to high performance routines.

Low-level Interface to CBLAS & LAPACKE

Owl has implemented the full interface to CBLAS and LAPACKE. Comparing to Julia which chooses to interface to BLAS/LAPACK, you might notice the extra `c` in `CBLAS` and `E` in `LAPACKE` because they are the corresponding C-interface of Fortran implementations. It is often believed that C-interface may introduce some extra overhead. However, it turns out that we cannot really notice any difference at all in practice when dealing with medium or large problems.

- `Owl_cblas` module provides the raw interface to CBLAS functions, from level-1 to level-3. The interfaced functions have the same names as those in CBLAS.
- `Owl_lapacke_generated` module provides the raw interface to LAPACKE functions (over 1,000) which also have the same names defined in `lapacke.h`.

- Owl_lapacke module is a very thin layer of interface between Owl_lapacke_generated module and Linalg module. The purpose is to provide a unified function to make generic functions over different number types.

The functions in Owl_cblas and Owl_lapacke_generated are very low-level, e.g., you need to deal with calculating parameters, allocating workspace, post-processing results, and many other tedious details. You do not really want to use them directly unless you have enough background in numerical analysis and chase after the performance. So for example, the LU factorisation is performed using the `sgetrf` or `dgetrf` function in the `owl_lapacke_generated` module, the signature of which look like this:

```
val sgetrf: layout:int -> m:int -> n:int -> a:float ptr -> lda:int -> ipiv:int32
ptr -> int
```

Instead of worrying about all these parameters, the `getrf` function in the `owl_lapacke` module provides interface that are more straightforward:

```
val getrf : a:('a, 'b) t -> ('a, 'b) t * (int32, int32_elt) t
```

These low-level functions provides more general access for users. If this still looks a bit unfamiliar to your, in the `Linalg` module we have:

```
val lu : ('a, 'b) t -> ('a, 'b) t * ('a, 'b) t * (int32, int32_elt) t
```

Here the function `lu x -> (l, u, ipiv)` calculates LU decomposition of input matrix `x`, and returns the `L`, `U` matrix together with the pivoting index. In practice, you should always use `Linalg` module which gives you a high-level wrapper for frequently used functions.

Besides these function, the linear algebra module also provides some helper functions. For example, the `peakflops ~n ()` function returns the peak number of float point operations using `Owl_cblas_basic.dgemm` function. The default matrix size is `2000 x 2000`, but the user can change this by setting `n` arguments.

Sparse Matrices

What we have mentioned so far are dense matrix. But when the elements are sparsely distributed in the matrix, such as the identity matrix, the *sparse* structure might be more efficient. The sparse matrix is proived in the `Sparse.Matrix` module, and also support the four types of number in the `S`, `D`, `C`, and `Z` submodules.

(Perhaps these contents are better to discuss in Ndarray module.)

Very brief. Focusing on introducing the data structure (CSC, CSR, etc), no the method. Mention the owl_suitesparse TODO: Introduce the sparse data structure in owl, and introduce CSR, CSC, tuples, and other formats.

Summary

References

Strang, Gilbert. 2006. *Linear Algebra and Its Applications*. Belmont, CA: Thomson, Brooks/Cole. <http://www.amazon.com/Linear-Algebra-Its-Applications-Edition/dp/0030105676>.

Ordinary Differential Equations

A *differential equation* is an equation that contains a function and one or more of its derivatives. It is studied ever since the invention of calculus, driven by the applications in mechanics, astronomy, and geometry. Currently it has become an important branch of mathematics study and its application is widely extended to biology, engineering, economics, and much more fields. In a differential equation, if the function and its derivatives are about only one variable, we call it an *Ordinary Differential Equation* (ODE). It is often used to model one-dimensional dynamical systems. Otherwise it is an *Partial Differential Equation* (PDE). In this chapter we focus on the ODE and introduce what is it, how it can be solved numerically, and the tools we provide to do that in Owl.

What Is An ODE

Generally, an ODE can be expressed as:

$$F(x, y', y'', \dots, y^{(n)}) = 0. \quad (12)$$

The differential equations model dynamic systems, and the initial status of the system is often known. That is called *initial values*. They can be represented as:

$$y|_{x=x_0} = y_0, y'|_{x=x_1} = y_1, \dots, \quad (13)$$

where the y_0 , y_1 , etc. are known. The highest order of derivatives that are used in eq. 12 is the *order* of this differential equation. A first-order differential equation can be generally expressed as: $\frac{dy}{dx} = f(x, y)$, where f is any function that contains x and y . Solving eq. 12 that fits the given initial values as in eq. 13 is called the *initial value problem*. Solving problems of this kind is the main target of many numerical ODE solvers.

Exact Solutions

Solving a differential equation is often complex, but we do know how to solve part of them. Before looking at the computer solvers to a random ODEs, let's turn to the math first and look at some ODE forms that we already have analytical close-form solution to.

Table 15: Examples of solutions to certain types of ODE

ODE	Solution
$P(y)\frac{dy}{dx} + Q(x) = 0$	$\int^y P(y)dy + \int^x Q(x)dx = C$
$\frac{dy}{dx} + P(x)y = Q(x)$	$y = e^{-\sum_{x_0}^x P(x)dx} (y_0 + \sum_{x_0}^x Q(x)e^{\sum_{x_0}^x P(x)dx} dx)$

The tbl. 15 shows two examples. The first line is a type of ODEs that are called the “separable equations”. The second line represents the ODEs that are called the “linear first-order equations”. The solutions to both form of ODE are already well-known, as shown in the second column. Here C is a constant decided by initial condition x_0 and y_0 . $P(x)$ and $Q(x)$ are functions that contain only variable x . Note that in both types the derivative dy/dx can be expressed explicitly as a function of x and y , and therefore is called *explicit* ODE. Otherwise it is called an *implicit* ODE.

High order ODEs can be reduced to the first order ones that contains only y' , y , and x . For example, an ODE in the form $y^{(n)} = f(x)$ can be reduced by multiple integrations one both sides. If a two-order ODE is in the form $y'' = f(x, y')$, let $y' = g(x)$, and then $y'' = p'(x)$. Put them into the original ODE, and it can be transformed as: $p' = f(x, p)$. This is a first-order ODE that can be solved by normal solutions. Suppose we get $y' = p = h(x, C_0)$; this explicit form of ODE can be integrated to get: $y = \int h(x, C_0) dx + C_1$.

We have only scratched the surface of the ODE as traditional mathematics topic. This chapter does not aim to fully introduce how to solve ODEs analytically or simplify high-order ODEs. Please refer to classical calculus books or courses for more detail.

Linear Systems

ODEs are often used to describe various dynamic systems. In the previous examples there is only one function y that changes over time. However, a real world system often contains multiple interdependent components, and each can be described by a unique function that evolves over time. In the next of this chapter, we will talk about several ODE examples in detail, such as the two-body problem and the Lorenz attractor. For now, it suffices for us to look at eq. 22 and eq. 23 in the sections below and see how they are different from the single-variant ODE so far. For example, the Lorenz attractor system has three components that change with time: the rate of convection in the atmospheric flow, the horizontal and vertical temperature variation.

These two systems are examples of what is called the *first-order linear system of ODE* or just the *linear system of ODE*. Generally, if we have:

$$\mathbf{y}(t) = \begin{bmatrix} y_1(t) \\ \vdots \\ y_n(t) \end{bmatrix}, \mathbf{A}(t) = \begin{bmatrix} a_{11}(t) & \dots & a_{1n}(t) \\ \vdots & \dots & \vdots \\ a_{n1}(t) & \dots & a_{nn}(t) \end{bmatrix}, \text{ and } \mathbf{g}(t) = \begin{bmatrix} g_1(t) \\ \vdots \\ g_n(t) \end{bmatrix},$$

then a linear system can be expressed as:

$$\mathbf{y}'(t) = \mathbf{A}(t)\mathbf{y}(t) + \mathbf{g}(t). \quad (14)$$

This linear system contains n time-dependent components: $y_1(t), y_2(t), \dots, y_n(t)$. As we will be shown soon, the first-order linear system is especially suitable for the numerical ODE solver to solve. Therefore, transforming a high-order single-component ODE into a linear system is sometimes necessary, as we will show in the two body problem example. But before we stride too far away, let's get back to the ground and start with the basics of solving an ODE numerically.

Solving An ODE Numerically

This section introduces the basic idea of solving the initial value problem numerically. Let's start with an example:

$$y' = 2xy + x, \quad (15)$$

where the initial value is $y(0) = 0$. Without going deep into the whole math calculation process (hint: it's a separable first-order ODE), we give its analytical close-form solution:

$$y = 0.5(e^{x^2} - 1). \quad (16)$$

Now, pretending we don't know the solution in eq. 16, we want to answer the question: what is y 's value when $x = 1$ (or any other value)? How can we solve it numerically?

Enter the *Euler Method*, a first-order numerical procedure to solve initial value problems. The basic idea is simple: according to eq. 15, we know the derivative, i.e., the "slope" at any given point on the function curve. Besides, we also know the initial value x_0 and y_0 of this function. We can then simply move from the initial point to the target x value in small steps, and at every new point we adjust the direction according to derivative. Formally, the Euler method proposes to approximate the function y using a sequence of iterative steps:

$$y_{n+1} = y_n + \Delta f(x_n, y_n),$$

where Δ is a certain step size. This method is really easy to be implemented in OCaml, as shown below.

```

let x = ref 0.
let y = ref 0.
let target = 1.
let step = 0.001
let f x y = 2. *. x *. y +. x

let _ =
  while !x <= target do
    y := !y +. step *. (f !x !y);
  done;
  !y

```

```
x := !x +. step
done
```

In this case, we know that the analytical solution at $x = 1$ is $0.5(e^{1^2} - 1)$:

```
# (Owl_const.e -. 1.)/. 2.
- : float = 0.859140914229522545
```

and the solution given by the previous numerical code is about 0.8591862 , which is pretty close to the true answer.

However, this method is as easy as it is unsuitable to be used in practical applications. One reason is that this method is not very accurate, despite that it works well in our example here. We will show this point soon. Also, it is not very stable, nor does it provide error estimate. Therefore, we can modify the Euler's method to use a “midpoint” in stepping, hoping to curb the error in the update process:

$$\begin{aligned}s_1 &= f(x_n, y_n), \\ s_2 &= f(x_n + \Delta /2, y_n + s_1 \Delta /2), \\ y_{n+1} &= y_n + \Delta \frac{s_1 + s_2}{2}.\end{aligned}\tag{17}$$

This method is called the *Midpoint Method*, and we can also implement it in OCaml similarly. Let's compare the performance of Euler and Midpoint in approximating the true result in eq. 16:

```
let f x y = 2. *. x *. y +. x
let f' x = 0.5 *. (Maths.exp (x *. x) -. 1.)

let euler step target =
  let x = ref 0. in
  let y = ref 0. in
  while !x <= target do
    y := !y +. step *. (f !x !y);
    x := !x +. step
  done;
  !y

let midpoint step target =
  let x = ref 0. in
  let y = ref 0. in
  while !x <= target do
    let s1 = f !x !y in
    let s2 = f (!x +. step /. 2.) (!y +. step /. 2. *. s1) in
    y := !y +. step *. (s1 +. s2) /. 2.;
    x := !x +. step
  done;
```

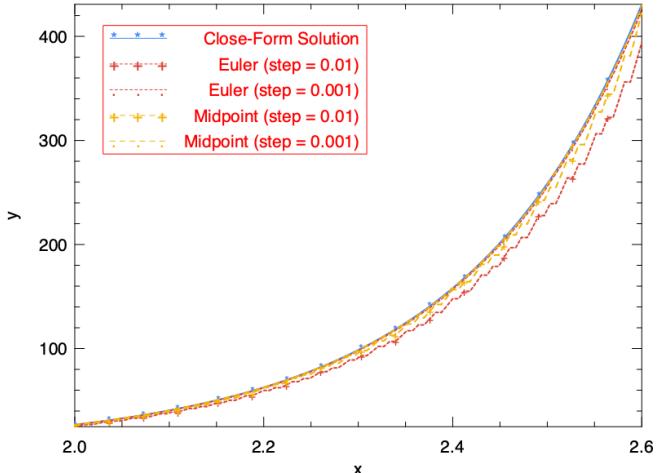


Figure 44: Comparing the accuracy of Euler method and Midpoint method in approximating solution to ODE

```
!y

let _ =
  let target = 2.6 in
  let h = Plot.create "plot_rk01.png" in
  Plot.(plot_fun ~h ~spec:[ RGB (66,133,244); LineStyle 1; LineWidth 2.; Marker
    "*" ] f' 2. target);
  Plot.(plot_fun ~h ~spec:[ RGB (219,68,55); LineStyle 2; LineWidth 2.; Marker
    "+" ] (euler 0.01) 2. target);
  Plot.(plot_fun ~h ~spec:[ RGB (219,68,55); LineStyle 2; LineWidth 2.; Marker
    "." ] (euler 0.001) 2. target);
  Plot.(plot_fun ~h ~spec:[ RGB (244,180,0); LineStyle 3; LineWidth 2.; Marker
    "+" ] (midpoint 0.01) 2. target);
  Plot.(plot_fun ~h ~spec:[ RGB (244,180,0); LineStyle 3; LineWidth 2.; Marker
    "." ] (midpoint 0.001) 2. target);
  Plot.(Legend_on h ~position:NorthWest [| "Close-Form Solution"; "Euler (step =
    0.01)";
    "Euler (step = 0.001)"; "Midpoint (step = 0.01)"; "Midpoint (step =
    0.001)" |]);
  Plot.output h
```

Let's see the result.

We can see that the choice of step size indeed matters to the precision. We use 0.01 and 0.001 for step size in the test, and for both cases the midpoint method outperforms the simple Euler method.

Should we stop now? Do we find a perfect solution in midpoint method? Surely no. We can follow the existing trend and add more intermediate stages in the

update sequence. For example, we can do this:

$$\begin{aligned}
 s_1 &= f(x_n, y_n), \\
 s_2 &= f(x_n + \Delta /2, y_n + s_1 \Delta /2), \\
 s_3 &= f(x_n + \Delta /2, y_n + s_2 \Delta /2), \\
 s_4 &= f(x_n + \Delta, y_n + s_3 \Delta), \\
 y_{n+1} &= y_n + \Delta \frac{s_1 + 2s_2 + 2s_3 + s_4}{6}.
 \end{aligned} \tag{18}$$

Here in each iteration four intermediate steps are computed: once at the initial point, once at the end, and twice at the midpoints. This method is often more accurate than the midpoint method.

We can keep going on like this, but hopefully you have seen the pattern so far. These seemingly mystical parameters are related to the term in Taylor series expansions. In the previous methods, e.g. Euler method, every time you update y_n to y_{n+1} , an error is introduced into the approximation. The *order* of a method is the exponent of the smallest power of Δ that cannot be matched. All these methods are called *Runge-Kutta Methods*. The basic idea is to remove the errors order by order, using the correct set of coefficients. A higher order of error indicates smaller error.

The Euler is the most basic form of Runge-Kutta (RK) method, and the Midpoint is also called the second-order Runge-Kutta Method (rk2). What eq. 18 shows is a fourth-order Runge-Kutta method (rk4). It is the most frequently used RK method and works surprisingly well in many cases, and it is often a good choice especially when computing f is not expensive.

However, as powerful as it may be, the classical rk4 is still a native implementation. A modern ODE solver, though largely follows the same idea, adds more “ingredients”. For example, the step size should be adaptively updated instead of being constant as in our example. Also, you may have seen solvers with names such as ode45 in MATLAB, and in their implementation, it means that this solver gets its error estimate at each step by comparing the 4th order solution and 5th order solution and then decides the direction.

Besides, other methods also exist. For example, the Adams-Bashforth Method and Backward Differentiation Formula (BDF) are both multi-step methods that utilise not just the information such as derivative of the current step, but also of previous time steps to compute the solution at next step. In recent years, the Bulirsch-Stoer method is known to be both accurate and efficient computation-wise. Discussion of these advanced numerical methods and techniques are beyond the scope of this book. Please refer to (Press et al. 2007) for more information.

Owl-ODE

Obviously, we cannot just rely on these manual solutions every time in practical use. It's time to use some tools. Based on the computation functionalities and ndarray data structures in Owl, we provide the package "owl_ode" to perform the tasks of solving the initial value problems. Let's start by seeing how the owl_ode package can be used to solve ODE problems.

Example: Linear Oscillator System

Here is a time independent linear dynamic system that contains two states:

$$\frac{dy}{dt} = Ay, \text{ where } A = \begin{bmatrix} 1 & -1 \\ 2 & -3 \end{bmatrix}. \quad (19)$$

This equation represents an oscillator system. In this system, y is the state of the system, and t is time. The initial state at $t = 0$ is $y_0 = [-1, 1]^T$. Now we want to know the system state at $t = 2$. The *function* can be expressed in Owl using the matrix module.

```
let f y t =
  let a = [| [1.; -1.]; [2.; -3.] |] > Mat.of_arrays in
  Mat.(a *@ y)
```

Next, we want to specify the *timespan* of this problem: from 0 to 2, at a step of 0.001.

```
let tspec = Owl_ode.Types.(T1 {t0 = 0.; duration = 2.; dt=1E-3})
```

One last thing to solve the problem is the *initial values*:

```
let x0 = Mat.of_array [| -1.; 1. |] 2 1
```

And finally we can provide all these information to the rk4 solver in `owl_ode` and get the answer:

```
# let ts, ys = Owl_ode.Ode.odeint Owl_ode.Native.D.rk4 f x0 tspec ()
val ts : Owl_dense_matrix_d.mat =
  C0 C1 C2 C3 C4 C1996 C1997 C1998 C1999 C2000
  R0 0 0.001 0.002 0.003 0.004 ... 1.996 1.997 1.998 1.999 2
val ys : Owl_dense_matrix_d.mat =
  C0 C1 C2 C3 C4 C1996 C1997 C1998 C1999 C2000
```

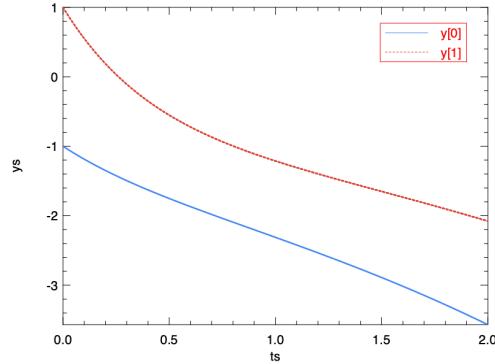


Figure 45: Visualise the solution of a simple linear system

```
R0 -1 -1.002 -1.00399 -1.00599 -1.00798 ... -3.56302 -3.56451 -3.566 -3.56749
      -3.56898
R1 1 0.995005 0.990022 0.985049 0.980088 ... -2.07436 -2.07527 -2.07617 -2.07707
      -2.07798
```

The `rk4` solver is short for “forth-order Runge-Kutta Method” that we have introduced before. The result shows both the steps ts and the system values at each step ys . We can visualise the oscillation according to the result:

Solver Structure

Hope that you have gotten the gist of how to use `owl-ode`. From these examples, we can see that the `owl-ode` abstracts the initial value problems as four different parts:

1. a function f to show how the system evolves in equation $y'(t) = f(y, t)$;
2. a specification of the timespan;
3. system initial values;
4. and most importantly, a solver.

If you look at the signature of a solver:

```
val rk4 : (module Types.Solver
  with type state = M.arr
  and type f = M.arr -> float -> M.arr
  and type step_output = M.arr * float
  and type solve_output = M.arr * M.arr)
```

it clearly indicates these different parts. Based on this uniform abstraction, you can choose a suitable solver and use it to solve many complex and practical ODE problems. Note that due to the difference of solvers, the requirement of different

solver varies. Some requires the state to be two matrices, while others process data in a more general ndarray format.

`owl-ode` provides a wide range of solvers. It implements native solvers which are based on the step-by-step update basic idea we have discussed. Currently there are already many mature off-the-shelf tools for solving ODEs. We choose to interface to two of them: sundials and ODEPACK. Both methods are well implemented and widely used in practical use. For example, the SciPy provides a Python wrap of the sundials, and the NASA also uses its CVODE/CVODES solvers for spacecraft trajectory simulations.

- `sundials`: a SUite of Nonlinear and DIfferential/ALgebraic equation Solvers. It contains six solvers, and we interface to its CVODE solver for solving initial value problems for ordinary differential equation systems.
- `odepack`: ODEPACK is a collection of FORTRAN solvers for the initial value problem for ordinary differential equation systems. We interface to its LSODA solver which is for solving the explicit form ODE.

For all these solvers, `owl-ode` provides an easy-to-use unified interface, as you have seen in the examples. [tbl. 16](#) is a table that lists all the solvers that are currently supported by `owl-ode`.

Table 16: Solvers provided by `owl-ode` and their types.

Solvers	Type	Function	Output
Euler/Midpoint	Native	mat -> float -> mat	mat * mat
rk4/rk23/rk45	Native	mat -> float -> mat	mat * mat
Cvode/Cvode_stiff	Sundials	arr -> float -> arr	arr * arr
LSODA	ODEPACK	mat -> float -> mat	mat * mat

The solvers share similar signature. The system evolve function takes state y and float time t as input, and the output is a tuple that contains two matrices or ndarrays that represent the time increment and the state of y at corresponding time step.

In most of the cases, `ode45` is a robust choice and the first solver to try. It is robust and has medium accuracy. If the problem has certain level of tolerance to accuracy, you can also try the other native solvers which are more efficient in computing. If the requirement for accuracy or stability is even higher, the Sundials/ODEPACK solvers can be used, especially when the ODEs to be solved are “stiff”. We will talk about the *stiffness* of ODEs in later section.

Symplectic Solvers

Besides what we have mentioned, `owl-ode` also implements the *symplectic solvers*, which are a bit different than the native solvers we have seen so far. Symplectic

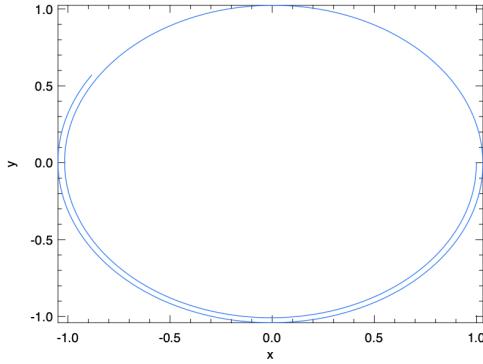


Figure 46: Visualise the circle trajectory by solving linear system

solvers are used to solve the *Hamiltonian dynamics* with much higher accuracy. We will show what it means with a example.

Think about a dynamic orbiting system where the trajectory is exactly a circle. Such a circle can be generated by solving the linear system:

$$y'_0 = y_1; y'_1 = -y_0 \quad (20)$$

This can also be expressed in the matrix form:

$$y' = Ay, \text{ where } A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}.$$

We can then apply the previous methods such as Euler to solve it:

```
let f y t =
  let a = [| [| 0.; 1.|]; [| -1.; 0.|] |]> Mat.of_arrays in
  Mat.(a *@ y)

let x0 = Mat.of_array [| 1.; 0.|] 2 1;;
let tspec = Owl_ode.Types.(T1 {t0 = 0.; duration = 10.; dt=1E-2});;
let ts, ys = Owl_ode.Ode.odeint Owl_ode.Native.D.euler f x0 tspec ()
```

The resulting trajectory by plotting the two components of the y_s is show in fig. 46. An exact solution would generate a perfect circle, but here we only get a spiral that is similar to a circle. (We have to admit that to better show this effect a large step size is used, but it still exists even using smaller step size.) Aside from improving the solver and precision, is there any other way to solve it better?

Look again at eq. 20. It shows an interesting pattern: it first uses the value of y_1 to update the y'_0 , and then uses y_0 to update the value of y'_1 . It belongs to a class of equations that are called the *Hamiltonian systems* which is often used in the classic mechanics. Such a system is represented by a set of coordinates (p, q) . The q_i 's are called the generalised coordinates, and p_i 's are corresponding conjugate momenta. The evolution of a Hamiltonian system is defined by the equations below:

$$p' = \frac{\partial H}{\partial q}, q' = \frac{\partial H}{\partial p}.$$

Here $H(p, q)$ is a function of these two components. In the example above, p and q are just the two coordinates, and $H(x, y) = (x^2 + y^2)/2$. To solve this system, the symplectic solvers are widely used in different fields, since they conserve the value of $H(p, q)$, to within the order of accuracy of the specific solver used. Most of the usual numerical solvers we have mentioned, such as the Euler and classical RK solvers, are not symplectic solvers.

In `owl_ode` we have implemented several symplectic solvers that are based on different integration algorithms: `Sym_Euler`, `Leapfrog`, `PseudoLeapFrog`, `Ruth3`, and `Ruth4`. Like the native solvers, they have different orders of error. These algorithms are implemented based on the same basic symplectic integration algorithm, with different parameters. If you are interested in more detail, please check the code and the paper (Candy and Rozmus 1991) which our implementation is based on. Later we will show an example of using the symplectic solver to solve a damped harmony oscillation problem.

Features and Limits

One feature of `owl_ode` is the automatic inference of state dimensionality from initial state. For example, the native solvers takes matrix as state. Suppose the initial state of the system is a row vector of dimension $1 \times N$. After T time steps, the states are stacked vertically, and thus have dimensions $T \times N$. If the initial state is a column vector of shape $N \times 1$, then the stacked state after T time steps will be inferred as $N \times T$.

The temporal integration of matrices, i.e. cases where the initial state is matrix instead of vector, is also supported. If the initial state is of shape $N \times M$, then the accumulated state stacks the flattened state vertically by time steps, which is of shape $T \times (NM)$. The `owl_ode` provides a helper function `Native.S.to_state_array` to unpack the output state into an array of matrices.

Another feature of `owl_ode` is that the users can easily define new solver module by creating a module of type `Solver`. For example, to create a custom Cvode solver that has a relative tolerance of 1E-7 as opposed to the default 1E-4, we can define and use the `custom_cvode` as follows:

```
let custom_ccode = Owl_ode_sundials.ccode ~stiff:false ~relative_tol:1E-7
  ~abs_tol:1E-4
let ts, xs = Owl_ode.Ode.odeint custom_ccode f x0 tspec ()
```

Here, we use the `ccode` function to construct a solver module `Custom_Owl_Code`. Similar helper functions like `ccode` have been also defined for native and symplectic solvers.

We aim to make the `owl-ode` to run across different backends such as JavaScript. One way to do this is to use tools like `js_of_ocaml` to convert OCaml bytecode into JavaScript. However, this approach only supports pure OCaml implementation. The sundial and ODEPACK solvers are therefore excluded. Part of the library, the `owl-ode-base` contains implementations of solvers that are purely written in OCaml. These parts are therefore suitable to be used for in JavaScript and executed on web browsers. Another backend option is the Unikernl virtual machine such as MirageOS. We will talk about these backends in the “Compiler Backends” chapter in the Part II of this book.

One limit of this library is that it is still in a development phase and thus a lot of features is not in place yet. For example, due to the lack of vector-valued root finding functions, we are currently limited to solving initial value problems of explicit ODEs.

Examples of using Owl-ODE

As with many good things in the world, mastering solving ODE requires practice. After getting to know `owl-ode`, in this section we will demonstrate more examples of using this tool.

Explicit ODE

Now that we have this powerful tool, we can use the solver in `owl-ode` to solve the motivative problem in eq. 15 with simple code.

```
let f y t = Mat.((2. *. y ** t) +$ t)
let tspec = Owl_ode.Types.(T1 {t0 = 0.; duration = 1.; dt=1E-3})
let y0 = Mat.zeros 1 1
let solver = Owl_ode.Native.D.rk45 ~tol:1E-9 ~dtmax:10.0
let _, ys = Owl_ode.Ode.odeint solver f y0 tspec ()
```

The code is mostly similar to previous example, the only difference is that we can now try another solver provided: the `rk45` solver, with certain parameters specified. You don’t have to worry about what the `tol` or `dtmax` means for now. Note that this solver (and the previous one) requires input to be of type `mat` in

Owl, and the function f be of type `mat -> float -> mat`. The result is shown below. You can verify the result with eq. 16, by setting the x to 1 in this equation, and the numerical value of y will be close to 0.859079.

```
# Mat.transpose ys
- : Mat.mat =
C0 C1 C2 C3 C4 C124 C125 C126 C127 C128
R0 0 7.62941E-06 9.91919E-05 0.000251833 0.00046561 ... 0.777984 0.797555 0.817586
0.83809 0.859079
```

Two Body Problem

In classical mechanics, the *two-body problem* is to predict the motion of two massive objects. It is assumed that the only force that is considered comes from each other, and both objects are not affected by any other object. This problem can be seen in the astrodynamics where the objects of interests are planets, satellites, etc. under the influence of only gravitation. Another case is the trajectory of electron around Atomic nucleus in a atom.

This classic problem is one of the earliest investigated mechanics problems, and was long solved from the age of Newton. It is also a typical integrable problem in classical mechanics. In this example, let's consider a simplified version of this problem. We assume that the two objects interact on a 2-dimensional plane, and one of them is so much more massive than the other one that it can be thought of as being static (think about electron and nucleus) and sits at the zero point of a Cartesian coordinate system $(0, 0)$ in the plane. In this system, let's consider the trajectory of the lighter object. This “one-body” problem is basis of the two body problem. For many forces, including gravitational ones, a two-body problem can be divided into a pair of one-body problems.

Given the previous assumption and newton's equation, it can be proved that the location of the lighter object $[y_0, y_1]$ with regard to time t can be described by:

$$\begin{aligned} y_0''(t) &= -\frac{y_0}{r^3}, \\ y_1''(t) &= -\frac{y_1}{r^3}, \end{aligned} \tag{21}$$

where $r = \sqrt{y_0^2 + y_1^2}$. This is a second-order ODE system, and to make it solvable using our tool, we need to make it into a first-order explicit ordinary differential equation system:

$$y_0' = y_2,$$

$$y_1' = y_3,$$

$$\begin{aligned} y_2' &= -\frac{y_0}{r^3}, \\ y_3' &= -\frac{y_1}{r^3}, \end{aligned} \tag{22}$$

Based on eq. 22, we can build up our code as below:

```
let f y _t =
  let y = Mat.to_array y in
  let r = Maths.(sqrt ((sqr y.(0)) +. (sqr y.(1)))) in
  let y0' = y.(2) in
  let y1' = y.(3) in
  let y2' = -.y.(0) /. (Maths.pow r 3.) in
  let y3' = -.y.(1) /. (Maths.pow r 3.) in
  [| [y0'; y1'; y2'; y3'] |] > Mat.of_arrays

let y0 = Mat.of_array [-1.; 0.; 0.5; 0.5] 1 4
let tspec = Owl_ode.Types.(T1 {t0 = 0.; duration = 20.; dt=1E-2})
let custom_solver = Native.D.rk45 ~tol:1E-9 ~dtmax:10.0
```

Here the y_0 provides initial status of the system: first two numbers denote the initial location of object, and the next two numbers indicate the initial momentum of this object. After building the function, initial status, timespan, and solver, we can then solve the system and visualise it.

```
let plot () =
  let ts, ys = Ode.odeint custom_solver f y0 tspec () in
  let h = Plot.create "two_body.png" in
  let open Plot in
  plot ~h ~spec:[ RGB (66, 133, 244); LineStyle 1 ] (Mat.col ys 0) (Mat.col ys 1);
  scatter ~h ~spec:[ Marker "#[0x229a]", MarkerSize 5. ] (Mat.zeros 1 1)
    (Mat.zeros 1 1);
  text ~h ~spec:[ RGB (51,51,51) (-.0.3) 0. "Massive Object"];
  output h
```

One example of this simplified two-body problem is the “planet-sun” system where a planet orbits the sun. Kepler’s law states that in this system the planet goes around the sun in an ellipse shape, with the sun at a focus of the ellipse. The orbiting trajectory in the result visually follows this theory.

Lorenz Attractor

Lorenz equations are one of the most thoroughly studied ODEs. This system of ODEs is proposed by Edward Lorenz in 1963 to model the flow of fluid (the air in particular) from hot area to cold area. Lorenz simplified the numerous atmosphere factors into the simple equations below.

$$x'(t) = \sigma (y(t) - x(t))$$

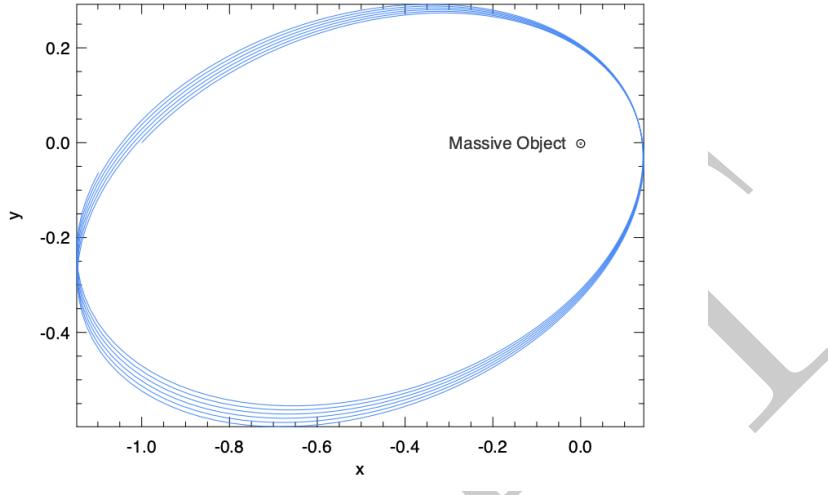


Figure 47: The trajectory of lighter object orbiting the massive object in a simplified two-body problem

$$\begin{aligned} y'(t) &= x(t)(\rho - z(t)) - y(t) \\ z'(t) &= x(t)y(t) - \beta z(t) \end{aligned} \tag{23}$$

Here x is proportional to the rate of convection in the atmospheric flow; y and z are proportional to the horizontal and vertical temperature variation. Parameter σ is the Prandtl number, and ρ is the normalised Rayleigh number. β is related to the geometry of the domain. The most commonly used parameter values are: $\sigma = 10$, $\rho = 20$, and $\beta = \frac{8}{3}$. Based on this information, we can use `owl-ode` to express the Lorenz equations with code.

```
let sigma = 10.
let beta = 8. /. 3.
let rho = 28.

let f y _t =
  let y = Mat.to_array y in
  let y0' = sigma *. (y.(1) -. y.(0)) in
  let y1' = y.(0) *. (rho -. y.(2)) -. y.(1) in
  let y2' = y.(0) *. y.(1) -. beta *. y.(2) in
  [| [y0'; y1'; y2'|] |] > Mat.of_arrays
```

We set the initial values of the system to -1 , -1 , and 1 respectively. The simulation timespan is set to 30 seconds, and the `rk45` solver is used.

```

let y0 = Mat.of_array [| -1.; -1.; 1. |] 1 3
let tspec = Owl_ode.Types.(T1 {t0 = 0.; duration = 30.; dt=1E-2})
let custom_solver = Native.D.rk45 ~tol:1E-9 ~dtmax:10.0

```

Now, we can solve the ODEs system and visualise the results. In the plots, we first show how the value of x , y and z changes with time; next we show the phase plane plots between each two of them.

```

let _ =
  let ts, ys = Ode.odeint custom_solver f y0 tspec () in
  let h = Plot.create ~m:2 ~n:2 "lorenz_01.png" in
  let open Plot in
  subplot h 0 0;
  set_xlabel h "time";
  set_ylabel h "value on three axes";
  plot ~h ~spec:[ RGB (66, 133, 244); LineStyle 1 ] ts (Mat.col ys 2);
  plot ~h ~spec:[ RGB (219, 68, 55); LineStyle 1 ] ts (Mat.col ys 1);
  plot ~h ~spec:[ RGB (244, 180, 0); LineStyle 1 ] ts (Mat.col ys 0);
  subplot h 0 1;
  set_xlabel h "x-axis";
  set_ylabel h "y-axis";
  plot ~h ~spec:[ RGB (66, 133, 244) ] (Mat.col ys 0) (Mat.col ys 1);
  subplot h 1 0;
  set_xlabel h "y-axis";
  set_ylabel h "z-axis";
  plot ~h ~spec:[ RGB (66, 133, 244) ] (Mat.col ys 1) (Mat.col ys 2);
  subplot h 1 1;
  set_xlabel h "x-axis";
  set_ylabel h "z-axis";
  plot ~h ~spec:[ RGB (66, 133, 244) ] (Mat.col ys 0) (Mat.col ys 2);
  output h

```

From fig. 48, we can imagine that the status of system keep going towards two “voids” in a three dimensional space, jumping from one to the other. These two voids are a certain type of *attractors* in this dynamic system, where a system tends to evolve towards.

Now, about Lorenz equation, there is an interesting question: “what would happen if I change the initial value slightly?” For some systems, such as a pendulum, that wouldn’t make much a difference, but not here. We can see that clearly with a simple experiment. Keeping function and timespan the same, let’s change only 0.1% of initial value and then solve the system again.

```

let y00 = Mat.of_array [| -1.; -1.; 1. |] 1 3
let y01 = Mat.of_array [| -1.001; -1.001; 1.001 |] 1 3
let ts0, ys0 = Ode.odeint custom_solver f y00 tspec ()
let ts1, ys1 = Ode.odeint custom_solver f y01 tspec ()

```

To make later calculation easier, we can set the two resulting matrices to be of the same shape using slicing.

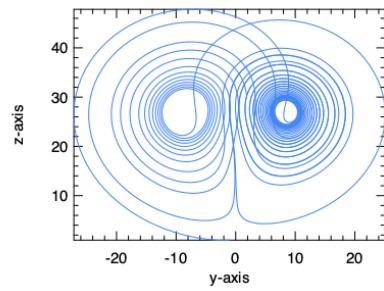


Figure 48: Three components and phase plane plots of Lorenz attractor

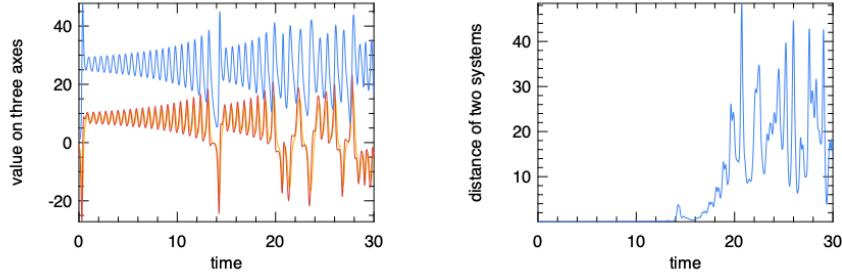


Figure 49: Change the initial states on three dimension by only 0.1%, and the value of Lorenz system changes visibly.

```

let r0, c0 = Mat.shape ys0
let r1, c1 = Mat.shape ys1
let r = if (r0 < r1) then r0 else r1
let ts = if (r0 < r1) then ts0 else ts1
let ys0 = Mat.get_slice [[0; r-1]; []] ys0
let ys1 = Mat.get_slice [[0; r-1]; []] ys1

```

Now, we can compare the Euclidean distance between the status of these two systems at certain time. Also, we show the value change of the three components after changing initial values along the time axis.

```

let _ =
let h = Plot.create ~m:1 ~n:2 "lorenz_02.png" in
let open Plot in
subplot h 0 0;
set_xlabel h "time";
set_ylabel h "value on three axes";
plot ~h ~spec:[ RGB (244, 180, 0); LineStyle 1 ] ts (Mat.col ys1 0);
plot ~h ~spec:[ RGB (219, 68, 55); LineStyle 1 ] ts (Mat.col ys1 1);
plot ~h ~spec:[ RGB (66, 133, 244); LineStyle 1 ] ts (Mat.col ys1 2);
subplot h 0 1;
let diff = Mat.(
  sqr ((col ys0 0) - (col ys1 0)) +
  sqr ((col ys0 1) - (col ys1 1)) +
  sqr ((col ys0 2) - (col ys1 2))
  |> sqrt
)
in
plot ~h ~spec:[ RGB (66, 133, 244); LineStyle 1 ] ts diff;
set_xlabel h "time";
set_ylabel h "distance of two systems";
output h

```

According to fig. 49, the first figure shows that, initially the systems looks quite like that in fig. 48, but after about 15 seconds, the system state begins to change. This change is then quantified using the Euclidean distance between these two systems. Clearly the difference two system changes sharply after a certain period of time, with no sign of converge. You can try to extend the timespan longer, and the conclusion will still hold.

This result shows that, in the Lorenz system, even a tiny bit of change of the initial state can lead to a large and chaotic change of future state after a while. It partly explains why weather prediction is difficult to do: you can only accurately predict the weather for a certain period of time, any day longer and the weather will be extremely sensitive to a tiny bit of perturbations at the beginning, such as, well, such as the flapping of the wings of a distant butterfly several weeks earlier. You are right, the Lorenz equation is closely related to the idea we now call “butterfly effect” in the pop culture.

Damped Oscillation

This oscillation system appears frequently in Physics and several other fields: charge flow in electric circuit, sound wave, light wave, etc. These phenomena all follow the similar pattern of ODEs. One example is the mass attached on a spring. The system is placed vertically. First the spring stretches to balance the gravity; once the system reaches equilibrium, we can then study the upward displacement of the mass from its original position, denoted with x . Once the mass is in motion, at any place x , the pulling force on the mass is proportional to displacement x : $F = -kx$. Here k is a positive parameter.

This system is called *simple harmonic oscillator*, which represents an ideal case where F is the only force acting on the mass. However, in a real harmonic oscillation system, there is also the frictional force. Such system is called *damped oscillation*. The frictional force is proportional to the velocity of the mass. Therefore the total force can be expressed as $F = -kx - cv$. Here c is the damping coefficient factor. Since both the force and velocity can be expressed as derivatives of x on time, we have:

$$m \frac{dx}{dt^2} = -kx - c \frac{dx}{dt} \quad (24)$$

Here m represents the mass of the object. Recall from the previous section that the state of the system in a symplectic solver is a tuple of two matrices, representing the position and momentum coordinates of the system. Therefore, we can express the oscillation equation with a function from this system.

```
let a = 1.0
let damped_noforcing a (xs, ps) _t : Owl.Mat.mat =
  Owl.Mat.((xs *$ -1.0) + (ps *$ (-1.0 *. a)))
```

The system evolve function takes the two matrices, position and momentum, and the float number time t as input states. For simplicity, we assume the coefficients are all the same and can be set to one. Let's then solve with the symplectic solver; we can use the `LeapFrog`, `ruth3`, and `Symplectic_Euler` to compare how their solutions differ.

```
let main () =
  let x0 = Owl.Mat.of_array [| -0.25 |] 1 1 in
  let p0 = Owl.Mat.of_array [| 0.75 |] 1 1 in
  let t0, duration = 0.0, 15.0 in
  let f = damped_noforcing a in
  let tspec = T1 { t0; duration; dt=0.1 } in
  let t, sol1, _ = Ode.odeint (module Symplectic.D.Leapfrog) f (x0, p0) tspec () in
  let _, sol2, _ = Ode.odeint Symplectic.D.ruth3 f (x0, p0) tspec () in
  let _, sol3, _ =
    Ode.odeint (module Symplectic.D.Symplectic_Euler) f (x0, p0) tspec ()
  in
  t, sol1, sol2, sol3
```

You should be familiar with this code by now. It defines the initial values, the time duration, time step, and then provides this information to the solvers. Unlike native solvers, these symplectic solvers return three values instead of two: the first is the time sequence, and the next two sequences indicate how the x and p values evolve at each time point.

```
let plot_sol fname t sol1 sol2 sol3 =
  let h = Plot.create fname in
  let open Plot in
  plot ~h ~spec:[ RGB (0, 0, 255); LineStyle 1 ] t (Mat.col sol1 0);
  plot ~h ~spec:[ RGB (0, 255, 0); LineStyle 1 ] t (Mat.col sol2 0);
  plot ~h ~spec:[ RGB (255, 0, 0); LineStyle 1 ] t (Mat.col sol3 0);
  set_xlabel h "time";
  set_ylabel h "x";
  legend_on h ~position:NorthEast [| "Leapfrog"; "Ruth3"; "Symplectic Euler" |];
  output h
```

Here we only plot the change of position x over time in the oscillation, and plot the solution provided by the three different solvers, as shown in the plotting code above. You can also try to visualise the change to the momentum p in a similar way. The result is shown in fig. 50. You can clearly see that the displacement decreases towards equilibrium position in this damped oscillation as the energy dissipating. The curves provided by three solvers are a bit different, especially at the peak of the curve, but keep close enough for most of the time.

Stiffness

Stiffness is an important concept in the numerical solution of ODE. Think about a function that has a “cliff” where at a point its nearby value changes rapidly. To find the solution with normal method as we have used, it may requires such



Figure 50: Step response of a damped harmonic oscillator

an extremely small stepping size that traversing the whole timespan may takes a very long time and lot of computation. Therefore, the solver of stiff equations needs to use different methods such as implicit differencing or automatic step size adjustment etc. to keep it accurate and robust.

The **Van der Pol equation** is a good example to show both non-stiff and stiff cases. In dynamics, the Van der Pol oscillator is a non-conservative oscillator with non-linear damping. Its behaviour with time can be described with a high order ODE:

$$y'' - \mu (1 - y^2)y' + y = 0, \quad (25)$$

where μ is a scalar parameter indicating the non-linearity and the strength of the damping. To make it solvable using our tool, we can change it into a pair of explicit one-order ODEs in a linear system:

$$\begin{aligned} y'_0 &= y_1, \\ y'_1 &= \mu (1 - y_0^2)y_1 - y_0. \end{aligned} \quad (26)$$

As we will show shortly, by varying the damping parameter, this group of equations can be either non-still or stiff.

We provide both stiff (`owl_cvode_Stiff`) and non-stiff (`owl_cvode`) solvers by interfacing to Sundials, and the `LSODA` solver of ODEPACK can automatically switch between stiff and non-stiff algorithms. We will try both in the example. We start with the basic function definition code that are shared by both cases:

```

open Owl_ode
open Owl_ode.Types
open Owl_plplot

let van_der_pol mu =
  fun y _t -
    let y = Mat.to_array y in
    [| [| y.(1); mu *. (1. -. Maths.sqr y.(0)) *. y.(1) -.y.(0) |] |]
  |> Mat.of_arrays

```

Solve Non-Stiff ODEs

When set the parameter to 1, the equation is a normal non-stiff one, and let's try to use the `Cvode` solver from sundials to do this job.

```

let f_non_stiff = van_der_pol 1.

let y0 = Mat.of_array [| 0.02; 0.03 |] 1 2

let tspec = T1 { t0 = 0.0; dt = 0.01; duration = 30.0 }

let ts, ys = Ode.odeint (module Owl_ode_sundials.Owl_Cvode) f_stiff y0 tspec ()

```

Everything seems normal. To see the “non-stiffness” clearly, we can plot how the two system states change over time, and a phase plane plot of their trajectory on the plane, using the two states as x- and y-axis values. The result is shown in fig. 51.

```

let () =
  let fname = "vdp_sundials_nonstiff.png" in
  let h = Plot.create ~n:2 ~m:1 fname in
  let open Plot in
  set_foreground_color h 0 0 0;
  set_background_color h 255 255 255;
  subplot h 0 0;
  plot ~h ~spec:[ RGB (0, 0, 255); LineStyle 1 ] (Mat.col ys 0) (Mat.col ys 1);
  subplot h 0 1;
  plot ~h ~spec:[ RGB (0, 0, 255); LineStyle 1 ] ts Mat.(col ys 1);
  plot ~h ~spec:[ RGB (0, 0, 255); LineStyle 3 ] ts Mat.(col ys 0);
  output h

```

Solve Stiff ODEs

Change the parameters to 1000, and now this function becomes *stiff*. We follow the same procedure as before, but now we use the `Lsoda` solver from `odepack`, and the timespan is extended to 3000. From fig. 52 we can see clearly what “stiff” means. Both lines in this figure contain very sharp “cliffs”.

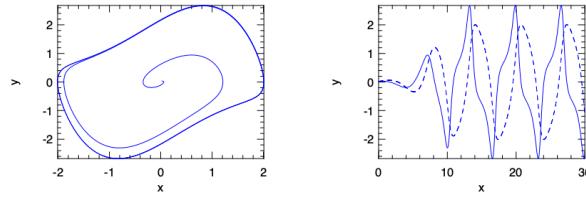


Figure 51: Solving Non-Stiff Van der Pol equations with Sundial CVode solver

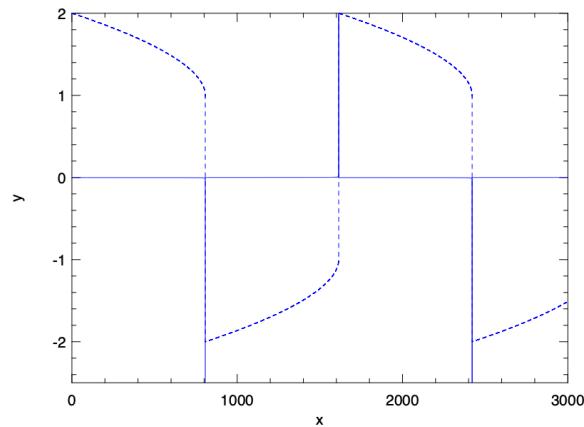


Figure 52: Solving Stiff Van der Pol equations with ODEPACK LSODA solver.

```

let f_stiff = van_der_pol 1000.
let y0 = Mat.of_array [| 2.; 0. |] 1 2
let tspec = T1 { t0 = 0.0; dt = 0.01; duration = 3000.0 }

let () =
  let ts, ys = Ode.odeint (module Owl_ode_odepack.Lsoda) f_stiff y0 tspec () in
  let fname = "vdp_odepack_stiff.png" in
  let h = Plot.create fname in
  let open Plot in
  set_foreground_color h 0 0 0;
  set_background_color h 255 255 255;
  set_yrange h (-2.) 2.;
  plot ~spec:[ RGB (0, 0, 255); LineStyle 1 ] ts Mat.(col ys 1);
  plot ~spec:[ RGB (0, 0, 255); LineStyle 3 ] ts Mat.(col ys 0);
  output h

```

Summary

This chapter discusses solving the initial value problem of ordinary differential equations (ODEs) numerically. We first present the general definition of the ODE and the initial value problem. There are already many existing mathematical solution to ODEs of certain format, but that is not the focus of this chapter. This chapter is centred on the `owl-ode` library. It implements several native and symplectic solvers. We have briefly explained the basic idea of these methods. Also, this library interfaces to existing off-the-shelf open source ODE solvers from the Sundials and ODEPACK library. Next, we demonstrate how these solvers are used to solve ODE from several real examples, including the Two-body problem, the Lorentz Attractor, and Damped Oscillation, etc. Finally, we introduce one important idea in the solving ODE numerically: stiffness, and then shows how we can solve stiff and non-stiff ODEs with the example of the van der pol equation. Hopefully, after studying this chapter, the readers can have a basic idea of how numerical ODE solver works and how to apply them into solving real-world problems.

References

- Candy, J, and W Rozmus. 1991. “A Symplectic Integration Algorithm for Separable Hamiltonian Functions.” *Journal of Computational Physics* 92 (1): 230–56.
- Press, William H, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge university press.

Signal Processing

We rely on signals such as sound and images to convey information. The signal processing is a field that's about analysing, generation, and transformation of signals. Its applications can be seen in a wide range of fields: audio processing, speech recognition, image processing, communication system, data science, etc. In this chapter we mainly focus on Fourier Transform, the core idea in signal processing and modern numerical computing. We introduce its basic idea, and then demonstrate how Owl supports FFT with examples and applications. We also cover the relationship between FFT and Convolution, and filters.

Discrete Fourier Transform

One theme in numerical applications is the transformation of equations into a coordinate system so that the original question can be easily decoupled and simplified. One of the most important such transformation is the *Fourier Transform*, which decomposes a function of time into its constituent frequencies.

All these sound too vague. Let's look at an example. Think about an audio that lasts for 10 seconds. This audio can surely be described in the *time domain*, which means plotting its sound intensity against time as x axis. On the other hand, maybe less obviously, the sound can also be described in the *frequency domain*. For example, if all the 10 seconds are filled with only playing the A# note, then you can describe this whole audio with one frequency number: 466.16 Hz. If it's a C note, then the number is 523.25 Hz, etc. The thing is that, the real-world sound is not always so pure, they are quite likely compounded from different frequencies. Perhaps this 10 seconds are about water flowing, or wind whispering, what frequencies it is built from then?

That's where Fourier Transform comes into play. It captures the idea of converting the two forms of representing a signal: in time domain and in frequency domain. We can represent a signal with the values of some quantity h as a function of time: $h(t)$, or this signal can be represented by giving its amplitude H as function of frequency: $H(f)$. We can think they are two representation of the same thing, and Fourier Transform change between them:

$$\begin{aligned} h(f) &= \int H(f)e^{-2\pi ift} df \\ H(f) &= \int h(t)e^{2\pi ift} dt \end{aligned} \tag{27}$$

To put it simply: suppose Alice mix an unknown number of colour together, and let Bob to guess what those colours are, then Bob can use a Fourier Transform machine to do that.

In computer-based numerical computation, signals are often represented in a discrete way, i.e. a finite sequence of sampled data, instead of continuous. In

that case, the method is called *Discrete Fourier Transform* (DFT). Suppose we have a complex vector y as signal, which contains n elements, then to get the Fourier Transform vector Y , the discrete form of eq. 27 can be expressed with:

$$\begin{aligned} Y_k &= \sum_{j=0}^{n-1} y_j \omega^{jk}, \\ y_k &= \frac{1}{n} \sum_{j=0}^{n-1} Y_k \omega^{-jk}, \end{aligned} \tag{28}$$

where $\omega = e^{-2\pi i/n}$ and $i = \sqrt{-1}$. j and k are indices that go from 0 to $n - 1$.

We highly recommend you to checkout the video that's named "But what is the Fourier Transform? A visual introduction" produced by 3Blue1Brown. It shows how this eq. 28 of Fourier Transform comes into being with beautiful and clear illustration. (TODO: follow the video, explain the idea of FT clearly, not just smashing an equation into readers' faces.)

What can we do if we know how a sound is composed? Think of a classic example where you need to remove some high pitch noise from some music. By using DFT, you can easily find out the frequency of this noise, remove this frequency, and turn the signal back to the time domain by using something a reverse process. You can also get a noisy image, recognise its noise by applying Fourier Transform, remove the noises, and reconstruct the image without noise. We will show such examples later.

Actually, the application of FT is more than on sound or image signal processing. We all use Fourier Transform every day without knowing it: mobile phones, image and audio compression, communication networks, large scale numerical physics and engineering, etc. It is the cornerstone of computational mathematics. One important reason of its popularity is that it has an efficient algorithm in implementation: Fast Fourier Transform.

Fast Fourier Transform

One problem with DFT is that if you follow its definition in implementation, the algorithm computation complexity would be $\mathcal{O}(n^2)$: computing each of the n component of Y in eq. 28 requires n multiplications and n additions. It means that DFT does not scale well with input size. The Fast Fourier Transform algorithm, first formulated by Gauss in 1805 and then developed by James Cooley and John Tukey in 1965, drops the complexity down to $\mathcal{O}(n \log n)$. To put it in a simple way, the FFT algorithm finds out that, any DFT can be represented by the sum of two sub-DFTs: one consists of the elements on even index in the signal, and the other consists of elements on odd positions:

$$\begin{aligned}
 Y_k &= \sum_{\text{even } j} \omega^{jk} y_j + \sum_{\text{odd } j} \omega^{jk} y_j \\
 &= \sum_{j=0}^{n/2-1} \omega^{2jk} y_{2j} + \omega^k \sum_{j=0}^{n/2-1} \omega^{2jk} y_{2j+1}.
 \end{aligned} \tag{29}$$

The key to this step is the fact that $\omega_{2n}^2 = \omega_n$. According to eq. 29, one FFT can be reduced into two FFTs, each on only half of the original length, and then the second half is multiplied by a factor ω^k and added to the first half. The half signal can further be halved, so on and so forth. Therefore the computation can be reduced to a logarithm level in a recursive process. At the end of this recursion is the fact that a FFT on input that contains only one number returns just the number itself.

To introduce Fourier Transform in detailed math and analysis of its properties is beyond the scope of this book, we encourage the readers to refer to other classic textbook on this topic (Phillips, Parr, and Riskin 2003). In this chapter, we focus on introducing how to use FFT in Owl and its applications with Owl code. Hopefully these materials are enough to interest you to investigate more.

The implementation of the FFT module in Owl interfaces to the FFTPack C implementation. The core functions in a FFT module is the `fft` function and its reverse, corresponding to the two equations in eq. 29. Owl provides these basic FFT functions, listed in Tabel tbl. 17. The parameter `otyp` is used to specify the output type. It must be the consistent precision with input `x`. You can skip this parameter by using a sub-module with specific precision such as `Owl.Fft.S` or `Owl.Fft.D`. The `axis` parameter is the highest dimension if not specified. The parameter `n` specifies the size of output.

Table 17: FFT functions in Owl

Functions	Description
<code>fft ~axis x</code>	Compute the one-dimensional discrete Fourier Transform
<code>ifft ~axis x</code>	Compute the one-dimensional inverse discrete Fourier Transform
<code>rfft ~axis otyp x</code>	Compute the one-dimensional discrete Fourier Transform for real input
<code>irfft ~axis ~n otyp x</code>	Compute the one-dimensional inverse discrete Fourier Transform for real input

Examples

We then show how to use these functions with some simple examples. More complex and interesting ones will follow in the next section.

1-D Discrete Fourier transforms

Let start with the most basic `fft` and it reverse transform function `ifft`. First, we create a complex 1-D ndarray that contains 6 elements as input to the `fft` function.

```
# let a = [|1.;2.;1.;-1.;1.5;1.0|]
val a : float array = [|1.; 2.; 1.; -1.; 1.5; 1.|]
# let b = Arr.of_array a [|6|] >> Dense.Ndarray.Generic.cast_d2z
val b : (Complex.t, complex64_elt) Dense.Ndarray.Generic.t =
  C0 C1 C2 C3 C4 C5
  R (1, 0i) (2, 0i) (1, 0i) (-1, 0i) (1.5, 0i) (1, 0i)

# let c = Owl_fft.D.fft b
val c : (Complex.t, complex64_elt) Owl_dense_ndarray_generic.t =
  C0 C1 C2 C3 C4 C5
  R (5.5, 0i) (2.25, -0.433013i) (-2.75, -1.29904i) (1.5, 1.94289E-16i) (-2.75,
  1.29904i) (2.25, 0.433013i)
```

The function `fft` takes a complex ndarray as input, and also returns a complex ndarray. In the result returned, the first half contains the positive-frequency terms, and the second half contains the negative-frequency terms, in order of decreasingly negative frequency. The negative frequency components are the phasors rotating in opposite direction. Typically, only the FFT corresponding to positive frequencies is used, so as to remove redundant frequencies, such as the 2.25 and -2.75 here.

```
# let d = Owl_fft.D.ifft c
val d : (Complex.t, complex64_elt) Owl_dense_ndarray_generic.t =
  C0 C1 C2 C3 C4 C5
  R (1, 1.38778E-17i) (2, 1.15186E-15i) (1, -8.65641E-17i) (-1, -1.52188E-15i) (1.5,
  1.69831E-16i) (1, 2.72882E-16i)
```

The function `ifft` takes the frequency domain result `c` produced by `fft` and reconstruct the original time domain signal. Since we do not change the frequencies, the inverse FFT should produce quite similar result as the original input, as shown in this example.

Perhaps only manipulating arrays still does not make very impressive example. The next example plots the FFT of the sum of two sine functions, showing the power of FFT to separate signals of different frequencies.

```
# module G = Dense.Ndarray.Generic
module G = Owl.Dense.Ndarray.Generic
```

```

# let n = 600. (* sample points *)
val n : float = 600.
# let t = 1. /. 800. (* sample spacing *)
val t : float = 0.00125
# let x = Arr.linspace 0. (n *. t) (int_of_float n)
val x : Arr.arr =
  C0 C1 C2 C3 C4 C595 C596 C597 C598 C599
  R 0 0.00125209 0.00250417 0.00375626 0.00500835 ... 0.744992 0.746244 0.747496
    0.748748 0.75

# let y1 = Arr.((50. *. 2. *. Owl_const.pi) $* x |> sin)
val y1 : Arr.arr =
  C0 C1 C2 C3 C4 C595 C596 C597 C598 C599
  R 0 0.383289 0.708033 0.92463 0.999997 ... 0.999997 0.92463 0.708033 0.383289
    1.27376E-14

# let y2 = Arr.(0.5 $* ((80. *. 2. *. Owl_const.pi) $* x |> sin))
val y2 : (float, float64_elt) Owl_dense_ndarray_generic.t =
  C0 C1 C2 C3 C4 C595 C596 C597 C598 C599
  R 0 0.294317 0.475851 0.47504 0.292193 ... -0.292193 -0.47504 -0.475851 -0.294317
    -2.15587E-14

```

Here we create two sine signals of different frequencies: $y_1(x) = \sin(100\pi x)$, $y_2(x) = \frac{1}{2} \sin(160\pi x)$. We then mix them together.

```

# let y = Arr.(y1 + y2) |> G.cast_d2z
val y : (Complex.t, complex64_elt) G.t =
  C0 C1 C2 C3 C4 C595 C596 C597 C598 C599
  R (0, 0i) (0.677606, 0i) (1.18388, 0i) (1.39967, 0i) (1.29219, 0i) ... (0.707804,
    0i) (0.449591, 0i) (0.232182, 0i) (0.0889723, 0i) (-8.82117E-15, 0i)

```

Next, we apply FFT on the mixed signal:

```

# let yf = Owl_fft.D.fft y
val yf : (Complex.t, complex64_elt) Owl_dense_ndarray_generic.t =
  C0 C1 C2 C3 C4 C595 C596 C597 C598 C599
  R (5.01874, 0i) (5.02225, 0.0182513i) (5.03281, 0.0366004i) (5.05051, 0.0551465i)
    (5.0755, 0.073992i) ... (5.108, -0.0932438i) (5.0755, -0.073992i) (5.05051,
    -0.0551465i) (5.03281, -0.0366004i) (5.02225, -0.0182513i)

```

In the results, each tuple can be seen as a frequency vector in the complex space. We can plot the length of these vectors. As we have said, we use only the first half, or the positive frequencies, of array yf .

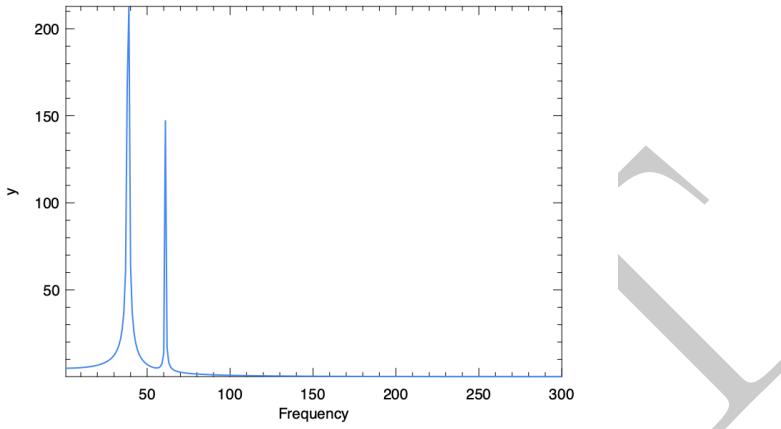


Figure 53: Using FFT to separate two sine signals from their mixed signal

```
# let z = Dense.Ndarray.Z.(abs yf |> re)
val z : Dense.Ndarray.Z.cast_arr =
  C0 C1 C2 C3 C4 C595 C596 C597 C598 C599
  R 5.01874 5.02228 5.03294 5.05081 5.07604 ... 5.10886 5.07604 5.05081 5.03294
  5.02228

# let h = Plot.create "plot_001.png" in
let xa = Arr.linspace 1. 600. 600 in
Plot.plot ~h ~spec:[ RGB (66,133,244); LineWidth 2.] xa z;
Plot.set_xrange h 1. 300;
Plot.set_xlabel h "Frequency";
Plot.output h;;
- : unit = ()
```

Next let's see `rfft` and `irfft`. Function `rfft` calculates the FFT of a real signal input and generates the complex number FFT coefficients for half of the frequency domain range. The negative part is implied by the Hermitian symmetry of the FFT. Similarly, `irfft` performs the reverse step of `rfft`. They are different to `fft` and `ifft` only in the data type, and may make the code cleaner sometimes, as shown in the example below.

```
# let a = [|1.; 2.; 1.; -1.; 1.5; 1.0|]
val a : float array = [|1.; 2.; 1.; -1.; 1.5; 1.|]
# let b = Arr.of_array a [|6|]
val b : Arr.arr =
  C0 C1 C2 C3 C4 C5
```

```
R 1 2 1 -1 1.5 1
```

```
# let c = Owl_fft.D.rfft b
val c : (Complex.t, complex64_elt) Owl_dense_ndarray_generic.t =
  C0 C1 C2 C3
  R (5.5, 0i) (2.25, -0.433013i) (-2.75, -1.29904i) (1.5, 0i)
```

```
# let d = Owl_fft.D.irfft c
val d : (float, float64_elt) Owl_dense_ndarray_generic.t =
  C0 C1 C2 C3 C4 C5
  R 1 2 1 -1 1.5 1
```

N-D Discrete Fourier transforms

The owl FFT functions also applies to multi-dimensional arrays, such as matrix. Example: the fft matrix.

(TODO: This is not the real N-D FFT. IMPLEMENTATION required. TODO: explain briefly how 2D FFT can be built with 1D. Reference: Data-Driven Book, Chap2.6. Implementation is not difficult: (1) do 1D FFT on each row (real to complex); (2) do 1D FFT on each column resulting from (1) (complex to complex))

```
# let a = Dense.Matrix.Z.eye 5
val a : Dense.Matrix.Z.mat =
  C0 C1 C2 C3 C4
  R0 (1, 0i) (0, 0i) (0, 0i) (0, 0i) (0, 0i)
  R1 (0, 0i) (1, 0i) (0, 0i) (0, 0i) (0, 0i)
  R2 (0, 0i) (0, 0i) (1, 0i) (0, 0i) (0, 0i)
  R3 (0, 0i) (0, 0i) (0, 0i) (1, 0i) (0, 0i)
  R4 (0, 0i) (0, 0i) (0, 0i) (0, 0i) (1, 0i)

# let b = Owl_fft.D.fft a
val b : (Complex.t, complex64_elt) Owl_dense_ndarray_generic.t =
  C0 C1 C2 C3 C4
  R0 (1, 0i) (1, 0i) (1, 0i) (1, 0i) (1, 0i)
  R1 (1, 0i) (0.309017, -0.951057i) (-0.809017, -0.587785i) (-0.809017, 0.587785i)
    (0.309017, 0.951057i)
  R2 (1, 0i) (-0.809017, -0.587785i) (0.309017, 0.951057i) (0.309017, -0.951057i)
    (-0.809017, 0.587785i)
  R3 (1, 0i) (-0.809017, 0.587785i) (0.309017, -0.951057i) (0.309017, 0.951057i)
    (-0.809017, -0.587785i)
  R4 (1, 0i) (0.309017, 0.951057i) (-0.809017, 0.587785i) (-0.809017, -0.587785i)
    (0.309017, -0.951057i)
```

IMAGE: plot x and y in to circle-like shape

Applications of FFT

As we said, the applications of FFT are numerous. Here we pick three to demonstrate the power of FFT and how to use it in Owl. The first is to find the period rules in the historical data of sunspots, and the second is about analysing the content of dial number according to audio information. Both applications are inspired by (Moler 2008). The third application is about image processing. These three applications together present a full picture about how the wide usage of FFT in various scenarios.

Find period of sunspots

On the Sun's photosphere, the *sunspots* are what appear as spots darker than the surrounding areas. Their number changes according to a certain cycle. The Sunspot Index and Long-term Solar Observations (SILS) is a world data center that preserves the data about the sunspot. In this example, we will use the data here to find out the sunspot cycle.

The datasets are all available on the website of SILS. Each contains the time and the sunspots number, measured by the “wolfer index”. The dataset provided here are of different granularity. Here we use the yearly data, from 1700 to 2020. You can also try the monthly data to get more detailed knowledge. First, load the data:

```
let data = Owl_io.read_csv ~sep:',' "sunspot_full.csv"
let data = Array.map (fun x -> Array.map float_of_string x) data |> Mat.of_arrays

let x = Mat.get_slice [[];[0]] data
let y = Mat.get_slice [[];[1]] data
```

We can then visualise the data:

```
let plot_sunspot x y =
  let h = Plot.create "plot_sunspot.png" in
  Plot.set_font_size h 8.;
  Plot.set_pen_size h 3.;
  Plot.set_xlabel h "Date";
  Plot.set_ylabel h "Sunspot number";
  Plot.plot ~h ~spec:[ RGB (255,0,0); LineStyle 1] x y;
  Plot.output h
```

We can see there is a cycle of about 10 years, but exactly how long is it? Let's start by applying the FFT on this signal. To process the data, we first remove the first element of the frequency vector y' , since it stores the sum of the data. The frequency is reduced to half, since we plot only half of the coefficients.

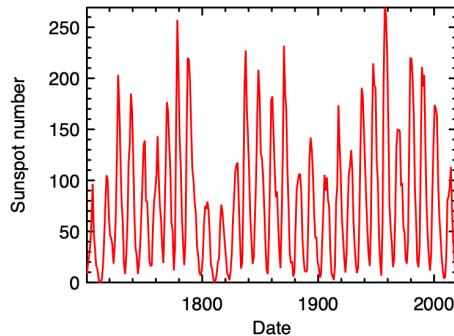


Figure 54: Yearly sunspot data

```
let get_frequency y =
  let y' = Owl_fft.D.rfft ~axis:0 y in
  let y' = Dense.Ndarray.Z.get_slice [[1; (Dense.Ndarray.Z.shape y').(0) - 1];[]]
    y' in
  Dense.Ndarray.Z.(abs y' > re)
```

The frequency (cycle/year) as unit of measurement seems a bit confusing. To get the cyclical activity that is easier to interpret, we also plot the squared power as a function of years/cycle. Both are plotted with code below.

```
let plot_sunspot_freq p =
  let n = (Arr.shape p).(0) in
  let f = Arr.(mul_scalar (linspace 0. 1. n) 0.5) in

  let h = Plot.create ~m:1 ~n:2 "plot_sunspot_freq.png" in
  Plot.set_pen_size h 3.;
  Plot.subplot h 0 0;
  Plot.set_xlabel h "cycle/year";
  Plot.set_ylabel h "squared power";
  Plot.plot ~h ~spec:[ RGB (255,0,0); LineStyle 1] f p;

  Plot.subplot h 0 1;
  Plot.set_xlabel h "year/cycle";
  Plot.set_ylabel h "squared power";
  let f' = Arr.(scalar_div 1. (get_slice [[1; Stdlib.(n-1)]] f)) in
  Plot.plot ~h ~spec:[ RGB (255,0,0); LineStyle 1] f' p;
  Plot.set_xrange h 0. 40.;
  Plot.output h
```

The result is shown in fig. 55. Now we can see clearly that the most prominent cycle is a little bit less than 11 years.



Figure 55: Find sunspot cycle with FFT

Decipher the Tone

When we are dialling a phone number, the soundwave can be seen as a signal. In this example, we show how to decipher which number is dialled according to the given soundwave. This example uses the data from (Moler 2008). Let's first load and visualise them.

```
let data = Owl_io.read_csv ~sep:',' "touchtone.csv"
let data = Array.map (fun x -> Array.map float_of_string x) data |> Mat.of_arrays
let data = Mat.div_scalar data 128.
```

The dataset specifies a sampling rate of 8192.

```
let fs = 8192.
```

We have a segment of signal that shows the touch tone of dialling a phone number. We can visualise the signal:

```
let plot_tone data filename =
  let x = Mat.div_scalar (Mat.sequential 1 (Arr.shape data).(1)) fs in
  let h = Plot.create filename in
  Plot.set_font_size h 8.;
  Plot.set_pen_size h 3.;
  Plot.set_xlabel h "time(s)";
  Plot.set_ylabel h "signal magnitude";
  Plot.plot ~h ~spec:[ RGB (0, 0, 255); LineStyle 1] x data;
  Plot.output h
```

The result is shown in fig. 56(a). Apparently, according to the dense area in this signal, there are 11 digits in this phone number. The question is: which numbers?

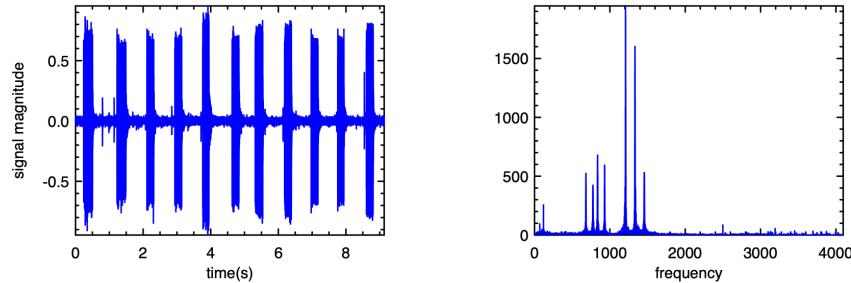


Figure 56: Recording of an 11-digit number and its FFT decomposition

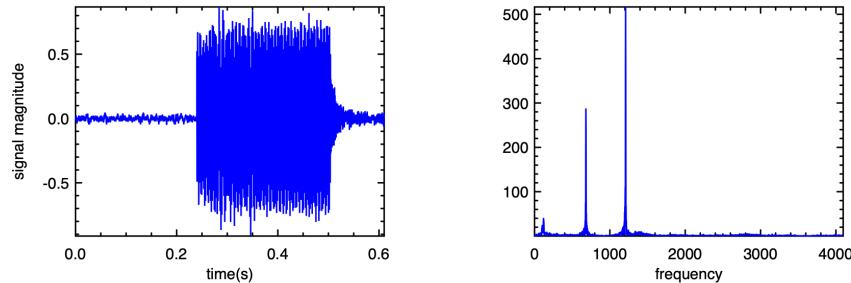


Figure 57: Recording of the first digit and its FFT decomposition

This is a suitable question for FFT. Let's start by applying the FFT to the original data.

```
let yf = Owl_fft.D.rfft data
let y' = Dense.Ndarray.Z.(abs yf |> re)
let n = (Arr.shape y').(1)
let x' = Mat.linspace 0. (fs /. 2.) n
```

We plot x' with y' similarly using the previous plotting function, and the result is shown in fig. 56(b). The tune of phone is combination of two different frequencies. All the 11 digits are composed from 7 prominent frequencies, as shown in tbl. 18. This frequency keypad is specified in the Dual-tone multi-frequency signalling (DTMF) standard.

Table 18: DTMF keypad frequencies

	1209 Hz	1336 Hz	1477 Hz
697Hz	1	2	3
770Hz	4	5	6
852Hz	7	8	9
941Hz	*	2	#

We can use the first tone as an example to find out which two frequencies it is composed from. Let's get a subset of the signal:

```
let data2 = Arr.get_slice [[], [0, 4999]] data
```

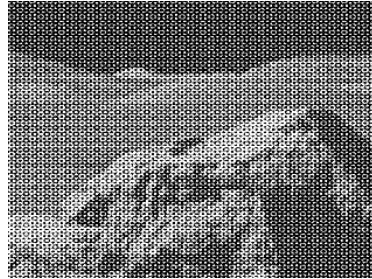


Figure 58: Noise Moonlanding image

are about 700 and 1200. Looking it up in tbl. 18, we can see that the first digit is 1. You can investigate the whole phone number following the same procedure.

Image Processing

When the data to be processed goes from being one dimensional to multiple dimensional, FFT is still a powerful tool. It can be used for a wide range of image processing tasks such as blurring, compression or denosing. In this section, we demonstrate an image denoising example. The basic idea is similar: many Fourier frequencies in the image are small and can be neglected via using filters. The image quality can be largely preserved by recreating the image from only the major frequencies.

As input, we use the noised version of the moon landing image, as shown in fig. 58.

As a tool for image processing, we use the `imageUtils.ml` script. It can be used for reading images of PPM format into ndarray, or saving ndarray into PPM image. You can copy this script to local directory.

```
#use "imageUtils.ml"
```

As the first step, we read in the image into Owl as a ndarray using the function `load_ppm`. All the elements in this matrix are within the range 0 to 255. Since this is an greyscale image, we only need to process on single channel of this array, which is a matrix.

```
module N = Dense.Ndarray.S
module C = Dense.Ndarray.C

let img_arr = load_ppm "moonlanding.ppm" |> N.get_slice [[];[];[0]]

let shp = N.shape img_arr
let h, w = shp.(0), shp.(1)
let img = N.reshape img_arr [|h; w|] |> Dense.Ndarray.Generic.cast_s2c
```

The next step is simple: applying the `fft2` function on the image.

```
| let img_fft = Owl_fft.S.fft2 img
```

Next, we can apply all kinds of filters on this frequency map `img_fft` to remove the unwanted frequencies. We choose use a simple one here: only keep the frequencies on the four “corner” in this matrix, and set the rest to zeros. We can achieve that with the slicing.

```
let sub_length x frac = (float_of_int x) *. frac |> int_of_float
let h1 = sub_length h 0.1
let h2 = sub_length h 0.9
let w1 = sub_length w 0.1
let w2 = sub_length w 0.9

let index_0 = [ R [h1; h2]; R []]
let index_1 = [ R [0; h1]; R [w1; w2] ]
let index_2 = [ R [h2; h-1]; R [w1; w2] ]

let slice_0 = C.get_fancy index_0 img_fft
let slice_1 = C.get_fancy index_1 img_fft
let slice_2 = C.get_fancy index_2 img_fft

let _ = C.set_fancy index_0 img_fft (C.shape slice_0 |> C.zeros)
let _ = C.set_fancy index_1 img_fft (C.shape slice_1 |> C.zeros)
let _ = C.set_fancy index_2 img_fft (C.shape slice_2 |> C.zeros)
```

Finally, let's apply the reverse FFT function on the processed frequency map, and save the result as image.

```
let img = Owl_fft.S.ifft img_fft |> C.re

let image = N.stack ~axis:2 [|img; img; img|]
let image = N.expand image 4
let _ = save_ppm_from_arr image "moonlanding_denoise.ppm"
```

As can be seen in fig. 59, though a bit blur, the output image remove the noise in the original image, and manage to keep most of the information in it.

Filtering

In the N-dimensional Array chapter, we have introduced the idea of `filter`, which allows to get target data from the input according to some function. *Filtering* in signal processing is similar. It is a generic name for any system that modifies input signal in certain ways, most likely removing some unwanted features from it. In this section, we introduce how FFT can be applied to perform some filtering tasks with examples.

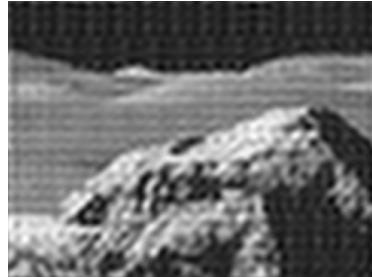


Figure 59: De-noised Moonlanding image

Example: Smoothing

Let's start with a simple and common filter task: smoothing. Suppose you have a segment of noisy signal: the stock price. In many cases we hope to remove the extreme trends and see a long-term trend from the historical data. We take the stock price of Google in the past year, April 09, from 2019 to 2020. The data is taken from Yahoo Finance. We can load the data into a matrix:

```
let data = Owl_io.read_csv ~sep:',' "goog.csv"
let data = Array.map (fun x ->
  Array.map float_of_string (Array.sub x 1 6))
  (Array.sub data 1 (Array.length data - 1))
|> Mat.of_arrays
```

The data y contains several columns, each representing opening price, volume, high price, etc. Here we use the daily closing price as example, which are in the fourth column.

```
let y = Mat.get_slice [][];[3]] data
```

To compute the moving average of this signal, I'll create a *window* with 10 elements. Here we only use a simple filter which normalises each element to the same 0.1.

```
let filter = Mat.of_array (Array.make 10 0.1 ) 1 10
```

Now, we can sliding this filter window along input signal to smooth the data step by step.

```
let y' = Mat.mapi (fun i _ ->
  let r = Mat.get_fancy [R [i; i+9]; R []] y in
  Mat.dot filter r |> Mat.sum'
) (Mat.get_slice [[0; (Arr.shape y).(0) - 10]; []] y)
```

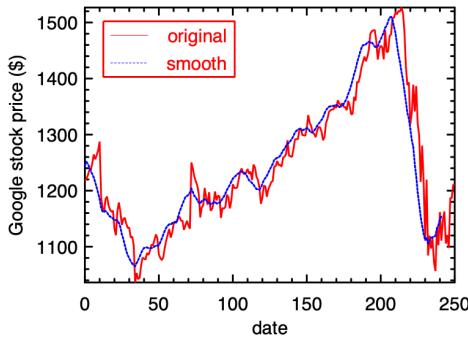


Figure 60: Smoothed stock price of Google

Finally, we can plot the resulting smoothed data with the original data.

```
let plot_goog data y y' =
  let n = (Arr.shape data).(0) in
  let x = Mat.sequential n 1 in
  let h = Plot.create "plot_goog.png" in
  Plot.set_font_size h 8.;
  Plot.set_pen_size h 3.;
  Plot.set_xlabel h "date";
  Plot.set_ylabel h "Google stock price ($)";
  Plot.plot ~h ~spec:[ RGB (255,0,0); LineStyle 1] x y;
  Plot.plot ~h ~spec:[ RGB (0,0,255); LineStyle 2] x y';
  Plot.(legend_on h ~position:NorthWest [| "original"; "smooth" |]);
  Plot.output h
```

The results are shown in fig. 60. The blue dotted line smooths the jagged original stock price line, which represents a general trend of the price. The sudden drop in the last month might be related with the COVID-19 pandemic.

Gaussian Filter

However, the filter we have used is not an ideal one. A common pattern in this line drops first, but then bounces around. To get a smoother curve, we can change this simple filter to another one: the gaussian filter. Instead of giving equal probability to each element in the moving window, the gaussian filter assigns probability according to the gaussian distribution: $p(x) = e^{-\frac{x^2}{2\sigma^2}}$.

The code below generates a simple 1-D gaussian filter. Similar to the previous simple example, the filter also needs to be normalised. For the filter window vector, its range of radius is set to truncate standard deviations. This implementation is similar to that in SciPy.



Figure 61: Smoothed stock price of Google with Gaussian filtering

```

let gaussian_kernel sigma =
  let truncate = 4. in
  let radius = truncate *. sigma +. 0.5 |> int_of_float in
  let r = float_of_int radius in
  let x = Mat.linspace (-r) r (2 * radius + 1) in
  let f a = Maths.exp (-0.5 *. a ** 2. /. (sigma *. sigma)) in
  let x = Mat.map f x in
  Mat.(div_scalar x (sum' x))

let filter = gaussian_kernel 3.

```

Computing the correlation between filter and the input data as before, we get a better smoothed curve in fig. 61.

Filters can be generally categorised by their usage into time domain filters and frequency domain filters. Time domain filters are used when the information is encoded in the shape of the signal's waveform, and can be used for tasks such as smoothing, waveform shaping, etc. It includes filter methods such as moving average and single pole. Frequency filters are used to divide a band of frequencies from signals, and its input information is in the form of sinusoids. It includes filter methods such as Windowed-sinc and Chebyshev. There are many filters, each with different shape (or *impulse response*) and application scenarios, and we cannot fully cover them here. Please refer to some classical textbooks on signal processing such as (Smith and others 1997) for more information.

Signal Convolution

So far we have not used FFT to perform filtering yet. What we have seen in the previous section is called *convolution*. Formally it is mathematical operation on two functions that produces a third function expressing how the shape of one is modified by the other:

$$f(t) * g(t) = \sum_{\tau=-\infty}^{\infty} f(\tau)g(t-\tau) \quad (30)$$

In equation eq. 30, $*$ denotes the convolution operation, and you can think of f as (discrete) input signal, and g be filter. Note that for computing $f(\tau)g(t-\tau)$ for each τ requires adding all product pairs. You can see that this process is computation-heavy. It is even more tricky for computing the convolution of two continuous signal following definition.

We have talked a lot about FFT, and here is the place we use it. Fourier Transformation can greatly reduce the complexity of computing the convolution and filtering. Specifically, the **Convolution Theorem** states that:

$$\text{DFT}(f * g) = \text{DFT}(f).\text{DFT}(g). \quad (31)$$

To put equation eq. 31 into plain words, to get DFT of two signals' convolution, we can simply get the DFT of each signal separately and then multiply them element-wise. Someone may also prefer to express it in this way: convolution in the time domain can be expressed in multiplication in the frequency domain. And multiplication we are very familiar with. Once you have the $\text{DFT}(f * g)$, you can naturally apply the inverse transform and get $f * g$.

Let's apply the DFT approach to the previous data, and move it to the frequency domain:

```
let yf = Owl_fft.D.rfft ~axis:0 y
```

The resulting data yf looks like this:

```
val yf : (Complex.t, Bigarray.complex64_elt) Owl_dense_ndarray_generic.t =
  C0
  R0 (312445, 0i)
  R1 (-2664.07, 17064.3i)
  R2 (-5272.52, 3899.16i)
  R3 (-2085.98, -3101.46i)
  ...
  R124 (23.0005, -38.841i)
  R125 (153.294, 68.7544i)
```

We only keep the five most notable frequencies, and set the rest to zero.

```
let n = (Dense.Ndarray.Z.shape yf).(0)
let z = Dense.Ndarray.Z.zeros [|n-5; 1|]
let _ = Dense.Ndarray.Z.set_slice [[5;n-1];[]] yf z
```

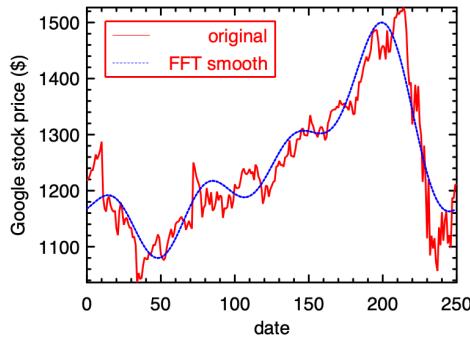


Figure 62: Smoothed stock price of Google using FFT method

Now, we can apply reverse FFT on this processed frequency vector and get the smoothed data:

```
let y2 = Owl_fft.D.irfft ~axis:0 yf
```

We can similarly check how the smoothing approach works in fig. 62. Compared to the previous two smoothing methods, FFT generates a better curve to describe the trend of the stock price.

FFT and Image Convolution

You might heard of the word “convolution” before, and yes you are right: convolution is also the core idea in the popular deep neural network (DNN) applications. The convolution in DNN is often applied on ndarrays. It is not complex: you start with an input image in the form of ndarray, and use another smaller ndarray called “kernel” to slide over the input image step by step, and at each position, an element-wise multiplication is applied, and the result is filled into corresponding position in an output ndarray. This process can be best illustrated in fig. 63 (inspired by the nice work by Andrej Karpathy):

Owl has provided thorough support of convolution operations:

```
val conv1d : ?padding:padding -> (float, 'a) t -> (float, 'a) t -> int array ->
  (float, 'a) t

val conv2d : ?padding:padding -> (float, 'a) t -> (float, 'a) t -> int array ->
  (float, 'a) t

val conv3d : ?padding:padding -> (float, 'a) t -> (float, 'a) t -> int array ->
  (float, 'a) t
```

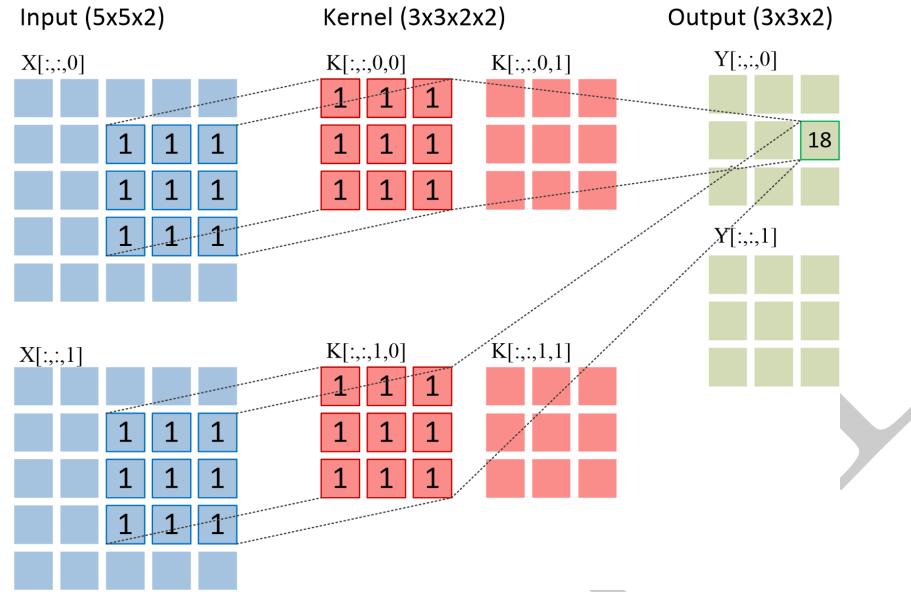


Figure 63: Image convolution illustration

They corresponds to different dimension of inputs. Besides, Owl also support other derived convolution types, including dilated convolutions, transpose convolutions, and backward convolutions etc.

It's OK if none of this makes sense to you now. We'll explain the convolution and its usage in later chapter in detail. The point is that, if you look closely, you can find that the image convolution is only a special high dimensional case of the convolution equation: a given input signal (the image), another similar but smaller filter signal (the kernel), and the filter slides across the input signal and perform element-wise multiplication.

Therefore, we can implement the convolution with FFT and vice versa. For example, we can use `conv1d` function in Owl to solve the previous simple smoothing problem:

```
let y3 = Arr.reshape y [|1;251;1|]
let f3 = Arr.reshape filter [|10;1;1|]
let y3' = Arr.conv1d y3 f3 [|1|]
```

If you are interested to check the result, this vector $y3'$ contains the data to plot a smoothed curve. The smoothed data would be similar to that in fig. 60 since the calculation is the same, only with more concise code.

Also, FFT is a popular implementation method of convolution. There has been a lot of research on optimising and comparing its performance with other

implementation methods such as Winograd, with practical considerations such as kernel size and implementation details of code, but we will omit these technical discussion for now.

Summary

This chapter centres around a fundamental idea behind signal processing: the Fourier Transform. We started with its definition, and then introduce a crucial idea behind its efficient implementation: the Fast Fourier Transform (FFT). Owl provides support to FFT by linking to existing FFTPack library. We showed how the FFT functions can be used in Owl, first with some simple examples, and then with three real applications: finding the frequency of sunspot events, decipher the dial tone number according to its sound, and de-noising an image. Finally, we discussed filtering in signal process using different techniques, including simple averaging smoothing, gaussian filtering, and FFT-based filtering. Here we also explained the relationship between the two most crucial computations in numerical applications: FFT and convolution. More about convolution can be find in the Neural Network chapter.

References

- Moler, Cleve B. 2008. *Numerical Computing with Matlab: Revised Reprint*. Vol. 87. Siam.
- Phillips, Charles L, John M Parr, and Eve A Riskin. 2003. *Signals, Systems, and Transforms*. Prentice Hall Upper Saddle River.
- Smith, Steven W, and others. 1997. “The Scientist and Engineer’s Guide to Digital Signal Processing.”

Algorithmic Differentiation

In science and engineering it is often necessary to study the relationship between two or more quantities, where changing of one quantity leads to changes of others. For example, in describing the motion an object, we denote velocity v of an object with the change of the distance regarding time:

$$v = \lim_{\Delta t} \frac{\Delta s}{\Delta t} = \frac{ds}{dt}. \quad (32)$$

This relationship $\frac{ds}{dt}$ is called “*derivative* of s with respect to t ”. This process can be extended to higher dimensional space. For example, think about a solid block of material, placed in a cartesian axis system. You heat it at some part of it and cool it down at some other place, and you can imagine that the temperature T at different position of this block: $T(x, y, z)$. In this field, we can describe this change with partial derivatives along each axis:

$$\nabla T = \left(\frac{\partial T}{\partial x}, \frac{\partial T}{\partial y}, \frac{\partial T}{\partial z} \right). \quad (33)$$

Here, we call the vector ∇T *gradient* of T . The procedure to calculating derivatives and gradients is referred to as *differentiation*.

Differentiation is crucial to many scientific related fields: find maximum or minimum values using gradient descent; ODE; Non-linear optimisation such as KKT optimality conditions is still a prime application. One new crucial application is in machine learning. The training of a supervised machine learning model often requires the forward propagation and back propagation phases, where the back propagation can be seen as the derivative of the whole model as a large function. We will talk about these applications in the next chapters.

Differentiation often requires complex computation, and in these applications we surely need to rely on some computing framework to support it. Differentiation module is built into the core of Owl. In this chapter, starting from the basic computation rule in performing differentiation, we will introduce how Owl supports this important feature step by step.

Chain Rule

Before diving into how to do differentiation on computers, let's recall how to do it with a pencil and paper from our Calculus 101. One of the most important rules in performing differentiation is the *chain rule*. In calculus, the chain rule is a formula to compute the derivative of a composite function. Suppose we have two functions f and g , then the chain rule states that:

$$F'(x) = f'(g(x))g'(x). \quad (34)$$

This seemingly simple rule is one of the most fundamental rules in calculating derivatives. For example, let $y = x^a$, where a is a real number, and then we can get y' using the chain rule. Specifically, let $y = e^{\ln x^a} = e^{a \ln x}$, and then we can set $u = a \ln x$ so that now $y = e^u$. By applying the chain rule, we have:

$$y' = \frac{dy}{du} \frac{du}{dx} = e^u \cdot a \cdot \frac{1}{x} = ax^{a-1}.$$

Besides the chain rule, it's helpful to remember some basic differentiation equations, as shown in *tbl. 19*. Here x is variable and both u and v are functions with regard to x . C is constant. These equations are the building blocks of differentiating more complicated ones. Of course, this very short list is incomplete. Please refer to the calculus textbooks for more information. Armed with the chain rule and these basic equations, we can begin to solve more differentiation problems than you can imagine.

Table 19: A Short Table of Basic Derivatives

Function	Derivatives
$(u(x) + v(x))'$	$u'(x) + v'(x)$
$(C \times u(x))'$	$C \times u'(x)$
$(u(x)v(x))'$	$u'(x)v(x) + u(x)v'(x)$
$\left(\frac{u(x)}{v(x)}\right)'$	$\frac{u'(x)v(x) - u(x)v'(x)}{v^2(x)}$
$\sin(x)$	$\cos(x)$
e^x	e^x
$\log_a(x)$	$\frac{1}{x \ln a}$

Differentiation Methods

As the models and algorithms become increasingly complex, sometimes the function being implicit, it is impractical to perform manual differentiation. Therefore, we turn to computer-based automated computation methods. There are three different ways widely used to automate differentiation: numerical differentiation, symbolic differentiation, and algorithmic differentiation.

Numerical Differentiation

The numerical differentiation comes from the definition of derivative in *eq. 32*. It uses a small step δ to approximate the limit in the definition:

$$f'(x) = \lim_{\delta \rightarrow 0} \frac{f(x + \delta) - f(x)}{\delta}. \quad (35)$$

This method is pretty easy to follow: evaluate the given f at point x , and then choose a suitable small amount δ , add it to the original x and then re-evaluate the function. Then the derivative can be calculated using *eq. 35*. As

long as you know how to evaluate function f , this method can be applied. The function f per se can be treated a black box. The implementation is also straightforward. However, the problem with this method is prone to truncation errors and round-off errors.

The *truncation error* comes from the fact that eq. 35 is only an approximation of the true gradient value. We can see their difference with Taylor expansion:

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(\sigma_h)$$

Here h is the step size and σ_h is in the range of $[x, x+h]$. This can be transformed into:

$$\frac{h^2}{2}f''(\sigma_h) = f'(x) - \frac{f(x+h) - f(x)}{h}.$$

This represents the truncation error in the approximation. For example, for function $f(x) = \sin(x)$, $f''(x) = -\sin(x)$. Suppose we want to calculate the derivative at $x = 1$ numerically using a step size of 0.01, then the truncation error should be in the range $\frac{0.01^2}{2}[\sin(1), \sin(1.01)]$.

We can see the effect of this truncation error in an example, by using an improperly large step size. Let's say we want to find the derivative of $f(x) = \cos(x)$ at point $x = 1$. Basic calculus tells us that it should be equals to $-\sin(1) = 0.84147$, but the result is obviously a bit different.

```
# let d =
  let _eps = 0.1 in
  let diff f x = (f (x +. _eps) -. f x) /. _eps in
  diff Maths.cos 1.
val d : float = -0.867061844425624506
```

Another source of error is the *round-off error*. It is caused by representing numbers approximately in numerical computation during this process. Looking back at eq. 35, we need to calculate $f(x+h) - f(x)$, the subtraction of two almost identical number. That could lead to a large round-off errors in a computer. For example, let's choose a very small step size this time:

```
# let d =
  let _eps = 5E-16 in
  let diff f x = (f (x +. _eps) -. f x) /. _eps in
  diff Maths.cos 1.
val d : float = -0.888178419700125121
```

It is still significantly different from the expected result. Actually if we use a even smaller step size $1e - 16$, the result becomes 0, which means the round-off error is large enough that $f(x)$ and $f(x + h)$ are deemed the same by the computer.

Besides these sources of error, the numerical differentiation method is also slow due to requiring multiple evaluations of function f . Some discussion about numerically solving derivative-related problems is also covered in the Ordinary Differentiation Equation chapter, where we focus on introducing solving these equations numerically, and how the impact of these errors can be reduced.

Symbolic Differentiation

Symbolic Differentiation is the opposite of numerical solution. It does not involve numerical computation, only math symbol manipulation. The rules we have introduced in tbl. 19 are actually expressed in symbols. Think about this function: $f(x_0, x_1, x_2) = x_0 * x_1 * x_2$. If we compute ∇f symbolically, we end up with:

$$\nabla f = \left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2} \right) = (x_1 * x_2, x_0 * x_2, x_1 * x_2).$$

It is nice and accurate, leaving limited space for numerical errors. However, you can try to extend the number of variables from 3 to a large number n , which means $f(x) = \prod_{i=0}^{n-1} x_i$, and then try to perform the symbolic differentiation again.

The point is that, symbolic computations tend to give a very large and complex result for even simple functions. It's easy to have duplicated common sub computations, and produce exponentially large symbolic expressions. Therefore, as intuitive as it is, the symbolic differentiation method can easily takes a lot of memory in computer, and it is slow.

The explosion of computation complexity is not the only limitation of symbolic differentiation. In contrast to the numerical differentiation, we have to treat the function in symbolic differentiation as a white box, knowing exactly what is inside of it. This further indicates that it cannot be used for arbitrary functions.

Algorithmic Differentiation

Algorithmic differentiation (AD) is a chain-rule based technique for calculating the derivatives with regards to input variables of functions defined in a computer programme. It is also known as automatic differentiation, though strictly speaking AD does not fully automate differentiation and can sometimes lead to inefficient code.

AD can generate exact results with superior speed and memory usage, therefore highly applicable in various real world applications. Even though AD also follows the chain rule, it directly applies numerical computation for intermediate results. It is important to point out that AD is neither numerical nor symbolic differentiation. It takes the best parts of both worlds, as we will see in the next



Figure 64: Graph expression of function

section. Actually, according to (Griewank and others 1989), the reverse mode of AD yields any gradient vector at no more than five times the cost of evaluating the function f itself. AD has already been implemented in various popular languages, including the `ad` in Python, `JuliaDiff` in Julia, and `ADMAT` in MATLAB, etc. In the rest of this chapter, we focus on introducing the AD module in Owl.

How Algorithmic Differentiation Works

We have seen the chain rules being applied on simple functions such as $y = x^a$. Now let's check how this rule can be applied to more complex computations. Let's look at the function below:

$$y(x_0, x_1) = (1 + e^{x_0 \cdot x_1 + \sin(x_0)})^{-1}. \quad (36)$$

This function is based on a sigmoid function. Our goal is to compute the partial derivative $\frac{\partial y}{\partial x_0}$ and $\frac{\partial y}{\partial x_1}$. To better illustrate this process, we express eq. 36 as a graph, as shown in fig. 64. At the right side of the figure, we have the final output y , and at the roots of this graph are input variables. The nodes between them indicate constants or intermediate variables that are gotten via basic functions such as `sine`. All nodes are labelled by v_i . An edge between two nodes represents an explicit dependency in the computation.

Based on this graphic representation, there are two major ways to apply the chain rules: the forward differentiation mode, and the reverse differentiation mode (not “backward differentiation”, which is a method used for solving ordinary differential equations). Next, we introduce these two methods.

Forward Mode

Our target is to calculate $\frac{\partial y}{\partial x_0}$ (partial derivative regarding x_1 should be similar). But hold your horse, let's start with some earlier intermediate results that might be helpful. For example, what is $\frac{\partial x_0}{\partial x_1}$? 1, obviously. Equally obvious is $\frac{\partial x_1}{\partial x_1} = 0$.

It's just elementary. Now, things gets a bit trickier: what is $\frac{\partial v_3}{\partial x_0}$? It is a good time to use the chain rule:

$$\frac{\partial v_3}{\partial x_0} = \frac{\partial (x_0 x_1)}{\partial x_0} = x_1 \frac{\partial (x_0)}{\partial x_0} + x_0 \frac{\partial (x_1)}{\partial x_0} = x_1.$$

After calculating $\frac{\partial v_3}{\partial x_0}$, we can then process with derivatives of v_5, v_6 , all the way to that of v_9 which is also the output y we are looking for. This process starts with the input variables, and ends with output variables. Therefore, it is called *forward differentiation*. We can simplify the math notations in this process by letting $v_i = \frac{\partial (v_i)}{\partial x_0}$. The v_i here is called *tangent* of function $v_i(x_0, x_1, \dots, x_n)$ with regard to input variable x_0 , and the original computation results at each intermediate point is called *primal* values. The forward differentiation mode is sometimes also called “tangent linear” mode.

Now we can present the full forward differentiation calculation process, as shown in tbl. 20. Two simultaneous computing processes take place, shown as two separated columns: on the left side is the computation procedure specified by eq. 36; on the right side shows computation of derivative for each intermediate variable with regard to x_0 . Let's find out \dot{y} when setting $x_0 = 1$, and $x_1 = 1$.

Table 20: Computation process of forward differentiation

Step	Primal computation	Tangent computation
0	$v_0 = x_0 = 1$	$\dot{v}_0 = 1$
1	$v_1 = x_1 = 1$	$\dot{v}_1 = 0$
2	$v_2 = \sin(v_0) = 0.84$	$\dot{v}_2 = \cos(v_0) * \dot{v}_0 = 0.54 * 1 = 0.54$
3	$v_3 = v_0 v_1 = 1$	$\dot{v}_3 = v_0 \dot{v}_1 + v_1 \dot{v}_0 = 1 * 0 + 1 * 1 = 1$
4	$v_4 = v_2 + v_3 = 1.84$	$\dot{v}_4 = \dot{v}_2 + \dot{v}_3 = 1.54$
5	$v_5 = 1$	$\dot{v}_5 = 0$
6	$v_6 = \exp(v_4) = 6.30$	$\dot{v}_6 = \exp(v_4) * \dot{v}_4 = 6.30 * 1.54 = 9.70$
7	$v_7 = 1$	$\dot{v}_7 = 0$
8	$v_8 = v_5 + v_6 = 7.30$	$\dot{v}_8 = \dot{v}_5 + \dot{v}_6 = 9.70$
9	$y = v_9 = \frac{1}{v_8}$	$\dot{y} = \frac{-1}{v_8^2} * \dot{v}_8 = -0.18$

This procedure shown in this table can be illustrated in fig. 65.

Of course, all the numerical computations here are approximated with only two significant figures. We can validate this result with algorithmic differentiation module in Owl. If you don't understand the code, don't worry. We will cover the details of this module in later sections.

```
| # open Algodiff.D
```

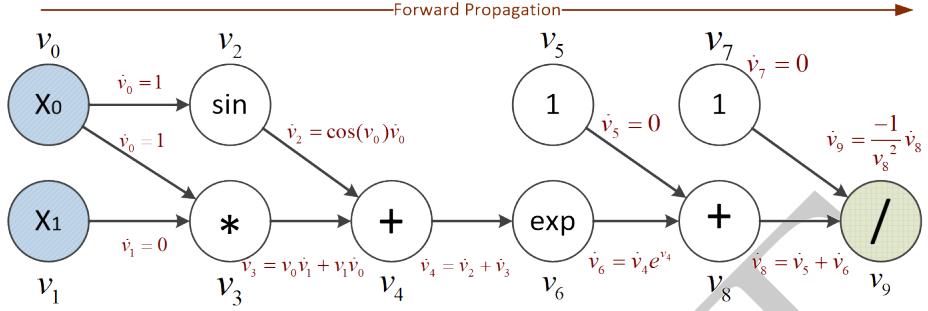


Figure 65: Example of forward accumulation with computational graph

```
# let f x =
  let x1 = Mat.get x 0 0 in
  let x2 = Mat.get x 0 1 in
  Maths.(div (F 1.) (F 1. + exp (x1 * x2 + (sin x1))))
val f : t -> t = <fun>

# let x = Mat.ones 1 2
val x : t = [Arr(1,2)]

# let _ = grad f x |> unpack_arr
- : A.arr =
  C0 C1
R0 -0.181974 -0.118142
```

Reverse Mode

Now let's rethink about this problem from the other direction, literally. Question remains the same, i.e. calculating $\frac{\partial y}{\partial x_0}$. We still follow the same “step by step” idea from the forward mode, but the difference is that, we calculate it backward. For example, here we reduce the problem in this way: since in this graph $y = v_7/v_8$, if only we can have $\frac{\partial y}{\partial v_7}$ and $\frac{\partial y}{\partial v_8}$, then this problem should be one step closer towards my target problem.

First of course, we have $\frac{\partial y}{\partial v_9} = 1$, since y and v_9 are the same. Then how do we get $\frac{\partial y}{\partial v_7}$? Again, time for chain rule:

$$\frac{\partial y}{\partial v_7} = \frac{\partial y}{\partial v_9} * \frac{\partial v_9}{\partial v_7} = 1 * \frac{\partial v_9}{\partial v_7} = \frac{\partial (v_7/v_8)}{\partial v_7} = \frac{1}{v_8}. \quad (37)$$

Hmm, let's try to apply a substitution to the terms to simplify this process. Let

$$\bar{v}_i = \frac{\partial y}{\partial v_i}$$

be the derivative of output variable y with regard to intermediate node v_i . It is called the *adjoint* of variable v_i with respect to the output variable y . Using this notation, eq. 37 can be expressed as:

$$\bar{v}_7 = \bar{v}_9 * \frac{\partial v_9}{\partial v_7} = 1 * \frac{1}{v_8}.$$

Note the difference between tangent and adjoint. In the forward mode, we know v_0 and v_1 , then we calculate v_2 , v_3 , ... and then finally we have v_9 , which is the target. Here, we start with knowing $\bar{v}_9 = 1$, and then we calculate \bar{v}_8 , \bar{v}_7 , ... and then finally we have $\bar{v}_0 = \frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial x_0}$, which is also exactly our target. Again, $v_9 = \bar{v}_0$ in this example, given that we are talking about derivative regarding x_0 when we use v_9 . Following this line of calculation, the reverse differentiation mode is also called *adjoint mode*.

With that in mind, let's see the full steps of performing reverse differentiation. First, we need to perform a forward pass to compute the required intermediate values, as shown in tbl. 21.

Table 21: Forward pass in the reverse differentiation mode

Step	Primal computation
0	$v_0 = x_0 = 1$
1	$v_1 = x_1 = 1$
2	$v_2 = \sin(v_0) = 0.84$
3	$v_3 = v_0 v_1 = 1$
4	$v_4 = v_2 + v_3 = 1.84$
5	$v_5 = 1$
6	$v_6 = \exp(v_4) = 6.30$
7	$v_7 = 1$
8	$v_8 = v_5 + v_6 = 7.30$
9	$y = v_9 = \frac{1}{v_8}$

You might be wondering, this looks the same as the left side of tbl. 20. You are right. These two are exactly the same, and we repeat it again to make the point that, this time you cannot perform the calculation with one pass. You must compute the required intermediate results first, and then perform the other “backward pass”, which is the key point in reverse mode.

Table 22: Computation process of the backward pass in reverse differentiation

Step	Adjoint computation
10	$\bar{v}_9 = 1$



Figure 66: Example of reverse accumulation with computational graph

Step Adjoint computation

11	$\bar{v}_8 = \bar{v}_9 \frac{\partial (v_7/v_8)}{\partial v_8} = 1 * \frac{-v_7}{v_8^2} = \frac{-1}{7.30^2} = -0.019$
12	$\bar{v}_7 = \bar{v}_9 \frac{\partial (v_7/v_8)}{\partial v_7} = \frac{1}{v_8} = 0.137$
13	$\bar{v}_6 = \bar{v}_8 \frac{\partial v_8}{\partial v_6} = \bar{v}_8 * \frac{\partial (v_6+v_5)}{\partial v_6} = \bar{v}_8$
14	$\bar{v}_5 = \bar{v}_8 \frac{\partial v_8}{\partial v_5} = \bar{v}_8 * \frac{\partial (v_6+v_5)}{\partial v_5} = \bar{v}_8$
15	$\bar{v}_4 = \bar{v}_6 \frac{\partial v_6}{\partial v_4} = \bar{v}_8 * \frac{\partial \exp(v_4)}{\partial v_4} = \bar{v}_8 * e^{v_4}$
16	$\bar{v}_3 = \bar{v}_4 \frac{\partial v_4}{\partial v_3} = \bar{v}_4 * \frac{\partial (v_2+v_3)}{\partial v_3} = \bar{v}_4$
17	$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2} = \bar{v}_4 * \frac{\partial (v_2+v_3)}{\partial v_2} = \bar{v}_4$
18	$\bar{v}_1 = \bar{v}_3 \frac{\partial v_3}{\partial v_1} = \bar{v}_3 * \frac{\partial (v_0*v_1)}{\partial v_1} = \bar{v}_4 * v_0 = \bar{v}_4$
19	$\bar{v}_{02} = \bar{v}_2 \frac{\partial v_2}{\partial v_0} = \bar{v}_2 * \frac{\partial (\sin(v_0))}{\partial v_0} = \bar{v}_4 * \cos(v_0)$
20	$\bar{v}_{03} = \bar{v}_3 \frac{\partial v_3}{\partial v_0} = \bar{v}_3 * \frac{\partial (v_0*v_1)}{\partial v_0} = \bar{v}_4 * v_1$
21	$\bar{v}_0 = \bar{v}_{02} + \bar{v}_{03} = \bar{v}_4(\cos(v_0) + v_1) = \bar{v}_8 * e^{v_4}(0.54 + 1) = -0.019 * e^{1.84} * 1.54 = -0.18$

Note that things a bit different for x_0 . It is used in both intermediate variables v_2 and v_3 . Therefore, we compute the adjoint of v_0 with regard to v_2 (step 19) and v_3 (step 20), and accumulate them together (step 20).

Similar to the forward mode, reverse differentiation process in [] can be clearly shown in figure fig. 66.

This result $\bar{v}_0 = -0.18$ agrees what we have gotten using the forward mode. However, if you still need another fold of insurance, we can use Owl to perform a numerical differentiation. The code would be similar to that of using algorithmic differentiation as shown before.

```
module D = Owl_numdiff_generic.Make (Dense.Ndarray.D);;
```

```

let x = Arr.ones [|2|]

let f x =
  let x1 = Arr.get x [|0|] in
  let x2 = Arr.get x [|1|] in
  Maths.(div 1. (1. +. exp (x1 *. x2 +. (sin x1))))

```

And then we can get the differentiation result at the point $(x_0, x_1) = (0, 0)$, and it agrees with the previous results.

```

# D.grad f x
- : D.arr =
  C0 C1
R -0.181973 -0.118142

```

Before we move on, did you notice that we get $\frac{\partial y}{\partial x_1}$ for “free” while calculating $\frac{\partial y}{\partial x_0}$. Noticing this will help you to understand the next section, about how to decide which mode (forward or backward) to use in practice.

Forward or Reverse?

Since both modes can be used to differentiate a function, the natural question is which mode we should choose in practice. The short answer is: it depends on your function. In general, given a function that you want to differentiate, the rule of thumb is:

- if the number of input variables is far larger than that of the output variables, then use reverse mode;
- if the number of output variables is far larger than that of the input variables, then use forward mode.

For each input variable, we need to seed individual variable and perform one forward pass. The number of forward passes increase linearly as the number of inputs increases. However, for backward mode, no matter how many inputs there are, one backward pass can give us all the derivatives of the inputs. I guess now you understand why we need to use backward mode for f . One real-world example of f is machine learning and neural network algorithms, wherein there are many inputs but the output is often one scalar value from loss function.

Backward mode needs to maintain a directed computation graph in the memory so that the errors can propagate back; whereas the forward mode does not have to do that due to the algebra of dual numbers. Later we will show example about choosing between these two methods.

A Strawman AD Engine

Surely you don’t want to make these tables every time you are faced with a new computation. Now that you understand how to use forward and reverse

propagation to do algorithmic differentiation, let's look at how to do it with computer programmes. In this section, we will introduce how to implement the differentiation modes using pure OCaml code. Of course, these will be elementary straw man implementations compared to the industry standard module provided by Owl, but nevertheless are important to the understanding of the latter.

We will again use the function in eq. 36 as example, and we limit the computation in our small AD engine to only these basic operations: `add`, `div`, `mul`.

Simple Forward Implementation

How can we represent tbl. 20? An intuitive answer is to build a table when traversing the computation graph. However, that's not a scalable: what if there are hundreds and thousands of computation steps? A closer look at the tbl. 20 shows that an intermediate node actually only need to know the computation results (primal value and tangent value) of its parents nodes to compute its own results. Based on this observation, we can define a data type that preserve these two values:

```
type df = {
  mutable p: float;
  mutable t: float
}

let primal df = df.p
let tangent df = df.t
```

And now we can define operators that accept type `df` as input and outputs the same type:

```
let sin_ad x =
  let p = primal x in
  let t = tangent x in
  let p' = Owl_maths.sin p in
  let t' = (Owl_maths.cos p) *. t in
  {p=p'; t=t'}
```

The core part of this function is to define how to compute its function value p' and derivative value t' based on the input `df` data. Now you can easily extend it towards the `exp` operation:

```
let exp_ad x =
  let p = primal x in
  let t = tangent x in
  let p' = Owl_maths.exp p in
  let t' = p' *. t in
  {p=p'; t=t'}
```

But what about operators that accept multiple inputs? Let's see multiplication.

```
let mul_ad a b =
  let pa = primal a in
  let ta = tangent a in
  let pb = primal b in
  let tb = tangent b in
  let p' = pa *. pb in
  let t' = pa *. tb +. ta *. pb in
  {p=p'; t=t'}
```

Though it require a bit more unpacking, its forward computation and derivative function are simple enough. Similarly, you can extend that towards similar operations: the add and div.

```
let add_ad a b =
  let pa = primal a in
  let ta = tangent a in
  let pb = primal b in
  let tb = tangent b in
  let p' = pa +. pb in
  let t' = ta +. tb in
  {p=p'; t=t'}

let div_ad a b =
  let pa = primal a in
  let ta = tangent a in
  let pb = primal b in
  let tb = tangent b in
  let p' = pa /. pb in
  let t' = (ta *. pb -. tb *. pa) /. (pb *. pb) in
  {p=p'; t=t'}
```

Based on these functions, we can provide a tiny wrapper named `diff`:

```
let diff f =
  let f' x y =
    let r = f x y in
    primal r, tangent r
  in
  f'
```

And that's all! Now we can do differentiation on our previous example.

```
let x0 = {p=1.; t=1.}
let x1 = {p=1.; t=0.}
```

These are inputs. We know the tangent of `x1` with regard to `x0` is zero, and so are the other constants used in the computation.

```
# let f x0 x1 =
  let v2 = sin_ad x0 in
  let v3 = mul_ad x0 x1 in
  let v4 = add_ad v2 v3 in
  let v5 = {p=1.; t=0.} in
  let v6 = exp_ad v4 in
  let v7 = {p=1.; t=0.} in
  let v8 = add_ad v5 v6 in
  let v9 = div_ad v7 v8 in
  v9
val f : df -> df -> df = <fun>

# let pri, tan = diff f x0 x1
val pri : float = 0.13687741466075895
val tan : float = -0.181974376561731321
```

The results are just as calculated in the previous section.

Simple Reverse Implementation

The reverse mode is a bit more complex. As shown in the previous section, forward mode only needs one pass, but the reverse mode requires two passes, a forward pass followed by a backward pass. This further indicates that, besides computing primal values, we also need to “remember” the operations along the forward pass, and then utilise these information in the backward pass. There are multiple ways to do that, e.g. a stack or graph structure. What we choose here is bit different though. Let’s start with the data types we use.

```
type dr = {
  mutable p: float;
  mutable a: float ref;
  mutable adj_fun : float ref -> (float * dr) list -> (float * dr) list
}

let primal dr = dr.p
let adjoint dr = dr.a
let adj_fun dr = dr.adj_fun
```

The `p` is for primal while `a` stands for adjoint. It’s easy to understand. The `adj_fun` is a bit tricky. Let’s see an example:

```
let sin_ad dr =
  let p = primal dr in
  let p' = Owl_maths.sin p in
  let adjfun' ca t =
    let r = !ca *. (Owl_maths.cos p) in
    (r, dr) :: t
  in
  {p=p'; a=ref 0.; adj_fun=adjfun'}
```

It's an implementation of `sin` operation. The `adj_fun` here can be understood as a *placeholder* for the adjoint value we don't know yet in the forward pass. The `t` is a stack of intermediate nodes to be processed in the backward process. It says that, if I have the adjoint value `ca`, I can then get the new adjoint value of my parents `r`. This result, together with the original data `dr`, is pushed to the stack `t`. This stack is implemented in OCaml list.

Let's then look at the `mul` operation with two variables:

```
let mul_ad dr1 dr2 =
  let p1 = primal dr1 in
  let p2 = primal dr2 in

  let p' = Owl_maths.mul p1 p2 in
  let adjfun' ca t =
    let r1 = !ca *. p2 in
    let r2 = !ca *. p1 in
    (r1, dr1) :: (r2, dr2) :: t
  in
  {p = p'; a = ref 0.; adj_fun = adjfun'}
```

The difference is that, this time both of its parents are added to the task stack. For the input data, we need a helper function:

```
let make_reverse v =
  let a = ref 0. in
  let adj_fun _ a t = t in
  {p=v; a; adj_fun}
```

With this function, we can perform the forward pass like this:

```
# let x = make_reverse 1.
val x : dr = {p = 1.; a = {contents = 0.}; adj_fun = <fun>}
# let y = make_reverse 2.
val y : dr = {p = 2.; a = {contents = 0.}; adj_fun = <fun>}
# let v = mul_ad (sin_ad x) y
val v : dr = {p = 1.68294196961579301; a = {contents = 0.}; adj_fun = <fun>}
```

After the forward pass, we have the primal values at each intermediate node, but their adjoint values are all set to zero, since we don't know them yet. And we have this adjoin function. Noting that executing this function would create a list of past computations, which in turn contains its own `adj_fun`. This resulting `adj_fun` remembers all the required information, and know we need to recursively calculate the adjoint values we want.

```
let rec reverse_push xs =
  match xs with
```

```
| [] -> ()
| (v, dr) :: t ->
  let aa = adjoint dr in
  let adjfun = adj_fun dr in
  aa := !aa +. v;
  let stack = adjfun aa t in
  reverse_push stack
```

The `reverse_push` does exactly that. Starting from a list, it gets the top element `dr`, gets the adjoint value we already calculated `aa`, updates it with `v`, and then gets the `adj_fun`. Now that we know the adjoint value, we can use that as input parameter to the `adj_fun` to execute the data of current task and recursively execute more nodes until the task stack is empty.

Now, let's add some other required operations basically by copy and paste:

```
let exp_ad dr =
  let p = primal dr in
  let p' = Owl_maths.exp p in
  let adjfun' ca t =
    let r = !ca *. (Owl_maths.exp p) in
    (r, dr) :: t
  in
  {p=p'; a=ref 0.; adj_fun=adjfun'}
```



```
let add_ad dr1 dr2 =
  let p1 = primal dr1 in
  let p2 = primal dr2 in
  let p' = Owl_maths.add p1 p2 in
  let adjfun' ca t =
    let r1 = !ca in
    let r2 = !ca in
    (r1, dr1) :: (r2, dr2) :: t
  in
  {p = p'; a = ref 0.; adj_fun = adjfun'}
```



```
let div_ad dr1 dr2 =
  let p1 = primal dr1 in
  let p2 = primal dr2 in

  let p' = Owl_maths.div p1 p2 in
  let adjfun' ca t =
    let r1 = !ca /. p2 in
    let r2 = !ca *. (-.p1) /. (p2 *. p2) in
    (r1, dr1) :: (r2, dr2) :: t
  in
  {p = p'; a = ref 0.; adj_fun = adjfun'}
```

We can express the differentiation function `diff` with the reverse mode, with first a forward pass and then a backward pass.

```

let diff f =
  let f' x =
    (* forward pass *)
    let r = f x in
    (* backward pass *)
    reverse_push [(1., r)];
    (* get result values *)
    let x0, x1 = x in
    primal x0, !(adjoint x0), primal x1, !(adjoint x1)
  in
  f'

```

Now we can do the calculation, which are the same as before, and the only difference is the way to build constant values.

```

# let x1 = make_reverse 1.
val x1 : dr = {p = 1.; a = {contents = 0.}; adj_fun = <fun>}
# let x0 = make_reverse 1.
val x0 : dr = {p = 1.; a = {contents = 0.}; adj_fun = <fun>}

# let f x =
  let x0, x1 = x in
  let v2 = sin_ad x0 in
  let v3 = mul_ad x0 x1 in
  let v4 = add_ad v2 v3 in
  let v5 = make_reverse 1. in
  let v6 = exp_ad v4 in
  let v7 = make_reverse 1. in
  let v8 = add_ad v5 v6 in
  let v9 = div_ad v7 v8 in
  v9
val f : dr * dr -> dr = <fun>

```

Now let's do the differentiation:

```

# let pri_x0, adj_x0, pri_x1, adj_x1 = diff f (x0, x1)
val pri_x0 : float = 1.
val adj_x0 : float = -0.181974376561731321
val pri_x1 : float = 1.
val adj_x1 : float = -0.118141988016545588

```

Again, their adjoint values are just as expected.

Unified Implementations

We have shown how to implement forward and reverse AD from scratch separately. But in the real world applications, we often need a system that supports both differentiation modes. How can we build it then? We start with combining the previous two record data types `df` and `dr` into a new data type `t` and its related operations:

```

type t =
| DF of float * float
| DR of float * float ref * adjoint

and adjoint = float ref -> (float * t) list -> (float * t) list

let primal = function
| DF (p, _) -> p
| DR (p, _, _) -> p

let tangent = function
| DF (_, t) -> t
| DR (_, _, _) -> failwith "error: no tangent for DR"

let adjoint = function
| DF (_, _) -> failwith "error: no adjoint for DF"
| DR (_, a, _) -> a

let make_forward p a = DF (p, a)

let make_reverse p =
  let a = ref 0. in
  let adj_fun _a t = t in
  DR (p, a, adj_fun)

let rec reverse_push xs =
  match xs with
  | [] -> ()
  | (v, x) :: t ->
    (match x with
    | DR (_, a, adjfun) ->
      a := !a +. v;
      let stack = adjfun a t in
      reverse_push stack
    | _ -> failwith "error: unsupported type")

```

Now we can operate on one unified data type. Based on this new data type, we can then combine the forward and reverse mode into one single function, using a `match` clause.

```

let sin_ad x =
  let ff = Owl_maths.sin in
  let df p t = (Owl_maths.cos p) *. t in
  let dr p a = !a *. (Owl_maths.cos p) in
  match x with
  | DF (p, t) ->
    let p' = ff p in
    let t' = df p t in
    DF (p', t')
  | DR (p, _, _) ->
    let p' = ff p in
    let adjfun' a t =
      let r = dr p a in
      (r, x) :: t

```

```
in
DR (p', ref 0., adjfun')
```

The code is mostly taken from the previous two implementations, so should be not very alien to you now. Similarly we can also build the multiplication operator:

```
let mul_ad xa xb =
  let ff = Owl_maths.mul in
  let df pa pb ta tb = pa *. tb +. ta *. pb in
  let dr pa pb a = !a *. pb, !a *. pa in
  match xa, xb with
  | DF (pa, ta), DF (pb, tb) ->
    let p' = ff pa pb in
    let t' = df pa pb ta tb in
    DF (p', t')
  | DR (pa, _, _), DR (pb, _, _) ->
    let p' = ff pa pb in
    let adjfun' a t =
      let ra, rb = dr pa pb a in
      (ra, xa) :: (rb, xb) ::: t
    in
    DR (p', ref 0., adjfun')
  | _, _ -> failwith "unsupported op"
```

Now before moving forward, let's pause and think about what's so different among different math functions. First, they may take different number of input arguments; they could be unary functions or binary functions. Second, they have different computation rules. Specifically, three type of computations are involved: `ff`, which computes the primal value; `df`, which computes the tangent value; and `dr`, which computes the adjoint value. The rest are mostly fixed.

Based on this observation, we can utilise the first-class citizen in OCaml, the “module”, to reduce a lot of copy and paste in our code. We can start with two types of modules: unary and binary:

```
module type Unary = sig
  val ff : float -> float
  val df : float -> float -> float
  val dr : float -> float ref -> float
end

module type Binary = sig
  val ff : float -> float -> float
  val df : float -> float -> float -> float -> float
  val dr : float -> float -> float ref -> float * float
end
```

They express both points of difference: first, the two modules differentiate between unary and binary ops; second, each module represents the three core operations: `ff`, `df`, and `dr`. We can focus on the computation logic of each computation in each module:

```
module Sin = struct
  let ff = Owl_maths.sin
  let df p t = (Owl_maths.cos p) *. t
  let dr p a = !a *. (Owl_maths.cos p)
end

module Exp = struct
  let ff = Owl_maths.exp
  let df p t = (Owl_maths.exp p) *. t
  let dr p a = !a *. (Owl_maths.exp p)
end

module Mul = struct
  let ff = Owl_maths.mul
  let df pa pb ta tb = pa *. tb +. ta *. pb
  let dr pa pb a = !a *. pb, !a *. pa
end

module Add = struct
  let ff = Owl_maths.add
  let df _pa _pb ta tb = ta +. tb
  let dr _pa _pb a = !a, !a
end

module Div = struct
  let ff = Owl_maths.div
  let df pa pb ta tb = (ta *. pb -. tb *. pa) /. (pb *. pb)
  let dr pa pb a =
    !a /. pb, !a *. (.-pa) /. (pb *. pb)
end
```

Now we can provide a template to build math functions:

```
let unary_op (module U: Unary) = fun x ->
  match x with
  | DF (p, t) ->
    let p' = U.ff p in
    let t' = U.df p t in
    DF (p', t')
  | DR (p, _, _) ->
    let p' = U.ff p in
    let adjfun' a t =
      let r = U.dr p a in
      (r, x) :: t
    in
    DR (p', ref 0., adjfun')
```

```

let binary_op (module B: Binary) = fun xa xb ->
  match xa, xb with
  | DF (pa, ta), DF (pb, tb) ->
    let p' = B.ff pa pb in
    let t' = B.df pa pb ta tb in
    DF (p', t')
  | DR (pa, _, _), DR (pb, _, _) ->
    let p' = B.ff pa pb in
    let adjfun' a t =
      let ra, rb = B.dr pa pb a in
      (ra, xa) :: (rb, xb) :: t
    in
    DR (p', ref 0., adjfun')
  | _, _ -> failwith "unsupported op"

```

Each template accepts a module, and then returns the function we need. Let's see how it works with concise code.

```

let sin_ad = unary_op (module Sin : Unary)
let exp_ad = unary_op (module Exp : Unary)
let mul_ad = binary_op (module Mul : Binary)
let add_ad = binary_op (module Add: Binary)
let div_ad = binary_op (module Div : Binary)

```

As you can expect, the diff function can also be implemented in a combined way. In this implementation we focus on the tangent and adjoint value of x_0 only.

```

let diff f =
  let f' x =
    let x0, x1 = x in
    match x0, x1 with
    | DF (.., ..), DF (.., ..) ->
      f x |> tangent
    | DR (.., .., ..), DR (.., .., ..) ->
      let r = f x in
      reverse_push [(1., r)];
      !(adjoint x0)
    | _, _ -> failwith "error: unsupported operator"
  in
  f'

```

That's all. We can move on once again to our familiar examples.

```

# let x0 = make_forward 1. 1.
val x0 : t = DF (1., 1.)
# let x1 = make_forward 1. 0.
val x1 : t = DF (1., 0.)

```

```
# let f_forward x =
  let x0, x1 = x in
  let v2 = sin_ad x0 in
  let v3 = mul_ad x0 x1 in
  let v4 = add_ad v2 v3 in
  let v5 = make_forward 1. 0. in
  let v6 = exp_ad v4 in
  let v7 = make_forward 1. 0. in
  let v8 = add_ad v5 v6 in
  let v9 = div_ad v7 v8 in
  v9
val f_forward : t * t -> t = <fun>
# diff f_forward (x0, x1)
- : float = -0.181974376561731321
```

That's just forward mode. With only tiny change of how the variables are constructed, we can also do the reverse mode.

```
# let x0 = make_reverse 1.
val x0 : t = DR (1., {contents = 0.}, <fun>)
# let x1 = make_reverse 1.
val x1 : t = DR (1., {contents = 0.}, <fun>)
# let f_reverse x =
  let x0, x1 = x in
  let v2 = sin_ad x0 in
  let v3 = mul_ad x0 x1 in
  let v4 = add_ad v2 v3 in
  let v5 = make_reverse 1. in
  let v6 = exp_ad v4 in
  let v7 = make_reverse 1. in
  let v8 = add_ad v5 v6 in
  let v9 = div_ad v7 v8 in
  v9
val f_reverse : t * t -> t = <fun>
# diff f_reverse (x0, x1)
- : float = -0.181974376561731321
```

Once again, the results agree and are just as expected.

Forward and Reverse Propagation API

So far we have talked a lot about what is algorithmic differentiation and how it works, with theory, example illustration, and code. Now finally let's turn to using it in Owl. Owl provides both numerical differentiation (in Numdiff.Generic module) and algorithmic differentiation (in Algodiff.Generic module). We have briefly used them in previous sections to validate the calculation results of our manual forward and reverse differentiation examples.

Algorithmic Differentiation is a core module built in Owl, which is one of Owl's special features among other similar numerical libraries. Algodiff.Generic is a functor that accepts a Ndarray modules. By plugging in Dense.Ndarray.S and

Dense.Ndarray.D modules we can have AD modules that supports float32 and float64 precision respectively.

```
module S = Owl_algodiff_generic.Make (Owl_algodiff_primal_ops.S)
module D = Owl_algodiff_generic.Make (Owl_algodiff_primal_ops.D)
```

This `Owl_algodiff_primal_ops` module here might seem unfamiliar to you, but in essence it is mostly an alias of the Ndarray module, with certain matrix and linear algebra functions added in.

We will mostly use the double precision `Algodiff.D` module, but of course using other choices is also perfectly fine.

Expressing Computation

Let's look at the previous example of eq. 36, and express it with the AD module. Normally, the code below should do.

```
open Owl

let f x =
  let x0 = Mat.get x 0 0 in
  let x1 = Mat.get x 0 1 in
  Owl_maths.(1. /. (1. +. exp (sin x0 +. x0 *. x1)))
```

This function accepts a vector and returns a float value, which is exactly what we are looking for. However, the problem is that we cannot directly differentiate this programme. Instead, we need to do some minor but important change:

```
module AD = Algodiff.D

let f x =
  let x0 = AD.Mat.get x 0 0 in
  let x1 = AD.Mat.get x 0 1 in
  AD.Maths.((F 1.) / (F 1. + exp (sin x0 + x0 *. x1)))
```

This function looks very similar, but now we are using the operators provided by the AD module, including the `get` operation and math operations. In AD, all the input/output are of type `AD.t`. There is no difference between scalar, matrix, or ndarray for the type checking.

The `F` is special packing mechanism in AD. It makes a type `t` float. Think about wrapping a float number inside a container that can be recognised in this factory called "AD". And then this factory can produce the differentiation result you want. The `Arr` operator is similar. It wraps an ndarray (or matrix) inside this same container. And as you can guess, there are also unpacking mechanisms. When this AD factory produces some result, to see the result you need to first unwrap this container with the functions `unpack_flt` and `unpack_arr`. For example,

we can directly execute the AD functions, and the results need to be unpacked before being used.

```
open AD

# let input = Arr (Dense.Matrix.D.ones 1 2)
# let result = f input |> unpack_flt
val result : float = 0.13687741466075895
```

Despite this slightly cumbersome number packing mechanism, we can now perform the forward and reverse propagation on this computation in Owl, as will be shown next.

Example: Forward Mode

The forward mode is implemented with the `make_forward` and `tangent` function:

```
val make_forward : t -> t -> int -> t
val tangent : t -> t
```

The forward process is straightforward.

```
open AD

# let x = make_forward input (F 1.) (tag ());; (* seed the input *)
# let y = f x;; (* forward pass *)
# let y' = tangent y;; (* get all derivatives *)
```

All the derivatives are ready whenever the forward pass is finished, and they are stored as tangent values in `y`. We can retrieve the derivatives using `tangent` function.

Example: Reverse Mode

The reverse mode consists of two parts:

```
val make_reverse : t -> int -> t
val reverse_prop : t -> t -> unit
```

Let's look at the code snippet below.

```

open AD

# let x = Mat.ones 1 2;; (* generate random input *)
# let x' = make_reverse x (tag ());; (* init the reverse mode *)
# let y = f x';; (* forward pass to build computation graph *)
# let _ = reverse_prop (F 1.) y;; (* backward pass to propagate error *)
# let y' = adjval x';; (* get the gradient value of f *)

- : A.arr =
  C0 C1
R0 -0.181974 -0.118142

```

The `make_reverse` function does two things for us: 1) wrapping `x` into type `t` that `Algodiff` can process 2) generating a unique tag for the input so that input numbers can have nested structure. By calling `f x'`, we construct the computation graph of `f` and the graph structure is maintained in the returned result `y`. Finally, `reverse_prop` function propagates the error back to the inputs. In the end, the gradient of `f` is stored in the adjacent value of `x'`, and we can retrieve that with `adjval` function. The result agrees with what we have calculated manually.

High-Level APIs

What we have seen is the basic of AD modules. There might be cases you do need to operate these low-level functions to write up your own applications (e.g., implementing a neural network), then knowing the mechanisms behind the scene is definitely a big plus. However, using these complex low level function hinders daily use of algorithmic differentiation in numerical computation task. In reality, you don't really need to worry about forward or reverse mode if you simply use high-level APIs such as `diff`, `grad`, `hessian`, and etc. They are all built on the forward or reverse mode that we have seen, but provide clean interfaces, making a lot of details transparent to users. In this section we will introduce how to use these high level APIs.

Derivative and Gradient

The most basic and commonly used differentiation functions is used for calculating the *derivative* of a function. The AD module provides `diff` function for this task. Given a function `f` that takes a scalar as input and also returns a scalar value, we can calculate its derivative at a point `x` by `diff f x`, as shown in this function signature.

```
val diff : (t -> t) -> t -> t
```

The physical meaning of derivative is intuitive. The function `f` can be expressed as a curve in a cartesian coordinate system, and the derivative at a point is the tangent on a function at this point. It also indicate the rate of change at this point.

Suppose we define a function f_0 to be the triangular function \tanh , we can calculate its derivative at position $x = 0.1$ by simply calling:

```
open Algodiff.D

let f0 x = Maths.(tanh x)
let d = diff f0 (F 0.1)
```

Moreover, the AD module is much more than that; we can easily chain multiple `diff` together to get a function's high order derivatives. For example, we can get the first to fourth order derivatives of f_0 by using the concise code below.

```
let f0 x = Maths.(tanh x);;
let f1 = diff f0;;
let f2 = diff f1;;
let f3 = diff f2;;
let f4 = diff f3;;
```

We can further plot these five functions using Owl, and the result is show in fig. 67.

```
let map f x = Owl.Mat.map (fun a -> a |> pack_flt |> f |> unpack_flt) x;;
let x = Owl.Mat.linspace (-4.) 4. 200;;
let y0 = map f0 x;;
let y1 = map f1 x;;
let y2 = map f2 x;;
let y3 = map f3 x;;
let y4 = map f4 x;;

let h = Plot.create "plot_00.png" in
Plot.plot ~h x y0;
Plot.plot ~h x y1;
Plot.plot ~h x y2;
Plot.plot ~h x y3;
Plot.plot ~h x y4;
Plot.output h;;
```

If you want, you can play with other functions, such as $\frac{1-e^{-x}}{1+e^{-x}}$ to see what its derivatives look like.

A close-related idea to derivative is the *gradient*. As we have introduced in eq. 33, gradient generalises derivatives to multivariate functions. Therefore, for a function that accepts a vector (where each element is a variable) and returns a scalar, we can use the `grad` function to find its gradient at a point. Imagine a 3D surface. At each points on this surface, a gradient consists of three element that each represents the derivative along the x, y or z axis. This vector shows the direction and magnitude of maximum change of a multivariate function.



Figure 67: Higher order derivatives

One important application of gradient is the *gradient descent*, a widely used technique to find minimum values on a function. The basic idea is that, at any point on the surface, we calculate the gradient to find the current direction of maximal change at this point, and move the point along this direction by a small step, and then repeat this process until the point cannot be further moved. We will talk about it in detail in the Regression and Optimisation chapters in our book.

Jacobian

Just like gradient extends derivative, the gradient can also be extended to the *Jacobian matrix*. The `grad` can be applied on functions with vector as input and scalar as output. The `jacobian` function on the other hand, deals with functions that has both input and output of vectors. Suppose the input vector is of length n , and contains m output variables, the jacobian matrix is defined as:

$$\mathbf{J}(y) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

The intuition of Jacobian is similar to that of the gradient. At a particular point in the domain of the target function, If you give it a small change in the input vector, the Jacobian matrix shows how the output vector changes. One application field of Jacobian is in the analysis of dynamical systems. In

a dynamic system $\vec{y} = f(\vec{x})$, suppose $f : \mathbf{R}^n \rightarrow \mathbf{R}^m$ is differentiable and its jacobian is \mathbf{J} .

According to the Hartman-Grobman theorem, the stability of a dynamic system near a stationary point is decided by the eigenvalues of \mathbf{J} . It is stable if all the eigenvalues have negative real parts, otherwise its unstable, with the exception that when the largest real part of the eigenvalues is zero. In that case, the stability cannot be decided by eigenvalues.

Let's revise the two-body problem from Ordinary Differential Equation Chapter. This dynamic system is described by a group of differential equations:

$$\begin{aligned} y'_0 &= y_2, \\ y'_1 &= y_3, \\ y'_2 &= -\frac{y_0}{r^3}, \\ y'_3 &= -\frac{y_1}{r^3}, \end{aligned} \tag{38}$$

We can express this system with code:

```
open Algodiff.D

let f y =
  let y0 = Mat.get y 0 0 in
  let y1 = Mat.get y 0 1 in
  let y2 = Mat.get y 0 2 in
  let y3 = Mat.get y 0 3 in

  let r = Maths.(sqrt ((sqr y0) + (sqr y1))) in
  let y0' = y2 in
  let y1' = y3 in
  let y2' = Maths.( neg y0 / pow r (F 3.)) in
  let y3' = Maths.( neg y1 / pow r (F 3.)) in

  let y' = Mat.ones 1 4 in
  let y' = Mat.set y' 0 0 y0' in
  let y' = Mat.set y' 0 1 y1' in
  let y' = Mat.set y' 0 2 y2' in
  let y' = Mat.set y' 0 3 y3' in
  y'
```

For this functions $f : \mathbf{R}^4 \rightarrow \mathbf{R}^4$, we can then find its Jacobian matrix. Suppose the given point of interest of where all four input variables equals one. Then we can use the `Algodiff.D.jacobian` function in this way.

```

let y = Mat.ones 1 4
let result = jacobian f y

let j = unpack_arr result;;
- : A.arr =
  C0 C1 C2 C3
R0 0 0 1 0
R1 0 0 0 1
R2 0.176777 0.53033 0 0
R3 0.53033 0.176777 0 0

```

Next, we find the eigenvalues of this jacobian matrix with the Linear Algebra module in Owl that we have introduced in previous chapter.

```

let eig = Owl_linalg.D.eigvals j
val eig : Owl_dense_matrix_z.mat =
  C0 C1 C2 C3
R0 (0.840896, 0i) (-0.840896, 0i) (0, 0.594604i) (0, -0.594604i)

```

It turns out that one of the eigenvalue is real and positive, so at current point the system is unstable.

Hessian and Laplacian

Another way to extend the gradient is to find the second order derivatives of a multivariate function which takes n input variables and outputs a scalar. Its second order derivatives can be organised as a matrix:

$$\mathbf{H}(y) = \begin{bmatrix} \frac{\partial^2 y_1}{\partial x_1^2} & \frac{\partial^2 y_1}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 y_1}{\partial x_1 \partial x_n} \\ \frac{\partial^2 y_2}{\partial x_2 \partial x_1} & \frac{\partial^2 y_2}{\partial x_2^2} & \cdots & \frac{\partial^2 y_2}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 y_m}{\partial x_n \partial x_1} & \frac{\partial^2 y_m}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 y_m}{\partial x_n^2} \end{bmatrix}$$

This matrix is called the *Hessian Matrix*. As an example of using it, consider the *newton's method*. It is also used for solving the optimisation problem, i.e. to find the minimum value on a function. Instead of following the direction of the gradient, the newton method combines gradient and second order gradients: $\frac{\nabla f(x_n)}{\nabla^2 f(x_n)}$. Specifically, starting from a random position x_0 , and it can be iteratively updated by repeating this procedure until converge, as shown in eq. 39.

$$x(n+1) = x_n - \alpha \mathbf{H}^{-1} \nabla f(x_n) \quad (39)$$

This process can be easily represented using the `Algodiff.D.hessian` function.

```
open Algodiff.D

let rec newton ?(eta=0.01) ?(eps=1e-6) f x =
  let g = grad f x in
  let h = hessian f x in
  if (Maths.l2norm' g |> unpack_flt) < eps then x
  else newton ~eta ~eps f Maths.(x - eta * g *@ (inv h))
```

We can then apply this method on a two dimensional triangular function to find one of the local minimum values, staring from a random initial point. Note that here the functions has to take a vector as input and output a scalar. We will come back to this method in the in Optimisation chapter with more details.

```
let _ =
  let f x = Maths.(cos x |> sum') in
  newton f (Mat.uniform 1 2)
```

Another useful and related function is `laplacian`, it calculate the *Laplacian operator* $\nabla^2 f$, which is the the trace of the Hessian matrix:

$$\nabla^2 f = \text{trace}(H_f) = \sum_{i=1}^n \frac{\partial^2 f}{\partial x_i^2}.$$

In Physics, the Laplacian can represent the flux density of the gradient flow of a function, e.g. the moving rate of a chemical in a fluid. Therefore, differential equations that contains Laplacian are frequently used in many fields to describe physical systems, such as the gravitational potentials, the fluid flow, wave propagation, and electric field, etc.

Other APIs

Besides, there are also many helper functions, such as `jacobianv` for calculating jacobian vector product; `diff'` for calculating both `f x` and `diff f x`, and etc. They will come handy in certain cases for the programmers. Besides the functions we have already introduced, the complete list of APIs can be found in the table below.

Table 23: List of other APIs in the AD module of Owl

API name	Explanation
<code>diff'</code>	similar to <code>diff</code> , but return <code>(f x, diff f x)</code>
<code>grad'</code>	similar to <code>grad</code> , but return <code>(f x, grad f x)</code>
<code>jacobian'</code>	similar to <code>jacobian</code> , but return <code>(f x, jacobian f x)</code>

API name	Explanation
jacobianv	jacobian vector product of $f : (\text{vector} \rightarrow \text{vector})$ at x along v ; it calculates $(\text{jacobian } f)_v$
jacobianv'	similar to jacobianv, but return $(f_x, \text{jacobianv } f_x)$
jacobianTv	it calculates transpose $((\text{jacobianv } f)_x v)$
jacobianTv'	similar to jacobianTv, but return $(f_x, \text{transpose } (\text{jacobianv } f)_x v)$
hessian'	hessian vector product of $f : (\text{scalar} \rightarrow \text{scalar})$ at x along v ; it calculates $(\text{hessian } f)_v$
hessianv'	similar to hessianv, but return $(f_x, \text{hessianv } f_x v)$
laplacian'	similar to laplacian, but return $(f_x, \text{laplacian } f_x)$
gradhessian	return $(\text{grad } f_x, \text{hessian } f_x)$, $f : (\text{scalar} \rightarrow \text{scalar})$
gradhessian'	return $(f_x, \text{grad } f_x, \text{hessian } f_x)$
gradhessianv	return $(\text{grad } f_x v, \text{hessian } f_x v)$
gradhessianv'	return $(f_x, \text{grad } f_x v, \text{hessian } f_x v)$

Differentiation is an important topic in scientific computing, and therefore is not limited to only this chapter in our book. As we have already shown in previous examples, we use AD in the newton method to find extreme values in optimisation problem in the Optimisation chapter. It is also used in the Regression chapter to solve the linear regression problem with gradient descent. More importantly, the algorithmic differentiation is core module in many modern deep neural libraries such as PyTorch. The neural network module in Owl benefit a lot from our solid AD module. We will elaborate these aspects in the following chapters. Stay tuned!

Internal of Algorithmic Differentiation

Go Beyond Simple Implementation

Now that you know the basic implementation of forward and reverse differentiation, and you have also seen the high level APIs that Owl provides. You might be wondering, how are these APIs implemented in Owl?

It turns out that, the simple implementations we have are not very far away from the industry-level implementations in Owl. There are of course many details that need to be taken care of in Owl, but by now you should be able to understand the gist of it. Without digging too deep into the code details, in this section we give an overview of some of the key differences between the Owl implementation and the simple version we built in the previous sections.

The fig. 68 shows the structure of AD module in Owl, and they will be introduced one by one below. Let's start with the **type definition**.

```
| type t =
```

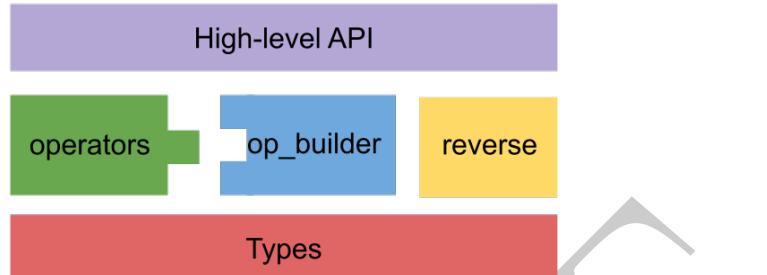


Figure 68: Architecture of the AD module

```

| F of A_elt
| Arr of A_arr
(* primal, tangent, tag *)
| DF of t * t * int
(* primal, adjoint, op, fanout, tag, tracker *)
| DR of t * t ref * op * int ref * int * int ref

and adjoint = t -> t ref -> (t * t) list -> (t * t) list
and register = t list -> t list
and label = string * t list
and op = adjoint * register * label

```

You will notice some differences. First, besides `DF` and `DR`, it also contains two constructors: `F` and `Arr`. This points out one shortcoming of our simple implementation: we cannot process ndarray as input, only float. That's why, if you look back at how our computation is constructed, we have to explicitly say that the computation takes two variable as input. In a real-world application, we only need to pass in a 1×2 vector as input.

You can also note that some extra information fields are included in the `DF` and `DR` data types. The most important one is `tag`, it is mainly used to solve the problem of high order derivative and nested forward and backward mode. This problem is called *perturbation confusion* and is important in any AD implementation. Here we only scratch the surface of this problem. Think about this: what if we want to compute the derivative of:

$$f(x) = x \frac{d(x+y)}{dy},$$

i.e. a function that contains another derivative function? It's simple, since $\frac{d(x+y)}{dy} = 1$, so $f'(x) = x' = 1$. Elementary. There is no way we can do it wrong, even with our strawman AD engine, right?

Well, not exactly. Let's follow our previous simple implementation:

```

# let diff f x =
  match x with
  | DF _, _ ->
    f x |> tangent
  | DR _, _, _ ->
    let r = f x in
    reverse_push [(1., r)];
    !(adjoint x)
val diff : (t -> t) -> t -> float = <fun>

# let f x =
  let g = diff (fun y -> add_ad x y) in
  mul_ad x (make_forward (g (make_forward 2. 1.)) 1.)
val f : t -> t = <fun>

# diff f (make_forward 2. 1.)
- : float = 4.

```

Hmm, the result is 3 at point ($x = 2, y = 2$) but the result should be 1 at any point as we have calculated, so what has gone wrong?

Notice that $x=DF(2,1)$. The tangent value equals to 1, which means that $\frac{dx}{dx} = 1$. Now if we continue to use this same x value in function g , whose variable is y , the same $x=DF(2,1)$ can be translated by the AD engine as $\frac{dx}{dy} = 1$, which is apparently wrong. Therefore, when used within function g , x should actually be treated as $DF(2,0)$.

The tagging technique is proposed to solve this nested derivative problem. The basic idea is to distinguish derivative calculations and their associated attached values by using a unique tag for each application of the derivative operator. More details of method is explained in (Siskind and Pearlmutter 2005).

Now we move on to a higher level. Its structure should be familiar to you now. The **builder** module abstract out the general process of forward and reverse modes, while the **ops** module contains all the specific calculation methods for each operations. Keep in mind that not all operations can follow exact math rules to perform differentiation. For example, what is the tangent and adjoint of the convolution or concatenation operations? These are all included in the `ops.ml`.

We have shown how the unary and binary operations can be built by providing two builder modules. But of course there are many operators that have other type of signatures. Owl abstracts more operation types according to their number of input variables and output variables. For example, the `qr` operations calculates QR decomposition of an input matrix. This operation uses the SIPO (single-input-pair-output) builder template.

In `ops`, each operation specifies three kinds of functions for calculating the primal value (`ff`), the tangent value (`df`), and the adjoint value (`dr`). However, actually some variants are required. In our simple examples, all the constants are either

DF or DR, and therefore we have to define two different functions `f_forward` and `f_reverse`, even though only the definition of constants are different. Now that the float number is included in the data type `t`, we can define only one computation function for both modes:

```
let f_forward x =
  let x0, x1 = x in
  let v2 = sin_ad x0 in
  let v3 = mul_ad x0 x1 in
  let v4 = add_ad v2 v3 in
  let v5 = F 1. in (* change *)
  let v6 = exp_ad v4 in
  let v7 = F 1. in (* change *)
  let v8 = add_ad v5 v6 in
  let v9 = div_ad v7 v8 in
v9
```

Now, we need to consider the question: how to compute DR and F data types together? To do that, we need to consider more cases in an operation. For example, in the previous implementation, one multiplication use three functions:

```
module Mul = struct
  let ff a b = Owl_maths.mul a b
  let df pa pb ta tb = pa *. tb +. ta *. pb
  let dr pa pb a = !a *. pb, !a *. pa
end
```

But now things get more complicated. For `ff a b`, we need to consider, e.g. what if `a` is float and `b` is an ndarray, or vice versa. For `df` and `dr`, we need to consider what happens if one of the input is constant (F or Arr). The builder module also has to take these factors into consideration accordingly.

Finally, in the `reverse` module (`Owl_algodiff_reverse`), we have the `reverse_push` functions. Compared to the simple implementation, it performs an extra `shrink` step. This step checks adjoint `a` and its update `v`, ensuring rank of `a` must be larger than or equal with rank of `v`. Also, the initialisation of the computation is required. An extra `reverse_reset` functions is actually required before the reverse propagation begins to reset all adjoint values to the initial zero values.

Above these parts are the high level APIs. One thing we need to notice is that although `diff` functions looks straightforward to be implemented using the forward and backward mode, the same cannot be said of other functions, especially `jacobian`. Another thing is that, our simple implementation does not support multiple precisions. It is solved by functors in Owl. In `Algodiff.Generic`, all the APIs are encapsulated in a module named `Make`. This module takes in an ndarray-like module and generate AD modules with corresponding precision. If it accepts a `Dense.Ndarray.S` module, it generate AD modules of single precision; if it

is `Dense.Ndarray.D` passed in, the functor generates AD module that uses double precision. (To be precise, this description is not quite correct; the required functions actually has to follow the signatures specified in `Owl_types_ndarray_algodiff.Sig`, which also contains operation about scalar, matrix, and linear algebra, besides `ndarray` operations. As we have seen previously, the input modules here are acutally `Owl_algodiff_primal_ops.S/D`, the wrapper modules for `Ndarray`.)

Extend AD module

The module design shown above brings one large benefit: it is very flexible in supporting adding new operations on the fly. Let's look at an example: suppose the Owl does not provide the operation `sin` in AD module, and to finish our example in eq. 36, what can we do? We can use the `Builder` module in AD.

```
open Algodiff.D

module Sin = struct
  let label = "sin"
  let ff_f a = F A.Scalar.(sin a)
  let ff_arr a = Arr A.(sin a)
  let df _cp ap at = Maths.(at * cos ap)
  let dr a _cp ca = Maths.(!ca * cos (primal a))
end
```

As a user, you need to know how the three types of functions work for the new operation you want to add. These are defined in a module called `sin` here. This module can be passed as parameters to the builder to build a required operation. We call it `sin_ad` to make it different from what the AD module actually provides.

```
let sin_ad = Builder.build_siso (module Sin : Builder.Siso)
```

The `siso` means “single input, single output”. That's all! Now we can use this function as if it is a native operation. You will find that this new operator works seamlessly with existing ones.

```
# let f x =
  let x1 = Mat.get x 0 0 in
  let x2 = Mat.get x 0 1 in
  Maths.(div (F 1.) (F 1. + exp (x1 * x2 + (sin_ad x1))))
val f : t -> t = <fun>

# let x = Mat.ones 1 2
val x : t = [Arr(1,2)]

# let _ = grad f x |> unpack_arr
- : A.arr =
  C0 C1
R0 -0.181974 -0.118142
```

Lazy Evaluation

Using the `Builder` enables users to build new operations conveniently, and it greatly improve the clarity of code. However, with this mechanism comes a new problem: efficiency. Imagine that a large computation that consists of hundreds and thousands of operations, with a function occurs many times in these operations. (Though not discussed yet, in a neural network which utilises AD, it is quite common to create a large computation where basic functions such as `add` and `mul` are repeated tens and hundreds of times.) With the current `Builder` approach, every time this operation is used, it has to be created by the builder again. This is apparently not efficient. We need some mechanism of caching. This is where the *lazy evaluation* in OCaml comes to help.

```
val lazy: 'a -> 'a lazy_t

module Lazy :
  sig
    type 'a t = 'a lazy_t
    val force : 'a t -> 'a
  end
```

As shown in the code above, OCaml provides a built-in function `lazy` that accept an input of type '`a`' and returns a '`'a lazy_t`' object. It is a value of type '`'a`' whose computation has been delayed. This lazy expression won't be evaluated until it is called by `Lazy.force`. The first time it is called by `Lazy.force`, the expresion is evaluated and the result is saved; and thereafter every time it is called by `Lazy.force`, the saved results will be returned without evaluation.

Here is an example:

```
# let x = Printf.printf "hello world!"; 42
hello world!
val x : int = 42
# let lazy_x = lazy (Printf.printf "hello world!"; 42)
val lazy_x : int lazy_t = <lazy>
# let _ = Stdlib.Lazy.force lazy_x
hello world!
- : int = 42
# let _ = Stdlib.Lazy.force lazy_x
- : int = 42
# let _ = Stdlib.Lazy.force lazy_x
- : int = 42
```

In this example you can see that building `lazy_x` does not evaluate the content, which is delayed to the first `Lazy.force`. After that, ever time `force` is called, only the value is returned; the `x` itself, including the `printf` function, will not be evaluated.

We can use this mechanism to improve the implementation of our AD code.

Back to our previous section where we need to add a `sin` operation that the AD module supposedly “does not provide”. We can still do:

```
open Algodiff.D

module Sin = struct
  let label = "sin"
  let ff_f a = F A.Scalar.(sin a)
  let ff_arr a = Arr A.(sin a)
  let df _cp ap at = Maths.(at * cos ap)
  let dr a _cp ca = Maths.(!ca * cos (primal a))
end
```

This part is the same, but now we need to utilise the lazy evaluation:

```
let _sin_ad = lazy Builder.build_siso (module Sin : Builder.Siso)
let sin_ad = Lazy.force _sin_ad
```

In this way, regardless of how many times this `sin` function is called in a massive computation, the `Builder.build_siso` process is only invoked once.

What we have talked about is the lazy evaluation at the compiler level, and do not mistake it with another kind of lazy evaluation that are also related with the AD. Think about that, instead of computing the specific numbers, each step accumulates on a graph, so that computation like primal, tangent, adjoint etc. all generate a graph as result, and evaluation of this graph can only be executed when we think it is suitable. This leads to delayed lazy evaluation. Remember that the AD functor takes an ndarray-like module to produce the `Algodiff.S` or `Algodiff.D` modules, and to do what we have described, we only need to plugin another ndarray-like module that returns graph instead of numerical value as computation result. This module is called the *computation graph* module. It is also a quite important idea, and we will talk about it in detail in the second part of this book.

Summary

In this chapter, we introduce the idea of algorithmic differentiation (AD) and how it works in Owl. First, we start with the classic chain rule in differentiation and different methods to do it, and the benefit of AD. Next, we use an example to introduce two basic modes in AD: the forward mode and the reverse mode. This section is followed by a step-by-step coding of a simple strawman AD engine using OCaml. You can see that the core idea of AD can be implemented with surprisingly simple code. Then we turn to the Owl side: first, how Owl support what we have done in the strawman implementation with the forward and reverse propagation APIs; next, how Owl provides various powerful high level APIs to enable users to directly perform AD. Finally, we give an in-depth

introduction to the implementation of the AD module in Owl, including some details that enhance the simple strawman code, how to build user-defined AD computation, and using lazy evaluation to improve performance, etc. Hopefully, after finishing this chapter, you can have a solid understanding of both its theory and implementation.

References

- Griewank, Andreas, and others. 1989. “On Automatic Differentiation.” *Mathematical Programming: Recent Developments and Applications* 6 (6): 83–107.
- Siskind, Jeffrey Mark, and Barak A. Pearlmutter. 2005. “Perturbation Confusion and Referential Transparency: Correct Functional Implementation of Forward-Mode Ad.” In *17th International Workshop, Ifl’05 (2005)*, edited by Jerey Mark Siskind and Barak A. Pearlmutter. Springer. <http://mural.maynoothuniversity.ie/566/>.

Optimisation

Optimisation is one of the fundamental functionality in numerical computation. In this chapter, we will briefly introduce the optimisation methods, and how Owl can be used to supports some of them.

Introduction

Mathematical optimisation deals with the problem of finding minimums or maximums of a function. The solution can be numerical if closed-form expression does not exist. An optimisation problem has the form:

$$\begin{aligned} & \text{minimise } f_0(\mathbf{x}), \\ & \text{subject to } f_i(\mathbf{x}) \leq b_i, i = 1, 2, \dots, m. \end{aligned} \tag{40}$$

Here \mathbf{x} is a vector that contains all the *optimisation variable*: $\mathbf{x} = [x_0, x_1, \dots, x_n]$. Function $f_0 : \mathbf{R}^n \rightarrow \mathbf{R}$ is the optimisation target, and is called an *objective function*, or *cost function*. An optimisation problem could be bounded by zero or more *constraints*. $f_i : \mathbf{R}^n \rightarrow \mathbf{R}$ in a constraint is called a *constraint function*, which are bounded by the b_i 's. The target is to find the optimal variable values \mathbf{x}^* so that f_0 can take on a maximum or minimum value.

An optimisation problem formalises the idea “maximum benefit/minimise cost with given constraint”, which is a widely applicable topic in many real world problems: scheduling computation/network resources, optimisation of investment portfolio, fitting math model based on observed data, logistics, aero engineering, competitive games... Optimisation has already been applied in many areas.

An optimisation problem can be categorised into different types. In eq. 40, if for all the objective functions and constraint functions, we have:

$$f_i(\alpha x + \beta y) = \alpha f_i(x) + \beta f_i(y), \tag{41}$$

the optimisation problem is then called *linear optimisation*. It is an important class of optimisation problems. If we change the “=” to “ \leq ” in eq. 41, it would make all the functions to be *convex*, and the problem then becomes *convex optimisation*, which can be seen as a generalised linear optimisation. In the optimisation world, convexity is considered as the watershed between easy and difficult problems; because for most convex problems, there exist efficient algorithmic solutions.

Linear optimisation is important because non-negativity is a usual constraint on real world quantities, and that people are often interested in additive bounds. Besides, many problems can be approximated by a linear model. Though still limited by actual problem size, the solution of most linear optimisation problems are already known and provided by off-the-shelf software tools. The

text book (Boyd and Vandenberghe 2004) focuses exclusively on the topic of convex optimisation.

Looking back at eq. 40, if we remove the constraints, it becomes an *unconstrained optimisation* problem. If f is convex and differentiable, this problem can be seen as finding the root of the derivative of f so that $f'(x^*) = 0$. As for the constrained version, one commonly used type is the *linear programming problem* where all the functions are linear. There are also other types of optimisations such as quadratic programming, semi-definite programming, etc. One subset of constrained optimisation is the *equality constrained optimisation* where all the constraints are expressed in the form of equality $Ax = b$. This set of problem can be simplified into the corresponding unconstrained problems.

Optimisation covers a wide range of topics and we can only give a very brief introduction here. In the rest of this chapter, we mostly cover the unconstrained and local optimisation. We will cover the other more advanced content briefly in the end of this chapter, and refer readers to classic books such as (Boyd and Vandenberghe 2004) and (Fletcher 2013) for more information.

This chapter uses differentiation techniques we have introduced in the previous chapter. Compared to numerical differentiation, the algorithmic differentiation guarantees a true derivative value without loss of accuracy. For the rest of this chapter, we prefer to use the algorithmic differentiation to compute derivatives when required, but of course you can also use the numerical differentiation.

Root Finding

We have seen some examples of root finding in the Mathematical Functions chapter. *Root finding* is the process which tries to find zeroes or *roots* of continuous functions. It is not an optimisation problem, but these two topics are closely related. It is beneficial for users to learn about the methods used in optimisation if they understand how the root finding algorithms work, e.g. how to find the root by bracketing and how to find target in an iterative manner.

The bisection method is a simple iterative method to find roots. Let's use $f(x) = x^2 - 2$ as an example. We know that the root of this equation is $x = \sqrt{2} = 1.4142135623\dots$. To find this solution, the bisection methods works like this: we know the solution must be between 1 and 2, so first we set x to the middle point $x = 1.5$; since $x^2 = 2.25$ is larger than 2, it means this x is too large; so next we try $x = 1\frac{1}{4}$, which is too small; so next we try $1\frac{3}{8}\dots$. Bisection is slow to converge. It takes about 50 iterations to reach a precision of 14 digits after the decimal point. However, it is a solid and reliable method to find roots, as long as the function is continuous in the given region. Owl provides the `owl_maths_root.bisec` function that implements this method.

Next is the Newton method. It utilises the derivative of objective function f . It starts with a initial value x_0 , and iterative update it following this process until converges:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \quad (42)$$

We can use the Algorithm Differentiation module in Owl to perform the derivative calculation. Again, let's find the root of $x^2 - 2 = 0$ using this method. The Owl code here is just a plain translation of eq. 42.

```
open Algodiff.D

let f x = Maths.(x ** (F 2.) - (F 2.))

let _ =
  let x = ref 1. in
  for _ = 0 to 6 do
    let g = diff f (F !x) |> unpack_elt in
    let v = f (F !x) |> unpack_elt in
    x := !x -. v /. g;
    Printf.printf "%,.15f\n" !x
  done
```

The resulting sequence is very short compared to the bisection method:

```
1.500000000000000
1.416666666666667
1.414215686274510
1.414213562374690
1.414213562373095
1.414213562373095
1.414213562373095
```

The Newton method is very efficient: it has quadratic convergence which means the square of the error at one iteration is proportional to the error at the next iteration. It is the basis of many powerful numerical methods, such as optimisation, multiplicative inverses of numbers and power series, and solving transcendental equations, etc.

The Newton method requires the function to be smooth. If f is not smooth or computing derivative is not always available, we need to approximate the tangent at one point with a secant through two points. This is called a *Secant Method*:

$$f'(x) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}. \quad (43)$$

This method does not need to compute derivatives, and has similar convergence property as the Newton method. In the Secant method, two points are used at each iteration in calculating derivatives. As an improvement, the *Inverse*

Quadratic Interpolation (IQI) method uses three points to do the interpolation. The benefit of the method is that it's fast at the end of iterating, but unstable in during process.

These methods can be combined together as an even more powerful method: the *Brent's Method*. It is generally considered the best of the root-finding routines. It combines the robustness of Bisection methods, and the iteration speed of Secant and IQI methods. The idea is to use the fast algorithm if possible, and turn to the slow but reliable method when in doubt. This method is indeed what we implement in Owl for the `owl_maths_root.brent` method. For example, the above example, can be simply solved with a one-liner.

```
# let f x = x *. x -. 2.
val f : float -> float = <fun>
# Owl_maths_root.brent f 0. 2.
- : float = 1.41421331999550248
```

Univariate Function Optimisation

Now that we have briefly introduced how root-finding works and some classic methods, let's move on to the main topic of this chapter: unconstrained optimisation problems. Let's start with the simple case that there is only one variable in the objective function. We will introduce the optimisation methods for multivariate functions in the next section, and they all apply to the univariate case, but the specific algorithms can work faster. Besides, understanding the optimisation of univariate functions can be a good step before getting to know the multivariate ones.

Use Derivatives

If a function is continuous and differentiable, then one obvious solution to find extreme values is to locate where the derivatives equals 0:

$$f'(x) = 0$$

This leads us back to our root finding solutions. If you already know the analytical form of $f'(x)$, it's good. For example, if $f(x) = x^2 - x$, then you can directly find root for $g(x) = 2x - 1$. Otherwise, you can use the differentiation functions in owl. Let's look at an example. The objective function is in a hump shape:

$$f(x) = \frac{1}{(x - 0.3)^2 + 0.01} + \frac{1}{(x - 0.9)^2 + 0.04} - 6$$

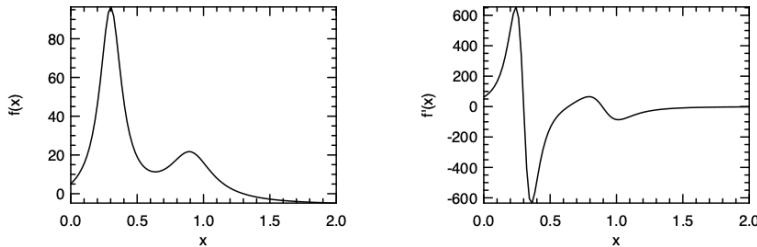


Figure 69: The hump function and its derivative function

```
open Algodiff.D

let f x = Maths.(
  (F 1.) / ((x - F 0.3) ** (F 2.) + F 0.01) +
  (F 1.) / ((x - F 0.9) ** (F 2.) + F 0.04) - F 6.)

let g = diff f

let f' x = f (F x) |> unpack_flt

let g' x = g (F x) |> unpack_flt
```

To better understand the optimisation along this function, we can visualise the image as below:

```
let _ =
  let h = Plot.create ~m:1 ~n:2 "plot_hump.png" in
  Plot.set_pen_size h 1.5;
  Plot.subplot h 0 0;
  Plot.plot_fun ~h f' 0. 2.;
  Plot.set_ylabel h "f'(x)";
  Plot.subplot h 0 1;
  Plot.plot_fun ~h g' 0. 2.;
  Plot.set_ylabel h "f(x)";
  Plot.output h
```

And then you can find the extreme values using the root finding algorithm, such as Brent's:

```
# Owl_maths_root.brent g' 0. 0.4
- : float = 0.30037562625819042
# Owl_maths_root.brent g' 0.4 0.7
- : float = 0.63700940626897
# Owl_maths_root.brent g' 0.7 1.0
- : float = 0.892716303287079405
```

It seems to work fine and find the extreme values correctly in the given range. However, the problem of this method is that you cannot be certain which is maximum and which is minimum.

Golden Section Search

Here we face the similar question again: what if computing derivatives of the function is difficult or not available? That leads us to some search-based approach. We have seen how we can keep reducing a pair of ranges to find the root of a function. A close analogue in optimisation is also a search method called *Golden Section Search*. It's an optimisation method that does not require calculating derivatives. It is one choice to do optimisation if your function has a discontinuous first or second derivative.

The basic idea is simple. It also relies on keeping reducing a “range” until it is small enough. The difference is that, instead of using only two numbers, this search method uses three numbers: $[a, b, c]$. It contains two ranges: $[a, b]$ and $[b, c]$. For every iteration, we need to find a new number d within one of the two ranges. For example, if we choose the d within $[b, c]$, and if $f(b) > f(d)$, then the new triplet becomes $[b, d, c]$, otherwise the new triplet is chosen as $[a, d, b]$. With this approach, the range of this triplet keeps reducing until it is small enough and the minimum value can thus be found.

Then the only question is how to choose the suitable d point at each step. This approach first chooses the larger of the two ranges, either $[a, b]$ or $[b, c]$. And then instead of choosing the middle point in that range, it uses the fractional distance 0.38197 from the central point of the triplet. The name comes from the Golden Ratio and length of range is also closely related with it. This method is slow but robust. It guarantees that each new iteration will bracket the minimum to a range just 0.61803 times the size of the previous one.

Multivariate Function Optimisation

The methods for univariate scenarios can be extended to solving multivariate optimisation problems. The analogue of derivative in this multi-dimensional space is the *gradient*, which we have already seen in previous chapters. To find the extreme values of a function with multiple independent values, you also have the same two options: to use gradients, or not.

Nelder-Mead Simplex Method

First, similar to the Golden Section Search or Brent's, you can always opt for a non-gradient method, which is as slow as it is robust. One such method we can use is the *Nelder-Mead Simplex Method*. As its name suggests, it is probably the simplest way to minimise a fairly well-behaved multi-variate function. It simply goes downhill in a straightforward way, without special assumptions about the objective function.

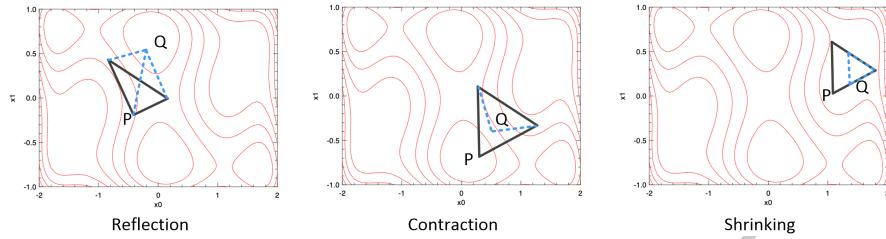


Figure 70: Different movement of simplex in Nelder-Mead optimisation method

The basic idea of this method is to move a “simplex” gradually towards the lowest point on the function. In a N -dimensional space (where the function contains N variables), this simplex consists of $N + 1$ points. For example, for a function that has 2 variables, the simplex is a triangle; for 3 variables, a tetrahedron is used, etc.

From a starting point, the simplex move downwards step by step. At each step, the “highest” point p that has the largest value on the function is found, then p can be moved in three possible ways:

1. through the opposite face of the simplex to a new point; if this *reflected* point is now not the “worst” point (point that leads to largest values on the function) among the $N + 1$ simplex points, accept it;
2. if the reflected point is the worst, then try to *contract* it towards the remaining points; if it is not the worst point, accept it;
3. if the contracted point is still the worst, then you have to *shrink* the simplex.

Repeat this process until it reaches a “valley”, where the method “contracts itself in the transverse direction and tries to ooze down the valley”. This three different methods are illustrated in fig. 70, where there are two variables and the simplex is a triangle.

There are some other method that does not rely on computing gradients such as Powell’s method. If the function is kind of smooth, this method can find the direction in going downhill, but instead of computing gradient, it relies on a one-dimensional optimisation method to do that, and therefore faster than the simplex method. But this method is always a robust and cost-effective way to try solving an optimisation problem at the beginning.

Gradient Descent Methods

A *descent method* is an iterative optimisation process. The idea is to start from an initial value, and then find a certain *search direction* along a function to decrease the value by certain *step size* until it converges to a local minimum. This process can be illustrated in fig. 71.

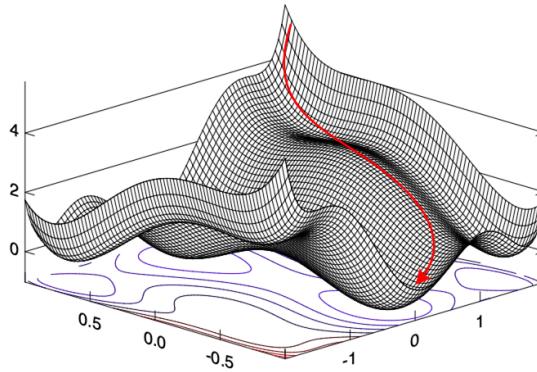


Figure 71: Reach the local minimum by iteratively moving downhill

Therefore, we can describe the n -th iteration of descent method as:

1. calculate a descent direction d ;
2. choose a step size α ;
3. update the location: $x_{n+1} = x_n + \alpha d$.

Repeat this process until a stopping condition is met, such as the update is smaller than a threshold.

Among the descent methods, the *Gradient Descent* method is one of the most widely used algorithms to perform optimisation and the most common way to optimise neural networks, which will be discussed in detail in the Neural Network chapter. Based on the descent process above, Gradient Descent method uses the function gradient to decide its direction d . The process can be described as:

1. calculate a descent direction $-\nabla f(x_n)$;
2. choose a step size α ;
3. update the location: $x_{n+1} = x_n + \alpha \nabla f(x_n)$.

Here ∇ denotes the gradient. The distance α along a certain direction is also called *learning rate*. In a gradient descent process, when looking for the minimum, the point always follows the direction that is against the direction that is represented by the negative gradient.

We can easily implement this process with the algorithmic differentiation module in Owl. Let's look at one example. Here we use the Rosenbrock function which is usually used as a performance test for optimisation problems. The function is defined as:

$$f(x, y) = (a - x)^2 + b(y - x^2)^2. \quad (44)$$

The parameters are usually set as $a = 1$ and $b = 100$.

```
open Algodiff.D
module N = Dense.Ndarray.D

let rosenbrock a =
  let x = Mat.get a 0 0 in
  let y = Mat.get a 0 1 in
  Maths.( (F 100.) * (y - (x ** (F 2.))) ** (F 2.) + (F 1. - x) ** (F 2.) |> sum')
```

Now we hope to apply the gradient descent method and observe the optimisation trajectory.

```
let a = N.of_array [|2.; -0.5|] [|1; 2|]
let traj = ref (N.copy a)
let a = ref a
let eta = 0.0001
let n = 200
```

As preparation, we use the initial starting point [2, -0.5]. The step size eta is set to 0.0001, and the iteration number is 100. Then we can perform the iterative descent process. You can also run this process in a recursive manner.

```
let _ =
  for i = 1 to n - 1 do
    let u = grad rosenbrock (Arr !a) |> unpack_arr in
    a := N.(sub !a (scalar_mul eta u));
    traj := N.concatenate [|!traj; (N.copy !a)|]
done
```

We apply the `grad` method on the Rosenbrock function iteratively, and the updated data `a` is stored in the `traj` array. Finally, let's visualise the trajectory of the optimisation process.

```
let plot () =
  let a, b = Dense.Matrix.D.meshgrid (-2.) 2. (-1.) 3. 50 50 in
  let c = N.(scalar_mul 100. (pow_scalar (sub b (pow_scalar a 2.)) 2.) +
             (pow_scalar (scalar_sub 1. a) 2.)) in

  let h = Plot.create ~m:1 ~n:2 "plot_gradients.png" in
  Plot.subplot h 0 0;
  Plot.(mesh ~h ~spec:[ NoMagColor ] a b c);

  Plot.subplot h 0 1;
  Plot.contour ~h a b c;

  let vx = N.get_slice [[]; [0]] !traj in
  let vy = N.get_slice [[]; [1]] !traj in
  Plot.plot ~h vx vy;
  Plot.output h
```



Figure 72: Optimisation process of gradient descent on multivariate function

We first create a mesh grid based on the Rosenbrock function to visualise the 3D image, and then on the 2D contour image of the same function we plot how the result of the optimisation is updated, from the initial starting point towards a local minimum point. The visualisation results are shown in fig. 72. On the right figure the black line shows the moving trajectory. You can see how it moves downwards along the slope in the right side figure.

Optimisation lays at the foundation of machine learning and neural network training. In the `owl.Optimise` module, we provide a `minimise_fun` function to perform this task. This function is actually an internal function that aims mainly to serve the Neural Network module, but nevertheless we can still try to use this function to solve a optimisation problem with gradient descent method.

This function works based on the Algorithm Differentiation module. It minimises function in the form of $f : x \rightarrow y$ with regard to x . x is an AD ndarray, and y is an AD scalar value. This function is implemented following the iterative descent approach. Let's use the previous Rosenbrock function example to demonstrate how it works.

```
let p = Owl_optimise.D.Params.default ()
let _ = p.epochs <- 10.
let _ = p.gradient <- Owl_optimise.D.Gradient.GD
```

First, we set the optimisation parameters. The `owl_optimise.D.Params` module contains several categories of parameters, including the gradient method, learning rate, loss functions, regularisation method, momentum method, epoch and batch etc. We will introduce these different parts in the Neural Network Chapter. Currently, it suffices to just set the iteration number `epochs` to something like 10 or 20 iterations. Then we set the gradient method to be the gradient descent. Then we can just executing the code, starting from the same starting point:

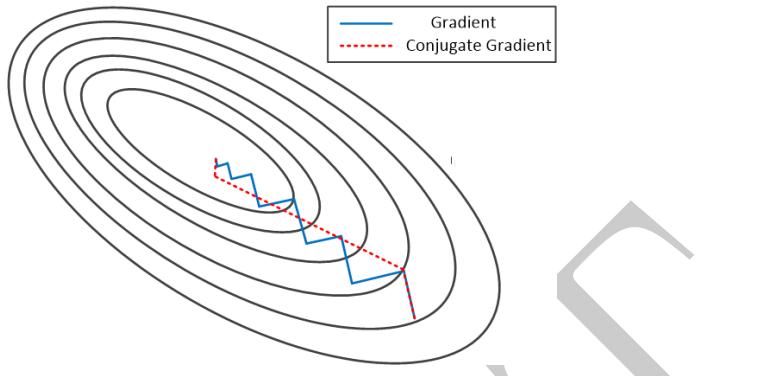


Figure 73: Compare conjugate gradient and gradient descent

```
let init_value = N.of_array [|2.;-0.5|] [|1;2|] |> pack_arr
let _ = Owl_optimise.D.minimise_fun p rosenbrock init_value
```

This function output enhanced log result which in part looks like below. It shows how the function value, starting at the initial point, is quickly reduced to the bottom within only 10 steps using gradient descent.

```
...
2020-09-13 10:46:49.805 INFO : T: 00s | E: 1.0/10 | B: 1/10 | L: 2026.000000
2020-09-13 10:46:49.806 INFO : T: 00s | E: 2.0/10 | B: 2/10 | L: 476.101033
2020-09-13 10:46:49.807 INFO : T: 00s | E: 3.0/10 | B: 3/10 | L: 63.836145
2020-09-13 10:46:49.807 INFO : T: 00s | E: 4.0/10 | B: 4/10 | L: 37.776798
2020-09-13 10:46:49.808 INFO : T: 00s | E: 5.0/10 | B: 5/10 | L: 21.396863
2020-09-13 10:46:49.809 INFO : T: 00s | E: 6.0/10 | B: 6/10 | L: 11.742345
2020-09-13 10:46:49.809 INFO : T: 00s | E: 7.0/10 | B: 7/10 | L: 6.567733
2020-09-13 10:46:49.809 INFO : T: 00s | E: 8.0/10 | B: 8/10 | L: 4.085909
2020-09-13 10:46:49.810 INFO : T: 00s | E: 9.0/10 | B: 9/10 | L: 3.016714
2020-09-13 10:46:49.810 INFO : T: 00s | E: 10.0/10 | B: 10/10 | L: 2.594318
...
```

Conjugate Gradient Method

One problem with the Gradient Descent is that it does not perform well on all functions. For example, if the function forms a steep and narrow valley, gradient descent takes many small steps to reach the minimum, bouncing back and forth, even if the function is in a perfect quadratic form.

The *Conjugate Gradient* method can solve this problem. It was first proposed by Hestenes and Stiefel in their work “Methods of Conjugate Gradients for Solving Linear Systems” in 1952. It is similar to the gradient descent, but at each step, the new direction does not totally follow the new gradient, but somehow *conjugated* to the old gradients and to all previous directions traversed.

For example, fig. 73 compares the different descent efficiency of the conjugate gradient with gradient descent. Both methods start from the same position and go for the same direction. At the next point, the gradient descent follows the direction of the descent, which is a blunt one since this function is steep. But the conjugate method thinks, “hmm, this seems like a steep turn of direction, and I would prefer following the previous momentum a little bit”. As a result, the conjugate method follows a direction in between (the red dotted line), and it turns out that the new direction avoids all the bouncing and finds the minimum more efficiently than the gradient descent method.

In computation, instead of $-\nabla f(x_n)$, conjugate gradient method chooses another way to calculate the descent direction. It maintains two sequences of updates:

$$\begin{aligned} x_{n+1} &= x_n - \alpha_n A y_n \\ y_{n+1} &= x_{n+1} + \beta_n y_n \end{aligned} \quad (45)$$

where

$$\begin{aligned} \alpha_n &= \frac{x_n^T y_n}{y_n^T A y_n} \\ \beta_n &= \frac{x_{n+1}^T x_{n+1}}{x_n^T x_n}. \end{aligned}$$

Here x_n is the function value to be minimised. A is a symmetric and positive-definite real matrix that denotes the system (describe in detail). Similar to the gradient descent method, the conjugate gradient is supported in the Owl optimisation module as `owl_optimise.D.Gradient.CG`.

Newton and Quasi-Newton Methods

There is also a Newton Method in optimisation (it is not to be confused with the newton method used in root-finding). Still following the basic process of descent method, newton method starts from a initial point and then repeat the process:

1. compute the descent direction: $d = -\frac{\nabla f(x_n)}{\nabla^2 f(x_n)}$
2. choose a step size α ;
3. update the location: $x_{n+1} = x_n + \alpha d$.

Here $\nabla^2 f(x_n)$ denotes the second-order derivatives of function f . For a scalar-valued function, its 2nd order derivatives can be represented by its Hessian matrix, which is introduced in the Algorithmic Differentiation chapter. With the hessian matrix \mathbf{H} , the update process of newton method can be expressed as:

$$x_{n+1} = x_n - \alpha \mathbf{H}_n^{-1} \nabla f(x_n).$$

This can also be implemented in Owl with algorithmic differentiation.

```
open Owl
open Algodiff.D

let rec newton ?(eta=0.01) ?(eps=1e-6) f x =
  let g, h = (gradhessian f) x in
  if (Maths.l2norm' g |> unpack_flt) < eps then x
  else newton ~eta ~eps f Maths.(x - eta * g *@ (inv h))

let _ =
  let f x = Maths.(cos x |> sum') in
  let y = newton f (Mat.uniform 1 2) in
  Mat.print y
```

One nice property about the Newton's method is its rate of convergence: it converges quadratically. However, one big problem with the newton method is the problem size. In the real world applications, it is not rare to see optimisation problems with thousands, millions or more variants. In these cases, it is impractical to compute the Hessian matrix, not to mention its inverse.

Towards this end, the *Quasi-newton* methods are proposed. The basic idea is to iteratively build up an approximation of the inverse of the Hessian matrix. Their convergence is fast, but not as efficient as the Newton's method. It takes about n quasi-newton iterations to progress similarly as the Newton's method. The most important method in this category is BFGS (Broyden-Fletcher-Goldfarb-Shanno), named after its four authors. In the BFGS algorithm, the search direction d_i at each iteration i is calculated by

$$A_i p_i = -\nabla f(x_i).$$

Here A_i is the approximation of Hessian matrix which is of the same shape $m \times m$. It is iteratively updated at each step. As a practical enhancement to the algorithm, the Limited-BFGS (L-BFGS) address the memory usage issue in BFGS. Instead of the large approximate matrix A_i , this method only stores a small number of vectors to represent this matrix, and also keeps updates from the last several iterations.

Global Optimisation and Constrained Optimisation

This chapter mainly focuses on unconstrained optimisation, mostly to find local optima. In the rest of this chapter we will give a very very brief introduction to global optimisation and constrained optimisation.

The basic idea of global optimisation is to provide effective search methods and heuristics to traverse the search space effectively. One method is to start from a sufficient number of initial points and find the local optima, then choose the smallest/largest value from them. Another heuristic is to try stepping away from

a local optimal value by taking a finite amplitude step away from it, perform the optimisation method, and see if it leads to a better solution or still the same.

One example of algorithm: *Simulated Annealing Methods*. A suitable system to apply Simulated Annealing consists of several elements. First, it contains a finite set S , and a cost function f that is defined on this set. There is also a non-increasing function T that projects the set of positive integers to real positive value. $T(t)$ is called the *temperature* at time t . Suppose at time t , the current state is i in S . It choose one of its neighbours j randomly. Next, if $f(i) < f(j)$ then j is used as the next state. If not so, then j is chosen as the next state with a probability of $e^{-\frac{f(j)-f(i)}{T(t)}}$, otherwise the next state stays to be i . Starting from an initial state x_0 , this process is repeated for a finite number of steps to find the optimum.

This method is inspired by thermodynamics, where metals cool and anneal starting from a high temperature. During this process, the atoms can often find the minimum energy state of the system automatically, just like finding the global optimum. The simulated annealing has already been used to solve the famous NP-hard traveling salesman problem to find the shortest path.

The *Constrained Optimisation* is another large topic we haven't covered in this chapter. Unlike all the optimisation problems we have seen so far, in a real-world problem it is hardly the case that you can free explore the optimum without any restriction, local or global. More often than not, there are certain constraints on the variables.

For example, we can have an objective function: $f(\mathbf{x}) = \mathbf{c}^T \mathbf{x}$, where \mathbf{c} is a parameter vector. This function is subject to non-negativity condition: $x_i \geq 0$ and $A\mathbf{x} \leq b$, where b and A are known vector and matrix parameters. Under such constraints, this optimisation problem is called Linear Programming (LP). Sometimes the variables are only allowed to take integer values, then it is called Integer Linear Programming (ILP). The linear programming is widely used to solve problems in various fields, such as portfolio optimisation in investment, manufacturing and transportation, route and traffic planning. With the help of algorithms such as the simplex algorithm and interior point methods, LP problems can often be efficiently solved. The 10th chapter of (Press et al. 2007) explains in detail how these two methods work.

Compared to the linear optimisation, solving *non-linear optimisation* problems can still be very challenging, especially the non-convex and non-quadratic problems. If the both objective function and the constraints are continuous, then the Lagrangian optimisation, with or without Karush-Kuhn-Tucker (KKT) condition, can be used; otherwise we can only rely on heuristics, such as the simulated annealing methods we have just mentioned. The tools such as LINDO provide functionalities to solve various sorts of optimisation problems.

Finding a global solution that maximises or minimises the non-linear objective function is often time-consuming, even for only a small set of variables. Therefore,

global optimisation of a non-linear problem is normally only used when it is absolutely necessary. For example, if a system pressure test is modelled as an optimisation problem, given a small number of variants in the system, and a global extreme value has to find to test if the system is robust enough. Otherwise, a local maximum or minimum is normally used instead as an approximation. In most engineering applications, a local extreme value is good enough. Even though optimisation cannot promise a true extremism, and is easily affected by algorithm parameters and initial guess in iterative algorithms, as a trade-off, local optimisation is much faster and thus still widely used.

Summary

In this chapter we give an overview of the mathematical optimisation field. At first we introduce the categorisation of optimisation problem. It can be viewed in two dimensions: local or global, unconstrained or constrained. This chapter mainly focuses on the local unconstrained problem. Before this however, we introduce the techniques to find roots of function, since the basic idea of root finding is similar to that of optimisation: it can be solved with or without calculating derivatives. The local unconstrained problem is further explained in two parts: the univariate and multivariate optimisation. Of all the methods introduced here, the gradient descent is especially important, and we will see it again in the Regression and Neural Network chapters. Finally, we give a brief peek at the topic of global and constrained optimisation problems.

References

- Boyd, Stephen, and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge university press.
- Fletcher, Roger. 2013. *Practical Methods of Optimization*. John Wiley & Sons.
- Press, William H, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge university press.

Regression

Regression is an important topic in statistical modelling and machine learning. It's about modelling problems which include one or more variables (also called "features" or "predictors") and making predictions of another variable ("output variable") based on previous data of predictors. Regression analysis includes a wide range of models, from linear regression to isotonic regression, each with different theory background and application fields. Explaining all these models are beyond the scope of this book. In this chapter, we focus on several common forms of regressions, mainly linear regression and logistic regression. We introduce their basic ideas, how they are supported in Owl, and how to use them to solve real problems.

Part of the material used in this chapter is attributed to the Coursera course ML004: "Machine Learning" by Andrew Ng. This is a great introductory course if you are interested to learn machine learning.

Linear Regression

Linear regression models the relationship of the features and output variable with a linear model. It is the most widely used regression model in research and business and is the easiest to understand, so it makes an ideal starting point for us to build understanding of regression. Let's start with a simple problem where only one feature needs to be considered.

Problem: Where to locate a new McDonald's restaurant?

McDonald's is undoubtedly one of the most successful fast food chains in the world. By 2018, it has already opened more than 37,000 stores worldwide, and surely more is being built as you are reading this book. One question you might be interested in then is: where to locate a new McDonald's branch?

According to its website, a lot of factors are in play: area population, existing stores in the area, proximity to retail parks, shopping centres, etc. Now let's simplify this problem by asserting that the potential profit is only related to area population. Suppose you are the decision maker in McDonald's, and also have access to data of each branch store (profit, population around this branch), what would be your decision about where to locate your next branch? Linear regression would be a good friend when you are deciding.

Part of the data is listed in [tbl. 24](#). However, note that this data set (and most of the dataset used below) is not taken from real data source but taken from that of the ML004 course by Andrew Ng. So perhaps you will be disappointed if you are looking for real data from running McDonald's.



Figure 74: Visualise data for regression problem

Table 24: Sample of input data: single feature

Profit	Population
20.27	21.76
5.49	4.26
6.32	5.18
5.56	3.08
18.94	22.63
...	...

Visualising these data can give a clear view about the relationship between profit and population. We can use the code below to do that. It first extracts the two columns data from the data file, converts it to dense matrix, and then visualise the data using the scatter plot.

```

let extract_data csv_file =
  let data = Owl_io.read_csv ~sep:',' csv_file in
  let data = Array.map (fun x -> Array.map float_of_string x) data
  |> Mat.of_arrays in
  let x = Mat.get_slice [[];[1]] data in
  let y = Mat.get_slice [[];[0]] data in
  x, y

let plot_data x y =
  let h = Plot.create "regdata.png" in
  Plot.scatter ~h ~spec:[MarkerSize 6.] x y;
  Plot.set_xlabel h "population";
  Plot.set_ylabel h "profit";
  Plot.output h

```



Figure 75: Find possible regression line for given data

The visualisation result is shown in fig. 74. As can be expected, there is a clear trend that larger population and larger profit are co-related with each other. But precisely how?

Cost Function

Let's start with a linear model that assumes the relationship between these two variables be formalised as:

$$y = \theta_0 + \theta_1 x_1 + \epsilon, \quad (46)$$

where y denotes the profit we want to predict, and input variable x_1 is the population number in this example. Since modelling can hardly make a perfect match with the real data, we use ϵ to denote the error between our prediction and the data. Specifically, we represent the prediction part as $h(\theta_0, \theta_1)$:

$$h_{\theta_0, \theta_1}(x_1) = \theta_0 + \theta_1 x_1 \quad (47)$$

The θ_0 and θ_1 are the parameters of this model. Mathematically they decide a line on a plane. We can now choose randomly these parameters and see how the result works, and some of these guesses are just bad intuitively, as shown in fig. 75. Our target is to choose suitable parameters so that the line is *close* to data we observed.

How do we define the line being “close” to the observed data then? One frequently used method is to use the *ordinary least square* to minimise the sum of squared distances between the data and line. We have shown the “ x - y ” pairs in the data above, and we represent the total number of data pairs with n , and thus the i 'th pair of data can be represented with x_i and y_i . With these notations, we can represent a metric to represent the *closeness* as:

$$J_{\theta_0, \theta_1}(\mathbf{x}, \mathbf{y}) = \frac{1}{2n} \sum_{i=1}^n (h_{\theta_1, \theta_0}(x_i) - y_i)^2 \quad (48)$$

Here \mathbf{x} and \mathbf{y} are both vectors of length n . In regression, we call this function the *cost function*. It measures how close the models are to an ideal one, and our target is thus clear: find suitable θ parameters to minimise the cost function.

Why do we use least square in the cost function? Physically, the cost function J represents the average distance of each data point to the line. By “distance” we mean the Euclidean distance between a data point and the point on the line with the same x-axis. A reasonable solution can thus be achieved by minimising this average distance.

Solving Problem with Gradient Descent

To give a clearer view, we can visualise the cost function with a contour graph. According to eq. 48, the cost function j is implemented as below:

```
let j x_data y_data theta0 theta1 =
  let f x = x *. theta1 +. theta0 in
  Mat.(pow_scalar (map f x_data - y_data) 2. |> mean') *. 0.5
```

Here x_data and y_data are the two columns of data from tbl. 24. We can then visualise this cost function within a certain range using surface and contour graphs:

```
let plot_surface x_data y_data =
  let x, y = Mat.meshgrid (-20.) 10. (-20.) 10. 100 100 in
  let z = Mat.(map2 (j x_data y_data) x y) in
  let h = Plot.create ~m:1 ~n:2 "reg_cost.png" in
  Plot.subplot h 0 0;
  Plot.(mesh ~h ~spec:[ NoMagColor ] x y z);
  Plot.set_xlabel h "theta0";
  Plot.set_ylabel h "theta1";
  Plot.set_zlabel h "cost";
  Plot.subplot h 0 1;
  Plot.contour ~h x y z;
  Plot.set_xlabel h "theta0";
  Plot.set_ylabel h "theta1";
  Plot.output h
```

In fig. 76 we can see that cost function varies with parameters θ_0 and θ_1 with a bowl-like shape curve surface. The minimum point lies at somewhere at the bottom of the “valley”. It is thus natural to recall the gradient descent we have introduced in the previous chapter, and use it to find the minimal point in this bowl-shape surface.

Recall from previous chapter that gradient descent works by starting at one point on the surface, and move in the *direction* of steepest descent at some *step size*, then gradually approach to a local minimum, hopefully as fast as possible. Let’s use a fixed step size α , and the direction at certain point on the surface

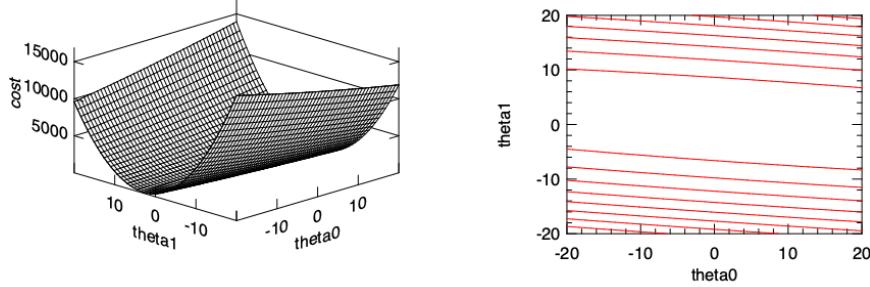


Figure 76: Visualise the cost function in linear regression problem

can be obtained by using partial derivative on the surface. Therefore, what we need to do is to apply this update process iteratively for both θ parameters:

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J_{\theta_0, \theta_1}(\mathbf{x}, \mathbf{y}), \quad (49)$$

where i is 1 or 2.

This process may seem terribly complex at first sight, but by solving the partial derivative we can calculate it as two parts:

$$\theta_0 \leftarrow \theta_0 - \frac{\alpha}{n} \sum_{i=1}^m (h_{\theta_0, \theta_1}(x_i) - y_i) x_i^{(0)}, \quad (50)$$

and

$$\theta_1 \leftarrow \theta_1 - \frac{\alpha}{n} \sum_{i=1}^m (h_{\theta_0, \theta_1}(x_i) - y_i) x_i^{(1)}. \quad (51)$$

Here the $x_i^{(0)}$ and $x_i^{(1)}$ are just different input features of the i -th row in data. Since currently we only focus on one feature in our problem, $x_i^{(0)} = 1$ and $x_i^{(1)} = x_i$. Following these equations, you can manually perform the gradient descent process until it converges.

```

let gradient_desc x y =
  let alpha = 0.01 in
  let theta0 = ref 10. in
  let theta1 = ref 10. in

  for i = 0 to 500 do
    let f x = x *. !theta1 +. !theta0 in
    theta0 := !theta0 -. Mat.(map f x - y |> mean') *. alpha;
    theta1 := !theta1 -. Mat.((map f x - y) * x |> mean') *. alpha
  done;
  theta0, theta1

```

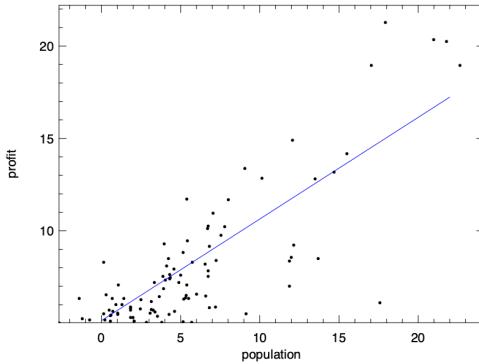


Figure 77: Validate regression result with original dataset

In the code above, we set the step size $\alpha = 0.01$, and start from a set of initial parameters: $\theta_0 = 10$ and $\theta_1 = 10$, and aim to improve them gradually. We then iteratively update the parameters using 500 iterations. Note that instead of manual summation in the equations, we use the vectorised operations with ndarray. By executing the code, we can get a pair of parameters: $\theta_0 = 5.14$ and $\theta_1 = 0.55$. To check if they indeed are suitable parameters, we can visualise them against the input data. The resulting figure fig. 77 shows a line that aligns with input data quite nicely.

Of course, there is no need to use to manually solve a linear regression problem in Owl. It has already provided high-level regression functions. For example, the `ols` function in the `Regression` module uses the ordinary least square method we have introduced to perform linear regression.

```
val ols : ?i:bool -> arr -> arr -> arr array
```

Here the Boolean parameter i denotes if the constant parameter θ_0 is used or not. By default it is set to `false`. We can use this function to directly solve the problem, and the resulting parameters are similar to what we have get manually:

```
# let theta = Regression.D.ols ~i:true x y

val theta : Owl_algodiff_primal_ops.D.arr array =
[|
  C0
  R0 0.588442
;
  C0
  R0 4.72381
|]
```

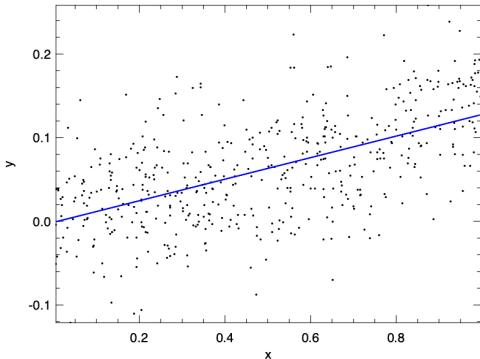


Figure 78: An example of using linear regression to fit data

The API is not limited to the regression module. The `linreg` function in the Linear Algebra module can also be used to perform the same task. The code snippet below first generates some random data, then using `linreg` function to perform a simple linear regression and plots the data as well as the regression line.

```
let generate_data () =
  let x = Mat.uniform 500 1 in
  let p = Mat.uniform 1 1 in
  let y = Mat.(x *@ p + gaussian ~sigma:0.05 500 1) in
  x, y

let t1_sol () =
  let x, y = generate_data () in
  let h = Plot.create "plot_00.png" in
  let a, b = Linalg.D.linreg x y in
  let y' = Mat.(x *$ b +$ a) in
  Plot.scatter ~h x y;
  Plot.plot ~h ~spec:[ RGB (0,255,0) ] x y';
  Plot.output h
```

Of course, since the process of finding suitable parameters can be performed using gradient descent methods, another approach to the regression problem is from the perspective of function optimisation instead of regression. We can use the gradient descent optimisation methods introduced in the Optimisation chapter, and apply them directly on the cost function eq. 48. As a matter of fact, the regression functions in Owl are mostly implemented using the `minimise_weight` function from the Optimisation module.

Multiple Regression

Back to our McDonald's problem. We have seen how a new store's profit can be related to the population of its surrounding, and we can even predict it given previous data. Now, remember that in the real world, population is not the only input features that affect the store's profit. Other factors such as existing stores in the area, proximity to retail parks, shopping centres, etc. also play a role. In that case, how can we extend our one-variable linear regression to the case of multiple variables?

The answer is very straight forward. We just use more parameters, so the model becomes:

$$h_{\theta_0, \theta_1, \theta_2, \theta_3, \dots}(x_1, x_2, x_3, \dots) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 \dots \quad (52)$$

However, to list all the parameters explicitly is not a good idea, especially when the question requires considering thousands or even more features. Therefore, we use the vectorised format in the model:

$$h_{\boldsymbol{\theta}}(X^{(i)}) = \boldsymbol{\theta} X^{(i)}, \quad (53)$$

where $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \theta_3, \dots]$, and $X^{(i)} = [1, x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, \dots]^T$ contains all the features from the i th row in data.

Accordingly, the cost function can be represented as:

$$J_{\boldsymbol{\theta}}(X, \mathbf{y}) = \frac{1}{2n} \sum_{i=1}^n (\boldsymbol{\theta} X^{(i)} - y^{(i)})^2, \quad (54)$$

where $y^{(i)}$ is the output variable value on the i th row of input data.

The derivative and manual gradient descent are left as exercise. Here we only show an example of using the regression function Owl has provided. Similar to the previous problem, we provide some data to this multiple variable problem. Part of the data is listed below:

Table 25: Sample of input data: multiple features

x_1	x_2	y
1888	2	255000
1604	3	242900
1962	4	259900
3890	3	573900
1100	3	249900
1458	3	464500
...

The problem has two different features. Similar to the single-variable regression problem in the previous section, by using the `ols` regression function in Owl, we can easily get the multi-variable linear model. The data loading method is exactly the same as before.

```
let multi_regression csv_file =
  let data = Owl_io.read_csv ~sep:',' csv_file in
  let data = Array.map (fun x -> Array.map float_of_string x) data
  |> Mat.of_arrays in
  let x = Mat.get_slice [[];[0; 1]] data in
  let y = Mat.get_slice [[];[2]] data in
  Regression.D.ols ~i:true x y
```

The resulting parameters are shown below:

```
val theta : Owl_algodiff_primal_ops.D.arr array =
  []
  C0
R0 57.342
R1 57.6741
;
  C0
R0 57.6766
[]
```

The result means that the linear model we get is about:

$$y = 57 + 57x_0 + 57x_1$$

Hmm, it might be right, but something about this model feels wrong. If you apply any line of data in `tbl_25` to this model, the prediction result deviates too much from the true `y` value. So what have gone wrong? To address this problem, we move on to an important issue: normalisation.

Feature Normalisation

Getting a result doesn't mean the end. Using the multi-variable regression problem as example, we would like to discuss an important issue about regression: *feature normalisation*. Let's look at the multi-variable data again. Apparently, the first feature is magnitudes larger than the second feature. That means the model and cost function are dominated by the first feature, and a minor change of this column will have a disproportionately large impact on the model. That's why the model we get in the previous section is wrong.

To overcome this problem, we hope to pre-process the data before the regression, and normalise every features within about $[-1, 1]$. This step is also called *feature scaling*. There are many ways to do this, and one of them is the *mean*

normalisation: for a column of features, calculate its mean, and divided by the difference between the largest value and smallest value, as shown in the code below:

```
let norm_ols data =
  let m = Arr.mean ~axis:0 data in
  let r = Arr.(sub (max ~axis:0 data) (min ~axis:0 data)) in
  let data' = Arr.((data - m) / r) in
  let x' = Mat.get_slice [[];[0; 1]] data' in
  let y' = Mat.get_slice [[];[2]] data' in
  let theta' = Regression.D.ols ~i:true x' y' in
  theta'
```

Here `data` is the matrix we get from loading the csv file from the previous section. This time we get a new set of parameters for the normalised data:

```
val theta' : Owl_algodiff_primal_ops.D.arr array =
  []
  C0
  R0 0.952411
  R1 -0.0659473
  ;
  C0
  R0 -1.93878E-17
  []
```

These parameters set the model as: $\bar{y} = 0.95\bar{x}_0 - 0.06\bar{x}_1$. This result can be cross-validated with the analytical solution shown in the next section. You can also manually check this result with the normalised data:

```
val data' : (float, Bigarray.float64_elt) Owl_dense_ndarray_generic.t =
  C0 C1 C2
  R0 0.68321 0.457447 0.678278
  R1 -0.202063 -0.0425532 -0.151911
  ...
```

Another benefit of performing data normalisation is to accelerate gradient descent. The illustration in fig. 79 shows the point. We have already seen that, in a “slim” slope, the Gradient Descent, which always trying to find the steepest downward path, may perform bad. Normalisation can reshape the slope to a more proper shape.

Normalisation is not only used in regression, but also may other data analysis and machine learning tasks. For example, in computer vision tasks, an image is represented as an ndarray with three dimensions. Each element represents a pixel in the image, with a value between 0 and 255. More often than not, this ndarray needs to be normalised in data pre-processed for the next step processing such as image classification.

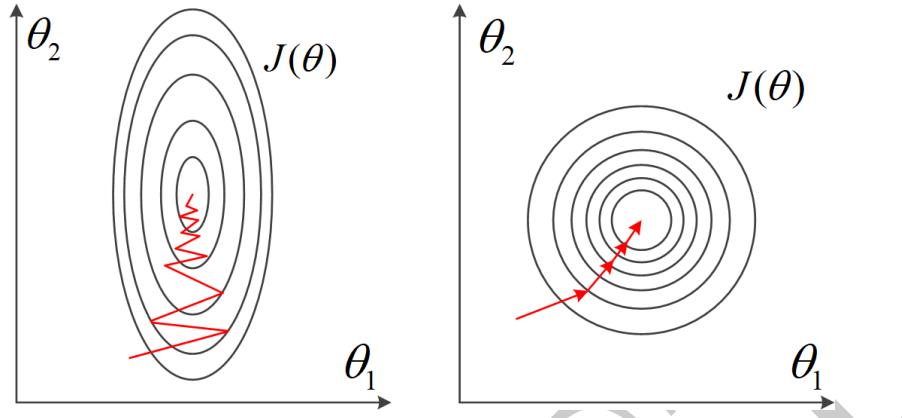


Figure 79: Compare gradient descent efficiency with and without data normalisation

Analytical Solution

Before taking a look at some other forms of regression, let's discuss solution to the linear regression besides gradient descent. It turns out that there is actually one close form solution to linear regression problems:

$$\theta = (X^T X)^{-1} X^T y \quad (55)$$

Chapter 3 of *The elements of statistical learning* (Friedman, Hastie, and Tibshirani 2001) covers how this solution is derived if you are interested. Suppose the linear model contains m features, and the input data contains n rows, then here X is a $n \times (m + 1)$ matrix representing the features data, and the output data y is a $n \times 1$ matrix. The reason there is $m + 1$ columns in X is that we need an extra constant feature for each data, and it equals to one for each data point.

With this method, we don't need to iterate the solutions again and again until converge. We can just compute the result with one pass with the given input data. This calculation can be efficiently performed in Owl using its Linear Algebra module. Let's use the dataset from multi-variable regression again and perform the computation.

```

let o = Arr.ones [| (Arr.shape x).(0); 1 |]
let z = Arr.concatenate ~axis:1 [|o; x'|];;

let solution = Mat.dot (Mat.dot
  (LinAlg.D.inv Mat.(dot (transpose z) z)) (Mat.transpose z)) y'

```

Here the x' , y' are the normalised data from the previous section. The result is close to what we have gotten using the gradient descent method:

```

val solution : Mat.mat =
  C0
  R0 -3.28614E-17
  R1 0.952411
  R2 -0.0659473

```

Compared to gradient descent, this method does not require multiple iterations, and you also don't need to worry about hyper-parameters settings such as the choice of learning rate. On the other hand, however, this approach has its own problems. When the size of X , or the input data, becomes very large, the computation of large linear algebra operations such as matrix multiplication and inversion could become really slow. Or even worse: your computer might not even have enough memory to perform the computation. Compare to it, gradient descent proves to work well even when the dataset is large.

Besides, there could be no solution at all using this method. That's when the $X^T X$ matrix is non-invertible, e.g. a singular matrix. That could be caused by multiple reasons. Perhaps some of the features are linear dependent, or that there are many redundant features. Then techniques such as choosing feature or regularisation are required. Most importantly, there is not always a close-form solution for you to use in other regression or machine learning problems. Gradient descent is a much more general solution.

Non-linear regressions

As powerful as it is, not all the regression problems can be solved with the linear model above. A lot of data can follow other patterns than a linear one. We can show this point with data from the Boston Housing Dataset. This dataset contains information collected by the U.S Census Service concerning housing in the area of Boston Mass. It contains 506 cases. Each case contains 14 properties, such as crime rate, nitric oxides concentration, average number of rooms per dwelling, etc. For this example, we observe the relationship between percentage of lower status of the population ("LSTAT") and the median value of owner-occupied homes in \$1000's ("MDEV").

```

let f ?(csv_file="boston.csv") () =
  let data = Owl_io.read_csv ~sep:' ' csv_file in
  let data = Array.map (fun x -> Array.map float_of_string x) data
    |> Mat.of_arrays in
  let lstat = Mat.get_slice [[];[12]] data in
  let medv = Mat.get_slice [[];[13]] data in
  lstat, medv

```

We can then visualise the data to see the trend clearly. As fig. 80 shows, the relationship basically follows a convex curve. You can try to fit a line into these data, but it's quite likely that the result would not be very fitting. And

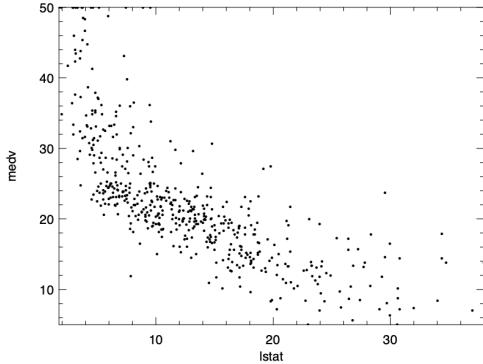


Figure 80: Visualise part of the boston housing dataset

that requires us to use non-linear models. In this section, we present a type of non-linear regression, the *polynomial regression*. We show how to use them with examples, without going into details of the math.

In polynomial regression, the relationship between the feature x and the output variable is modelled as an n -th degree polynomial in the feature x :

$$h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 \dots \quad (56)$$

Owl provides a function to perform this forms of regression:

```
| val poly : arr -> arr -> int -> arr
```

In the code, we use the `poly` function in the `Regression` module to get the model parameter. We limit the model to be 2nd order.

```
let poly lstat medv =
  let a = Regression.D.poly lstat medv 2 in
  let a0 = Mat.get a 0 0 in
  let a1 = Mat.get a 1 0 in
  let a2 = Mat.get a 2 0 in
  fun x -> a0 +. a1 *. x +. a2 *. x *. x
```

By executing the `poly` function, the parameters we can get are:

```
- : Owl_algodiff_primal_ops.D.arr =
C0
```

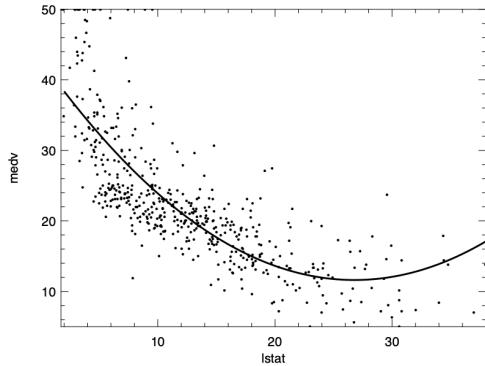


Figure 81: Polynomial regression based on Boston housing dataset

```
R0 42.862
R1 -2.33282
R2 0.0435469
```

That gives us the polynomial model:

$$f(x) = 42.8 - 2.3x + 0.04x^2 + \epsilon$$

We can visualise this model to see how well it fits the data:

Regularisation

Regularisation is an important issue in regression, and is widely used in various regression models. The motivation of using regularisation comes from the problem of *over-fitting* in regression. In statistics, over-fitting means a model is tuned too closely to a particular set of data and it may fail to predict future observations reliably.

Let's use the polynomial regression as an example. Instead of using an order of 2, now we use an order of 4. Note that we take only a subset of 50 of the full data set to better visualise the over-fitting problem:

```
let subdata, _ = Mat.draw_rows ~replacement:false data 50
```

we can get the new model:

$$f(x) = 63 - 10.4x + 0.9x^2 - 0.03x^3 + 0.0004x^4.$$

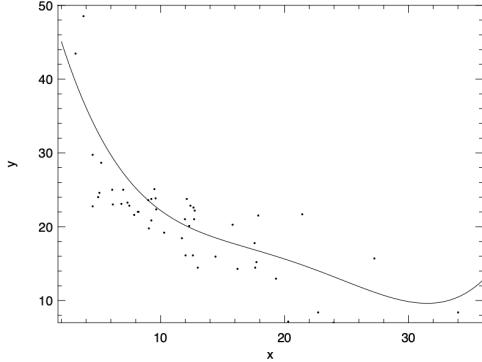


Figure 82: Polynomial regression with high order

This model could be visualised as in fig. 82. Apparently, this model fits too closely with the given data, even the outliers. Therefore, this model does not make a good prediction for future output values.

To reduce the effect of higher order parameters, we can penalise these parameters in the cost function. We design the cost function so that the large parameter values leads to higher cost, and therefore by minimising the cost function we keep the parameters relatively small. Actually we don't need to change the cost functions dramatically. All we need is to add some extra bit at the end. For example, we can add this:

$$J_{\theta}(x, y) = \frac{1}{2n} \left[\sum_{i=1}^n n(h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^m \theta_j^2 \right]. \quad (57)$$

Here the sum of squared parameter values is the penalty we add to the original cost function, and λ is a regularisation control parameter. That leads to a bit of change in the derivative of $J(\theta)$ in using gradient descent:

$$\theta_j \leftarrow \theta_j - \frac{\alpha}{n} \left[\sum_{i=1}^m (h_{\Theta}(x_i) - y_i) x_i^{(j)} - \lambda \theta_j \right]. \quad (58)$$

We can now apply the new update procedure in gradient descent code, with a polynomial model with 4th order.

Currently not all the regression methods in Owl support regularisation. But we can implement one easily. For the polynomial regression, we can implement it this way:



Figure 83: Revised polynomial model by applying regularisation in regression

```

open Optimise.D
open Optimise.D.Algodiff

let poly_ridge ~alpha x y n =
  let z =
    Array.init (n + 1) (fun i -> A.(pow_scalar x (float_of_int i |> float_to_elt)))
  in
  let x = A.concatenate ~axis:1 z in
  let params =
    Params.config
      ~batch:Batch.Full
      ~learning_rate:(Learning_Rate.Const 1.)
      ~gradient:Gradient.Newton
      ~loss:Loss.Quadratic
      ~regularisation:(Regularisation.L2norm alpha)
      ~verbosity:false
      ~stopping:(Stopping.Const 1e-16)
    100.
  in
  (Regression.D._linear_reg false params x y).(0)

```

The implementation is based on the optimisation module and the general low level _linear_reg function. It utilises the optimisation parameter module, which will be explained in detail in the Neural Network chapter. For now, just know that the key point is to use the L2-norm function as regularisation method. By using this regularised version of polynomial regression, we can have an updated model as shown in fig. 83.

Here we choose the alpha parameter to be 20. We can see that by using regularisation the model is less prone to the over-fitting problem, compared to fig. 82. Note that linear regression is used as an example in the equation, but regularisation is widely use in all kinds of regressions.

Ols, Ridge, Lasso, and Elastic_net

You might notice that Owl provides a series of functions other than `ols` in the regression module:

```
val ridge : ?i:bool -> ?alpha:float -> arr -> arr -> arr array
val lasso : ?i:bool -> ?alpha:float -> arr -> arr -> arr array
val elastic_net : ?i:bool -> ?alpha:float -> ?l1_ratio:float -> arr -> arr -> arr
array
```

What are these functions? The short answer is that: they are for regularisation in regression using different methods. The `ridge` cost function adds the L2 norm of θ as the penalty term: $\lambda \sum \theta^2$, which is what we have introduced. The `lasso` cost function is similar. It adds the *L1 norm*, or absolute value of the parameter as penalty: $\lambda \sum |\theta|$. This difference makes `lasso` allow for some coefficients to be zero, which is very useful for feature selection. The `elastic_net` is proposed by Hui Zou and Trevor Hastie of the Stanford University to combine the penalties of the previous two. What it adds is this:

$$\lambda \left(\frac{1-a}{2} \sum \theta^2 + a \sum |\theta| \right), \quad (59)$$

where a is a control parameter between `ridge` and `lasso`. The elastic net method aims to make the feature selection less dependent on input data. We can thus choose one of these functions to perform regression with regularisation on the dataset in the previous chapter.

Logistic Regression

So far we have been predicting a value for our problems, whether using linear, polynomial or exponential regression. What if we care about is not the value, but a classification? For example, we have some historical medical data, and want to decide if a tumour is cancer or not based on several features. These kind of variables are called *categorical variables*. They represent data that can be divided into groups. Such variables include “race”, “age group”, “religion”, etc. To predicting if a given data belongs to which group for a categorical variable, the previous regression methods do not apply. Instead, we need to use the *Logistic Regression*. Its point is to figure out certain “boundaries” among the data space so that different data can be divided into corresponding variable group. Let’s start with introducing how it works.

Sigmoid Function

As a naive solution, we can still try to continue using linear regression, and the model can be interpreted as the possibility of one of these results. But one problem is that, the prediction value could well be out of the bounds of [0,



Figure 84: The logistic function curve

[1]. Then maybe we need some way to normalise the result to this range? The solution is to use the sigmoid function (or logistic function): $\sigma(x) = \frac{1}{1+e^{-x}}$.

As shown in fig. 84, this function projects value within the range of [0, 1]. Applying this function on the returned value of a regression, we can get a model that returns value within [0, 1].

$$h_{\theta}(x) = \sigma(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}. \quad (60)$$

Now we can interpret this model easily. The function value can be seen as possibility. If it is larger than 0.5, then the classification result is 0, otherwise it returns 1. Remember that in logistic regression we only care about the classification. So for a 2-class classification, returning 0 and 1 is enough.

Cost Function

With the new model comes new cost function. Recalls that we can find suitable parameters by minimising the cost function with training data. The cost function of linear regression in eq. 48 indicates the sum of squared distances between the data and the model line. We can continue to use it here, with the new $h(x)$ function defined as sigmoid function. But the problem is that, in this case it will end up being a non-convex function, and the gradient descent can only give us one of many local minimums.

Therefore, in the logistic regression, we define its cost function as:

$$J_{\theta}(\mathbf{x}, \mathbf{y}) = \frac{1}{m} \sum_{i=1}^m g(h(x^{(i)}) - y^{(i)}), \quad (61)$$

where the function g is defined as:

$$g(h_{\theta}(x), y) = -\log(h_{\theta}(x)), \text{ if } y = 1, \quad (62)$$

or

$$g(h_{\theta}(x), y) = -\log(1 - h_{\theta}(x)), \text{ if } y = 0. \quad (63)$$

Both forms of function $g()$ capture the same idea. Since the h function is in the range $[0, 1]$, the range of g is $[0, \infty]$. When the value of $h(x)$ and y are close, then the item within the summation in eq. 61 $g(h(x)) - y$ will be close to 0; on the other hand, if the prediction result $h(x)$ and y are different, then $g(h(x)) - y$ will incur a large value to the cost function as penalty.

The previous three equations can be combined as one:

$$J_{\theta}(\mathbf{x}, \mathbf{y}) = \frac{1}{m} \sum_{i=1}^m (y^{(i)} \log(h(x^{(i)})) + (1 - y^{(i)}) \log(1 - h(x^{(i)}))) \quad (64)$$

The next step is to follow eq. 49 to find the partial derivative of this cost function and then iteratively minimise it to find suitable parameters Θ . It turns out that the partial derivative of this cost function is similar as that in linear regression:

$$\frac{\partial J_{\theta}(\mathbf{x}, \mathbf{y})}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m (\sigma_{\theta}(x^{(i)}) - y^{(i)})^2 x_j^{(i)} \quad (65)$$

This solution benefits from the fact that the sigmoid function has a simple derivative: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

With this derivative at hand, the rest is similar to what we have done with linear regression: follow eq. 49 and repeat this gradient descent step until it converges. Owl also provides a logistic function in the Regression module. In the next section, we will show an example of binary categorisation with logistic regression.

Example

To perform the logistic regression, let's first prepare some data. We can generate the data with the code below:

```
let generate_data () =
  let open Mat in
  let c = 500 in
  let x1 = (gaussian c 2 *$ 2.) in
  let a, b = float_of_int (Random.int 5), float_of_int (Random.int 5) in
  let x1 = map_at_col (fun x -> x +. a) x1 0 in
  let x1 = map_at_col (fun x -> x +. b) x1 1 in
```

```

let x2 = (gaussian c 2 *$ 1.) in
let a, b = float_of_int (Random.int 5), float_of_int (Random.int 5) in
let x2 = map_at_col (fun x -> x -. a) x2 0 in
let x2 = map_at_col (fun x -> x -. b) x2 1 in
let y1 = create c 1 ( 1.) in
let y2 = create c 1 ( 0.) in
let x = concat_vertical x1 x2 in
let y = concat_vertical y1 y2 in
x, y

let x, y = generate_data ()

```

Basically this code creates two groups of random data with `gaussian` function. Data `x` is of shape `[|1000; 2|]`, and is equally divided into two groups. The first group is at a higher position, and the corresponding `y` label is positive. The nodes in lower group are labelled as negative. Our task is to try to divide a given data point into one of these two categories.

With the `logistic` function, we train a model:

$$h_{\theta}(x_0, x_1) = \sigma(\theta_0 x_0 + \theta_1 x_1 + \theta_2).$$

In the linear model within the sigmoid function, we have two parameters θ_0 and θ_1 for the two variables that represent the two coordinates of a data point. The `logistic` functions takes an `i` argument. If `i` is set to `true`, the linear model contains an extra parameter θ_2 . Based on the data, we can get the parameters by simply executing:

```

# let theta =
  Owl.Regression.D.logistic ~i:true x y
val theta : Owl_algodiff_primal_ops.D.arr array =
[|
  C0
  R0 16.4331
  R1 12.4031
;
  C0
  R0 20.7909
|]

```

Therefore, the model we get is:

$$h(x_0, x_1) = \sigma(16x_0 + 12x_1 + 20). \quad (66)$$

We can validate this model by comparing the inference result with the true label `y`. Here any prediction value larger than 0.5 produced by the model is deemed as positive, otherwise it's negative.

```

let test_log x y =
  let p' = Owl.Regression.D.logistic ~i:true x y in
  let p = Mat.(p'.(0) @= p'.(1)) in
  let x = Mat.(concat_horizontal x (ones (row_num x) 1)) in
  let y' = Mat.(sigmoid (x *@ p)) in
  let y' = Mat.map (fun x -> if x > 0.5 then 1. else 0.) y' in
  let e = Mat.((mean' (abs (y - y')))) in
  Printf.printf "accuracy: %.4f\n" (1. -. e)

# test_log x y
accuracy: 0.9910
- : unit = ()

```

The result shows that, the trained model has more than 99% prediction accuracy when applied on the original dataset. Of course, a more suitable approach would be to use a new set of test data set, but you can get a basic idea about how well it works.

Decision Boundary

As we have said, the physical meaning of classification is to draw a *decision boundary* in a hyperplane to divide different groups of data points. For example, if we are using a linear model h within the sigmoid function, the linear model itself divides the points into two halves in the plane. Use eq. 66 as an example, any x_0, x_1 that makes the $h(x_0, x_1) > 0$ is taken as positive, otherwise it's negative. Therefore, the boundary line we need to draw is: $16x_0 + 12x_1 + 20 = 0$, or $x_1 = -(4x_0 + 5)/3$. We can visualise this decision boundary on a 2D-plane and how it divides the two groups of data.

```

open Owl
let data = Mat.concat_horizontal x y

let f x = -(4. *. x +. 5.) /. 3.

let plot_logistic data =
  let neg_idx = Mat.filter_rows (fun m -> Mat.get m 0 2 = 0.) data in
  let neg_data = Mat.get_fancy [ L (Array.to_list neg_idx); R [] ] data in
  let pos_idx = Mat.filter_rows (fun m -> Mat.get m 0 2 = 1.) data in
  let pos_data = Mat.get_fancy [ L (Array.to_list pos_idx); R [] ] data in
  (* plot dataset *)
  let h = Plot.create "reg_logistic.png" in
  Plot.(scatter ~h ~spec:[ Marker "#[0x2217]"; MarkerSize 5. ]
    (Mat.get_slice [[],[]] neg_data)
    (Mat.get_slice [[],[]] neg_data));
  Plot.(scatter ~h ~spec:[ Marker "#[0x2295]"; MarkerSize 5. ]
    (Mat.get_slice [[],[]] pos_data)
    (Mat.get_slice [[],[]] pos_data));
  (* plot line *)
  Plot.plot_fun ~h ~spec:[ RGB (0,0,255); LineWidth 2. ] f (-5.) 5. ;
  Plot.output h

```



Figure 85: Visualise the logistic regression dataset

The code above visualises the data, two types of points showing the negative and positive data, and the line shows the decision boundary we get from the logistic model. The result is shown in fig. 85. There are some wrong categorisations, but you can see that this model works well for most the data points.

Of course, we can use more than linear model within the sigmoid function. for example, we can use to set the model as $h_{\theta}(x) = \sigma(\theta_0 + \theta_1 x + \theta_2 x^2)$. If we use a non-linear polynomial model, the plane is divided by curve lines.

Logistic regression uses the linear model. If you believe your data won't be linearly separable, or you need to be more robust to outliers, you should look at SVM (see sections below) and look at one of the non-linear models.

Multi-class classification

We have seen how to classify objects into one of two classes using logistic regression. What if our target consists of multiple classes? For this problem, one approach is called *one-vs-all*. Its basic idea is simple. Suppose we need to classify one object into three different classes; we can still use the logistic regression, but instead of one classifier, we train three binary classifiers, each one is about one class against the other two. In the prediction phase, using an object as input, each classifier yields a probability as output. We then choose the largest probability as the classification result.

The multi-class classification problem is prevalent in the image recognition tasks, which often includes classifying one object in the image into of ten, hundred, or more different classes. For example, one popular classification problem is the hand-written recognition task based on the MNIST dataset. It requires the model to recognise a 28x28 grey scale image, representing a hand-written number, to be one of ten numbers, from 0 to 9. It is a widely used ABC task for Neural Networks. We will discuss this example in detail later in the beginning

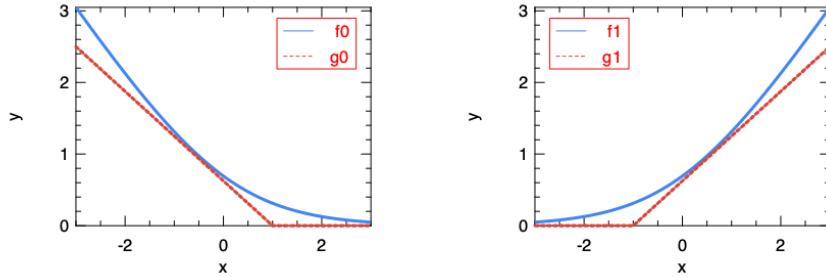


Figure 86: Simplifying the cost function of logistic regression

of the Neural Network chapter. You will see how this logistic regression line of thought is extended, and how the real-world multi-class classification problems are solved using neural networks.

Support Vector Machine

The *Support Vector Machines* (SVMs) are a group of supervised learning algorithms that can be used for classification and regression tasks. The SVM is one of the most powerful and widely used Machine Learning methods in both academia and industry. It is used in solve real-world problems in various fields, including the classification of text, image, satellite data, protein data, etc.

Similar to logistic regression, the SVM creates a hyperplane (decision boundary) that separates data into classes. However, it tries to maximise the “margin” between the hyperplane and the boundary points of both classes in the data. By use the “kernels”, the SVM can also fit non-linear boundaries. Besides, the SVM works well with unstructured and semi-structured data such as text and images, with less risk of over-fitting. In this section, we will introduce the basic idea behind SVM and how it works in example.

Again, let’s start with the objective cost function. It turns out that the cost function of the SVM is very similar to that of logistic regression in eq. 64, with some modifications:

$$J_{\theta}(x, y) = \frac{1}{m} \sum_{i=1}^m (y^{(i)} g_0(\boldsymbol{\theta}^T x^{(i)}) + (1 - y^{(i)}) g_1(\boldsymbol{\theta}^T x^{(i)})) + \frac{1}{2} \sum_{j=1}^m \theta_j^2 \quad (67)$$

Function $g_0()$ and $g_1()$ are simplification of the logarithm function:

Here $f_0(x) = -\log(\sigma(x))$ is what is used in the cost function of the logistic regression. This computation-heavy logarithm is replaced with $g_0(x)$, a simple segmented function. Similarly, $f_1(x) = -\log(1 - \sigma(x))$ is replaced by $g_1(x)$.



Figure 87: Margins in the Supported Vector Machines

Also, another difference is that a regularisation item is added to the cost function in eq. 67. Therefore, considering the properties of $g_0(x)$ and $g_1(x)$, by minimising this function, we are actually seeking parameters set θ to minimise $\sum_{j=1}^m \theta_j^2$, with the limitation that $\theta^T x > 1$ when $y = 1$, or $\theta^T x < -1$ when $y = 0$.

It turns out that, by solving this optimisation problem, SVM tends to get a *large margin* between different categories of data points. One example is shown in fig. 87. It shows two possible decision boundaries, both can effectively divide the two groups of training data. But the blue boundary has a larger distance towards the positive and negative training samples, denoted with dotted lines. These dotted lines indicate the *margin* of the SVM. As to the inference phase, any data x that makes $\theta^T x > 0$ is deemed positive, i.e. $y = 1$, or negative if $\theta^T x < 0$. It is intuitive to see that a model with larger margin tends to predict the test data better.

Kernel and Non-linear Boundary

So far we have talked about the linear boundary, but that's surely not the limit of SVM. In fact, it is normally the case that we use SVM to train a non-linear boundary in categorising different groups of points in the space. To do that, we can simply update the linear part $\theta^T x$ in the cost function to make it a non-linear function, e.g.:

$$f_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2 + \dots \quad (68)$$

This function and the linear function $\theta^T x$ are both examples of a *Kernel Function*, which are to be used within $g()$ in the cost function. With the trained parameters θ , if eq. 68 is larger than zero, then the inference result is positive, otherwise it's negative. However, this model is apparently not scalable with regard to the number of features of the input.

Currently, one common way to model this function is to choose k reference points

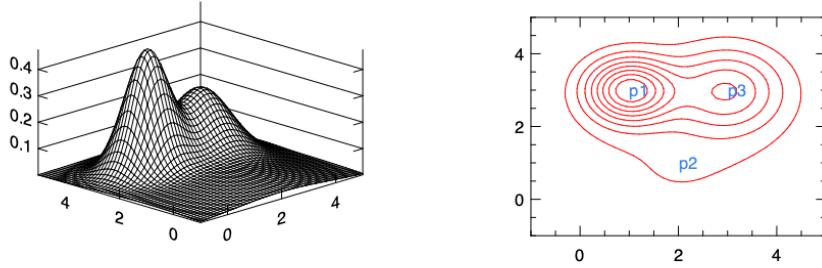


Figure 88: Using the gaussian kernel to locate non-linear boundary in categorisation

in the space, and use the *distances* to these points as feature. In other words, for data \mathbf{x} that contains any number of features, the objective function is reduced to use a fixed k features:

$$\theta_0 + \sum_{i=1}^k \theta_i d_i, \quad (69)$$

where d_i is the “distance” of the current point to the reference point p_k . In inference phase, if this function is larger than zero, then the point is predicted to be positive, otherwise it is negative.

So how exactly is this “distance” calculated? There are many ways to do that, and one of the most used is the gaussian distance, as shown in eq. 70. Here the total number of feature is n . x_j is the j -th feature of x and $p_k^{(j)}$ is the j -th feature of reference point p_k .

$$d_i = \exp\left(-\frac{\sum_{j=1}^n (x_j - p_k^{(j)})^2}{2\sigma^2}\right). \quad (70)$$

The intuition is that, with suitable trained parameters θ , this approach can represent different regions in the classification. For example, in fig. 88 we choose only three reference points: p_0 , p_1 , and p_2 . In this example, we set the parameters for p_1 and p_3 to be larger than that of p_2 . The figure left shows the summation of distances of a point to all three of them. The contour graph on the right then shows clearly how this model lead to a prediction that’s obviously large around a region that’s close to the point p_1 and p_3 . That’s the boundary we use to decide if a point is positive (close to p_1 or p_3) or negative. You can imagine how this non-linear boundary can be changed with new parameters.

Only three reference points is normally not enough to support a complex non-linear boundary. In fact, one common practice is to use all the training data, each as one point, as the reference points.

In this example, the distance function d_i is called the *Kernel Function*. The Gaussian kernel we have just used is a commonly used one, but other kernel functions can also be used in eq. 69, such as the polynomial kernel, the Laplacian kernel, etc. The previous linear model $\theta^T x$ is called the linear kernel, or “no kernel” as is sometimes called.

Example

The SVM is a very important and widely used machine learning method, and that is accompanied by highly efficient implementation in libraries. For example, the Libsvm is an open source library that devotes solely to SVMs. It is widely interfaced to many languages such as Matlab and Python.

Using its optimisation engine, Owl provides initial support for SVM using linear kernel. Let's look at an example. Here we apply SVM to another randomly generated dataset in the similar way. The only difference is that previous we have label 0 and 1, but now we have label 1 and -1. After applying `Regression.D.svm ~i:true x y`, for a certain data we get, the result we get is:

```
val theta : Owl_algodiff_primal_ops.D.arr array =
[|
  C0
  R0 -0.438535
  R1 -0.922763
;
  C0
  R0 5.7011
|]
```

That means the hypothesis function $\theta^T x$ we have is: $f(x) = 5.7 - 0.43x_1 - 0.92x_2$. If $f(x) > 0$, the categorisation result is positive, otherwise it's negative. We can visualise this boundary line by setting $f(x) = 0$, as shown in fig. 89. Here the y axis is x_2 , and x axis is x_1 .

Model error and selection

Error Metrics

We have introduced using the least square as a target in minimising the distance between model and data, but it is by no means the only way to assess how good a model is. In this section, we discuss several error metrics for assessing the quality of a model and comparing different models. In testing a model, for each data point, its real value y and predicted value y' . The difference between these two is called *residual*. In this section, when we say error, we actually mean residual,

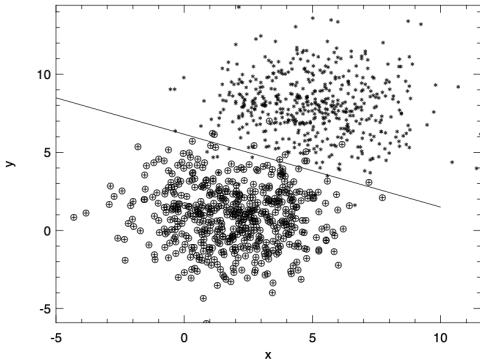


Figure 89: Visualise the SVM dataset

and do not confuse it with the ϵ item in the linear model. The latter is the deviation of the observed value from the unobservable true value, and residual means the difference between the observed value and the predicted value.

First, let's look at two most commonly used metrics:

- **Mean absolute error (MAE)**: average absolute value of residuals, represented by: $MAE = \frac{1}{n} \sum |y - y'|$.
- **Mean square error (MSE)**: average squared residuals, represented as: $MSE = \frac{1}{n} \sum (y - y')^2$. This is the method we have previously used in linear regression in this chapter. The part before applying average is called **Residual Sum of Squares (RSS)**: $RSS = \sum (y - y')^2$.

The difference between using absolute value and squared value means different sensitivity to outliers. Using the squared residual value, MSE grows quadratically with error. As a result, the outliers are taken into consideration in the regression so as to minimise MSE. On the other hand, by using the absolute error, in MAE each residual contribute proportionally to the metric, and thus the outliers do not have especially large impact on the model fitting. How to choose one of these metrics depends on how you want to treat the outliers in data.

Based on these two basic metrics, we can derive the definition of other metrics:

- **Root mean squared error (RMSE)**: it is just the square root of MSE. By applying square root, the unit of error is back to normal and thus easier to interpret. Besides, this metric is similar to the standard deviation and denotes how wide the residuals spread out.
- **Mean absolute percentage error (MAPE)**: based on MAE, MAPE changes it into percentage representation: $MAPE = \frac{1}{n} \sum \left| \frac{y - y'}{y} \right|$. It denotes the average distance between a model's predictions and their corresponding outputs in percentage format, for easier interpretation.

- **Mean percentage error** (MPE): similar to MAPE, but does not use the absolute value: $MPE = \frac{1}{n} \sum \left(\frac{|y - y'|}{y} \right)$. Without the absolute value, the metric can represent if the predicted value is larger or smaller than the observed value in data. So unlike MAE and MSE, it's a relative measurement of error.

Model Selection

We have already mentioned the issue of feature selection in previous sections. It is common to see that in a multiple regression model, many variables are used in the data and modelling, but only a part of them are actually useful. For example, we can consider the weather factor, such as precipitation quantity, in choosing the location of McDonald's store, but I suspect its contribution would be marginal at best. By removing these redundant features, we can make a model clearer and increase its interpretability. Regularisation is one way to downplay these features, and in this section we briefly introduce another commonly used technique: *feature selection*.

The basic idea of feature selection is simple: choose features from all the possible combinations, test the performance of each model using metric such as RSS. Then choose the best one from them. To put it into detail, suppose we have n features in a multi-variable regression, then for each $i = 1, 2, \dots, n$, test all the $\binom{n}{i}$ possible models with i variable(s), choose a best one according to its RSS, and we call this model M_i . Once this step is done, we can select the best one from the n models: M_1, M_2, \dots, M_n using *certain methods*.

You might have already spotted one big problem in this approach: computation complexity. To test all 2^n possibilities is a terribly large cost for even medium number of features. Therefore, some computationally efficient approaches are proposed. One of them is the *stepwise selection*.

The idea of stepwise selection is to build models based on existing best models. We start with one model with zero parameters (always predict the same value regardless of input data) and assume it is the best model. Based on this one, we increase the number of features to one, choose among all the n possible models according to their RSS, name the best one M_1 . And based on M_1 , we consider adding another feature. Choose among all the $n - 1$ possible models according to their RSS, name the best one M_2 . So on and so forth. Once we have all the models $M_i, i = 1, 2, \dots, n$, we can select the best one from them using suitable methods. This process is called "Forward stepwise selection", and similarly there is also a "Backward stepwise selection", where you build the model sequence from full features selection M_n down to M_1 .

You might notice that we mention using "certain methods" in selecting the best one from these n models. What are these methods? An obvious answer is to continue to use RSS etc. as the metric, but the problem is that the model with full features always has the smallest error and then gets selected every time.

Instead, we need to estimate the test error. We can directly do that using a validation dataset. Otherwise we can make adjustment to the training error such as RSS to include the bias caused by over-fitting. Such methods include: C_p , Akaike information criterion (AIC), Bayesian information criterion (BIC), adjusted R^2 , etc. To further dig into these statistical methods is beyond the scope of this book. We recommend specific textbooks such as (James et al. 2013).

Summary

This chapter introduces different techniques of regression. It starts with the most basic linear regression with one variant, and then extends it multiple variants. We introduced the basic idea of linear regression, its theoretical support, and shows how it is supported in Owl with examples. Furthermore, we extend it to more complex regressions with non-linear models, such as the polynomial regression. Unlike linear regression, the logistic regression model categorises data into different groups by finding out a decision boundary. With a bit of change in its cost function, we venture to introduce a type of advanced machine learning techniques, the Support Vector Machines. They can be used for both linear and non-linear decision boundary by using different kernel functions. We have also talked about related issues, such as the regularisation, model error and selection.

References

- Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2001. *The Elements of Statistical Learning*. Vol. 1. 10. Springer series in statistics New York.
- James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning*. Vol. 112. Springer.

Deep Neural Networks

The Neural Network has been a hot research topic and widely used in engineering and social life. The name “neural network” and its original idea comes from modelling how (the computer scientists think) the biological neural systems work. The signal processing in neurons are modelled as computation, and the complex transmission and triggering of impulses are simplified as activations, etc.

Since the inception of this idea in about 1940’s, the neural network has been revised and achieved astounding result. In this chapter, we will first explain that, as complex as it seems, the neural network is nothing more than a step forward based on the regression we have introduced. Then we will present the neural network module, including how to use it and how this module is designed and built based on existing mechanisms such as Algorithmic Differentiation and Optimisation. After the basic feedforward neural network and the Owl module, we then proceed to introduce some more advanced type of neural networks, including the Convolutional Neural Network, the Recurrent Neural Network, and Generative Adversarial Network. Let’s begin.

Perceptron

Before diving into the complex neural network structures, let’s briefly recount where everything begins: the *perceptron*. The definition is actually very similar to that of logistic regression. Look at fig. 90(a) and remember that logistic regression can be expressed as:

$$h_{\theta}(x) = g(x^T \theta),$$

where the g function is a sigmoid function: $g(x) = \frac{1}{1+e^{-x}}$. This function projects a number in $[-\infty, \infty]$ to $[0, 1]$. To get a perceptron, all we need to do is to change the function to:

$$g(x) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + \mathbf{b} > 0 \\ 0 & \text{otherwise} \end{cases}$$

This function is called a *Unit Step Function*, or heaviside/binary step function. Instead of a range $[0, 1]$, the result can only be either 0 or 1. It is thus suitable for binary classification. In the perceptron learning algorithm, we can still follow the previous parameter update method:

$$\theta_i = \theta_i - \lambda (y - h_{\theta}(\mathbf{x}))x_i$$

for each pair of training data \mathbf{x} and y .

The perceptron was first proposed in 1950’s to perform binary image classification. Back then it was thought to model how individual neuron in the brain works.



Figure 90: Extend logistic regression to neural network with one hidden layer

Though initially deemed promising, people quickly realise that perceptrons could not be trained to recognise many classes of patterns, which is almost the case in image recognition. To fix this problem requires introducing more layers of interconnected perceptrons. That is called *feedforward neural network*, and we will talk about it in detail in the two sections below.

Yet Another Regression

To some extent, a deep neural network is nothing but a regression problem in a very high-dimensional space. We need to minimise its cost function by utilising higher-order derivatives. Before looking into the actual Neural module, let's build a small neural network from scratch.

Following the previous logistic regression, in this section we build a simple neural network with a hidden layer, and train its parameters. The task is hand-written recognition. The starting example is also inspired by the Machine Learning course by Andrew Ng.

Model Representation

In logistic regression we have multiple parameters as one layer to decide if the input data belongs to one type or the other, as shown in fig. 90(a). Now we need to extend it towards multiple classes, with a new hidden layer.

The data we will use is from MNIST dataset. You can use `Owl.Dataset.download_all()` to download the dataset.

```
| let x, _, y = Dataset.load_mnist_train_data_arr ()
```



Figure 91: Visualise part of MNIST dataset

```
# let x_shape, y_shape =
  Dense.Ndarray.S.shape x, Dense.Ndarray.S.shape y

val x_shape : int array = [|60000; 28; 28; 1|]
val y_shape : int array = [|60000; 10|]
```

The label is in the one-hot format:

```
val y : Owl_dense_matrix.S.mat =
  C0 C1 C2 C3 C4 C5 C6 C7 C8 C9
  R0 0 0 0 0 0 1 0 0 0 0
  R1 1 0 0 0 0 0 0 0 0 0
  R2 0 0 0 0 1 0 0 0 0 0
  ....
```

It shows the first three labels are 5, 0, and 4.

Forward Propagation

Specifically we use a hidden layer of size 25, and the output class is 10. Since we will use derivatives in training parameters, we construct all the computation using the Algorithmic Differentiation module. The computation is repeated logistic regression:

$$h_{\Theta}(x) = f(f(x^T \theta_0) \theta_1).$$

Here Θ denotes the collection of parameters θ_0 and θ_1 . It can be implemented as:

```
open Algodiff.D
module N = Dense.Ndarray.D

let input_size = 28 * 28
let hidden_size = 25
let classes = 10

let theta0 = Arr (N.uniform [|input_size; hidden_size|])
let theta1 = Arr (N.uniform [|hidden_size; classes|])

let h theta0 theta1 x =
```

```
let t = Arr.dot x theta0 |> Maths.sigmoid in
Arr.dot t theta1 |> Maths.sigmoid
```

That's it. We can now classify an input 28x28 array into one of the ten classes... except that we can't. Currently we only use random content as the parameters. We need to train the model and find suitable θ_0 and θ_1 parameters.

Back propagation

Training a network is essentially a process of minimising the cost function by adjusting the weight of each layer. The core of training is the backpropagation algorithm. As its name suggests, backpropagation algorithm propagates the error from the end of a network back to the input layer, in the reverse direction of evaluating the network. Backpropagation algorithm is especially useful for those functions whose input parameters are far larger than output parameters.

Backpropagation is the core of all neural networks; actually it is just a special case of reverse mode AD. Therefore, we can write up the backpropagation algorithm from scratch easily with the help of Algodiff module.

Recall in the Regression chapter, training parameters is the process is to find the parameters that minimise the cost function of iteratively. In the case of this neural network, its cost function J is similar to that of logistic regression. Suppose we have m training data pairs, then it can be expressed as:

$$J_{\Theta}(\mathbf{x}, \mathbf{y}) = \frac{1}{m} \sum_{i=1}^m (-y^{(i)} \log(h_{\Theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\Theta}(x^{(i)}))). \quad (71)$$

It can be translated to Owl code as:

```
let j t0 t1 x y =
  let z = h t0 t1 x in
  Maths.add
    (Maths.cross_entropy y z)
    (Maths.cross_entropy Arr.(sub (ones (shape y)) y)
      Arr.(sub (ones (shape z)) z))
```

Here the “cross_entropy y x ” means $-y \log(x)$.

In the regression chapter, to find the suitable parameters that minimise J , we iteratively apply:

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial J}{\partial \theta_j}$$

until it converges. The same also applies here. But the partial derivative is not intuitive to give an analytical solution. But actually we don't have to now that

we are using the AD module. The partial derivatives of both parameters can be correctly calculated. We have shown in the Algorithmic Differentiation chapter how it can be done in Owl:

```
let x', y' = Dataset.draw_samples x y 1
let cost = j t0 t1 (Arr x') (Arr y')
let _ = reverse_prop (F 1.) cost
let theta0' = adjval t0 |> unpack_arr
let theta1' = adjval t1 |> unpack_arr
```

That's it for one iteration. We get $\frac{\partial J}{\partial \theta_j}$, and then can iteratively update the θ_0 and θ_1 parameters.

Feed Forward Network

This example works well, but nevertheless has several problems. In the next step, we revise it to add more details.

Layers

First, the previous example mixes all the computation together. We need to add the abstraction of *layer* as the building block of neural network instead of numerous basic computations. The following code defines the layer and network type, both are OCaml record types.

Also note that for each layer, besides the matrix multiplication, we also added an extra *bias* parameter. The bias vector influences the output without actually interacting with the data. Each linear layer performs the following calculation where *a* is a non-linear activation function.

$$y = a(x \times w + b)$$

Each layer consists of three components: weight *w*, bias *b*, and activation function *a*. A network is just a collection of layers.

```
open Algodiff.S

type layer = {
  mutable w : t;
  mutable b : t;
  mutable a : t -> t;
}

type network = { layers : layer array }

```
Despite of the complicated internal structure, we can treat a neural network as a function, which takes input data and generates predictions. The question is how to evaluate a network. Evaluating a network can be decomposed as a sequence of evaluation of layers.

```

The output `of` one layer will be given `to` the next layer `as` its input, moving forward until it reaches the `end`. The following two lines show how `to` evaluate a neural network `in` the `*forward mode*`.

```
```ocaml
let run_layer x l = Maths.((x *@ l.w) + l.b) |> l.a

let run_network x nn = Array.fold_left run_layer x nn.layers
```

The `run_network` can generate what equals to the $h_{\Theta}(x)$ function in the previous section.

Activation Functions

Let's step back and check again the similarity between neural network and what we think biological neuron works. So far we have seen how to connect every node (or "neuron") to every node between two layers. That's hardly how the real neuron works. Instead of activating all the connecting neurons during information transformation, different neurons are electrically activated differently in a kind of irregular fashion.

Biology aside, think about it in the mathematical way: if we keep fully connect multiple layers, that's in essence multiple matrix multiplication, and that would equals to only one single matrix multiplication.

Therefore, to make adding more layers mean something, we frequently use the *activation* functions to simulate how neuron works in biology and introduce *non-linearity* into the network. Non-linearity is a property we are looking for in neural network, since most real world data demonstrate non-linear features. If it is linear, then we human can often simply observe it and use something like linear regression to find the solution.

Actually we have already seen two types of activation function so far. The first is the Unit Step Function. It works like a simple on/off digital gate that allows part of neurons to be activated. Then there is the familiar `sigmoid` function. It limits the value to be within 0 and 1, and therefore we can think of it as a kind of probability of being activated.

Besides these two, there are many other types of non-linear activation functions, as shows in fig. 92. The $\tanh(x)$ function computes $\frac{e^x - e^{-x}}{e^x + e^{-x}}$. Softsign computes $\frac{x}{1+|x|}$. The $\text{relu}(x)$ computes:

$$\text{relu}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

And there is the `softmax` function. It takes a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers:



Figure 92: Different activation functions in neural network

$$f_i(x) = \frac{e^{x_i}}{\sum_{k=1}^K e^{x_k}} \text{ for } i = 1, 2, \dots, K.$$

We will keep using these activation functions in later network structures.

Initialisation

In this small example, we will only use two layers, 10 and 11. 10 uses a 784×40 matrix as weight, and `tanh` as activation function. 11 is the output layer and `softmax` is the cost function.

```
let 10 = {
    w = Maths.(Mat.uniform 784 40 * F 0.15 - F 0.075);
    b = Mat.zeros 1 40;
    a = Maths.tanh;
}

let 11 = {
    w = Maths.(Mat.uniform 40 10 * F 0.15 - F 0.075);
    b = Mat.zeros 1 10;
    a = Maths.softmax ~axis:1;
}

let nn = {layers = [|10; 11|]}
```

This definition is plain to see, but there is still one thing to say: the *initialisation* of parameters. From the regression chapter we have seen that how finding a good initial starting point can be beneficial to the performance of gradient descent. You might be thinking that uniformly generated parameters should work fine, but that's not the case.

Now we know that the essence of a layer is basically a matrix multiplication. If we use randomly initialise the parameter using uniform or normal distribution, you will find out that the results will soon explode after several layers. Even if by using `sigmoid` activation function we can control the number within $[0, 1]$, that still means the results are very near to 1, and the gradient will be extremely small. On the other hand, if we choose initial parameters that are close to 0,

then the output result from the network itself would be close zero. It is call the “vanishing gradient” problem, and in both cases, the network cannot learn well.

There are many works that aim to solve this problem. One common solution is to use ReLU as activation functions since it is more robust to this issue. As to initialisation itself, there are multiple heuristics that can be used.

For example, the commonly used Xavier initialization approach proposes to scale the uniformly generated parameters with: $\sqrt{\frac{1}{n}}$. This parameter is shared by two layers, and n is the size the first layer. This approach is especially suitable to use with `tanh` activation function. It is provided by the `Init.Standard` method in the initialisation module. The `Init.LecunNormal` is similar, but it uses $\sqrt{\frac{1}{n}}$ as the standard deviation of the Gaussian random generator.

In (Glorot and Bengio 2010) the authors propose to use $\sqrt{\frac{2}{n_0+n_1}}$ as the standard deviation in gaussian random generation. Here n_0 and n_1 is the input and output size of the current layer, or the length of two edges of the parameter matrix. It can be used with `Init.GlorotNormal`. If we want to use the uniformly generation approach, then the parameters should be scaled by $\sqrt{\frac{6}{n_0+n_1}}$. For this method we use `Init.GlorotUniform` or `Init.Tanh`.

Of course, besides these methods, we still provide the mechanism to use the vanilla uniform (`Init.Uniform`) or gaussian (`Init.Gaussian`) randomisation, or a custom method (`Init.Custom`).

Training

The loss function is constructed in the same way.

```
let loss_fun nn x y =
  let t = tag () in
  Array.iter (fun l ->
    l.w <- make_reverse l.w t;
    l.b <- make_reverse l.b t;
  ) nn.layers;
  Maths.(cross_entropy y (run_network x nn) / (F (Mat.row_num y |> float_of_int)))
```

The backprop also uses the same procedure as the previous example. The partial derivative is gotten using `adjval`, and the parameter `w` and `b` of each layer are updated accordingly. It then uses the gradient descent method, and the learning rate `eta` is fixed.

```
let backprop nn eta x y =
  let loss = loss_fun nn x y in
  reverse_prop (F 1.) loss;
  Array.iter (fun l ->
    l.w <- Maths.((primal l.w) - (eta * (adjval l.w))) |> primal;
```

```

l.b <- Maths.((primal l.b) - (eta * (adjval l.b))) |> primal;
) nn.layers;
loss |> unpack_flt

```

Test

We need to see how well our trained model works. The test function performs model inference and compares the predictions with the labelled data. By doing so, we can evaluate the accuracy of a neural network.

```

let test nn x y =
Dense.Matrix.S.iter2_rows (fun u v =>
  let p = run_network (Arr u) nn |> unpack_arr in
  Dense.Matrix.Generic.print p;
  Printf.printf "prediction: %i\n" (let _, i = Dense.Matrix.Generic.max_i p in
    i.(1))
) (unpack_arr x) (unpack_arr y)

```

Finally, we can put all the previous parts together. The following code starts the training for 999 iterations.

```

let main () =
  let x, _, y = Dataset.load_mnist_train_data () in
  for i = 1 to 999 do
    let x', y' = Dataset.draw_samples x y 100 in
    backprop nn (F 0.01) (Arr x') (Arr y')
    |> Owl_log.info "#%03i : loss = %g" i
  done;
  let x, y, _ = Dataset.load_mnist_test_data () in
  let x, y = Dataset.draw_samples x y 10 in
  test nn (Arr x) (Arr y)

```

When the training starts, our application keeps printing the value of loss function at the end of iteration. From the output, we can see the value of loss function keeps decreasing quickly after training starts.

```

2019-11-12 01:04:14.632 INFO : #001 : loss = 2.54432
2019-11-12 01:04:14.645 INFO : #002 : loss = 2.48446
2019-11-12 01:04:14.684 INFO : #003 : loss = 2.33889
2019-11-12 01:04:14.696 INFO : #004 : loss = 2.28728
2019-11-12 01:04:14.709 INFO : #005 : loss = 2.23134
2019-11-12 01:04:14.720 INFO : #006 : loss = 2.21974
2019-11-12 01:04:14.730 INFO : #007 : loss = 2.0249
2019-11-12 01:04:14.740 INFO : #008 : loss = 1.96638

```

After the training is finished, we test the accuracy of the network. Here is one example where we input hand-written 3. The vector below shows the prediction. The model says with 90.14 chance it is a number 3, which is quite accurate.

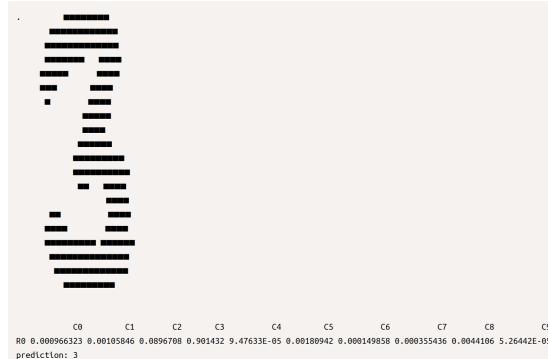


Figure 93: Prediction from the model

Neural Network Module

The simple implementation looks promising enough, but we cannot really leave it for the users to define layers, networks, and the training procedure all by themselves. That would hard be a scalable approach. Therefore, now is finally the time to introduce the Neural Network module provided by Owl. It is actually very similar to the naive framework we just built, but with more complete support to various neurons.

Module Structure

First thing first: a bit of history about neural network module in Owl. Owl has always been designed as a general-purpose numerical library, and we never planned to make it yet another framework for deep neural networks. The original motivation of including such a neural network module was simply for demo purpose, since in almost every presentation we had been to, there was always the same question from the audience: “*can Owl do deep neural network by the way?*” In the end, we became curious about this question ourselves, although the perspective was slightly different. We were sure we could implement a proper neural network framework atop of Owl, but we didn’t know how easy it is. We take it as an excellent opportunity to test Owl’s capability and expressiveness in developing complicated analytical applications.

The outcome is wonderful. It turns out with Owl’s architecture and its internal functionality (algodiff, optimisation, etc.), combined with OCaml’s powerful module system, implementing a full featured neural network module only requires approximately 3500 LOC. Yes, you heard me, 3500 LOC, and it reach TensorFlow’s level of performance on CPU (by the time we measured in 2018).

To understand how we do that, let’s look at fig. 94. It shows the basic module architecture of the neural network module. The neural network in Owl mainly consists of two sub modules: `Neuron` and `Graph`. In the module system, they

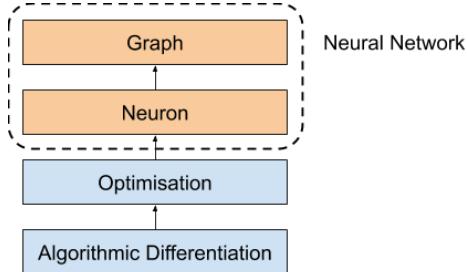


Figure 94: Neural network module structure

are built based on the Optimisation module, which are in turn based on the Algorithmic Differentiation module (`Algodiff`).

`Algodiff` is the most powerful part of Owl and offers great benefits to the modules built atop of it. In neural network case, we only need to describe the logic of the forward pass without worrying about the backward propagation at all, because the `Algodiff` figures it out automatically for us thus reduces the potential errors. This explains why a full-featured neural network module only requires less than 3.5k lines of code. Actually, if you are really interested, you can have a look at Owl's Feedforward Network which only uses a couple of hundreds lines of code to implement a complete Feedforward network. We have already introduced the `Algodiff` module in Owl in previous chapter.

Neurons

The basic unit in neural network is *Neuron*. We have implemented a set of commonly used neurons in `Owl.Neural.Neuron`. Each neuron is a standalone module. We have encapsulated the computation introduced above into a neuron. In previous chapter we have seen how the input and hidden layer can be connected, and we abstract it into the `FullyConnected` layer. We take this neuron as an example.

```

module FullyConnected = struct
  type neuron_typ =
  { mutable w : t;
    mutable b : t;
    mutable init_typ : Init.typ;
    mutable in_shape : int array;
    mutable out_shape : int array
  }
  ...
end

```

This module contains the two parameters we have seen: `w` and `b`, both of type `t` which means the `Algodiff` array. Besides, we also need to specify the input and output shape. They are actually the length of input vector and the length of

the hidden layer itself. The last part in the definition of this neuron `init_typ` is about what kind of initialisation we use, as has been discussed before.

Then this module contains several standard functions that are shared by all the neuron modules.

```
let create ?inputs o init_typ =
  let in_shape =
    match inputs with
    | Some i -> [| i |]
    | None -> [| 0 |]
  in
  { w = Mat.empty 0 o; b = Mat.empty 1 o; init_typ; in_shape; out_shape = [| o |] }
```

After definition of its type, a neuron is created using the `create` function. Here we only need to specify the output shape, or the size of hidden layer `o`.

```
let connect out_shape l =
  assert (Array.length out_shape > 0);
  l.in_shape <- Array.copy out_shape
```

The input shape is actually taken from the previous layer in the `connect` function. We will see why we need this function later. Next we initialise the parameters accordingly:

```
let init l =
  let m = Array.fold_left (fun a b -> a * b) 1 l.in_shape in
  let n = l.out_shape.(0) in
  l.w <- Init.run l.init_typ [| m; n |] l.w;
  l.b <- Mat.zeros 1 n
```

There is nothing magical in the `init` function. The `m` is a flattened input size in case input ndarray is of multiple dimensions. The `w` parameter is initialised with predefined initialisation function, and we can just make `b` all zero, which means no bias at the beginning. Then we have the forward propagation part, in the `run` function:

```
let run x l =
  let m = Mat.row_num l.w in
  let n = Arr.numel x / m in
  let x = Maths.reshape x [| n; m |] in
  let y = Maths.((x *@ l.w) + l.b) in
  y
```

It's the familiar matrix multiplication and summation we have shown previously. The only thing we add is to reshape the possible multiple dimension input into a matrix.

Finally, we divide the backpropagation into several parts: tagging the parameters, get the derivatives, and update parameters. They are also included in the neuron module:

```
let mktag t l =
  l.w <- make_reverse l.w t;
  l.b <- make_reverse l.b t

let mkpri l = [| primal l.w; primal l.b |]

let mkadj l = [| adjval l.w; adjval l.b |]

let update l u =
  l.w <- u.(0) |> primal';
  l.b <- u.(1) |> primal'
```

That's about the main part of the `FullyConnected` neuron and the other similar neurons. Adding a new type of neuron is quite easy thanks to Owl's `Algodiff` module. As other modules such as `Ndarray` and `Algodiff`, the `Owl.Neural` provides two submodules `s` and `d` for both single precision and double precision neural networks.

Neural Graph

Neuron is the core of the neural network module, but we cannot work directly on the neurons. In a neural network, the individual neuron has to be instantiated into a node and constructed into a *graph*. And it's the `Graph` module we users have access to.

The `node` in a neural network is defined as:

```
type node = {
  mutable name : string;
  mutable prev : node array;
  mutable next : node array;
  mutable neuron : neuron;
  mutable output : t option;
  mutable network : network;
  mutable train : bool;
}
```

Besides the neuron itself, a node also contains information such as its parents, children, output, the network this node belongs to, a flag if this node is only for training, etc.

In the `Graph` module, most of the time we need to deal with functions that build node and connect it to existing network. For example:

```
let fully_connected ?name ?(init_typ = Init.Standard) outputs input_node =
  let neuron = FullyConnected (FullyConnected.create outputs init_typ) in
  let nn = get_network input_node in
  let n = make_node ?name [| |] [| |] neuron None nn in
  add_node nn [| input_node |] n
```

What this function does is simple: instantiate the `FullyConnected` neuron using its `create` function, and wrap it into a node `n`. The current network `nn` is found from its input node `input_node`. Then we add `n` as a child node to `nn` and connect it to its parents using the `add_node` function. This step uses the `connect` function of the neuron, and also update the child's input and output shape during connection.

Finally, after understanding the `Graph` module of Owl, we can now “officially” re-define the network in previous example with the Owl neural network module:

```
open Neural.S
open Neural.S.Graph
open Neural.S.Algodiff

let make_network () =
  input [|28; 28; 1|]
  |> fully_connected 40 ~act_typ:Activation.Tanh
  |> linear 10 ~act_typ:Activation.(Softmax 1)
  |> get_network
```

We can see how the input, the hidden layer, and the output from previous example are concisely expressed using the Owl neural network graph API. The `linear` is similar to `fully_connected`, only that it accepts one-dimensional input. The parameter `act_typ` specifies the activation function applied on the output of this node.

Usually, the network definition always starts with `input` neuron and ends with `get_network` function which finalises and returns the constructed network. We can also see the input shape is reserved as a passed in parameter so the shape of the data and the parameters will be inferred later whenever the `input_shape` is determined.

Owl provides a convenient way to construct neural networks. You only need to provide the shape of the data in the first node (often `input` neuron), then Owl will automatically infer the shape for you in the downstream nodes which save us a lot of efforts and significantly reduces the potential bugs.

Training Parameters

Now the last thing to do is to train the model. Again, we want to encapsulate all the manual back-propagation and parameter update into one simple function. It is mainly implemented in the `minimise_network` function in the `Optimise` module. This module provides the `Params` submodule which maintains a set of training hyper-parameters. Without getting into the sea of implementation details, we

focus on one single i -th update iteration and see how these hyper-parameters work. Let's start with the first step in this iteration: training data bataching.

```
| let xt, yt = batch_fun x y i
```

In parameters we can use `Batch` module to specify how the training data are batched. In the definition of cost functions, we often assume that to update the parameters, we need to include all the data. This approach is `Batch.Full`. However, for large scale training task, there can be millions of training data. Besides the memory issue, it is also a waste to wait for all the data to be processed to get an updated parameter.

The mostly commonly used batching method is mini-batch (`Batch.Mini`). It only takes a small part of the training data. As long as the data is fully covered after certain number of iterations, this approach is mathematically equivalent to the full batch. Actually this method is usually more efficient since the training data are often correlated. You don't need to cover all the training data to train a good model. For example, if the model have seen 10 cat images in training, then probably it does not need to be trained on another 10 cat images to get a fairly good model to recognise cat.

To move this method to extreme where only one data sample is used every time, we get the *stochastic* batch (`Batch.Stochastic`) method. It is often not a very good choice, since the vectorised computation optimisation will then not be efficiently utilised.

Another batching approach is `Batch.Sample`. It is the same as mini batch, except that every mini batch is randomly chosen from the training data. It is especially important for the data that are “in order”. Imagine that in the MNIST task, all the training data are ordered according to the digit value. In that case, you may have a model that only works for the lower digits like 0, 1, and 2 at the beginning.

```
| let yt', ws = forward xt
```

There is nothing magical about the `forward` function. It executes the computation layer by layer, and accumulates the result in `yt'`. Note that `yt'` is not simply an `ndarray`, but an `Algodiff` data type that contains all the computation graph information. The `ws` is an array of all the parameters in the neural network.

```
| let loss = loss_fun yt yt'
| let loss = Maths.(loss / _f (Mat.row_num yt |> float_of_int))
```

To compare how different the inference result `y'` is from the true label `y`, we need the loss function. Previously we have used the `cross_entropy`, and in the `Loss` module, the optimisation module provides other popular loss function:

- Loss.L1norm: $\sum |y - y'|$
- Loss.L2norm: $\sum \|y - y'\|_2$
- Loss.Quadratic: $\sum \|y - y'\|_2^2$
- Loss.Hinge: $\sum \max(0, 1 - y^T y')$

```
let reg =
  match params.regularisation <>> Regularisation.None with
  | true -> Owl_utils.aarr_fold (fun a w -> Maths.(a + regl_fun w)) (_f 0.) ws
  | false -> _f 0.
let loss = Maths.(loss + reg)
```

In the regression chapter we have talked about the idea of regularisation and its benefit. We have also introduced different types of regularisation methods. In the optimisation module, we can use `Regularisation.L1norm`, `Regularisation.L2norm`, or `Regularisation.Elastic_net` in training. We can also choose not to apply regularisation method by using the `None` parameter.

In the `Graph` module, Owl provides a `train` function that is a wrapper of this optimisation function. As a result, we can train the network by simply calling:

```
let train () =
  let x, _, y = Dataset.load_mnist_train_data_arr () in
  let network = make_network () in
  let params = Params.config
    ~batch:(Batch.Mini 100)
    ~learning_rate:(Learning_Rate.Adagrad 0.005) 0.1
  in
  Graph.train ~params network x y |> ignore;
  network
```

The `Adagrad` part may seem unfamiliar. So far we keep using a constant learning rate (`Learning_rate.Const`), but the problem is that, this is hardly an ideal setting. We want the gradient descent to be fast with large step at the beginning, but we also want it to be in small steps when it reaches the minimum point. Therefore, Owl provides the `Decay` and `Exp_decay` learning rate methods; both method reduce the base learning rate according to the iteration. The first reduces the learning rate by a factor of $\frac{1}{1+ik}$, where i is the iteration number and k is the reduction rate. Similarly, the second method reduces the leaning rate by a factor of e^{-ik} .

We also implement the other more advanced learning methods. The `Adagrad` we use here adapts the learning rate to the parameters, not just iteration number. It uses smaller step for parameters associated with frequently occurring features. Therefore, it is very suitable for sparse training data. The `Adagrad` achieves this by storing all the past squared gradients. Based on this method, the `RMSprop` proposes to restrict the window of accumulated past gradients by keeping an exponentially decaying average of past squared gradients, so as to reduce the aggressive learning rate reduction strategy. Furthermore, besides the squared

gradients, the `Adam` method also keeps an exponentially decaying average of gradients themselves.

The one last thing we need to notice in the training parameter is the last number `0.1`. It denotes the training epochs, or how many times we should repeat on the whole dataset. Here by taking a `0.1` epoch, we process only a tenth of all the training data for once.

After the training is finished, you can call `Graph.model` to generate a functional model to perform inference. Moreover, `Graph` module also provides functions such as `save`, `load`, `print`, `to_string` and so on to help you in manipulating the neural network. Finally we can test the trained parameter on test set, by comparing the accuracy of correct inference result.

```
let test network =
  let imgs, _, labels = Dataset.load_mnist_test_data () in
  let m = Dense.Matrix.S.row_num imgs in
  let imgs = Dense.Ndarray.S.reshape imgs [|m;28;28;1|] in

  let mat2num x = Dense.Matrix.S.of_array (
    x |> Dense.Matrix.Generic.max_rows
    |> Array.map (fun (_,_,num) -> float_of_int num)
  ) 1 m
  in

  let pred = mat2num (Graph.model network imgs) in
  let fact = mat2num labels in
  let accu = Dense.Matrix.S.(elt_equal pred fact |> sum') in
  Owl_log.info "Accuracy on test set: %f" (accu /. (float_of_int m))
```

The result shows that we can achieve an accuracy of 71.7% with only 0.1 epochs. Increase the training epoch to 1, and the accuracy will be improved to 88.2%. Further changing the epoch number to 2 can lead to an accuracy of about 90%. This result is OK, but not very ideal. Next we will see how a new type of neuron can improve the performance of the network dramatically.

Convolutional Neural Network

So far we have seen how an example of fully connected feed forward neural network evolves step by step. However, there is so much more than just this kind of neural networks. One of the most widely used is the *convolution neural network* (CNN).

We have seen the 1D convolution from signal processing chapter. The 2D convolution is similar, and the only difference is that now the input and filter/kernel are both matrices instead of vectors. As shown in the Signal chapter, the kernel matrix moves along the input matrix in both directions, and the sub-matrix on the input matrix is element-wisely multiplied with the kernel. This operation is especially good at capturing the features in images. It is the key to image process in neural networks.

To perform computer vision we need more types of neurons, and so far we have implemented most of the common type of neurons such as convolution (both 2D and 3D), pooling, batch normalisation, etc. They are enough to support building many state-of-the-art network structures. These neurons should be sufficient for creating from simple MLP to the most complicated convolution neural networks.

Since the MNIST handwritten recognition task is also a computer vision task, let's use the CNN to do it again. The code below creates a small convolutional neural network of six layers.

```
let make_network input_shape =
  input input_shape
  |> lambda (fun x -> Maths.(x / F 256.))
  |> conv2d [|5;5;1;32|] [|1;1|] ~act_typ:Activation.Relu
  |> max_pool2d [|2;2|] [|2;2|]
  |> dropout 0.1
  |> fully_connected 1024 ~act_typ:Activation.Relu
  |> linear 10 ~act_typ:Activation.(Softmax 1)
  |> get_network
```

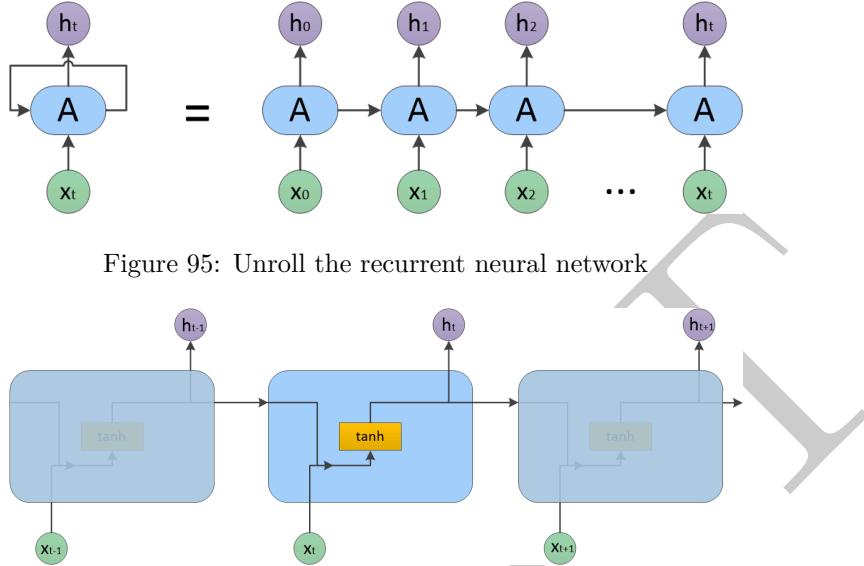
The training method is exactly the same as before, and the training accuracy we can achieve is about 93.3%, within only one 0.1 epoch. Compare this result to the previous simple feedforward network, and we can see the effectiveness of CNN.

Actually, the convolutional neural network is such an important driving force in the computer vision field that in the Part III of this book we have prepared three cases: *image recognition*, *instance segmentation*, and *neural style transfer* to demonstrate how we can use Owl to implement these state-of-art computer vision networks. We will also introduce how these different neurons such as pooling work, how these networks are constructed etc. Please check these chapters for more examples on building CNN in Owl. Another great reference on this topic is the Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition. We refer you to its course notes for more detailed information.

Recurrent Neural Network

In all the previous examples, even the computer vision tasks, one pattern is obvious to see: given one input, the trained network generates another output. However, that's not how every real world task works; in many cases the input data is in a sequence, and the output is updated based on the previous data in the sequence. For example, if we need to generate English based on French, or label each frame in a video, only focusing on the current word/frame is not enough.

That's where the Recurrent Neural Network (RNN) comes to help. It allows the input and output to be in sequence. The basic structure of RNN is quite simple: it is a neural network with loops, and the output of the previous loop is fed into



the next loop as input, together with the current data in sequence. In this way, the information from previous data in the sequence is kept.

As shown in fig. 95, a RNN can actually be unrolled into a chain of multiple connected neural networks. Here the x_i 's are sequential input, and the h_i 's are the *hidden status*, or output of the RNN. The function of RNN therefore mainly relies on the processing logic in A .

In a vanilla recurrent neural network, the function can be really simple and familiar:

$$h_i = \text{activation}(w(h_{i-1}x_i) + b). \quad (72)$$

This is exactly what we have seen in the feed forward networks. Here w and b are the parameters to be trained in this RNN. This process is shown in fig. 96. The activation function here is usually the \tanh function to keep the value within range of $[-1, 1]$.

However, this unit has a problem. Think about it: if you keep updating a diary based on input data; after a while, the information from the old days will certainly be flooded out by the new information. It's like what Sherlock Holmes describes about how brain works: if you keep dumping information into your head, you will have to throw away existing stuff out, be it useful or not. As a result, in this RNN the old data in the sequence would have diminishing effect on the output, which means the output would be less sensitive to the context.

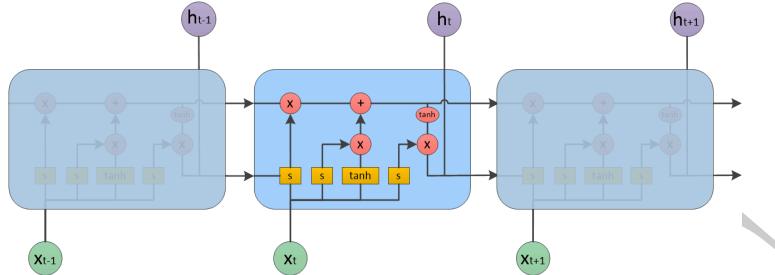


Figure 97: Basic processing unit in LSTM

That's why currently when put into practical use, we often use a special kind of RNN: the Long/Short Term Memory (LSTM).

Long Short Term Memory (LSTM)

LSTM is proposed by Hochreiter & Schmidhuber (1997) and since widely used and refined by many work. Based on RNN, the basic idea of LSTM is simple. We still need to pass in the output from previous loop, but instead of take it as is, the processing unit makes three choices: 1) what to forget, 2) what to remember, and 3) what to output. In this way, the useful information from previous data can be kept longer and the RNN would then have a “longer memory”.

Let's see how it achieves this effect. The process unit of LSTM is shown in fig. 97. It consists of three parts that corresponds to the three choices listed above. Unlike standard RNN, each unit also takes in and produces a state C that flows along the whole loop process. This state is modified twice within the unit.

The first part is called *forget gate layer*. It combines the output h_{t-1} from previous loop and the data x_t , and outputs a probability number between $[0, 1]$ to decide how much of the existing information should be kept. This probability, as you may have guessed, is achieved using the `sigmoid` activation function, denoted by σ .

Next, we need to decide “what to remember” from the existing data. This is done with two branches. The first branch uses the `sigmoid` function to denote which part of the new data $h_{t-1} + x_t$ should be updated, and the second branch using the `tanh` function decides how much value to update for the vector. Both branches follow the procedure in eq. 72, but with different w and b parameters.

By multiplying these two branches together, we know how much new information we should add to the information flow C . The flow C is therefore first multiplied with the output from the *forget gate* to remove unnecessary information, and it adds the output from the second step to gain necessary know knowledge.

Now the only step left is to decide what to output. This time it first runs a

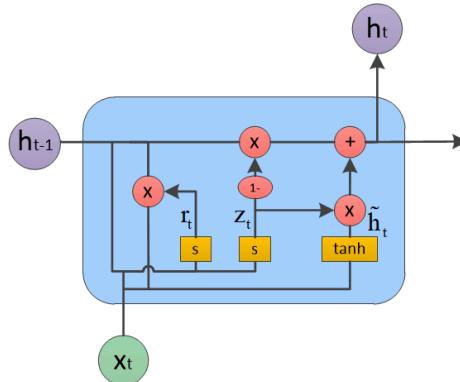


Figure 98: Basic processing unit in GRU

sigmoid function again to decide which part of information flow C to keep, and then applies this filter to a \tanh -scaled information flow to finally get the output h_t .

LSTM is widely used in time-series related applications such as speech recognition, time-series prediction, human action recognition, robot control, etc. Using the neural network module in Owl, we can easily built a RNN that generates text by itself, following the style of the input text.

```
open Neural.S
open Neural.S.Graph

let make_network wndsz vocabsz =
  input [|wndsz|]
  |> embedding vocabsz 40
  |> lstm 128
  |> linear 512 ~act_typ:Activation.Relu
  |> linear vocabsz ~act_typ:Activation.(Softmax 1)
  |> get_network
```

That's it. The network is even simpler than that of the CNN. The only parameter we need to specify in building the LSTM is the length of vectors. However, the generated computation graph is way more complicated due to LSTM's internal recurrent structure. You can download the high-resolution file to take a look if you are interested.

Gated Recurrent Unit (GRU)

The LSTM has been refined in later work since its proposal. There are many variants of it, and one of them is the *Gated Recurrent Unit* (GRU) which is proposed by Cho, et al. in 2014. Its processing unit is shown in fig. 98.

Compared to LSTM, the GRU consists of two parts. The first is a “reset gate”

that decides how much information to forget from the past, and the “update gate” behaves like a combination of LSTM’s forget and input gate. Besides, it also merges the information flow C and output status h . With these changes, the GRU can achieve the same effect as LSTM with fewer operations, and therefore is a bit faster than LSTM in training. In the LSTM code above, we can just replace the `lstm` node to `gru`.

Generative Adversarial Network

There is one more type of neural network we need to discuss. Actually it’s not a particular type of neural network with new neurons like DNN or RNN, but more like a huge family of networks that shows a particular pattern. A Generative Adversarial Network (GAN) consists of two parts: generator and discriminator. During training, the generator tries its best to synthesizes images based on existing parameters, and the discriminator tries its best to separate the generated data and true data. This mutual deception process is iterated until the discriminator can no longer tell the difference between the generated data and the true data (which means us human beings are also not very like to do that).

It might still be difficult to fathom how does a GAN work in action only by text introduction. Let’s look at an example. Previously we have used the MNIST dataset extensively in image recognition task, but now let’s try something different with it. Say we want to build a neural network that can produce a digit picture that looks like it’s taken from the MNIST dataset. It doesn’t matter which digits; the point is being “real”, since this output is actually NOT in the dataset.

To generate such an image does not really need too complicated network structure. For example, we can use something like below (Reference):

```
open Neural.S
open Neural.S.Graph
open Neural.S.Algodiff

let make_generator input_shape =
  let out_size = Owl_utils_array.fold_left ( * ) 1 input_shape in
  input input_shape
  |> fully_connected 256 ~act_typ:(Activation.LeakyRelu 0.2)
  |> normalisation ~decay:0.8
  |> linear 512 ~act_typ:(Activation.LeakyRelu 0.2)
  |> normalisation ~decay:0.8
  |> linear 1024 ~act_typ:(Activation.LeakyRelu 0.2)
  |> normalisation ~decay:0.8
  |> linear out_size ~act_typ:Activation.Tanh
  |> reshape input_shape
  |> get_network
```

We pile up multiple linear layers, activation layers, and normalisation layers. We

don't even have to use the convolution layer. By now you should be familiar with this kind of network structure. This network accepts an ndarray of image shape 28×28 and outputs an ndarray of the same shape, i.e. a black and white image.

Besides this generator, the other half of the GAN is the discriminator. The structure is also quite simple:

```
let make_discriminator input_shape =
    input input_shape
    |> fully_connected 512 ~act_typ:(Activation.LeakyRelu 0.2)
    |> linear 256 ~act_typ:(Activation.LeakyRelu 0.2)
    |> linear 1 ~act_typ:Activation.Sigmoid
    |> get_network
```

The discriminator takes in the image as input. The output from this network is only one value. Since we apply the sigmoid activation function on it, this output means the probability how good the discriminator thinks the outputs from generator are. An output of 1 means the discriminator think this output is taken from MNIST and 0 means the input is obviously a fake.

The question is: how to train these two parts so that they can do their own job perfectly? Here is how the loss values are constructed. Let's assume that we each time we only take one picture from MNIST and training data. First, to train the discriminator, we consider the *ground truth*: we know that the data taken from MNSIT must be true, so it is labelled 1; on the other hand, we know that anything that comes from generator, however good it is, must be a fake, and thus labelled 0. By adding these two parts together, we can get loss value for training the discriminator. The point of this step is to make the discriminator to tell the output from generators from the true images as effectively as possible.

With the same batch of training data, we also want to train the generator. The strategy is totally the reverse now. We combine the generator network and discriminator network together, give it a random noise image as input, and label the true output as 1, even though we know that at the begin the output from generator would be totally fake. The loss value is got from comparing the output of this combined network and the true label 1. During the training of this loss value, we need to make the discriminator as non-trainable. The point of this step is to make generator produce images that can fool the discriminator as convincingly as possible.

That's all. It's kind of like a small scale Darwinism simulation. By iteratively strengthening both parties, the generator can finally become so good that even a good discriminator cannot tell if an input image is faked by the generator or really taken from MNIST. At that stage, the generator is trained well and the job is done.

This approach is successfully applied in many applications, such as Pix2Pix, face ageing, increase photo resolution, etc. For Pix2Pix, you give it a pencil-drawn

picture of bag and it can render it into a real-looking bag. Or think about the popular applications that create an animation character that does not really exist previously. In these applications, the generators are all required to generate images that just do not exist but somehow are real enough to fool the people to think that they do exist in the real world. They may require much more complex network structure, but the general idea of GAN would be the same as what we have introduced.

Summary

This chapter is not yet another “hello world” level tutorial about using the neural network API of a framework. Instead, in this chapter we give a detail introduction to the theory behind neural networks and how a neural network module is constructed in Owl.

We start with the most basic and early form of neural network: perceptron, and then manually build a feedforward network to perform multi-class handwritten digit recognition task by extending the logistic regression. Next, we introduce the Neural Network module, including how its core part works and how it is built step by step. Here we use the Owl API to solve the same example, including training and testing. The training parameter setting regression module in Owl is explained in detail. We then introduce two important types of neural network: the convolutional neural network, together its superior performance against simple feedforward network, and the recurrent neural network, including two of its variants: the LSTM and GRU. We finish this chapter with a brief introduction of the basic idea behind Generative Adversarial Network, another type of neural network that has gained a lot of momentum in research and application recently.

References

- Glorot, Xavier, and Yoshua Bengio. 2010. “Understanding the Difficulty of Training Deep Feedforward Neural Networks.” In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–56.

Natural Language Processing

Text is a dominant media type on the Internet along with images, videos, and audios. Many of our day-to-day tasks involve text analysis. Natural language processing (NLP) is a powerful tool to extract insights from text corpora.

Introduction

NLP is a field of research that helps computers understand, interpret and manipulate human language. It combines many disciplines, including linguistics, computer science, information engineering, and artificial intelligence, etc. NLP is often considered difficult in computer science, since the rules that lie behind natural languages are not always easy for computers to understand. For example, the abstract ideas such as sarcastic and humour are still difficult to convey to computers. Besides, in the real world the data that generated by conversations and tweets etc. are unstructured and cannot be fit well into the traditional row and column structure. The unstructured data are difficult to manipulate.

NLP is a large topic that covers many different advanced problems. *Information Retrieval* focuses on recognising structured information such as key relations or event types from given unstructured information. The Named Entity Recognition task belongs to this category. *Machine Translation* is one of the most important fields in NLP. It involves translating text or speech from one language to another using computer programs. Currently we are still far from being able to build translation systems that can match the quality of human translation. *Text generation* also covers a lot of NLP tasks. The generated text can be used to explain or describe certain input, combining information from multiple sources into a summary, or for interactive conversation with human participants. Research in these fields often needs linguistic knowledge, and deep learning approaches have also achieved good performance on many NLP tasks.

We surely cannot cover all of them in this one single chapter, perhaps not even a whole book. To this end, in this chapter we mainly focus on information retrieval, and specifically, topic modelling. In this chapter, we will use a news dataset crawled from the Internet. It contains 130000 pieces of news from various sources, each line in the file representing one entry. For example we the first line/document is:

a brazilian man who earns a living by selling space for tattoo adverts on his body
is now looking for a customer for his forehead , it appears ... borim says
when clients do n't pay or cancel an ad , he crosses them out . "
skinvertising " caused a stir in the mid-2000s , when many dot.com companies
experimented with it...

Text Corpus

Normally we call a collection of documents a *text corpus*, which contains a large and structured set of texts. For example, for the English language there are the Corpus of Contemporary American English, Georgetown University Multilayer Corpus, etc. Our news collection is also one such example. To perform NLP tasks such as topic modelling, the first and perhaps the most important thing is to represent a text corpus as format that the models can process, instead of directly using natural language.

For the task of topic modelling, we perform the tokenisation on the input English text. The target is to represent each word as an integer index so that we can further process the numbers instead of words. This is called the *tokenisation* of the text. Of course we also need to have a mapping function that from index to word.

Step-by-step Operation

The NLP module in Owl supports building a proper text corpus from given text dataset. In this section we will show how we can build a corpus from a collection of documents, in a step by step way.

In the first step, remove the special characters. We define a regular expression `regexp_split` for special characters such as , , ?, \t etc. First remove them, and then convert all the text into lower-case. The code below defines such a process function, and the `Nlp.Corporus.preprocess` apply it to all the text. Note this function will not change the number of lines in a corpus.

```
let simple_process s =
  Str.split Owl_nlp_utils.regexp_split s
  |> List.filter (fun x -> String.length x > 1)
  |> String.concat " "
  |> String.lowercase_ascii
  |> Bytes.of_string

let preprocess input_file =
  let output_file = input_file ^ ".output" in
  Nlp.Corporus.preprocess simple_process input_file output_file
```

Based on the processed text corpus, we can build the *vocabulary*. Each word is assigned a number id, or index, and we have the dictionary to map word to index, and index to word. This is achieved by using the `Nlp.Vocabulary.build` function.

```
let build_vocabulary input_file =
  let vocab = Nlp.Vocabulary.build input_file in
  let output_file = input_file ^ ".vocab" in
  Nlp.Vocabulary.save vocab output_file
```

The `build` function returns a vocabulary. It contains three hash tables. The first maps a word to an index, and the second index to word. The last hash table is a map between index and its frequency, i.e. number of occurrence in the whole text body. We can check out the words of highest frequency with:

```
let print_freq vocab =
  Nlp.Vocabulary.top_vocab 10 |>
  Owl.Utils.Array.to_string ~sep:", "
```

Unsurprisingly, the “the”’s and “a”’s are most frequently used:

```
- : string =
"the, to, of, a, and, in, \", s, that, on"
```

Changing `Nlp.Vocabulary.top` to `Nlp.Vocabulary.bottom` can show the words of lowest frequency:

```
"eichorst, gcs, freeross, depoliticisation, humping, shopable, appurify,
intersperse, vyaecheslav, raphaelle"
```

However, in a topic modelling task, we don’t want these too frequent but meaningless words and perhaps also the least frequent words that are not about the topic of this document. Now let’s trim off some most and least frequency words. You can trim either by absolute number or by percent. We use percent here, namely trimming off top and bottom 1% of the words.

```
let trim_vocabulary vocab =
  Nlp.Vocabulary.trim_percent ~lo:0.01 ~hi:0.01 vocab
```

With a proper vocabulary at hands, now we are ready to tokenise a piece of text.

```
let tokenise vocab text =
  String.split_on_char ' ' text |>
  List.map (Nlp.Vocabulary.word2index vocab)
```

For example, if we tokenise “this is owl book”, you will get the following output.

```
tokenise vocab "this is an owl book";
- : int list = [55756; 18322; 109456; 90661; 22362]
```

Furthermore, we can now tokenise the whole news collection.

```

let tokenise_all vocab input_file =
  let doc_s = Owl_utils.Stack.make () in
  Owl_io.iteri_lines_of_file
    (fun i s =>
      let t =
        Str.split Owl_nlp_utils.regexp_split s
        |> List.filter (Owl_nlp_vocabulary.exists_w vocab)
        |> List.map (Owl_nlp_vocabulary.word2index vocab)
        |> Array.of_list
      in
      Owl_utils.Stack.push doc_s i)
  input_file;
doc_s

```

The process is simple: in the text corpus each line is a document and we iterate through the text line by line. For each line/document, we remove the special characters, filter out the words that exist in the vocabulary, and map each word to an integer index accordingly. Even though this is a simplified case, it well illustrates the typical starting point of text analysis before delving into any topic modelling.

Use the Corpus Module

But we don't have to build a text corpus step by step. We provide the `Nlp.Corpus` module for convenience. By using the `Nlp.Corpus.build` we perform both tasks we have introduced: building vocabulary, and tokenising the text corpus. With this function we can also specify how to trim off the high-frequency and low-frequency words. Here is an example:

```

let main () =
  let ids = Nlp.Corpus.unique "news.txt" "clean.txt" in
  Printf.printf "removed %i duplicates." (Array.length ids);
  let corpus = Nlp.Corpus.build ~lo:0.01 ~hi:0.01 "clean.txt" in
  Nlp.Corpus.print corpus

```

The `Nlp.Corpus.unique` function is just one more layer of pre-processing. It removes the possible duplicated lines/documents. The output prints out the processing progress, and then a summary of the corpus is printed out.

```

2020-01-28 19:07:05.461 INFO : build up vocabulary ...
2020-01-28 19:07:10.461 INFO : processed 13587, avg. 2717 docs/s
2020-01-28 19:07:15.463 INFO : processed 26447, avg. 2644 docs/s
...
2020-01-28 19:08:09.125 INFO : convert to binary and tokenise ...
2020-01-28 19:08:34.130 INFO : processed 52628, avg. 2104 docs/s
2020-01-28 19:08:39.132 INFO : processed 55727, avg. 1857 docs/s
...
corpus info
  file path : news.txt
  # of docs : 129968

```

```
doc minlen : 10
- : unit = ()
```

The corpus contains three parts: the vocabulary, token, and text string. By calling the `build` function, we also save them for later use. It creates several files in the current directory. First, there is the vocabulary file `news.txt.voc` and `news.txt.voc.txt`. They are the same; only that the latter is in a human-readable format that has each line a word and the corresponding index number. We can get the vocabulary with `Corpus.get_vocab`.

The tokenised text corpus is marshalled to the `news.txt.tok` file, and the string format content is saved as binary file to `news.txt.bin`. We choose to save the content as binary format to save file size. To get the i -th document, we can use `Corpus.get_corpus i` to get the text string, or `Corpus.get_tok corpus i` to get an integer array that is tokenised version of this document.

To access different documents efficiently by the document index (line number), we keep track of the accumulated length of text corpus and token array after processing each document. These two types of indexes are saved in the `news.txt.mdl` file. This file also contains the document id. We have seen the `minlen` value in the output of corpus information. Each document with less than 10 words will not be included in the corpus. The document id is an int array that shows the index (line number) of each document in the original text corpus so that it can be traced back. The document id can be retrieved by `corpus.get_docid corpus`

In the `corpus` module, we provide three mechanisms to iterate through the text corpus: `next`, `iteri`, `mapi`. The `next` function is a generator that yields the next line of text document string in the text corpus until it hits the end of file. The `iteri` and `mapi` functions work exactly like in the normal `Array` module. The first function iterates all the documents one by one in the corpus, and the second maps all the documents in a corpus into another array. The `iteri_tk` and `mapi_tk` work the same, except that the function should work on integer array instead of string. Their signatures is shown below:

```
val iteri : (int -> string -> unit) -> t -> unit
val iteri_tk : (int -> int array -> unit) -> t -> unit
val mapi : (int -> string -> 'a) -> t -> 'a array
val mapi_tk : (int -> 'a -> 'b) -> t -> 'b array
```

The `Corpus` module is designed to support a large number of text corpus. With this tool in hand, we can further proceed with the discussion of topic modelling.

Vector Space Models

Based on the tokenised text corpus, the next thing we need is a mathematical model to express abstract ideas such as “this sentence makes sense and that one does not”, “these two documents are similar”, or “the key word in that paragraph is such and such”. To perform NLP tasks such as text retrieval and topic modelling, we use the *Vector Space Model* (VSM) to do that.

According to the Wikipedia, a VSM is “an algebraic model for representing text documents (and any objects, in general) as vectors of identifiers”. It may sound tricky but the basic idea is actually very simple. For example, let’s assume we only care about three topics in any news: covid19, economics, and election. Then we can represent any news article with a three-element vector, each representing the weight of this topic in it. For the BBC news “Coronavirus: Millions more to be eligible for testing”, we can represent it with vector $(100, 2.5, 0)$. The specific value does not actually matter here. The point is that now instead of a large chunk of text corpus, we only need to deal with this vector for further processing.

The vector space model proposes a framework that maps a document to a vector $d = (x_1, x_2, \dots, x_N)$. This N -dimensional vector space is defined by N basic terms. Under this framework, we mainly have to decide on three factors. The first is to choose the meaning of each dimension, or the N basic concepts in the vector space. The second is to specify the weight of each dimension for a document. In our simple example, why do we assign the first weight to 100 instead of 50? There should be rules about it. That means we need a proper mapping function f defined. Finally, after learning the vector representation, we can cluster or search the documents based on their *similarity*. Some common metrics of similarity are Euclidean distance and cosine similarity. We will talk about it later.

In this chapter we focus on mapping a document to a vector space. However, VSM is not limited to only documents. We can also map a word into a vector that represents a point in a certain vector space. This vector is also called *word embedding*. In a proper representation, the similar words should be clustered together, and can even be used for calculation such as:

$$V_{\text{king}} - V_{\text{man}} + V_{\text{woman}} \approx V_{\text{queen}}.$$

One of the most widely used methods for word embedding is the `word2vec` proposed in (Mikolov, Le, and Sutskever 2013). It includes different algorithms such as the skip-gram for computing the vector representation of words. For general purpose use, Google has already published a pre-trained word2vec-based word embedding vector set based on part of the GoogleNews dataset. This vector set contains 300-dimensional vectors for 3 million words and phrases.

Now, let’s return to the theme of mapping documents to vector space. In the next chapter, we will start with a simple method that instantiate the VSM: the

Bag of Words.

Bag of Words (BOW)

The Bag of Words is a simple way to map docs into a vector space. This space uses all the vocabulary as the dimensions. Suppose there are totally N different words in the vocabulary, then the vector space is of N dimension. The mapping function is simply counting how many times each word in the vocabulary appears in a document.

For example, let's use the five words "news", "about", "coronavirus", "test", and "cases" as the five dimensions in the vector space. Then if a document is "...we heard news a new coronavirus vaccine is being developed which is expected to be tested about September..." will be represented as [1, 1, 1, 1, 0] and the document "...number of positive coronavirus cases is 100 and cumulative cases are 1000..." will be projected to vector [0, 0, 1, 0, 2].

This Bag of Words method is easy to implement based on the text corpus. We first define a function that count the term occurrence in a document and return a hash table:

```
let term_count htbl doc =
  Array.iter
    (fun w =>
      match Hashtbl.mem htbl w with
      | true ->
          let a = Hashtbl.find htbl w in
          Hashtbl.replace htbl w (a +. 1.)
      | false -> Hashtbl.add htbl w 1.)
  doc
```

The hash table contains all the counts of words in this document. Of course, we can also represent the returned results as an array of integers, though the array would likely be sparse. Then we can apply this function to all the documents in the corpus using the map function:

```
let build_bow corpus =
  Nlp.Corporus.mapi_tok
    (fun i doc ->
      let htbl = Hashtbl.create 128 in
      term_count htbl doc;
      htbl)
  corpus
```

Based on this bag of words, the similarity between two vectors can be measured using different methods, e.g. with a simple dot product.

This method is easy to implement and the computation is inexpensive. It may be simple, but for some tasks, especially those that have no strict requirement for

context or position of words, this method proves to work well. For example, to cluster spam email, we only need to specify proper keywords as dimensions, such as “income”, “bonus”, “extra”, “cash”, “free”, “refund”, “promise” etc. We can expect that the spam email texts will be clustered closely and easy to recognise in this vector space using the bag of words.

Actually, one even simpler method is called Boolean model. Instead of term frequency (count of word), the table only contains 1 or 0 to indicate if a word is present in a document. This approach might also benefit from its simplicity and proved to be useful in certain tasks, but it loses the information about the importance of the word. One can easily construct a document that is close to everyone else, by putting all the vocabulary together. The bag of word method fixes this problem.

On the other hand, this simple approach does have its own problems. Back to the previous example, if we want to get how close a document is to “news about coronavirus test cases”, then the doc “...number of positive coronavirus cases is 100 and cumulative cases are 1000...” is scored the same as “hey, I got some good news about your math test result...”. This is not what we expected. Intuitively, words like “coronavirus” should matter more than the more normal words like “test” and “about”. That’s why we are going to introduce an improved method in the next section.

Term Frequency–Inverse Document Frequency (TF-IDF)

In this previous section, we use the count of each term in representing document as vector. It is a way to represent the frequency the term in the document, and we can call it *term frequency*. In the previous section we have seen the intuition that the meaning of different word should be different. This cannot be fixed by simply using term count. In this section we introduce the idea of *Inverse Document Frequency* (IDF) to address this problem.

The basic idea is simple. The IDF is used to represent how common a word is across all the documents. You can imagine that if a word is used throughout all the documents, then it must be of less importance in determining a feature of a document. On the other hand, if a word exists in only 1-2 documents, and where it exists, this word must be of crucial importance to determine its topic. Therefore, the IDF factor can be multiplied with the term frequency to present a more accurate metric for representing a document as vector. This approach is called TF-IDF.

Actually, the two parts TF and IDF just provide frameworks for different computation methods. To compute the term frequency, we can use the count of words c , or the percentage of word in the current document $\frac{c}{N}$ where N is the total number of words in the document. Another computation method is logarithm normalisation which use $\log(c+1)$. We can even use the boolean count that take the frequency of word that exists to be 1 that the ones that are not to be 0. These methods are all defined in the `owl_nlp.Tfidf` module.

```
type tf_typ =
| Binary
| Count
| Frequency
| Log_norm
```

The same goes for the IDF. To measure how common a word w is across all the document, a common way to compute is to do: $\log(\frac{N_D}{n_w})$, where N_D is the total number of documents and n_w is the number of documents with term w in it. This metric is within the range of $[0, \infty)$. It increases with larger total document number or smaller number of documents that contain a specific word. An improved version is called `Idf_Smooth`. It is calculated as $\log(\frac{N_D}{n_w+1})$. This method avoid the n_w to be zero to cause divide error, and also avoid getting a 0 for a word just because it is used across all the documents. In Owl they are included in the type `df_typ`. Here the `Unary` method implies not using IDF, only term frequency.

```
type df_typ =
| Unary
| Idf
| Idf_Smooth
```

In Owl we have the `owl_nlp.Tfidf` module to perform the TF-IDF method. The corpus we have built in the previous section is used as input to it. Specifically, we use the `Nlp.Tfidf.build` function to build the TFIDF model:

```
let build_tfidf corpus =
let tf = Nlp.Tfidf.Count in
let df = Nlp.Tfidf.Idf in
let model = Nlp.Tfidf.build ~tf ~df corpus in
Nlp.Tfidf.save model "news.tfidf";
model
```

In this code, we configure to use the bag-of-words style word count method to calculate term frequency, and use the normal logarithm method to compute inverse document frequency. The model can be saved for later use. After the model is build, we can search similar documents according to a given string. As a random example, let's just use the first sentence in our first piece of news in the dataset as search target: "a brazilian man who earns a living by selling space for tattoo adverts on his body is now looking for a customer for his forehead".

```
let query model doc k =
let typ = Owl_nlp_similarity.Cosine in
let vec = Nlp.Tfidf.apply model doc in
let knn = Nlp.Tfidf.nearest ~typ model vec k in
knn
```

Recall the three ingredients in vector space model: choosing dimension topic words, mapping document to vector, and the measurement of similarity. Here we use the *cosine similarity* as a way to measure how aligned two vectors A and B are. We will talk about the similarity measurement in detail later.

Next, the `vec` returned by the `apply` functions return an array of `(int * float)` tuples. For each item, the integer is the tokenised index of a word in the input document `doc`, and the float number is the corresponding TF-IDF value, based on the `model` we get from previous step. Finally, the `nearest` function searches all the documents and finds the vectors that have the largest similarity with the target document. Let's show the top-10 result by setting `k` to 10:

```
val knn : (int * float) array =
[(11473, -783.546068863270875); (87636, -669.76533603535529);
(121966, -633.9255557720907); (57239, -554.838541799660675);
(15810, -550.95468134048258); (15817, -550.775276912183131);
(15815, -550.775276912183131); (83282, -547.322385552312426);
(44647, -526.074567425088844); (0, -496.924176137374445)]
```

The returned result shows the id of the matched documents. We can retrieve each document by running e.g. `Owl_nlp.Corpora.get_corpus 11473`. To save you some effort to do that, here we list link to some of the original news that are matched to be similar to the target document:

1. *Every tattoo tells a story*, doc id: 11473. [\[Link\]](#)
2. *The Complete Guide to Using Social Media for Customer Service*, doc id: 87636. [\[Link\]](#)
3. *Murder ink? Tattoos can be tricky as evidence*, doc id: 57239. [\[Link\]](#)
4. *Scottish independence: Cinemas pull referendum adverts*, doc id: 15810. [\[Link\]](#)
5. *The profusion of temporarily Brazilian-themed products*, doc id: 44647. [\[Link\]](#)

If you are interested, the input document comes from this BBC news: *Brazil: Man ‘earns a living’ from tattoo ads*. Then you can see that, the searched result is actually quite related to the input document, especially the first one, which is exactly the same story written in another piece of news. The second result is somewhat distant. The word “customer” is heavily used in this document, and we can guess that it is also not frequently seen throughout the text corpus. The fourth news is not about the tattoo guy, but this news features the topic of “customer” and “adverts”. The fifth news is chosen apparently because of the non-frequent word “brazilian” carries a lot of weight in TF-IDF. The interesting thing is that the same document, the first document, is ranked only 10th closest. Note that we just simply take a random sentence without any pre-processing or keyword design; also we use the un-trimmed version of text corpus. Even so, we can still achieve a somewhat satisfactory matching result, and the result fits nicely with the working mechanisms of the TF-IDF method.

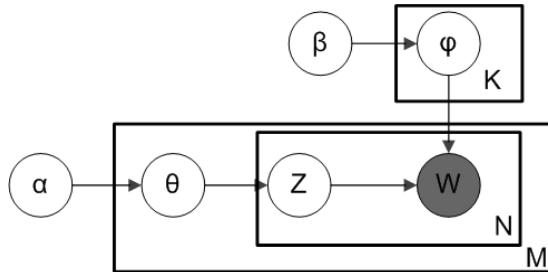


Figure 99: Plate notation for LDA with Dirichlet-distributed topic-word distributions

Latent Dirichlet Allocation (LDA)

In the previous section, we have seen that by specifying a document and using it as a query, we can find out the similar documents as the query. The query document itself is actually seen as a collection of words. However, the real world text, article or news, are rarely as simple as collections of words. More often than not, an article contains one or more *topics*. For example, it can involve the responsibility of government, the protection of environment, and a recent protest in the city, etc. Moreover, each of these topics can hardly be totally covered by just one single word. To this end we introduce the problem *topic modelling*: instead of proposing a search query to find similar content in text corpus, we hope to automatically cluster the documents according to several topics, and each topic is represented by several words.

One of such method to do topic modelling is called *Latent Dirichlet Allocation* (LDA). The trained model of LDA contains two matrices. The first is called the “document-topic”, which contains the number of tokens assigned to each topic in each doc. What do these topics look like then? This concerns the other trained matrix in the model: the “word-topic table”. It contains the number of tokens assigned to each topic for each word. We will see how they work in a latter example. But first, some background theory.

Models

Let's take a look at the model of LDA that is proposed in (Blei, Ng, and Jordan 2003). That is to say, how the LDA thinks about the way a document is composed. The model is expressed in fig. 99.

This model uses the plate notation, the notation for describing probabilistic graphical models, to capture the dependencies among variables. In tbl. 26 we list the definition of the math notations used here and latter in this section.

Table 26: Variable notations in the LDA model

Variable	Meaning
K	number of topics
D	number of documents in text corpus
V	number of words in the vocabulary
N	total number of words in all documents
α	vector of length K , prior weight of the K topics in a document
β	vector of length V , prior weight of the V words in a topic
Θ	vector of length K , distribution of topics in a document
ϕ	vector of length V , distribution of words in a topic
Z	matrix of shape $D \times V$, topic assignment of all words in all documents
W	matrix of shape $D \times V$, token of words in all documents
$n_{d,k}$	how many times the document d uses topic k in the document-topic table
$m_{k,w}$	the number of times topic k uses word w in the topic-word table

In this model, to infer the topics in a corpus, we imagine a **generative process** to create a document. The core idea here is that each document can be described by the distribution of topics, and each topic can be described by distribution of words. This makes sense, since we don't need the text in order to find the topics in an article. The process is as follows:

1. Initialise the distribution of topics $\theta_d \sim \text{Dirichlet}(\alpha)$ in document d . Dirichlet(α) is a Dirichlet distribution parameterised by α . We will talk about it in detail later.
2. Initialise the distribution of words $\phi_k \sim \text{Dirichlet}(\beta)$ for topic k .
3. Iterate each document d and each word position w , and then perform the steps below:
 - first, picks one of these topics randomly (one of the elements in Z). Specifically, the choice of topic is actually taken according to a categorical distribution, parameterised by θ . Formally, this step is represented as $Z_{d,w} \sim \text{Categorical}(\theta_d)$;
 - second, according to the words this topic contains, we pick a word randomly according to ϕ . The picking process also follows categorical distribution: $W_{d,w} \sim \text{Categorical}(\phi_{Z_{d,w}})$.

After finishing this generative process, we now have a “fake” document. The total probability of the model is:

$$P(W, Z, \theta, \phi; \alpha, \beta) = \prod_{i=1}^K P(\phi_i; \beta) \prod_{j=1}^D P(\theta_j; \alpha) \prod_{t=1}^N P(Z_{j,t} | \theta_j) P(W_{j,t} | \phi_{Z_{j,t}}). \quad (73)$$

The eq. 73 corresponds to the above process and model in fig. 99 step by step. It is a multiplication of three parts: the probability of θ across all the documents, the probability of ϕ across all the topics, and that of the generated words across all documents. The LDA hopes to make this generated document to be close to a real document as much as possible. In another word, when we are looking at real document, LDA tries to maximise the possibility eq. 73 that this document can be generated from a set of topics.

Dirichlet Distribution

There is something we need to add to the generative process in the previous section. How *theta* and *phi* are generated? Randomly? No, that would not be a proper way. Think about what would happen if we randomly initialise the document-topic table: each document will be equally likely to contain any topic. But that's rarely the case. An article cannot talk about all the topics at the same time. What we really hope however, is that a single document belongs to a single topic, which is a more real-world scenario. The same goes for the word-topic table.

To that end, LDA uses the Dirichlet Distribution to perform this task. It is a family of continuous multivariate probability distribution parameterised by a vector α . For example, suppose we have only two topics in the whole world. The tuple $(0, 1)$ means it's totally about one topic, and $(1, 0)$ means its totally about the other. We can run the `stats.dirichlet_rvs` function to generate such a pair of float numbers. The results are shown in fig. 100. Both figures have the same number of dots. It shows that with smaller α value, the distribution is pushed to the corners, where it is obviously about one topic or the other. A larger α value, however, makes the topic concentrate around the middle where it's a mixture of both topics.

Therefore, in the model in fig. 99, we have two parameters α and β as prior weights to initialise Θ and ϕ respectively. We use reasonably small parameters to have skewed probability distributions where only a small set of topics or words have high probability.

Gibbs Sampling

Next, we will briefly introduce how the training algorithm works to get the topics using LDA. The basic idea is that we go through the documents one by one. Each word is initially assigned a random topic from the Dirichlet distribution. After that, we iterate over all the documents again and again. In each iterate, we look at each word, and try to find a hopefully a bit more proper topic for

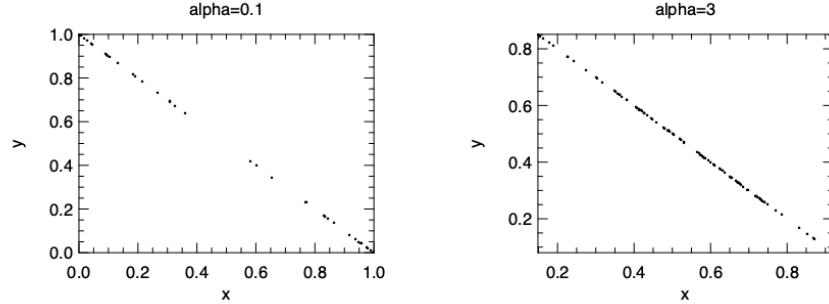


Figure 100: Two dimensional dirichlet distribution with different alpha parameters

this word. In this process, we assume that all the other topic assignments in the whole text corpus are correct except for the current word we are looking at. Then we move forward to the next word in this document. In one iteration, we process all the words in all the documents in the same way. After enough iteration, we can get a quite accurate assignment for each word. And then of course the topics of each document would be clear.

We need to further explain some details in this general description. The most important question is, in the sampling of a document, how exactly do we update the topic assignment of a word? We use the *Gibbs Sampling* algorithm to approximate the distribution of $P(Z|W; \alpha, \beta)$. For this word, we expect to get a vector of length k where k is the number of topics. It represents a conditional probability distribution of a one word topic assignment conditioned on the rest of the model. Based on eq. 73, it can be derived that, in this distribution vector, the k -th element is:

$$p(Z_{d,n} = k | Z_{-d,n}, W, \alpha, \beta) = \frac{n_{d,k} + \alpha_k}{\sum_{i=1}^K (n_{d,i} + \alpha_i)} \frac{m_{k,w_{d,n}} + \beta_{w_{d,n}}}{\sum_i m_{k,i} + \beta_i}.$$

Here $w_{d,n}$ is the current word we are looking at. To perform the sampling, we assume that only the current topic assignment to $w_{d,n}$ is wrong, so we remove the current assignment from the model before this round of iteration begins. Z is the topic assignment of all words in all documents, and W is the text corpus.

This computation is a multiplication of two parts. As shown in tbl. 26, in the first part, $n_{d,k}$ shows how many times the document d uses topic k , and α_k is the prior weight of topic k in document. Therefore, this item means the percentage of words that are also assigned the same topic in the whole document. To put it more simply, it shows how much this document likes topic k . The larger it is, the more likely we will assign the current word to topic k again. Similarly, the second part is the percentage of words that are also assigned the same topic in

the whole document. Therefore, this item indicates how does topic like the word w . Larger number means w will be assigned to this topic k again.

Finally, we multiply these two items to get the final distribution of probability of the word $w_{d,n}$, in the form of a vector of length K . Then we can uniformly draw a topic from this vector. We iterate this sampling process again and again until the model is good enough.

Topic Modelling Example

Owl contains the `owl_nlp.Lda` module to perform LDA method. Let's first use an example to demonstrate how LDA works.

```
let build_lda corpus topics =
  let model = Nlp.Lda.init ~iter:1000 topics corpus in
  Nlp.Lda.(train SimpleLDA model);
  model
```

The input to LDA is still the text corpus we have built. We also need to specify how many topics we want the text corpus to be divided into. Let's say we set the number of topics to 8. The process is simple, we first initialise the model using the `init` function and then we can train the model. Let's take a look at the document-topic table in this model, as shown below.

```
val dk : Arr.arr =
  C0 C1 C2 C3 C4 C5 C6 C7
R0 13 13 4 7 11 12 14 16
R1 35 196 15 42 31 23 122 4
R2 7 9 3 1 3 163 2 4
R3 10 22 23 140 18 11 17 143
...
```

This matrix shows the distribution of topics in each document, represented by a row. Each column represents a topic. For example, you can see that the fifth column of in the third document (R2) is obviously larger than the others. It means that dominantly talks about only the topic 6. Similarly, in the fourth document, the topic 4 and topic 8 are of equal coverage.

We can then check the topic-word table in this model:

```
val wk : Arr.arr =
  C0 C1 C2 C3 C4 C5 C6 C7
R0 1 0 0 0 0 0 0 0
R1 0 0 0 1 0 0 0 0
R2 0 0 0 0 3 0 0 0
R3 0 0 0 0 0 0 0 3
...
```

This is sparse matrix. Each row represents a word from the vocabulary. A topic in a column can thus be represented as the words that have the largest numbers in that column. For example, we can set that a topic be represented by 10 words. The translation from the word-topic table to text representation is straightforward:

```
let get_topics vocab wt =
  Mat.map_cols (fun col ->
    Mat.top col 10
    |> Array.map (fun ws ->
      Owl_nlp.Vocabulary.index2word vocab ws.(0))
  ) wt
```

As an example, we can take a look at the topics generated by the “A Million News Headlines” dataset.

```
Topic 1: police child calls day court says abuse dead change market
Topic 2: council court coast murder gold government face says national police
Topic 3: man charged police nsw sydney home road hit crash guilty
Topic 4: says wa death sa abc australian report open sex final
Topic 5: new qld election ban country future trial end industry hour
Topic 6: interview australia world cup china south accused pm hill work
Topic 7: police health govt hospital plan boost car minister school house
Topic 8: new water killed high attack public farmers funding police urged
```

Here each topic is represented by ten of its highest ranked words in the vocabulary, but you might “feel” a common theme by connecting these dots together, even though some words may stride away a bit far away from this theme. We cannot directly observe the topic, only documents and words. Therefore the topics are latent. The word-topic matrix shows that each word has different weight in the topic and the words in a topic are ranked according to the weight. Now that we know what each topic talks about, we can cluster the documents by their most prominent topic, or just discover what topics are covered in a document, with about how much percentage each.

We have introduced the basic mechanism of LDA. There are many work that extend based on it, such as the SparseLDA in (Yao, Mimno, and McCallum 2009), and LightLDA in (Yuan et al. 2015). They may differ in details but share similar basic theory.

Latent Semantic Analysis (LSA)

Besides LDA, another common technique in performing topic modelling is the Latent Semantic Analysis (LSA). Its purpose is the same as LDA, which is to get two matrices: the document-topic table, and the word-topic table to show the probability distribution of topics in documents and words in topics. The difference is that, instead of using an iterative update approach, LSA

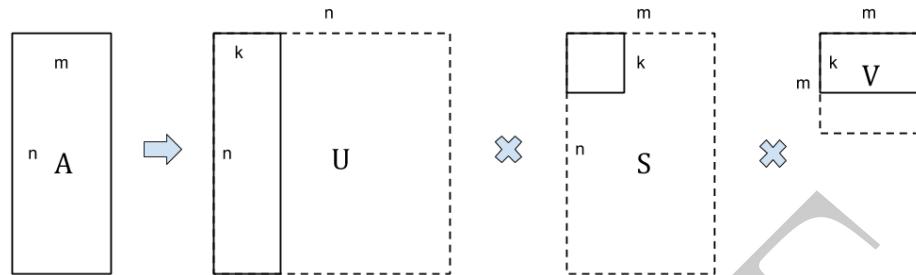


Figure 101: Applying SVD and then truncating on document-word matrix to retrieve topic model

explicitly builds the *document-word matrix* and then performs the singular value decomposition (SVD) on it to get the two aforementioned matrices.

Assume the text corpus contains n documents, and the vocabulary contains m words, then the document-word matrix is of size $n \times m$. We can use the simple word count as the element in this matrix. But as we have discussed in previous section, the count of words does not reflect the significance of a word, so a better way to fill in the document-word matrix is to use the TF-IDF approach for each word in a document.

Apparently, this matrix would be quite sparse. Also its row vectors are in a very high dimension. There is surely redundant information here. For example, if two documents talk about the same topic(s), then the words they contain will largely overlap. To this end, the SVD is then used to reduce the dimension and redundancy in this matrix.

We have seen the SVD in the linear algebra chapter. It is widely used for reducing the dimension of information, by rotating and scaling the coordinating system to find suitable dimensions. SVD decomposes a matrix A into $A = USV^T$. In this specific context of semantic analysis, A is the document-word composition. We can think of the U as representing the relationship between document and the topics, and V^T as the relationship between topics and words.

The columns of U and rows of V^T are both orthonormal bases, and the diagonal matrix S has eigenvalues along its diagonal, each representing the weight of a group of corresponding bases from U and V . Therefore, we can throw away the bases with less weight, truncating only K columns (rows) from each matrix. In that way, we can preserve a large part of the information from the original document-word table by choosing only a small number of topics. This process is shown in fig. 101. Once we have the document-topic table U and the topic-word table V , using the model will be the same as in LDA example.

Compared to LDA, this process is easy to understand and implement. However, SVD is computationally intensive and hard to iterate with new data. The result

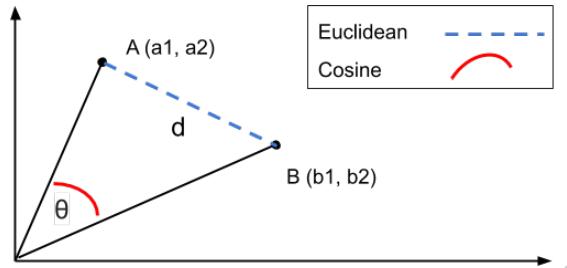


Figure 102: Euclidean distance and cosine similarity in a two dimensional space

is decent, but as this blog shows, it may not be as good as LDA in separating out the topic categories.

Application of topic modelling is wide. For example, it can be used for summarising the large corpus of text data, text categorisation, spam filter, the recommender system, or automatic tagging of articles, etc. It can even be used to effectively discover useful structure in large collection of biological data.

Search Relevant Documents

Topic models are effective tools for clustering documents based on their similarity or relevance. We can further use this tool to query relevant document given an input one. In this section, we will go through some techniques on how to query models built using the previous topic modelling method.

Euclidean and Cosine Similarity

In the previous sections, we see that the topic modelling techniques maps documents to a vector space of topics. We can use different metrics to compare the similarity between two vectors. Two of the commonly used are the *Euclidean* and *Cosine* distances. Suppose we have two vectors A and B , both of length of n . Then the Euclidean distance between these two are:

$$\sqrt{\sum_{i=1}^n (a_i - b_i)^2}. \quad (74)$$

The cosine similarity between two vectors A and B is defined as:

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}. \quad (75)$$

It is the dot product of two vectors divided by the product of the length of both vectors.

We have implemented both methods in the `nlp.Similarity` module as similarity metrics for use in NLP. The relationship between the Euclidean distance and Cosine similarity can be expressed in fig. 102. There are two points on this two dimensional space. The Euclidean measures the direct distance of these two points, while the cosine similarity is about the degree between these two vectors. Therefore, the cosine similarity is more suitable for cases where the magnitude of the vectors does not matter. For example, in topic modelling, we already have two vectors representing documents. If we multiply all the elements in one of them by a scalar 10, the Euclidean distance between these two would change greatly. However, since the probability distribution in the vector does not change, we don't expect the similarity between these two vectors to change. That's why in this case we would prefer to use the cosine similarity as a measurement.

Linear Searching

Suppose we have n documents, each represented by a vector of length m . They are then denoted with variable `corpus`, an array of arrays, each of which is a document. We provide a vector `doc` as the query to search for the top- k similar documents to it. First, we need a function to calculate pairwise distance for the whole model, and returns result in the form of array of `(id, dist)`. Here `id` is the original index of the document, `dist` is the distance between a document in `corpus` and the query document.

```
let all_pairwise_distance typ corpus x =
  let dist_fun = Owl_nlp_similarity.distance typ in
  let l = Array.mapi (fun i y -> i, dist_fun x y) corpus in
  Array.sort (fun a b -> Stdlib.compare (snd a) (snd b)) l;
  l
```

The results are sorted according to the distance, whichever distance metric we use. Based on this routine we can find the k most relevant document:

```
let query corpus doc k =
  let typ = Owl_nlp_similarity.Cosine in
  let l = all_pairwise_distance typ corpus doc in
  Array.sub l 0 k
```

Here we use the cosine similarity as measurement of distance between vectors. To improve the efficiency of computation, we can instead use matrix multiplication to implement the cosine similarity. Specifically, suppose we have the query document vector A , and the corpus of document vector as before, and this array of arrays has already been converted to a dense matrix B , where each row vector represents a document. Then we can compute the AB^T to get the cosine similarity efficiently. Of course, according to eq. 75, we also need to make sure that A and each row r in B are normalised by its own L2-norm before computations, so that for any vector v we can have $\|v\| = 1$.

```

let query corpus doc k =
  let vec = Mat.transpose doc in
  let l = Mat.(corpus *@ vec) in
  Mat.bottom l k

```

Compared to the previous direct element-by-element multiplication, the matrix dot multiplication is often implemented with highly optimised linear algebra library routines, such as in OpenBLAS. These methods utilise various techniques such as multi-processing and multi-threading so that the performance is much better than a direct pairwise computation according to definition.

Summary

In this chapter, we focus on topic modelling, one important natural language processing task, and introduce the basic idea and how Owl support it. First, we introduce how to tokenise text corpus for further mathematical processing. Then we introduce the basic idea of the vector space, and two different ways: the Bag of words (BOW), and Term Frequency–Inverse Document Frequency (TF-IDF), to project a document into a vector space as single vector. The BOW is straightforward to understand and implement, and the TF-IDF consider the how special a word is across the whole text corpus, and therefore usually gives more accurate representation.

Next, we present two different methods based on the vector representation to retrieve topics from the documents: the Latent Dirichlet Allocation (LDA), and Latent Semantic Analysis (LSA). The LSA relies on the singular value decomposition technique on a document-word matrix to do that, while LDA relies on a generative model to iteratively to get the topic model. Once we have the topic modelling, we can compare the similarity between documents, or search for similar documents in the text corpus using different measurement of vector distances. The cosine similarity is a common one in text analysis. The computation of search process can be optimised using matrix multiplication.

References

- Blei, David M, Andrew Y Ng, and Michael I Jordan. 2003. “Latent Dirichlet Allocation.” *Journal of Machine Learning Research* 3 (Jan): 993–1022.
- Mikolov, Tomas, Quoc V Le, and Ilya Sutskever. 2013. “Exploiting Similarities Among Languages for Machine Translation.” *arXiv Preprint arXiv:1309.4168*.
- Yao, Limin, David Mimno, and Andrew McCallum. 2009. “Efficient Methods for Topic Model Inference on Streaming Document Collections.” In *Proceedings of the 15th Acm Sigkdd International Conference on Knowledge Discovery and Data Mining*, 937–46.
- Yuan, Jinhui, Fei Gao, Qirong Ho, Wei Dai, Jinliang Wei, Xun Zheng, Eric Po Xing, Tie-Yan Liu, and Wei-Ying Ma. 2015. “Lightlda: Big Topic Models on

Modest Computer Clusters.” In *Proceedings of the 24th International Conference on World Wide Web*, 1351–61.

DRAFT

Dataframe for Tabular Data

Dataframe is a popular way to manipulate data. It originates from R's dataframe and is widely implemented in many mainstream libraries such as Pandas. Essentially, a dataframe is simple container of the data that can be represented as a table.

Different from the matrices in numerical computing, data stored in a dataframe are not necessarily numbers but a mixture of different types. The flexibility of dataframe largely comes from the dynamic typing inherently offered in a language. Due to OCaml's static type checking, this poses greatest challenges to Owl when we were trying to introduce the similar functionality.

It becomes an art when balancing between flexibility and efficiency in designing the programming interface. This article covers the design of Dataframe module and its basic usage.

Basic Concepts

The dataframe functionality is implemented in Owl's Dataframe module. Owl views a dataframe as a collection of time series data, and each series corresponds to one column in the table. All series must have the same length and each has a unique column head. In the following, we use series and column interchangeably.

Owl packs each series into a unified type called `series` and stores them in an array. As you can already see, dataframe is column-based so accessing columns is way more efficient than accessing rows. The Dataframe module only provides basic functionality to create, access, query, and iterate the data in a frame. We need to combine dataframe with the numerical functions in `Stats` module to reach its full capability. Essentially, Pandas is a bundle of table manipulation and basic statistical functions.

Create Frames

Dataframes can be created in various ways. `Dataframe.make` is the core function if we can to create a frame dynamically. For example, the following code creates a frame consisting of three columns include "name", "age", and "salary" of four people.

```
let name = Dataframe.pack_string_series [| "Alice"; "Bob"; "Carol"; "David" |]
let age = Dataframe.pack_int_series [| 20; 25; 30; 35 |]
let salary = Dataframe.pack_float_series [| 2200.; 2100.; 2500.; 2800. |]
let frame = Dataframe.make [| "name"; "age"; "salary" |] ~data:[| name; age; salary |]
```

If you run the code in `utop`, Owl can pretty print out the dataframe in the following format. If the frame grows too long or too wide, Owl is smart enough to truncate them automatically and present the table nicely in the toplevel.

```
# Owl_ppretty.pp_dataframe Format.std_formatter frame;;
+-----+-----+
| name age salary
+-----+-----+
R0 Alice 20 2200.
R1 Bob 25 2100.
R2 Carol 30 2500.
R3 David 35 2800.
- : unit = ()
```

In fact, you do not necessarily need to pass in the data when calling `make` function. You can make an empty frame by just passing in head names.

```
| let empty_frame = Dataframe.make [| "name"; "age"; "salary" |];;
```

Try the code, and you will see Owl prints out an empty table.

Manipulate Frames

There are a comprehensive set of table manipulation functions implemented in `Dataframe` module. We will go through them briefly in this section.

Now that Owl allows us to create empty frames, it certainly provides functions to dynamically add new columns.

```
let job = Dataframe.pack_string_series [| "Engineer"; "Driver"; "Lecturer";
                                         "Manager" |] in
Dataframe.append_col frame job "job";;

let gender = Dataframe.pack_string_series [| "female"; "male"; "female"; "male" |] in
Dataframe.append_col frame gender "gender";;

let location = Dataframe.pack_string_series [| "Cambridge, UK"; "Helsinki, FIN";
                                              "London, UK"; "Prague, CZ" |] in
Dataframe.append_col frame location "location";;
```

From the output, we can see that the “job” column has been appended to the end of the previously defined dataframe.

```
# Owl_ppretty.pp_dataframe Format.std_formatter frame;;
+-----+-----+-----+-----+
| name age salary job gender location
+-----+-----+-----+
R0 Alice 20 2200. Engineer female Cambridge, UK
R1 Bob 25 2100. Driver male Helsinki, FIN
R2 Carol 30 2500. Lecturer female London, UK
R3 David 35 2800. Manager male Prague, CZ
- : unit = ()
```

We can even concatenate two dataframes. Depending on concatenating direction, there are a couple of things worth our attention:

- when two dataframes are concatenated vertically, they must have the same number of columns and consistent column types; The head names of the first argument will be used in the new dataframe;
- when two dataframes are concatenated horizontally, they must have the same number of rows; all the columns of two dataframes must have unique names.

For example, the following code adds two new entries to the table by concatenating two dataframes vertically.

```
let name = Dataframe.pack_string_series [| "Erin"; "Frank" |];;
let age = Dataframe.pack_int_series [| 22; 24 |];;
let salary = Dataframe.pack_float_series [| 3600.; 5500.; |];
let job = Dataframe.pack_string_series [| "Researcher"; "Consultant" |];
let gender = Dataframe.pack_string_series [| "male"; "male" |];
let location = Dataframe.pack_string_series [| "New York, US"; "Beijing, CN" |];
let frame_1 = Dataframe.make [| "name"; "age"; "salary"; "job"; "gender";
    "location" |]
    ~data:[|name; age; salary; job; gender; location|];
let frame_2 = Dataframe.concat_vertical frame frame_1;;
```

The new dataframe looks like the following.

```
# Owl_pp_dataframe Format.std_formatter frame_2;;
+---+---+---+---+---+
| name age salary job gender location |
+---+---+---+---+---+
R0 Alice 20 2200. Engineer female Cambridge, UK
R1 Bob 25 2100. Driver male Helsinki, FIN
R2 Carol 30 2500. Lecturer female London, UK
R3 David 35 2800. Manager male Prague, CZ
R4 Erin 22 3600. Researcher male New York, US
R5 Frank 24 5500. Consultant male Beijing, CN
- : unit = ()
```

However, if you just want to append one or two rows, the previous method seems a bit overkill. Instead, you can call `Dataframe.append_row` function.

```
# let new_row = Dataframe.([|
    pack_string "Erin";
    pack_int 22;
    pack_float 2300.;
    pack_string "Researcher";
    pack_string "male";
    pack_string "New York, US" |])
in
```

```
| Dataframe.append_row frame new_row
- : unit = ()
```

There are also functions allow you to retrieve the properties, for example:

```
| val copy : t -> t (* return the copy of a dataframe. *)
| val row_num : t -> int (* return the number of rows. *)
| val col_num : t -> int (* return the number of columns. *)
| val shape : t -> int * int (* return the shape of a dataframe. *)
| val numel : t -> int (* return the number of elements. *)
| ...
| ...
```

The module applies several optimisation techniques to accelerate the operations on dataframes. You can refer to the API reference for the complete function list.

Query Frames

We can use various functions in the module to retrieve the information from a dataframe. The basic get and set function treats the dataframe like a matrix. We need to specify the row and column index to retrieve the value of an element.

```
| # Dataframe.get frame 2 1;;
- : Owl_dataframe.elt = Owl.Dataframe.Int 30
```

The `get_row` and `get_col` (also `get_col_by_name`) are used to obtain a complete row or column. For multiple rows and columns, there are also corresponding `get_rows` and `get_cols_by_name`.

Because each column has a name, we can also use `head` to retrieve information. However, we still need to pass in the row index because rows are not associated with names.

```
| # Dataframe.get_by_name frame 2 "salary";;
- : Owl_dataframe.elt = Owl.Dataframe.Float 2500.
```

We can use the `head` and `tail` functions to retrieve only the beginning or end of the dataframe. The results will be returned as a new dataframe. We can also use the more powerful functions like `get_slice` or `get_slice_by_name` if we are interested in the data within a dataframe. The slice definition used in these two functions is the same as that used in Owl's Ndarray modules.

```
# Dataframe.get_slice_by_name ([1;2], ["name"; "age"]) frame;;
- : Owl_dataframe.t =
+-----+
| name age
+-----+
R0 Bob 25
R1 Carol 30
```

Iterate, Map, and Filter

How can we miss the classic iteration functions in the functional programming? Dataframe includes the following methods to traverse the rows in a dataframe. We did not include any method to traverse columns because they can be simply extracted out as series then processed separately.

```
val iteri_row : (int -> elt array -> unit) -> t -> unit
val iter_row : (elt array -> unit) -> t -> unit
val mapi_row : (int -> elt array -> elt array) -> t -> t
val map_row : (elt array -> elt array) -> t -> t
val filteri_row : (int -> elt array -> bool) -> t -> t
val filter_row : (elt array -> bool) -> t -> t
val filter_mapi_row : (int -> elt array -> elt array option) -> t -> t
val filter_map_row : (elt array -> elt array option) -> t -> t
```

Applying these functions to a dataframe is rather straightforward. All the elements in a row are packed into `elt` type, it is a programmer's responsibility to unpack them properly in the passed-in function.

One interesting thing worth mentioning here is that there are several functions associated with extended indexing operators. This allows us to write quite concise code in our application.

```
val ( .%( ) ) : t -> int * string -> elt
(* associated with `get_by_name` *)

val ( .%( )<- ) : t -> int * string -> elt -> unit
(* associated with `set_by_name` *)

val ( .?( ) ) : t -> (elt array -> bool) -> t
(* associated with `filter_row` *)

val ( .?( )<- ) : t -> (elt array -> bool) -> (elt array -> elt array) -> t
(* associated with `filter_map_row` *)
```

```
val ( .$( ) ) : t -> int list * string list -> t
(* associated with `get_slice_by_name` *)
```

Let's present several examples to demonstrate how to use them. We can first pass in row index and head name tuple in %() to access cells.

```
open Dataframe;;
frame.(1, "age");;
(* return Bob's age. *)
frame.(2, "salary") <- pack_float 3000.;;
(* change Carol's salary to 3000. *)
```

The operator .?() provides a shortcut to filter out the rows satisfying the passed-in predicate and returns the results in a new dataframe. For example, the following code filters out the people who are younger than 30.

```
# frame.?(<fun r -> unpack_int r.(1) < 30);;
- : t =
+-----+
| name age salary job gender location
+-----+
R0 Alice 20 2200. Engineer female Cambridge, UK
R1 Bob 25 2100. Driver male Helsinki, FIN
R2 Erin 22 2300. Researcher male New York, US
```

The cool thing about .?() is that you can chain the filters up like below. The code first filters out the people younger than 30, then further filter out whose salary is higher than 2100.

```
frame.?(<fun r -> unpack_int r.(1) < 30)
.?(<fun r -> unpack_float r.(2) > 2100.);;
```

It is also possible to filter out some rows then make some modifications. For example, we want to filter out those people older than 25, then raise their salary by 5%. We can achieve this in two ways. First, we can use `filter_map_row` functions.

```
let predicate x =
  let age = unpack_int x.(1) in
  if age > 25 then (
    let old_salary = unpack_float x.(2) in
    let new_salary = pack_float (old_salary *. 1.1) in
    x.(2) <- new_salary;
```

```

    Some x
)
else
None
;;
filter_map_row predicate frame;;

```

Alternatively, we can use the `.?(<)-` indexing operator. The difference is that we now need to define two functions - one (i.e. `check` function) for checking the predicate and one (i.e. `modify` function) for modifying the passed-in rows.

```

let check x = unpack_int x.(1) > 25;;
let modify x =
  let old_salary = unpack_float x.(2) in
  let new_salary = pack_float (old_salary *. 1.1) in
  x.(2) <- new_salary;
x;;
frame.?(<check>) <- modify;;

```

Running the code will give you the same result as that of calling `filter_map_row` function, but the way of structuring code becomes slightly different.

Finally, you can also use `$.()` operator to replace `get_slice_by_name` function to retrieve a slice of dataframe.

```

# frame.$([0;2], ["name"; "salary"]);
- : t =
+----+
| name salary
+----+
R0 Alice 2200.
R1 Bob 2100.
R2 Carol 3000.

```

Read/Write CSV Files

CSV (Comma-Separated Values) is a common format to store tabular data. The module provides simple support to process CSV files. The two core functions are as follows.

```

val of_csv : ?sep:char -> ?head:string array -> ?types:string array -> string -> t
val to_csv : ?sep:char -> t -> string -> unit

```

`of_csv` function loads a CSV file into in-memory dataframe while `to_csv` writes a dataframe into CSV file on the disk. In both functions, we can use `sep` to specify the separator, the default separator is `tab` in Owl.

For `of_csv` function, you can pass in the head names using `head` argument; otherwise the first row of the CSV file will be used as head. `types` argument is used to specify the type of each column in a CSV file. If `types` is dropped, all the column will be treated as string series by default. Note the length of both `head` and `types` must match the actual number of columns in the CSV file.

The mapping between `types` string and actual OCaml type is shown below:

- `b`: boolean values;
- `i`: integer values;
- `f`: float values;
- `s`: string values;

The following examples are in a gist that contains code and several example CSV files. The first example simply loads the `funding.csv` file into a dataframe, then pretty prints out the table.

```
let fname = "funding.csv" in
let types = [| "s"; "s"; "f"; "s"; "s"; "s"; "s"; "f"; "s"; "s" |] in
let df = Dataframe.of_csv ~sep:',' ~types fname in
Owl_pretty.pp_dataframe Format.std_formatter df
```

The result should look like this. We have truncated out some rows to save space here.

	funding data in csv file									
	permalink	company	numEmps	category	city	state	fundedDate	raisedAmt	raisedCurrency	round
R0	lifelock	LifeLock	nan	web	Tempe	AZ	1-May-07	6850000.	USD	b
R1	lifelock	LifeLock	nan	web	Tempe	AZ	1-Oct-06	6000000.	USD	a
R2	lifelock	LifeLock	nan	web	Tempe	AZ	1-Jan-08	25000000.	USD	c
R3	mycityfaces	MyCityFaces	7.	web	Scottsdale	AZ	1-Jan-08	50000.	USD	seed
R4	flypaper	Flypaper	nan	web	Phoenix	AZ	1-Feb-08	3000000.	USD	a
R5	infusionsoft	Infusionsoft	105.	software	Gilbert	AZ	1-Oct-07	9000000.	USD	a
R1450	cozi	Cozi	26.	software	Seattle	WA	1-Jun-08	8000000.	USD	c
R1451	trusera	Trusera	15.	web	Seattle	WA	1-Jun-07	2000000.	USD	angel
R1452	alerts-com	Alerts.com	nan	web	Bellevue	WA	8-Jul-08	1200000.	USD	a
R1453	myrio	Myrio	75.	software	Bothell	WA	1-Jan-01	20500000.	USD	unattributed
R1454	grid-networks	Grid Networks	nan	web	Seattle	WA	30-Oct-07	9500000.	USD	a
R1455	grid-networks	Grid Networks	nan	web	Seattle	WA	20-May-08	10500000.	USD	b

The second example is slightly more complicated. It loads `estate.csv` file then filters out the some rows with two predicates. You can see how the two predicates are chained up with `.?()` indexing operator.

```
open Dataframe

let fname = "estate.csv" in
let d = (of_csv ~sep:',' fname)
  .?(fun row -> unpack_string row.(7) = "Condo")
  .?(fun row -> unpack_string row.(4) = "2")
in
Owl_pretty.pp_dataframe Format.std_formatter d
```

For more examples, please refer to the gist [dataframe.ml](#).

Infer Type and Separator

We want to devote a bit more text to CSV files. In the previous section, when we use `of_csv` function to load a CSV file, we explicitly pass in the separator and the types of all columns. However, both parameters are optional and can be skipped.

Dataframe is able to automatically detect the correct separator and the type of each column. Of course, it is possible that the detection mechanism fails but such probability is fairly low in many cases. Technically, Dataframe first tries a set of predefined separators to see which one can correctly separate the columns, and then it tries a sequence of types to find out which one is able to correctly unpack the elements of a column.

There are several technical things worth mentioning here:

- to be efficient, Dataframe only takes maximum the first 100 lines in the CSV file for inference;
- if there are missing values in a column of integer type, it falls back to float value because we can use `nan` to represent missing values;
- if the types have been decided based on the first 100 lines, any following lines containing the data of inconsistent type will be dropped.

With this capability, it is much easier to load a CSV to quickly investigate what is inside.

```
open Dataframe

let fname = "estate.csv" in
let df = Dataframe.of_csv fname in
Owl_pretty.pp_dataframe Format.std_formatter df
```

You can use the `Dataframe.types` function to retrieve the types of all columns in a dataframe.

Summary

This chapter introduces the dataframe module in Owl, including its creating, manipulation, query, loading and saving, etc. Comparing to those very mature libraries like Pandas, the Dataframe module in Owl is very young. We also try to keep its functionality minimal in the beginning to reserve enough space for future adjustment. The dataframe should only offer a minimal set of table manipulation functions, its analytical capability should come from the combination with other modules (e.g. `stats`) in Owl.

DRAFT

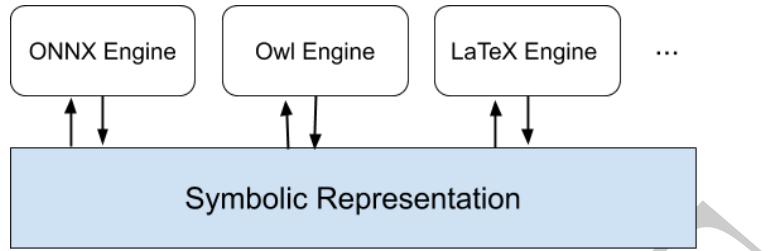


Figure 103: Architecture of the symbolic system

Symbolic Representation

Introduction

The development of `owl_symbolic` library is motivated by multiple factors. For one thing, scientific computation can be considered as consisting of two broad categories: numerical computation, and symbolic computation. Owl has achieved a solid foundation in the former, but as yet to support the latter one, which is heavily utilised in a lot of fields. For another, with the development of neural network compilers such as TVM, it is a growing trend that the definition of computation can be separated out, and the low level compilers to deal with optimisation and code generation etc. to pursue best computation performance. Besides, tasks such as visualising a computation also require some form or intermediate representation (IR). Owl has already provided a computation graph layer to separate the definition and execution of computation to improve the performance, but it's not an IR layer to perform these different tasks as mentioned before. Towards this end, we begin to develop an intermediate symbolic representation of computations and facilitate various tasks based on this symbol representation.

One thing to note is that do not mistake our symbolic representation as the classic symbolic computation (or Computer Algebra System) that manipulate mathematical expressions in a symbolic way, which is similar to the traditional manual computations. It is indeed one of our core motivations to pursue the symbolic computation with Owl. Currently we provide a symbolic representation layer as the first step towards that target. More discussion will be added in future versions with the development with the support of symbolic math in Owl.

Design

`owl_symbolic` is divided into two parts: the core symbolic representation that constructs a symbolic graph, and various engines that perform different task based on the graph. The architecture design of this system is shown in fig. 103.

The core abstraction is an independent symbolic representation layer. Based on this layer, we have various engines that can be translated to and from this

symbolic representation. Currently we support three engines: the ONNX binary format, the computation graph in Owl, and the LaTeX string. The CAS engine is currently still an on-going research project, and we envision that, once finished, this engine can be used to pre-process a symbolic representation so that it as a simplified canonical form before being processed by other engines.

Core abstraction

The core part is designed to be minimal and contains only necessary information. Currently it has already covered many common computation types, such as math operations, tensor manipulations, neural network specific operations such as convolution, pooling etc. Each symbol in the symbolic graph performs a certain operation. Input to a symbolic graph can be constants such as integer, float number, complex number, and tensor. The input can also be variables with certain shapes. An empty shape indicates a scalar value. The users can then provide values to the variable after the symbolic graph is constructed.

Symbol

The symbolic representation is defined mainly as array of `symbol`. Each `symbol` is a graph node that has an attribution of type `owl_symbolic_symbol.t`. It means that we can traverse through the whole graph by starting with one `symbol`. Besides symbols, the `name` field is the graph name, and `node_names` contains all the nodes' name contained in this graph.

```
type symbol = Owl_symbolic_symbol.t Owl_graph.node

type t =
  { mutable sym_nodes : symbol array
  ; mutable name : string
  ; mutable node_names : string array
  }
```

Let's look at `owl_symbolic_symbol.t`. It defines all the operations contained in the symbolic representation:

```
type t =
  | NOOP
  | Int of Int.t
  | Complex of Complex.t
  | Float of Float.t
  | Tensor of Tensor.t
  | Variable of Variable.t
  | RandomUniform of RandomUniform.t
  | Sin of Sin.t
  | Cos of Cos.t
  | Exp of Exp.t
  | ReduceSum of ReduceSum.t
  | Reshape of Reshape.t
```

```
| Conv of Conv.t
....
```

There are totally about 150 operations included in our symbolic representation. Each operation is implemented as a module. These modules share common attributes such as name, input operation names, output shapes, and then each module contains zero or more attributes of itself. For example, the `Sin` operation module is implemented as:

```
module Sin = struct
  type t =
    { mutable name : string
    ; mutable input : string array
    ; mutable out_shape : int array option array
    }

  let op_type = "Sin"

  let create ?name x_name =
    let input = [| x_name |] in
    let name = Owl_symbolic_utils.node_name ?name op_type in
    { name; input; out_shape = [| None |] }
end
```

The module provides properties such as `op_type` and functions such as `create` that returns object of type `Sin.t`. The `name`, `input` and `out_shape` are common attributes in the operation modules.

In implementing the supported operations, we follow the category used in ONNX. These operations can be generally divided into different groups as shown below.

- Generators: operations that generate data, taking no input. For example, the `Int`, `Float`, `Tensor`, `Variable`, etc.
- Logical: logical operations such as `Xor`.
- Math: mathematical operations. This group of operations makes a large part of the total operations supported.
- Neural Network: neural network related operations such as convolution and pooling.
- Object detection: also used in neural network, but the operations that are closely related with object detection applications, including `RoiAlign` and `NonMaxSuppression`.
- Reduction: reduction (or folding) math operations such as sum reduce.
- RNN: Recurrent neural network related operations such as `LSTM`.
- Tensor: Normal tensor operations, like the ones that are included in the `Ndarray` module, such as `concat`, `reshape`, etc.
- Sequence: take multiple tensors as one single object called `sequence`, and there are different corresponding functions on the sequence type data, such as `SequenceInsert`, `SequenceLength` etc.

Based on these operation modules, we provide several functions on the `Owl_symbolic_symbol.t` type:

- `name`: get the name of operation
- `op_type`: get the operation type string
- `input`: get the input nodes name of an operation
- `set_input`: update the input nodes name
- `output`: get the output nodes name
- `set_output`: update the output nodes name

There are also some functions that only apply to certain types of operations. The generator type of operations all need to specify the type of data it supports. Therefore, we use `dtype` function to check their data types. Another example is the `output` property. For most of the operation, it has only one output, and therefore its name is its output name. However, for operations such as `MaxPool` that contains multiple outputs, we need another function: `output`.

Type Checking

The type supported by `owl_symbolic` is listed as follows:

```
type number_type =
  | SNT_Noop
  | SNT_Float
  | SNT_Double
  | SNT_Complex32
  | SNT_Complex64
  | SNT_Bool
  | SNT_String
  | SNT_Int8
  | SNT_Int16
  | SNT_Int32
  | SNT_Int64
  | SNT_Uint8
  | SNT_Uint16
  | SNT_Uint32
  | SNT_Uint64
  | SNT_Float16
  | SNT_SEQ of number_type
```
This list of types covers most number and non-number types. `SNT_SEQ` means the type a list of the basic elements as inputs/outputs.

```

**\*\*Operators\*\***

All these operations are invisible to users.  
 What the users really use are the \*operators\*.  
 To build a graph, we first need to build the required attributes into an operation, and then put it into a graph node. This is what an operator does.  
 Take the `sin` operator as an example:

```
```clike
let sin ?name x =
  let xn = Owl_symbolic_graph.name x in
```

```
let s = Owl_symbolic_ops_math.Sin.create ?name xn in
make_node (Owl_symbolic_symbol.Sin s) [| x |]
```

Here the `sin` operator takes its parent node `x` as input, get its name as input property, and create a symbol node with the function `make_node`. This function takes an operation and an array of parent symbols, and then creates one symbol as return. What it does is mainly creating a child node using the given operation as node attribution, updating the child's input and output shape, and then connecting the child with parents before returning the child node. The connection is on both directions:

```
connect_ancestors parents [| child |];
let uniq_parents = Owl_utils_array.unique parents in
Array.iter (fun parent -> connect_descendants [| parent |] [| child |])
uniq_parents
```

Therefore, the users can use the operators to build a graph representation. Here is an example:

```
open Owl_symbolic
open Op
open Infix

let x = variable "x_0"
let y = exp ((sin x ** float 2.) + (cos x ** float 2.))
+ (float 10. * (x ** float 2.))
+ exp (pi () * complex 0. 1.)
```

Here we start with the `variable` operator, which creates a placeholder for incoming data later. You can specify the shape of the variable with `~shape` parameter. If not specified, then it defaults to a scalar. You can also choose to initialise this variable with a `tensor` so that even if you don't feed any data to the variable, the default tensor value will be used. A tensor in `owl-symbolic` is defined as:

```
type tensor =
{ mutable dtype : number_type
; mutable shape : int array
; mutable str_val : string array option
; mutable flt_val : float array option
; mutable int_val : int array option
; mutable raw_val : bytes option
}
```

A tensor is of a specific type of data, and then it contains the value: string array, float array, integer array, or bytes. Only one of these fields can be used. If initialised with a tensor, a variable takes the same data type and shape as that of the tensor.

Naming

Currently we adopt a global naming scheme, which is to add an incremental index number after each node's type. For example, if we have an `Add` symbol, a `Div` symbol, and then another `Add` symbol in a graph, then each node will be named `add_0`, `div_1`, and `add_1`. One exception is the variable, where a user has to explicitly name when create a variable. Of course, users can also optionally name any node in the graph, but the system will check to make sure the name of each node is unique. The symbolic graph contains the `node_names` field that includes all the nodes' names in the graph.

Shape Inferencing

One task the symbolic core needs to perform is shape checking and shape inferencing. Shape inference is performed in the `make_node` function and therefore happens every time a user uses an operation to construct a symbolic node and connect it with previous nodes. It is assumed that the parents of the current node are already known.

```
let (in_shapes : int array option array array)=
  Array.map (fun sym_node ->
    Owl_graph.attr sym_node |> Owl_symbolic_symbol.out_shape
  ) parents
  in
let (shape : int array option array) =
  Owl_symbolic_shape.infer_shape in_shapes sym
...
...
```

As the code shows, for each node, we first find the output shapes of its parents. The `in_shape` is of type `int array option array array`. You can understand it this way: `int array` is a shape array; `int array option` means this shape could be `None`. Then `int array option array` is one whole input from previous parent, since one parent may contain multiple outputs. Finally, `int array option array array` includes output from all parents. The main function `Owl_symbolic_shape.infer_shape` then infers the output shape of current node, and save it to the `out_shape` property of that symbol.

The `infer_shape` function itself checks the symbol type and then match with specific implementation. For example, a large number of operations actually takes one parent and keep its output shape:

```
let infer_shape input_shapes sym =
  | Sin _ -> infer_shape_01 input_shapes
  | Exp _ -> infer_shape_01 input_shapes
  | Log _ -> infer_shape_01 input_shapes
  ...
  ...
let infer_shape_01 input_shapes =
  match input_shapes.(0).(0) with
```

```
| Some s -> [| Some Array.(copy s) |]
| None -> [| None |]
```

This pattern `infer_shape_01` covers these operations. It simply takes the input shape, and returns the same shape.

There are two possible reasons for the input shape to be `None`. At first each node will be initialised with `None` output shape. During shape inference, in certain cases, the output shape depends on the runtime content of input nodes, not just the shapes of input nodes and attributions of the currents node. In that case, the output shapes is set to `None`. Once the input shapes contain `None`, the shape inference results hereafter will all be `None`, which means the output shapes cannot be decided at compile time.

Multiple output

Most of the operators are straightforward to implement, but some of them returns multiple symbols as return. In that case, an operation returns not a node, but a tuple or, when output numbers are uncertain, an array of nodes. For example, the `MaxPool` operation returns two outputs, one is the normal maxpooling result, and the other is the corresponding tensor that contains indices of the selected values during pooling. Or we have the `split` operation that splits a tensor into a list of tensors, along the specified axis. It returns an array of symbols.

Engines

Based on this simple core abstraction, we use different *engines* to provide functionalities: converting to and from other computation expression formats, print out to human-readable format, graph optimisation, etc. As we have said, the core part is kept minimal. If the engines require information other than what the core provides, each symbol has an `attr` property as extension point.

All engines must follow the signature below:

```
type t

val of_symbolic : Owl_symbolic_graph.t -> t
val to_symbolic : t -> Owl_symbolic_graph.t
val save : t -> string -> unit
val load : string -> t
```

It means that, each engine has its own core type `t`, be it a string or another format of graph, and it needs to convert `t` to and from the core symbolic graph type, or save/load a type `t` data structure to file. An engine can also contain extra functions besides these four.

Now that we have explained the design of `owl_symbolic`, let's look at the details of some engines in the next few sections.

ONNX Engine

The ONNX Engine is the current focus of development in `owl_symbolic`. ONNX is a widely adopted open neural network exchange format. A neural network model defined in ONNX can be, via suitable converters, can be run on different frameworks and thus hardware accelerators. The main target of ONNX is to promote the interchangeability of neural network and machine learning models, but it is worthy of noting that the standard covers a lot of basic operations in scientific computation, such as power, logarithms, trigonometric functions, etc. Therefore, ONNX engines serves as a good starting point for its coverage of operations.

Taking a symbolic graph as input, how would then the ONNX engine produce ONNX model? We use the `ocaml-protoc`, a protobuf compiler for OCaml, as the tool. The ONNX specification is defined in an `onnx.proto` file, and the `ocaml-protoc` can compile this protobuf files into OCaml types along with serialisation functions for a variety of encodings.

For example, the toplevel message type in `onnx.proto` is `ModelProto`, defined as follows:

```
message ModelProto {
    optional int64 ir_version = 1;
    repeated OperatorSetIdProto opset_import = 8;
    optional string producer_name = 2;
    optional string producer_version = 3;
    optional string domain = 4;
    optional int64 model_version = 5;
    optional string doc_string = 6;
    optional GraphProto graph = 7;
    repeated StringStringEntryProto metadata_props = 14;
};
```

And the generated OCaml types and serialisation function are:

```
open Owl_symbolic_specs.PT

type model_proto =
  { ir_version : int64 option
  ; opset_import : operator_set_id_proto list
  ; producer_name : string option
  ; producer_version : string option
  ; domain : string option
  ; model_version : int64 option
  ; doc_string : string option
  ; graph : graph_proto option
  ; metadata_props : string_string_entry_proto list
  }

val encode_model_proto : Onnx_types.model_proto -> Pbrt.Encoder.t -> unit
```

```Besides the meta information such as model version and IR version etc., a model is mainly a graph, which includes input/output information and an array of nodes.

A node specifies operator type, input and output node name, and its own attributions, such as the `axis` attribution in reduction operations.

Therefore, all we need is to build up a `model\_proto` data structure gradually from attributions to nodes, graph and model. It can then be serialised using `encode\_model\_proto` to generate a protobuf format file, and that is the ONNX model we want.

Besides building up the model, one other task to be performed in the engine is type checking and type inferencing. The [operator documentation](<https://github.com/onnx/onnx/blob/master/docs/Operators.md>) lists the type constraints of each operator. For example, the sine function can only accept input of float or double number types, and generate the same type of input as that of input.

Each type of operator has its own rules of type checking and inferencing. Starting from input nodes, which must contain specific type information, this chain if inferencing can thus verify the whole computation meets the type constraints for each node, and then yield the final output types of the whole graph.

The reason that type checking is performed at the engine side instead of the core is that each engine may have different type constraints and type inferencing rules for the operators.

### ### Example 1: Basic operations

Let's look at a simple example.

```
```ocaml
open Owl_symbolic
open Op
open Infix

let x = variable "X"
let y = variable "Y"
let z = exp ((sin x ** float 2.) + (cos x ** float 2.)) + (float 10. * (y ** float 2.))
let g = SymGraph.make_graph [| z |] "sym_graph"
let m = ONNX_Engine.of_symbolic g
let _ = ONNX_Engine.save m "test.onnx"
```

After including necessary library component, the first three line of code creates a symbolic representation z using the symbolic operators such as sin, pow and float. The x and y are variables that accept user input. It is then used to create a symbolic graph. This step mainly checks if there is any duplication of node names. Then the of_symbolic function in ONNX engine takes the symbolic graph as input, and generates a model_proto data structure, which can be further saved as a model named test.onnx.

To use this ONNX model we could use any framework that supports ONNX. Here we use the Python-based ONNX Runtime as an example. We prepare a simple Python script as follows:

```

import numpy as np
import math
import onnxruntime as rt

sess = rt.InferenceSession("test.onnx")
input_name_x = sess.get_inputs()[0].name
input_name_y = sess.get_inputs()[1].name
x = np.asarray(math.pi, dtype="float32")
y = np.asarray(3., dtype="float32")

pred_onx = sess.run(None, {input_name_x: x, input_name_y: y})[0]
print(pred_onx)

```This script is very simple: it loads the ONNX model we have just created, and then get the two input variables, and assign two values to them in the `sess.run` command. All the user need to know in advance is that there are two input variables in this ONNX model. Note that we could define not only scalar type input but also tensor type variables in `owl_symbolic`, and then assign NumPy array to them when evaluating.

```

### ### Example 2: Variable Initialisation

We can initialise the variables with tensor values so that these default values are used even if no data are passed in.

Here is one example:

```

```ocaml
open Owl_symbolic
open Op

let _ =
  let flt_val = [| 1.; 2.; 3.; 4.; 5.; 6. |] in
  let t = Type.make_tensor ~flt_val [| 2; 3 |] in
  let x = variable ~init:t "X" in
  let y = sin x in
  let g = SymGraph.make_graph [| y |] "sym_graph" in
  let z = ONNX_Engine.of_symbolic g in
  ONNX_Engine.save z "test.onnx"

```

This computation simply takes an input variable x and then apply the \sin operation. Let's look at the Python side.

```

import numpy as np
import onnxruntime as rt

sess = rt.InferenceSession("test.onnx")
pred_onx = sess.run(None, input_feed={})
print(pred_onx[0])

```

The expected output is:

```

[[ 0.84147096  0.9092974  0.14112 ]
 [-0.7568025 -0.9589243 -0.2794155 ]]

```

Note how the initializer works without user providing any input in the input feed dictionary. Of course, the users can still provide their own data to this computation, but the mechanism may be a bit different. For example, in `onnx_runtime`, using `sess.get_inputs()` gives an empty set this time. Instead, you should use `get_overridable_initializer()`:

```
input_x = sess.get_overridable_initializer()[0]
input_name_x = input_x.name
input_shape_x = input_x.shape
x = np.ones(input_shape_x, dtype="float32")
pred_onx = sess.run(None, {input_name_x: x})
```

Example 3: Neural network

The main purpose of the ONNX standard is for expressing neural network models, and we have already covered most of the common operations that are required to construct neural networks. However, to construct a neural network model directly from existing `owl_symbolic` operations requires a lot of details such as input shapes or creating extra nodes. For example, if you want to build a neural network with operators directly, you need to write something like:

```
let dnn =
  let x = variable ~shape:[| 100; 3; 32; 32 |] "X" in
  let t_conv0 = conv ~padding:Type.SAME_UPPER x
    (random_uniform ~low:(-0.138) ~high:0.138 [| 32; 3; 3; 3 |]) in
  let t_zero0 =
    let flt_val = Array.make 32 0. in
    let t = Type.make_tensor ~flt_val [| 32 |] in
    tensor t
  in
  let t_relu0 = relu (t_conv0 + t_zero0) in
  let t_maxpool0, _ = maxpool t_relu0 ~padding:VALID ~strides:[| 2; 2 |] [| 2; 2 |
    |] in
  let t_reshape0 = reshape [| 100; 8192 |] t_maxpool0 in
  let t_rand0 = random_uniform ~low:(-0.0011) ~high:0.0011 [| 8192; 512 |] in
  ...
```

Apparently that's too much information for the users to handle. To make things easier for the users, we create neural network layer based on existing symbolic operations. This light-weight layer takes only 180 LoC, and yet it provides an Owl-like clean syntax for the users to construct neural networks. For example, we can construct a MNIST-DNN model:

```
open Owl_symbolic_neural_graph
let nn =
  input [| 100; 3; 32; 32 |]
  |> normalisation
  |> conv2d [| 32; 3; 3 |] [| 1; 1 |]
```

```

|> activation Relu
|> max_pool2d [| 2; 2 |] [| 2; 2 |] ~padding:VALID
|> fully_connected 512
|> activation Relu
|> fully_connected 10
|> activation (Softmax 1)
|> get_network

let _ =
  let onnx_graph = Owl_symbolic_engine_onnx.of_symbolic nn in
  Owl_symbolic_engine_onnx.save onnx_graph "test.onnx"

```

Besides this simple DNN, we have also created the complex architectures such as ResNet, InceptionV3, SqueezeNet, etc. They are all adapted from existing Owl DNN models with only minor change. The execution of the generated ONNX model is similar:

```

import numpy as np
import onnxruntime as rt

sess = rt.InferenceSession("test.onnx")
input_name_x = sess.get_inputs()[0].name
input_name_shape = sess.get_inputs()[0].shape
input_x = np.ones(input_name_shape, dtype="float32")
pred_onx = sess.run(None, {input_name_x: input_x})[0]

```

For simplicity, we generate a dummy input for the execution/inference phase of this model. Of course, currently in our model the weight data is not trained. Training of a model should be completed on a framework such as TensorFlow. Combining trained weight data into the ONNX model remains to be a future work.

Furthermore, by using tools such as `js_of_ocaml`, we can convert both examples into JavaScript; executing them can create the ONNX models, which in turn can be executed on the browser using `ONNX.js` that utilises WebGL. In summary, using ONNX as the intermediate format for exchange computation across platforms enables numerous promising directions.

LaTeX Engine

The LaTeX engine takes a symbolic representation as input, and produce LaTeX strings which can then be visualised using different tools. Its design is simple, mainly about matching symbol type and projecting it to correct implementation. Again, let's look at an example that builds up a symbolic representation of a calculation $\exp(\sin(x_0)^2 + \cos(x_0)^2) + 10 \times x_0^2 + \exp(\pi i)$

```

open Owl_symbolic
open Op
open Infix

```

```

let make_expr0 () =
  let x = variable "x_0" in
  let y =
    exp ((sin x ** float 2.) + (cos x ** float 2.))
    + (float 10. * (x ** float 2.))
    + exp (pi () * complex 0. 1.)
  in
  SymGraph.make_graph [| y |] "sym_graph"

```

This expression can be converted into a corresponding LaTeX string:

```

# let () = make_expr0 ()
|> LaTeX_Engine.of_symbolic
|> print_endline
\exp(\sin(x_0) ^ 2 + \cos(x_0) ^ 2) + 10 \times x_0 ^ 2 + \exp(\pi \times 1.00i)

```

Simply putting it in the raw string form is not very helpful for visualisation. We have built a web UI in this Engine that utilises KaTeX, which renders LaTeX string directly on a browser. Below we use the `html` function provided by the engine to show this string on our web UI using the functionality the engine provides.

```

# let () =
  let exprs = [ make_expr0 () ] in
  LaTeX_Engine.html ~dot:true ~exprs "example.html"

```

The generated “example.html” webpage is a standalone page that contains all the required scripts. Once opened in a browser, it looks like this:

For each expression, the web UI contains its rendered LaTeX form and corresponding computation graph.

Owl Engine

An Owl Engine enables converting Owl computation graph to or from a symbolic representation. Symbolic graph can thus benefit from the concise syntax and powerful features such as Algorithm Differentiation in Owl.

The conversion between Owl CGraph and the symbolic representation is straightforward, since both are graph structures. We only need to focus on making the operation projection between these two systems correct.

```

let cnode_attr = Owl_graph.attr node in
match cnode_attr.op with
| Sin -> Owl_symbolic_operator.sin ~name sym_inputs.(0)
| Sub -> Owl_symbolic_operator.sub ~name sym_inputs.(0) sym_inputs.(1)
| SubScalar -> Owl_symbolic_operator.sub ~name sym_inputs.(0) sym_inputs.(1)

```

Owl-Symbolic L^AT_EX Engine

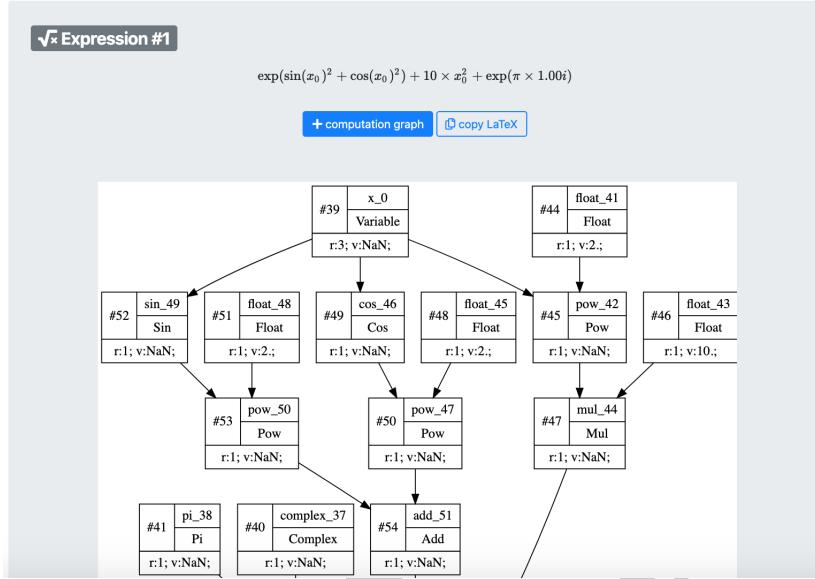


Figure 104: UI of L^AT_EX engine

```
| Conv2d (padding, strides) ->
  let pad =
    if padding = SAME then Owl_symbolic_types.SAME_UPPER else
      Owl_symbolic_types.VALID
  in
  Owl_symbolic_operator.conv ~name ~padding:pad ~strides sym_inputs.(0)
  sym_inputs.(1)
```

The basic logic is simple: find the type of symbol and its input node in CGraph, and then do the projection to symbolic representation. For most of the math operators such as sin, the projection is one-to-one, but that's not all the cases. For some operations such as subtraction, we have Sub, SubScalar and ScalarSub etc. depending on the type of input, but they can all be projected to the sub operator in symbolic representation. Or for the convolution operation, we need to first convert the parameters in suitable way before the projection.

Let's see an example of using the Owl engine:

```
open Owl_symbolic
module G = Owl_computation_cpu_engine.Make (Owl_algodiff_primal_ops.S)
module AD = Owl_algodiff_generic.Make (G)
module OWL_Engine = Owl_symbolic_engine_owl.Make (G)

let make_graph () =
```

```

let x = G.ones [| 2; 3 |] |> AD.pack_arr in
let y = G.var_elt "y" |> AD.pack_elt in
let z = AD.Maths.(sin x + y) in
let input = [| AD.unpack_elt y |> G_elt_to_node |] in
let output = [| AD.unpack_arr z |> G_arr_to_node |] in
G.make_graph ~input ~output "graph"

let g = make_graph () |> OWL_Engine.to_symbolic

```

Here we build a simple computation graph with the algorithmic differentiation module in Owl. Then we perform the conversion by calling `OWL_Engine.to_symbolic`.

We can also chain multiple engines together. For example, we can use Owl engine to converge the computation define in Owl to symbolic graph, which can then be converted to ONNX model and get executed on multiple frameworks. Here is such an example. A simple computation graph created by `make_graph()` is processed by two chained engines, and generates an ONNX model.

```

let _ =
let k = make_graph () |> OWL_Engine.to_symbolic |> ONNX_Engine.of_symbolic in
ONNX_Engine.save k "test.onnx"

```

And this `test.onnx` file can further be processed with Python code as introduced in the previous section.

Summary

To improve the performance of computation, it is necessary to utilise the power of hardware accelerators. We believe it is a growing trend that the definition and execution of computation can be separated out. Therefore, we build a symbolic representation based on Owl to facilitate exporting computations to other frameworks that supports multiple hardware accelerators. This representation can be executed by multiple backend engines. Currently it supports the ONNX, LaTeX, and Owl itself as engines. This chapter introduces the design of this symbolic representation, and uses several examples to demonstrate how the computation in Owl can be executed on other frameworks or visualised. Implementing a symbolic computation library based on Owl remains our future work.

Probabilistic Programming

TODO: <http://edwardlib.org/tutorials/>

Generative Model vs Discriminative Model

Bayesian Networks

Sampling Techniques

TODO: MCMC, Gibbs Sampling, MH ...

Inference

DRAFT

System Architecture

DRAFT

Architecture Overview

Owl is an emerging library developed in the OCaml language for scientific and engineering computing. It focuses on providing a comprehensive set of high-level numerical functions so that developers can quickly build up any data analytical applications. After over one-year intensive development and continuous optimisation, Owl has evolved into a powerful software system with competitive performance to mainstream numerical libraries. Meanwhile, Owl's overall architecture remains simple and elegant. Its small code base can be easily managed by a small group of developers.

In this chapter, we first present Owl's design, core components, and its key functionality. We show that Owl benefits greatly from OCaml's module system which not only allows us to write concise generic code with superior performance, but also leads to a very unique design to enable parallel and distributed computing. OCaml's static type checking significantly reduces potential bugs and accelerates the development cycle. We also share the knowledge and lessons learnt from building up a full-fledge system for scientific computing with the functional programming community.

Introduction

Thanks to the recent advances in machine learning and deep neural networks, there is a huge demand on various numerical tools and libraries in order to facilitate both academic researchers and industrial developers to fast prototype and test their new ideas, then develop and deploy analytical applications at large scale. Take deep neural network as an example, Google invests heavily in TensorFlow while Facebook promotes their PyTorch. Beyond these libraries focusing on one specific numerical task, the interest on general purpose tools like Python and Julia also grows fast. Python has been one popular choice among developers for fast prototyping analytical applications. One important reason is because SciPy and NumPy two libraries, tightly integrated with other advanced functionality such as plotting, offer a powerful environment which lets developers write very concise code to finish complicated numerical tasks. As a result, even for the frameworks which were not originally developed in Python (such as Caffe and TensorFlow), they often provide Python bindings to take advantage of the existing numerical infrastructure in NumPy and SciPy.

On the other hand, the supporting of basic scientific computing in OCaml is rather fragmented. There have been some initial efforts (e.g., Lacaml, Oml, Pareto, and etc.), but their APIs are either too low-level to offer satisfying productivity, or the designs overly focus on a specific problem domain. Moreover, inconsistent data representation and careless use of abstract types make it difficult to pass data across different libraries. Consequently, developers often have to write a significant amount of boilerplate code just to finish rather trivial numerical tasks. As we can see, there is a severe lack of a general purpose numerical library in OCaml ecosystem. We believe OCaml per se is a good candidate for developing

such a general purpose numerical library for two important reasons: 1) we can write functional code as concise as that in Python with type-safety; 2) OCaml code often has much superior performance comparing to dynamic languages such as Python and Julia.

However, designing and developing a full-fledged numerical library is a non-trivial task, despite that OCaml has been widely used in system programming such as MirageOS. The key difference between the two is obvious and interesting: system libraries provide a lean set of APIs to abstract complex and heterogeneous physical hardware, whilst numerical library offer a fat set of functions over a small set of abstract number types.

When Owl project started in 2016, we were immediately confronted by a series of fundamental questions like: “what should be the basic data types”, “what should be the core data structures”, “what modules should be designed”, and etc. In the following development and performance optimisation, we also tackled many research and engineering challenges on a wide range of different topics such as software engineering, language design, system and network programming, and etc.

In this chapter, We show that Owl benefits greatly from OCaml’s module system which not only allows us to write concise generic code with superior performance, but also leads to a very unique design to enable parallel and distributed computing. OCaml’s static type checking significantly reduces potential bugs and accelerate the development cycle. We would like to share the knowledge and lessons learnt from building up a full-fledge system for scientific computing with the functional programming community.

Architecture Overview

Owl is a complex library consisting of numerous functions (over 6500 by the end of 2017), we have strived for a modular design to make sure that the system is flexible enough to be extended in future. In the following, we will present its architecture briefly.

The fig. 105 gives a bird view of Owl’s system architecture, i.e. the two subsystems and their modules. The subsystem on the left part is Owl’s Numerical Subsystem. The modules contained in this subsystem fall into three categories: (1) core modules contains basic data structures and foreign function interfaces to other libraries (e.g., CBLAS and LAPACKE); (2) classic analytics contains basic mathematical and statistical functions, linear algebra, regression, optimisation, plotting, and etc.; (3) composable service includes more advanced numerical techniques such as deep neural network, natural language processing, data processing and service deployment tools.

The numerical subsystem is further organised in a stack of smaller libraries, as follows.

- **Base** is the basis of all other libraries in Owl. Base defines core data

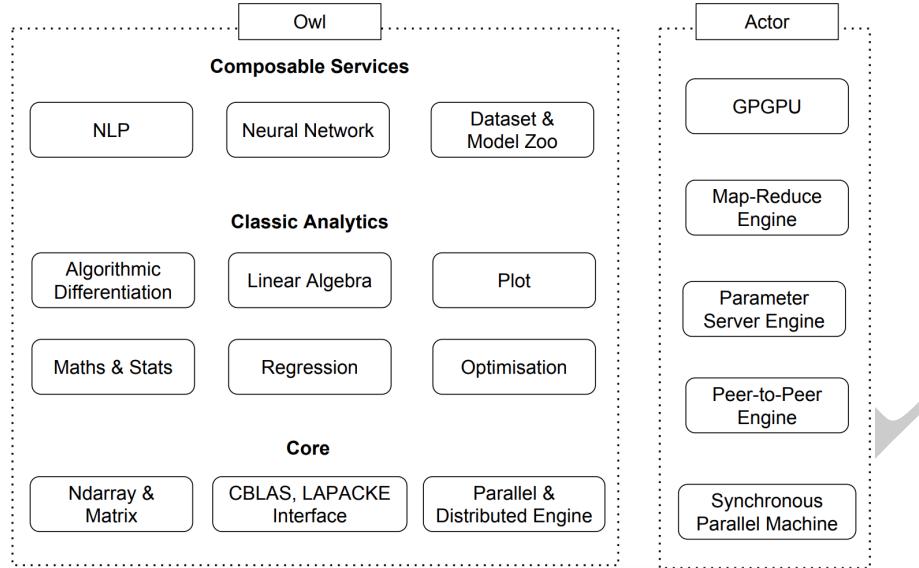


Figure 105: Owl system architecture

structures, exceptions, and part of numerical functions. Because it contains pure OCaml code so the applications built atop of Base can be safely compiled into native code, bytecode, JavaScript, even into unikernels. Fortunately, majority of Owl’s advanced functions are implemented in pure OCaml.

- **Owl** is the backbone of the numerical subsystem. It depends on Base but replaces some pure OCaml functions with C implementations (e.g. vectorised math functions in Ndarray module). Mixing C code into the library limits the choice of backends (e.g. browsers and MirageOS) but gives us significant performance improvement when running applications on CPU.
- **Zoo** is designed for packaging and sharing code snippets among users. This module targets small scripts and light numerical functions which may not be suitable for publishing on the heavier OPAM system. The code is distributed via gists on Github, and Zoo is able to resolve the dependency and automatically pull in and cache the code.
- **Top** is the Toplevel system of Owl. It automatically loads both Owl and Zoo, and installs multiple pretty printers for various data types.

The subsystem on the right is called Actor Subsystem which extends Owl’s capability to parallel and distributed computing. The addition of Actor subsystem makes Owl fundamentally different from mainstream numerical libraries such as SciPy and Julia. The core idea is to transform a user application from sequential

execution mode into parallel mode (using various computation engines) with minimal efforts. The method we used is to compose two subsystems together with functors to generate the parallel version of the module defined in the numerical subsystem.

Besides, there are other utility modules such as plotting. Plotting is an indispensable function in modern numerical libraries. We build Plot module on top of PLplot which is a powerful cross-platform plotting library. However PLPlot only provides very low-level functions to interact with its multiple plotting engines, even making a simple plot involves very lengthy and tedious control sequence. Using these low-level functions directly requires developers to understand the mechanisms in depth, which not only significantly reduces the productivity but also is prone to errors. Inspired by Matlab, we implement Plot module to provide developers a set of high-level APIs. The core plotting engine is very lightweight and only contains about 200 LOC. Its core design is to cache all the plotting operations as a sequence of function closures and execute them all when we output the figure.

Core Implementation

N-dimensional Array

N-dimensional array and matrix are the building blocks of Owl library, their functionality are implemented in Ndarray and Matrix modules respectively. Matrix is a special case of n-dimensional array, and in fact many functions in Matrix module call the corresponding functions in Ndarray directly.

For n-dimensional array and matrix, Owl supports both dense and sparse data structures. The dense data structure is built atop of OCaml's native Bigarray module hence it can be easily interfaced with other libraries like BLAS and LAPACK. Owl also supports both single and double precisions for both real and complex number. Therefore, Owl essentially has covered all the necessary number types in most common scientific computations.

- The first group contains the vectorised mathematical functions such as sin, cos, relu, and etc.
- The second group contains the high-level functionality to manipulate arrays and matrices, e.g., index, slice, tile, repeat, pad, and etc.
- The third group contains the linear algebra functions specifically for matrices. Almost all the linear algebra functions in Owl call directly the corresponding functions in CBLAS and LAPACKE.

These functions together provide a strong support for developing high-level numerical functions. Especially the first two groups turn out to be extremely useful for writing machine learning and deep neural network applications. Function polymorphism is achieved using GADT (Generalized algebraic data type),

therefore most functions in Generic module accept the input of four basic number types.

Optimisation with C Code

Interfacing to high performance language is not uncommon practice among numerical libraries. If you look at the source code of NumPy, more than 50% is C code. In SciPy, the FORTRAN and C code takes up more than 40%. Even in Julia, about 26% of its code is in C or C++, most of them in the core source code.

Besides interfacing to existing libraries, we focus on implementing the core operations in the Ndarray modules with C code. As we have seen in the N-Dimensional Arrays chapter, the n-dimensional array module lies in the heart of Owl, and many other libraries. NumPy library itself focuses solely on providing a powerful ndarray module to the Python world.

An ndarray is a container of items of the same type. It consists of a contiguous block of memory, combined with an indexing scheme that maps N integers into the location of an item in the block. A stride indexing scheme can then be applied on this block of memory to access elements. Once converted properly to the C world, a ndarray can be effectively manipulated with normal C code.

There is a big room for optimising the C code. We are trying to push the performance forward with multiple techniques. We mainly use the multiprocessing with OpenMP and parallel computing using SIMD intrinsics when possible.

Interfaced Libraries

Some functionality such as math and linear algebra is included into the system by interfacing to other libraries. Rather than simply exposing the low-level functions, we carefully design easy-to-use high-level APIs and this section will cover these modules. For example, the mathematical functions, especially the special functions, are interfaced from the Cephes Mathematical Functions Library, and the normal math functions rely on the standard C library.

Even though Fortran is no longer among the top choices as a programming language, there is still a large body of FORTRAN numerical libraries whose performance still remain competitive even by today's standard, e.g. BLAS and LAPACK. When designing the linear algebra module, we decide to interface to CBLAS and LAPACKE (i.e. the corresponding C interface of BLAS and LAPACK) then further build higher-level APIs atop of the low-level FORTRAN functions. The high-level APIs hides many tedious tasks such as setting memory layout, allocating workspace, calculating strides, and etc.

Advanced Functionality

Built on these core modules are the advanced functionalities in Owl. We have introduced many of them in the first part of this book.

Computation Graph

As a functional programmer, it is basic knowledge that a function takes an input then produces an output. The input of a function can be the output of another function which then creates dependency. If we view a function as one node in a graph, and its input and output as incoming and outgoing links respectively, as the computation continues, these functions are chained together to form a directed acyclic graph (DAG). Such a DAG is often referred to as a computation graph.

Computation graph plays a critical role in our system. Its benefits are many-fold: provides simulate lazy evaluation in a language with eager evaluation, reduce computation complexity by optimising the structure of a graph, reduce memory footprint, etc. It can be used for supporting multiple other high level modules e.g. algorithmic differentiation, and GPU computing modules all implicitly or explicitly use computation graph to perform calculations.

Algorithmic Differentiation

Atop of the core components, we have developed several modules to extend Owl's numerical capability. E.g., Maths module includes many basic and advanced mathematical functions, whilst stats module provides various statistical functions such as random number generators, hypothesis tests, and so on. The most important extended module is Algodiff, which implements the algorithmic differentiation functionality. Owl's Algodiff module is based on the core nested automatic differentiation algorithm and differentiation API of DiffSharp, which provides support for both forward and reverse differentiation and arbitrary higher order derivatives.

Algodiff module is able to provide the derivative, Jacobian, and Hessian of a large range of functions, we exploit this power to further build the optimisation engine. The optimisation engine is light and highly configurable, and also serves as the foundation of Regression module and Neural Network module because both are essentially mathematical optimisation problems. The flexibility in optimisation engine leads to an extremely compact design and small code base. For a full-fledge deep neural network module, we only use about 2500 LoC and its inference performance on CPU is superior to specialised frameworks such as TensorFlow and Caffe.

Regression

Regression is an important topic in statistical modelling and machine learning. It's about modelling problems which include one or more variables (also called "features" or "predictors") and making predictions of another variable ("output variable") based on previous data of predictors.

Regression analysis includes a wide range of models, from linear regression to isotonic regression, each with different theory background and application fields.

Explaining all these models are beyond the scope of this book. In this chapter, we focus on several common forms of regressions, mainly linear regression and logistic regression. We introduce their basic ideas, how they are supported in Owl, and how to use them to solve problems.

Neural Network

We have no intention to make yet another framework for deep neural networks. The original motivation of including such a neural network module was simply for demo purpose. It turns out that with Owl's architecture and its internal functionality (Algodiff, CGraph, etc.), combined with OCaml's powerful module system, implementing a full featured neural network module only requires approximately 3500 LoC.

Algodiff is the most powerful part of Owl and offers great benefits to the modules built atop of it. In neural network case, we only need to describe the logic of the forward pass without worrying about the backward propagation at all, because the Algodiff figures it out automatically for us thus reduces the potential errors. This explains why a full-featured neural network module only requires less than 3.5k LoC. Actually, if you are really interested, you can have a look at Owl's Feedforward Network which only uses a couple of hundreds lines of code to implement a complete Feedforward network.

Parallel Computing

Actor Engine

Parallelism can take place at various levels, e.g. on multiple cores of the same CPU, or on multiple CPUs in a network, or even on a cluster of GPUs. OCaml official release only supports single threading model at the moment, and the work on Multicore OCaml is still ongoing in the Computer Lab in Cambridge. In the following, we will present how parallelism is achieved in Owl to speed up numerical computations.

The design of distributed and parallel computing module essentially differentiates Owl from other mainstream numerical libraries. For most libraries, the capability of distributed and parallel computing is often implemented as a third-party library, and the users have to deal with low-level message passing interfaces. However, Owl achieves such capability through its Actor subsystem.

GPU Computing

Scientific computing involves intensive computations, and GPU has become an important option to accelerate these computations by performing parallel computation on its massive cores. There are two popular options in GPGPU computing: CUDA and OpenCL. CUDA is developed by Nvidia and specifically targets their own hardware platform whereas OpenCL is a cross platform solution

and supported by multiple vendors. Owl currently supports OpenCL and CUDA support is included in our future plan.

To improve performance of a numerical library such as Owl, it is necessary to support multiple hardware platforms. One idea is to “freeride” existing libraries that already support various hardware platforms. We believe that computation graph is a suitable IR to achieve interoperability between different libraries. Along this line, we develop a prototype symbolic layer system by using which the users can define a computation in Owl and then turn it into ONNX structure, which can be executed with many different platforms such as TensorFlow. By using the symbolic layer, we show the system workflow, and how powerful features of Owl, such as algorithmic differentiation, can be used in TensorFlow. We then briefly introduce system design and implementation.

OpenMP

OpenMP uses shared memory multi-threading model to provide parallel computation. It requires both compiler support and linking to specific system libraries. OpenMP support is transparent to programmers. It can be enabled by turning on the corresponding compilation switch in `dune` file. After enabling OpenMP, many vectorised math operators are replaced with the corresponding OpenMP implementation in the compiling phase. However, parallelism offered by OpenMP is not a free lunch. The scheduling mechanism adds extra overhead to a computation task. If the task per se is not computation heavy or the ndarray is small, OpenMP often slows down the computation. We therefore set a threshold on ndarray size below which OpenMP code is not triggered. This simple mechanism turns out to be very effective in practice. To further utilise the power of OpenMP, we build an automatic tuning module to decide the proper threshold value for different operations.

Community-Driven R&D

After three years of intense development, Owl currently contains about 130k lines of OCaml code and 100k lines of C code. As of March 2020, it contains about 4,200 commits and contains 29 releases. Owl has a small and concise team. These codes are mainly provided by three main developers, but so far more than 40 contributors have also participated in the project.

Owl is a large open source project, to guarantee quality of the software and sustainable development. We enforce the following rules in day-to-day research, development, and project management. Besides coding, there are many other ways you can contribute. Bug reporting, typo fix, asking/answering questions, and improvement of existing documents are all well welcome.

Coding Style

Coding style guarantees a consistent taste of code written by different people. It improves code readability and maintainability in large software projects. OCaml

is the main developing language in Owl. We use ocamlformat to enforce the style of OCaml code. There is also a significant amount of C code in the project. For the C code, we apply the Linux kernel coding style. The coding style does not apply to the vendor's code directly imported into Owl source code tree.

Unit Test

All the code must be well tested before submitting a pull request. If existing functions are modified, you need to run the unit tests to make sure the changes do not break any tests. If existing functions are modified, you may also need to add more unit tests for various edge cases which are not covered before. If new functions are added, you must add corresponding unit tests and make sure edge cases are well covered.

Pull Request

Minor improvement changes can be submitted directly in a pull request. The title and description of the pull request shall clearly describe the purpose of the PR, potential issues and caveats. For significant changes, please first submit a proposal on Owl's issue tracker to initialise the discussion with Owl Team. For sub libraries building atop of Owl, if you want the library to be included in the "owlbarn" organisation, please also submit the proposal on issue tracker. Note that the license of the library must be compliant with "owlbarn", i.e. MIT or BSD compliant. Exception is possible but must be discussed with Owl Team first. Pull requests must be reviewed and approved by at least two key developers in Owl Team. A designated person in Owl Team will be responsible for assigning reviewers, tagging a pull request, and final merging to the master branch.

Documentation

For inline documentation in the code, the following rules apply.

- Be concise, simple, and correct.
- Make sure the grammar is correct.
- Refer to the original paper whenever possible.
- Use both long documentation in mli and short inline documentation in code.

For serious technical writing, please contribute to Owl's tutorial book.

- Fixing typos, grammar issues, broken links, and improving tutorial tooling are considered as minor changes. You can submit pull requests directly to Tutorial Book repository.
- Extending existing chapters are medium changes and you need to submit a proposal to tutorial book issue tracker.
- Contributing a standalone chapter also requires submitting a chapter proposal. Alternatively, you can write to us directly to discuss about the chapter.

Summary

As the first chapter in Part II, this chapter gives a brief overview of the Owl architecture, including the core modules, the advanced functionalities, and parallel computation support. Some of these topics are covered in Part I, and we will talk about the rest in the second part of this book. This part is about Owl's internal working mechanism. Stay tuned if you are interested in how a numerical library is built, not just how it is used.

DRAFT

Core Optimisation

The study of the numerical methods is both new and old. There are always study that keeps extending numerical methods to more applications. On the other hand, we keep returning to the classical algorithms and libraries. For a high-level numerical library to achieve good performance, it is often necessary to interface its core code to classical C or Fortran code and libraries. That is true for NumPy, Julia, Matlab, and basically every other library of industrial level, and Owl is not an option. We interface part of the core operation to C code and highly optimised C libraries (such as the Lapacke from OpenBLAS). To better equip you with knowledge about how the low level is designed in Owl, in this chapter, we introduce how the core operations are implemented in C language for performance, and use some examples to show the techniques we use to optimise the C code.

TODO: update evaluations

TODO: logic is not very clear; paragraphs are fragmented.

Background

First, we briefly introduce some background information about numerical libraries and related optimisation.

Numerical Libraries

There are two widely used specifications of low level linear algebra routines. Basic Linear Algebra Subprograms (BLAS) consists of three levels of routines, from vector to matrix-vector and then to matrix-matrix operations. The other one, Linear Algebra Package (LAPACK), specifies routines for advanced numerical linear algebra, including solving systems of linear equations, linear least squares, eigenvalue problems, SVD, etc.

The implementations of these specifications vary in different libraries, e.g. OpenBLAS~[?] and Math Kernel Library (MKL). OpenBLAS is a popular open source optimised BLAS library. MKL is a proprietary library, and provides highly optimised mathematical functions on Intel processors. It implements not only BLAS and LAPACK but also FFT and other computationally intensive mathematical functions. Another implementation is Eigen, a C++ template linear algebra library. The CPU implementation of many kernels in TensorFlow uses the Eigen Tensor class. The Automatically Tuned Linear Algebra Software (ATLAS) is another BLAS implementation, featuring automatically-tuned routines on specific hardware.

These basic libraries focus on optimising the performance of operations in different hardware and software environment, but they don't provide APIs that are easy to use for end users. That requires libraries such as NumPy, Julia, Matlab, and Owl. NumPy is the fundamental package for scientific computing with Python.

It contains a powerful N-dimensional array abstraction. Julia is a high-level, high-performance dynamic programming language for numerical computing. Both are widely used and considered state of the art in numerical computing. Both NumPy and Julia rely on OpenBLAS or MKL for linear algebra backends. Matlab, the numerical computing library that has millions of uses worldwide, also belongs to this category.

Deep learning libraries such as TensorFlow, PyTorch, and MxNet are popular. Keras is a user-friendly neural networks API that can run on top of TensorFlow. Instead of the wide range of numerical functionalities that NumPy etc. provide, these libraries focus on building machine learning applications for both research and production. Owl library provides its own neural network module.

Optimisation of Numerical Computation

To achieve optimal performance has always been the target of numerical libraries. However, the complexity of current computation platforms is growing fast, and the “free” performance boost that benefits from hardware upgrade also stagnates. These factors have made it difficult to achieve the optimal performance. Below list some of the techniques that we use to optimise operations in Owl.

One method to utilise the parallelism of a computation platform is to use the Single Instruction Multiple Data (SIMD) instruction sets. They exploit data level parallelism by executing the same instruction on a set of data simultaneously, instead of repeating it multiple times on a single scalar value. One easy way to use SIMD is to rely on the automatic vectorisation capabilities of modern compilers, but in many cases developers have to manually vectorise their code with SIMD intrinsic functions. The Intel Advanced Vector Extensions (AVX) instruction set is offered on Intel and AMD processors, and the ARM processors provide the Advanced SIMD (NEON) extension.

Another form of parallelism is to execute instructions on multiple cores. OpenMP is a C/C++/FORTRAN compiler extension that allows shared memory multi-processing programming. It is widely supported on compilers such as GCC and Clang, on different hardware platforms. It is important for a numerical library to porting existing code to the OpenMP standard.

To achieve optimal performance often requires choosing the most suitable system parameters on different machines or for different inputs. Aiming at providing fast matrix multiplication routines, the ATLAS library runs a set of micro-benchmarks to decide hardware specifications, and then search for the most suitable parameters such as block size in a wide tuning space.

One general algorithm cannot always achieve optimal performance. One of the most important techniques the Julia uses is “multiple dispatch”, which means that the library provides different specific implementations according to the type of inputs.

Besides these techniques, the practical experience from others always worth

learning during development. These principles still hold true in the development of modern numerical libraries. An optimised routine can perform orders of magnitude faster than a naive implementation.

Interfacing to C Code

Despite the efficiency of OCaml, we rely on C implementation to deliver high performance for core functions. In the previous chapters in the Part I of this book, we have seen that how some of Owl modules, such as FFT and Linear Algebra, interface to existing C libraries. Optimising operations in these fields has been the classic topic of high performance computation for years, and thus there is no need to re-invent the wheels. We can directly interface to these libraries to provide good performance.

Ndarray Operations

Interfacing to high performance language is not uncommon practice among numerical libraries. If you look at the source code of NumPy, more than 50% is C code. In SciPy, the Fortran and C code takes up more than 40%. Even in Julia, about 26% of its code is in C or C++, most of them in the core source code.

Besides interfacing to existing libraries, we focus on implementing the core operations in the Ndarray modules with C code. As we have seen in the N-Dimensional Arrays chapter, the n-dimensional array module lies in the heart of Owl, and many other libraries. NumPy library itself focuses solely on providing a powerful ndarray module to the Python world.

A ndarray is a container of items of the same type. It consists of a contiguous block of memory, combined with an indexing scheme that maps N integers into the location of an item in the block. A stride indexing scheme can then be applied on this block of memory to access elements. Once converted properly to the C world, an ndarray can be effectively manipulated with normal C code.

Here we list the categories of operations that are optimised with C in Owl. Many operations are first implemented in OCaml but then updated to C driven by our practical experience and applications.

- mathematics operations, which are divided into map function, fold functions, and comparison functions.
- convolution and pooling operations, since they took up most of the computation resources in DNN-related application
- slicing, the basic operation for n-dimensional array
- matrix operations, including transpose, swapping, and check functions such as `is_hermitian`, `is_symmetric` etc.
- sorting operation
- other functions, including contraction, sliding, and repeat.

From OCaml to C

TODO: need more detail to show the layer-by-layer structure. A callgraph would be good.

Let's use examples to see exactly how we implement core operations with C and interface them to OCaml.

In Owl, ndarray is built on OCaml's native `Bigarray.Genarray`. The `Bigarray` module implements multi-dimensional numerical arrays of integers and floating-point numbers, and `Genarray` is the type of `Bigarrays` with variable numbers of dimensions.

`Genarray` is of type `('a, 'b) t`. It has three parameters: OCaml type for accessing array elements (`'a`), the actual type of array elements (`'b`), and indexing scheme (`'t`). The initial design of Owl supports both col-major and row-major indexing, but this choice leads to a lot of confusion, since the FORTRAN way of indexing starts from index 1, while the row-major starts from 0. Owl sticks with the row-major scheme now, and therefore in the core library the owl ndarray is defined as:

```
open Bigarray
type ('a, 'b) owl_arr = ('a, 'b, c_layout) Genarray.t
```

Now, let's look at the `'a` and `'b`. In the GADT type `('a, 'b) kind`, an OCaml type `'a` is for values read or written in the `Bigarray`, such as `int` or `float`, and `'b` represents the actual contents of the `Bigarray`, such as the `float32_elt` that contains 32-bit single precision floats. Owl supports four basic types of element: `float`, `double`, `float complex`, and `double complex` number. And we use the definition of type `('a, 'b) kind` in the `BigArray` module.

```
open Bigarray
type ('a, 'b) kind =
| Float32 : (float, float32_elt) kind
| Float64 : (float, float64_elt) kind
| Complex32 : (Complex.t, complex32_elt) kind
| Complex64 : (Complex.t, complex64_elt) kind
```

Suppose we want to implement the sine math function, which maps the `sin` function on every elements in the ndarray. We need to implement four different versions, each for one of these four number types. The basic code looks like this:

```
let _owl_sin : type a b. (a, b) kind -> int -> ('a, 'b) owl_arr -> ('a, 'b)
    owl_arr -> unit =
fun k l x y ->
match k with
| Float32 -> owl_float32_sin l x y
```

```
| Float64 -> owl_float64_sin l x y
| Complex32 -> owl_complex32_sin l x y
| Complex64 -> owl_complex64_sin l x y
| _ -> failwith "_owl_sin: unsupported operation"
```

The `_owl_sin` implementation takes four input parameters. The first is the number type kind, the second is the total number of elements `l` to apply the `sin` function, the third one `x` is the source ndarray, and the final one `y` is the target ndarray. This function applies the `sin` function on the first `l` elements from `x` and then put the results in `y`. Therefore we can simply add a simple layer of wrapper around this function in the `Dense` module:

```
let sin x =
  let y = copy x in
  _owl_sin (kind x) (numel y) x y;
  y
```

But wait, what are the `owl_float32_sin` and `owl_float64_sin` etc. in `_owl_sin` function? How are they implemented? Let's take a look:

```
external owl_float32_sin
: int
-> ('a, 'b) owl_arr
-> ('a, 'b) owl_arr
-> unit
= "float32_sin"
```

OCaml provides mechanism for interfacing with C using the `external` keyword: `external ocaml-function-name : type = c-function-name`. This defines the value name `ocaml-function-name` as a function with type `type` that executes by calling the given C function `c-function-name`. Here we already have a C function that is called “`float32_sin`”, and `owl_float32_sin` calls that function.

Now, finally, we venture into the world of C. We first need to include the necessary header files provided by OCaml:

```
#include <caml/mlvalues.h> // definition of the value type, and conversion macros
#include <caml/alloc.h> // allocation functions to create structured ocaml objects
#include <caml/memory.h> // miscellaneous memory-related functions and macros
#include <caml/fail.h> //functions for raising exceptions
#include <caml/callback.h> // callback from C to ocaml
#include <caml/threads.h> //operations for interfacing in the presence of multiple
threads
```

In the C file, the outlines of function `float32_sin` is:

```
CAMLprim value float32_sin(value vN, value vX, value vY) {
    ...
}
```

To define a C primitive to interface with OCaml, we use the `CAMLprim` macro. Unlike normal C functions, all the input types and output type are defined as `value` instead of `int`, `void` etc. It represents OCaml values and encodes objects of several base types such as integers, strings, floats, etc. as well as OCaml data structures. The specific type of input will be passed in when the function is called at runtime.

Now let's look at the content within this function. First, the input is of type `value` and we have to change them into the normal types for further processing.

```
CAMLparam3(vN, vX, vY);
int N = Long_val(vN);
struct caml_ba_array *X = Caml_ba_array_val(vX);
float *X_data = (float*) X->data;
struct caml_ba_array *Y = Caml_ba_array_val(vY);
float *Y_data = (float *) Y->data;
```

These “`value`” type parameters or local variables must be processed with one of the `CAMLparam` macros. Here we use the `CAMLparam3` macro since there are three parameters. There are six `CAMLparam` macros from `CAMLparam0` to `CAMLparam5`, taking zero to five parameters. For more than five parameters, you can first call `CAMLparam5`, and then use one or more `CAMLxparam1` to `CAMLxparam5` functions after that.

The next step we convert the `value` type inputs into normal types. The `Long_val` macro convert the `value` into `long int` type. Similarly, there are also `Double_val`, `Int32_val` etc. We convert the Bigarray to a structure to the structure of type `caml_ba_array`. The function `Caml_ba_array_val` returns a pointer to this structure. Its member `data` is a pointer to the data part of the array. Besides, the information of ndarray dimension is also included. The member `num_dims` of `caml_ba_array` is the number of dimensions, and `dim[i]` is the *i*-th dimension.

One more thing to do before the “real” coding. If the computation is complex, we don't want all the OCaml threads to be stuck. Therefore, we need to call the `caml_release_runtime_system` function to release the master lock and other OCaml resources, so as to allow other threads to run.

```
caml_release_runtime_system();
```

Finally, we can do the real computation, and now that we have finished converting the input data to the familiar types, the code itself is straight forward;

```

float *start_x, *stop_x;
float *start_y;

start_x = X_data;
stop_x = start_x + N;
start_y = Y_data;

while (start_x != stop_x) {
    float x = *start_x;
    *start_y = sinf(x);
    start_x += 1;
    start_y += 1;
};

```

That's all, we move the pointers forward and apply the `sinf` function from the C standard library one by one.

As you can expect, when all the computation is finished, we need to end the multiple threading.

```
caml_acquire_runtime_system();
```

And finally, we need to return the result with `CAMLreturn` macro – not normal type, but the `value` type. In this function we don't need to return anything, so we use the `Val_unit` macro:

```
CAMLreturn(Val_unit);
```

That's all for this function. But if we want to return a, say, long int, you can the use `Val_long` to wrap an int type into `value` type. In the Owl core C code, we normally finish the all the computation and copy the result in-place, and then returns `Val_unit`, as shown in this example.

Now that we finish `float32_sin`, we can copy basically all the code above and implement the rest three functions: `float64_sin`, `complex32_sin`, and `complex64_sin`. However, this kind of coding practice is apparently not ideal. Instead, in the core implementation, Owl utilises the macros and templates of C. In the above implementation, we abstract out the only three special part: function name, math function used, and data type. We assign macro `FUN` to the first one, `MAPFN` to the next, and `NUMBER` to the third. Then the function is written as a template:

```

CAMLprim value FUN(value vN, value vX, value vY) {
    ...
    NUMBER *X_data = (NUMBER *) X->data;
    ...
    *start_y = (MAPFN(x));
    ...
}

```

This template is defined in the file `owl_ndarray_maths_map.h` file. In another stub C file, these macros are defined as:

```
#define FUN float32_sin
#define NUMBER float
#define MAPFN(X) (sinf(X))
#include "owl_ndarray_maths_map.h"
```

In this way, we can easily extend this template to other data types. To extend it to complex number, we can use the `_Complex float` and `_Complex double` as number type, and the `csinf` and `csin` for math function function on complex data type.

```
#define FUN4 complex32_sin
#define NUMBER _Complex float
#define MAPFN(X) (csinf(X))
#include "owl_ndarray_maths_map.h"
```

Once finished the template, we can find that, this template does not only apply to `sin`, but also the other triangular functions, and many more other similar unary math function that accept one input, such as `exp` and `log`, etc.

```
#define FUN float32_log
#define NUMBER float
#define MAPFN(X) (logf(X))
#include "owl_ndarray_maths_map.h"
```

Of course, the template can become quite complex for other types of function. But by utilising the template and macros, the core C code of Owl is much simplified. A brief recap: in the core module we are talking about three files. The first one is a ocaml file that contains functions like `_owl_sin` that interfaces to C code using `external` keyword. Then the C implementation is divided into the template file, normally as a `.h` header file, and is named as `*_impl.h`. The stub that finally utilises these templates to generate functions are put into `*_stub.c` files.

Note that if the input parameters are more than 5, then two primitives should be implemented. The first `bytecode` function takes two arguments: a pointer to a list of `value` type arguments, and an integer that indicating the number of arguments provided. The other `native` function takes its arguments directly. The syntax of using `external` should also be changed to include both functions.

```
external name : type = bytecode-C-function-name native-code-C-function-name
```

For example, in our implementation of convolution we have a pair of functions:

```
CAMLprim value FUN_NATIVE (spatial) {
  value vInput_ptr, value vKernel_ptr, value vOutput_ptr,
  value vBatches, value vInput_cols, value vInput_rows, value vIn_channel,
  value vKernel_cols, value vKernel_rows,
  value vOutput_cols, value vOutput_rows, value vOut_channel,
  value vRow_stride, value vCol_stride,
  value vPadding, value vRow_in_stride, value vCol_in_stride
} {
  ....
}

CAMLprim value FUN_BYTEx (spatial) (value * argv, int argc) {
  return FUN_NATIVE (spatial) (
    argv[0], argv[1], argv[2], argv[3], argv[4], argv[5], argv[6], argv[7],
    argv[8], argv[9], argv[10], argv[11], argv[12], argv[13], argv[14],
    argv[15], argv[16]
);
}
```

In the stub we define the function name macros:

```
#define FUN_NATIVE(dim) stub_float32_ndarray_conv ## _ ## dim ## _ ## native
#define FUN_BYTEx(dim) stub_float32_ndarray_conv ## _ ## dim ## _ ## bytecode
```

And therefore in the OCaml interfacing code we interface to C code with:

```
external owl_float32_ndarray_conv_spatial
  : ('a, 'b) owl_arr -> ('a, 'b) owl_arr -> ('a, 'b) owl_arr -> int -> int -> int
  -> int -> int -> int -> int -> int -> int -> int -> int -> int ->
  int -> unit
  = "stub_float32_ndarray_conv_spatial_bytecode"
  "stub_float32_ndarray_conv_spatial_native"
```

More details of interfacing to C code OCaml can be found in the OCaml documentation. Another approach is to use the Foreign Function Interface, as explained here.

Optimisation Techniques

There is a big room for optimising the C code. We are trying to push the performance forward with multiple techniques. We mainly use the multiprocessing with OpenMP and parallel computing using SIMD intrinsics when possible. In this section, We choose some representative operations to demonstrate our optimisation of the core ndarray operations. Besides them, we have also applied other basic C code optimisation techniques such as avoiding redundant computation in for-loop.

To show how these optimisation works, we compare performance of an operation, in different numerical libraries: Owl, NumPy, Julia, and Eigen. The purpose is

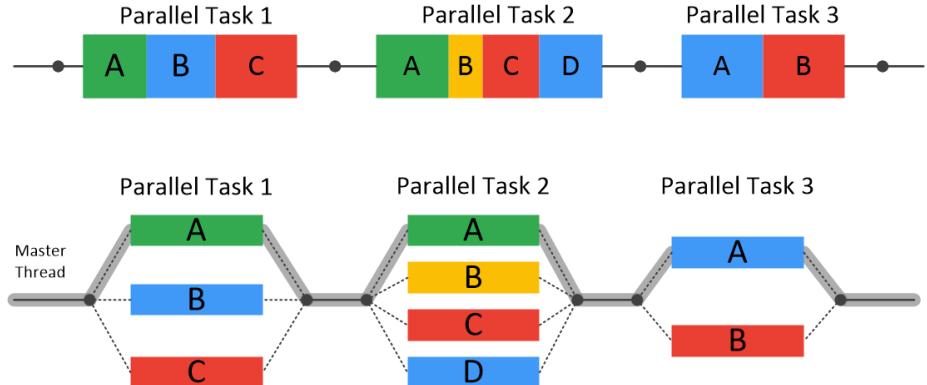


Figure 106: Fork-join model used by OpenMP

two-fold: first, to bring insight into the low-level structure design; second, to demonstrate the possible optimisations in implementing these operations.

In the performance measurements, we use multiple input sizes, and observe the execution time and memory usage. The experiments are conducted on both a laptop (Thinkpad T460s, Core i5 CPU) and a Raspberry Pi (rPi) 3B model. They represent different CPU architectures and computation power.

Map Operations

The `map` operations are a family of operations that accept `ndarray` as input, and apply a function on all the elements in the `ndarray`. Again, we use the trigonometric `sin` operation as a representative map arithmetic operation in this section. It requires heavy computation. In the implementation, it directly calls the low-level C functions via a single template. The performance of such operation is mainly decided by the linked low level library. Map function can also benefit from parallel execution on the multi-core CPU, such as using OpenMP.

OpenMP is one of the most common parallel programming models in use today. Unlike `pthread`, the low-level API to work with threads, OpenMP operate at a high-level and is much more portable. It uses a “Fork–join model” where the master thread spawns other threads as necessary, as shown in fig. 106.

In the C code we can create threads with the `omp parallel` pragma. For example, to create a four-thread parallel region, we can use:

```
omp_set_num_threads(4);
#pragma omp parallel
{
    /* parallel region */
    ...
}
```

The task in the region is assigned to the four threads and get executed in parallel. The most frequently used pattern in our core code is to move a for-loop into the parallel region. Each thread is assigned part of the whole input array, and apply the math computation on each element in parallel. Taking the implementation code from previous chapter, we only need to add a single line of OpenMP compiler directive:

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N; i++) {
    NUMBER x = *(start_x + i);
    *(start_y + i) = (MAPFN(x));
}
```

The for-loop is included in parallel region, and the N elements are scheduled to each thread. In the code we use the `static` scheduling, which means scheduling is done at compile time. It works best when the each iterations take roughly equal time. Otherwise we can consider using the “`dynamic`” scheduling that happens at runtime, or “`auto`” scheduling when the runtime can learn from previous executions of the same loop.

That’s all. We apply it simple techniques to the templates of many map function. Note that OpenMP comes with a certain overhead. What if we don’t want to use the OpenMP version?

Our solution is to provide two sets of C templates and switch depending on configuration flags. For example, for the map functions, we have the normal template file “`owl_ndarray_maths_map.h`”, and then a similar one “`owl_ndarray_maths_map_omp.h`” where each template uses the OpenMP derivative. We can then switch between these two implementation by simply define or un-define the `_OPENMP` macro, which can easily be done in the configuration file.

```
#ifdef _OPENMP
#define OWL_NDARRAY_MATHS_MAP "owl_ndarray_maths_map_omp.h"
#else
#define OWL_NDARRAY_MATHS_MAP "owl_ndarray_maths_map.h"
#endif
```

OpenMP is surely not only utilised in the map function. We also implement OpenMP-enhanced templates for the fold operations, comparison operations, slicing, and matrix swap, etc.

Another optimisation is to remove the memory copy phase by applying mutable operations. A mutable operation does not create new memory space before calculation, but instead utilise existing memory space of input ndarray. This kind of operations does not involve the C code, but rather in the ndarray module. For example:

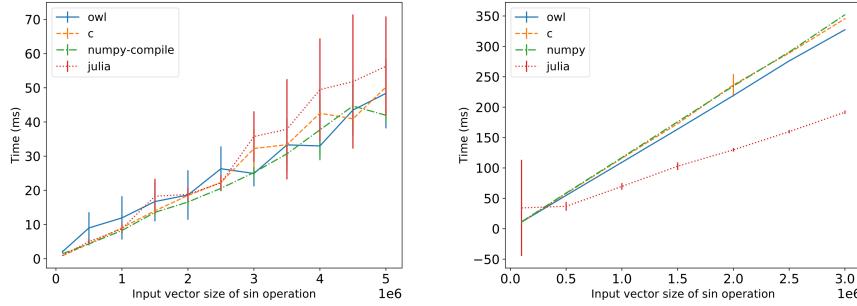


Figure 107: Sin operation performance.

```

let sin_ ?out x =
  let out =
    match out with
    | Some o -> o
    | None -> x
  in
  _owl_sin (kind x) (numel x) x out
  
```

The `_owl_sin` function is still the same, but in this mutable function `sin_` we choose the destination array and source array to be the same. Therefore, the existing memory is utilised and we don't have to copy the previous content to a new memory space before the calculation.

Both vectorisation and parallelisation techniques can be utilised to improve its performance. Computation-intensive operations such as sine in a for-loop can be vectorised using SIMD instructions. The computation performance can be boosted by executing single instruction on multiple data in the input ndarray. In that way, with only one core, 4 or 8 elements in the for-loop can be processed at the same time. However, unlike OpenMP, we cannot say "apply sine operation on these 4 elements". The SIMD intrinsics, such as the ones provided by Intel, only support basic operations such as copy, add, etc. To implement functions such as sine and exponential is non-trivial task. One simple implementation using the SSE instruction set is here. More and more libraries such as the Intel MKL provide SIMD version of these basic math operations instead of that provided in the standard C library.

Let's look at how our implementation of the `sin` operation performs compared with the other libraries. To measure performance, we compare the sine operation in Owl, NumPy, Julia, and C. The compiling flags in C and Owl are set to the same level 3 optimisation. The input is a vector of single-precision float numbers. We increase the input size from 100,000 to 5,000,000 gradually. The comparison results are shown in fig. 107.

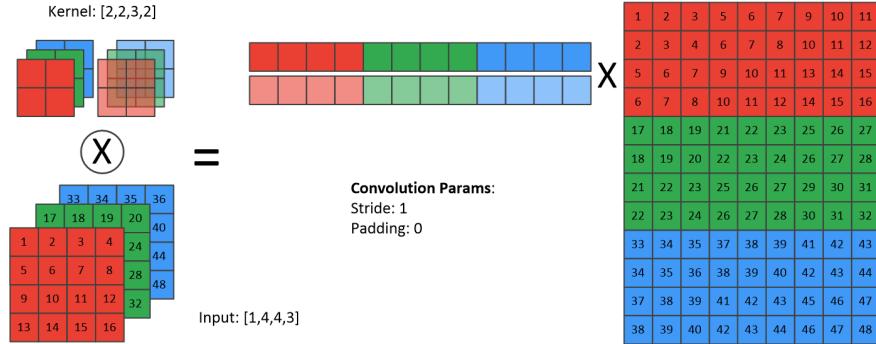


Figure 108: Basic implementation algorithm of convolution: im2col

It can be seen that the execution time in Owl grows linearly with input size, and very similar to that of C library. Julia has large deviation, but it performs fastest on rPi, even faster than C. It is because of Julia utilises NEON, the SIMD architecture extension on ARM. In some cases, NumPy can be compiled with MKL library. The MKL Vector Math functions provide highly optimised routines for trigonometric operations. In this evaluation we use NumPy library that is not compiled with MKL, and it performs close to Owl and C, with slightly larger deviation.

Convolution Operations

The convolution operations take up the majority of computation involved in deep neural network, and therefore is the main target of our core optimisation. We have seen how the convolution works and the Neural Network chapter. In this section, we would like to go a bit deeper and talk about its implementation. Starting with a short recap of how the convolution works.

A convolution operation takes two ndarrays as input: image (I) and kernel (F). In a 2-dimensional convolution, both ndarrays are of four dimensions. The image ndarray has B batches, each image has size $H \times W$, and has IC channels. The kernel ndarray has R rows, C columns, the same input channel IC , and output channel K . The convolution can then be expressed as:

$$CONV_{b,h,w,k} = \sum_{ic=1}^{IC} \sum_{r=1}^R \sum_{c=1}^C I_{b,h+r,w+c,ic} F_{r,c,ic,k}. \quad (76)$$

A naive convolution algorithm is to implement eq. 76 with nested for-loops. It is easy to see that this approach does not benefit from any parallelisation, and thus not suitable for production code.

The next version of implementation uses the `im2col` method. A `im2col`-based convolution transforms the input ndarray into a matrix with redundancy. This

process can be explained clearly with fig. 108. In this example, we start with an input image of shape 4x4, and has 3 output channels. Each channel is denoted by a different colour. Besides, the index of each element is also show in the figure. The kernel is of shape 2x2, has 3 input channels as the input image. Each channel has the same colour as the corresponding channel of input image. The 2 output channels are differentiated by various level of transparency in the figure. According to the definition of convolution operation, we use the kernel to slide over the input image step by step, and at each position, an element-wise multiplication is applied. Here in this example, we use a stride of 1, and a valid padding. In the first step, the kernel starts with the position where the element indices are [1, 2, 5, 6] in the first input channel, [17, 18, 21, 22] in the second input channel, and [33, 34, 37, 38] in the third input channel. The element-wise multiplication result is filled into corresponding position in the output ndarray. Moving on to the second position, the input indices become [2, 3, 6, 7, 18, 19, 22, 23, 34, 35, 38, 39]. So on and so forth. This process can be simplified as one matrix multiplication. The first matrix is just the flattened kernel. The second matrix is based on the input ndarray. Each column is a flattened sub-block of the same size as one channel of the kernel. This approach is the basic idea of the `im2col` algorithm. Since the matrix multiplication is a highly optimised operation in linear algebra packages such as OpenBLAS, this algorithm can be executed efficiently, and is easy to understand.

However, this algorithm requires generating a large temporary intermediate matrix. Its row number is `kernel_col * kernel_row * input_channel`, and its column number is `output_col * output_row * batches`. Even for a mediocre size convolution layer, the size of this intermediate input matrices is not small, not to mention for larger input/kernel sizes and with tens and hundreds of convolution layers together in a neural network. The memory usage can easily reach Gigabytes in DNN applications.

There are several methods proposed to mitigate this problem. If you look closely at the intermediate matrix, you will find that it contains a lot of redundant information: the columns overlap too much. Algorithms such as Memory-efficient Convolution aims to reduce the size of this intermediate matrix based on not generating the whole intermediate matrix, but only part of it to efficiently utilise the overlapped content. But even so, it may still fail with very large input or kernel sizes.

The implementation in Eigen provides another solution. Eigen a C++ template library for linear algebra. Think of it as an alternative to BLAS etc. Based on its core functionalities, it implements convolution operations as a unsupported module. Eigen is a widely used library to support high-performance convolution operations, and was used as computation backend of TensorFlow on CPU devices. The convolution operation is first implemented in Owl by interfacing to the Eigen library. We later turn to C implementation since interfacing to this C++ library proves to be problematic and leads to a lot of installation issues. In its implementation, Eigen solves this memory usage problem according to the

method proposed in (Goto and Geijn 2008).

It still generally follows the previous matrix multiplication approach, but instead of generating the whole intermediate matrix, it cuts the input and kernel matrices into small blocks one at a time so that the memory usage is limited no matter how large the input and kernel are.

Specifically, the block size can be chosen in a way to fit into the L1/L2 cache of CPU to do high-performance computation. Multiplication of two matrices can be divided into multiplication of small blocks. The L1/L2/L3 cache sizes are retrieved using the `CPUID` instruction on x86 architecture, and predefined constant value for non-x86 architectures.

To further improve the performance, we use the SIMD intrinsics during building those small temporary matrices from input ndarray. We focus on the main operation that copy input ndarray into the new input matrix: loading data, storing data, and adding two vectors. Currently we only support the most recent Advanced Vector Extensions (AVX) x86 extension on Intel and AMD architectures. We detect if the AVX extension is supported by detecting if the `__AVX__` is detected in GCC. If so, we include the header filer `immintrin.h`.

The extends commands to 256 bits, so that we can process eight float or 4 double elements at the same time. In the code we mainly use the `_mm256_store_ps`, `_mm256_load_ps`, and `_mm256_add_ps` intrinsics, for storing 256-bits variable from source into memory, loading 256-bits to memory, or adding two 256-bits into destination variable. Note that the load and store intrinsics require the source or destination address to be aligned on a 32-byte boundary. If not, we need to use the unaligned version `_mm256_storeu_ps` and `_mm256_loadu_ps`, with degraded performance.

To maximise the performance of caching, we need to make the memory access as consecutive as possible. Depending on the input channel is divisible by the supported data length of SIMD (e.g. 8 float numbers for AVX), we provide two set of implementations for filling the temporary blocks. If input channel is divisible by data length, the input matrix can always be loaded consecutively at a step of data length with the AVX intrinsics, otherwise we have to build the temporary matrix blocks with less AVX intrinsics, on only part of the matrix, and then take care of the edge cases.

We have described the implementation method we use to optimise the convolution operations. We recommend reading the full code in `owl_ndarray_conv_impl.h` file for more details. One more optimisation is that, we have shown the `im2col` method and its disadvantage with memory usage. However, it is still straightforward and fast with small input sizes. Therefore, we set a pre-defined threshold to decide if we use the `im2col` implementation or the one that inspired by Eigen.

As you know, convolution operations consists of three types: `Conv`, `ConvBackwardKernel`, `ConvBackwardInput`. The `Conv` operation calculates the output given input image and kernel. Similarly, `ConvBackwardKernel` calculates the kernel given the input and output ndarrays, and `ConvBackwardInput` gets

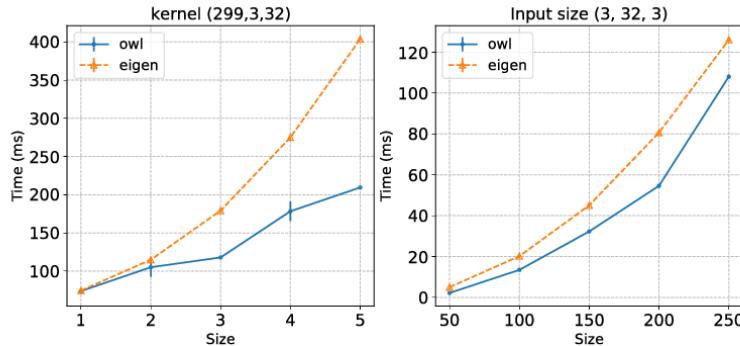


Figure 109: Compare the execution time of Conv2D operation of Owl and Eigen

input ndarray from kernel and output. The last two are mainly used in the backpropagation phase in training a DNN, but all three operations share a similar calculation algorithm. The backward convs are actually also implemented as matrix multiplication. For `ConvBackwardKernel`, it first reshape the output ndarray as matrix, and multiply it with the intermediate input matrix. Similarly, in `ConvBackwardInput`, we need to first multiply the kernel and output matrix to get the intermediate input matrix, and then re-construct the input ndarray based on it.

These implementation can then be easily extended to the three dimension and one dimension cases. Besides, the transpose convolutions and diluted convolutions are only variate of normal convolution and the code only needs to be slightly changed. At the OCaml level, mutable convolution operations are also provided, so as to further improve performance by utilising existing memory space.

To measure the performance of my convolution implementation, we compare the three convolution operations in Owl with that in the Eigen. We use two settings: fixed input size with varying kernel size; and fixed kernel size with varying input size. The Owl code is interfaced to existing implementation and Eigen library. The results shown below are performed on the single board computer Raspberry Pi 4. The results show the effectiveness of our implementation of convolution operations compared with that of the Eigen library. This good performance comes from the combination of multiple optimisation techniques as well as choosing suitable implementation according to the input.

Reduction Operations

As in the parallel programming model, the map operations are accompanied by another group: the reduction operations, or the fold operations as they are sometimes called. Reduction operations such as `sum` and `max` accumulate values in an ndarray along certain axes by certain functions. For example, a 1-dimension ndarray (vector) can be reduced to one single number along the row dimension.

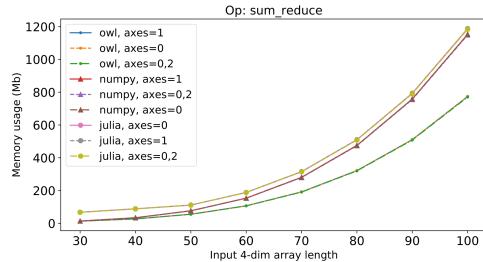


Figure 110: Sum reduction operation on laptop

The result can be the sum of all the elements if the `sum` operation is used, or the max of these elements if it is the `max` operation. The reduction operations are among the key operation that are key to high level applications. For example, `sum` is used for implementing the `BatchNormalisation` neuron, which is a frequently used neuron in DNN.

Apparently, the fold operations follow similar pattern, and that leads to the similar design choice as the map operations using templates. The implementation of the reduction operations are summarised into several patterns, which are contained in the `owl_ndarray_maths_fold.h` file as templates. In most cases these templates we only need to define the accumulation function `ACCFN`. Same with the map functions, these macros are defined in the stub file `owl_ndarray_maths_stub.c`. For example, for the sum function of float precision, I define the accumulation function as `#define ACCFN(A,X) A += X}`.

The reduction operation often needs a specified axis. One challenge we were faced with is the multi-axis reduction. A naive implementation is to repeat the operation along one axis for each axis specified, and then repeat this procedure on the next axis. However, each single-axis reduction needs extra temporary memory for storing the intermediate result. In applications that heavily utilises the reduction operation such as a DNN, the inefficiency of reduction operations becomes a memory and performance bottleneck.

In a single-axis reduction algorithm, it needs to reduce source ndarray x into a smaller destination ndarray y . Suppose the dimension to be reduced is of size a , and total number of elements in x is n . Then the basic idea is iterate their elements one by one, but the index in y keeps returning to 0 when it reaches $a/n - 1$. We revise this process so that the index in y can keep the re-iterating according to given axes, all using one single piece of intermediate memory.

One optimisation step before this algorithm is to combine adjacent axes. For example, if an ndarray of shape `[2,3,4,5]` is to be reduced along the second and third axis, then it can be simplified to reducing an ndarray of shape `[2,12,5]`.

Since it involves multiple axes, to evaluate the reduction operation, we use a four-dimensional ndarray of float numbers as input. All four dimensions are of

the same length. We measure the peak memory usage with increasing length, each for axis equals to 0, 1, and both 0 and 2 dimension. The evaluation result compared with NumPy and Julia is shown in fig. 110.

Repeat Operations

The `repeat` operation repeats elements of an ndarray along each axis for specified times. For example, a vector of shape `[2, 3]` can be expanded to shape `[4, 3]` if repeated along the first axis, or `[2, 6]` along the second axis. It consists of inner repeat and outer repeat (or tile). The former repeats elements of an input ndarray, while the later constructs an ndarray by repeating the whole input ndarray by specified number of times along each axis.

`Repeat` is another operation that is frequently used in DNN, especially for implementing the `upsampling` and `BatchNormalisation` neurons. While a reduction operation “shrinks” the input ndarray, a repeat operations expands it. Both operation require memory management instead of complex computation. Each repeat along one axis require creating extra memory space for intermediate result. Therefore, similar to the reduction functions, to perform multi-axis repeat simply using existing operations multiple times leads to memory bottleneck for the whole application.

To this end, I implement the multi-axis repeat operation in Owl. The optimisation I use in the algorithm follows two patterns. The first is to provide multiple implementations for different inputs. For example, if only one axis is used or only the highest dimension is repeated, a specific implementation for that case would be much faster than a general solution. The second is to reduce creating intermediate memory. A repeat algorithm is like a reverse of reduction: it needs expand the source ndarray `x` into a larger destination ndarray `y`. Using the elements to be repeated as a block, the repeat operation copies elements from `x` to `y` block by block. The index in both ndarrays move by a step of block size, though at different cycles. In the revised implementation, the intermediate memory is only created once and the all the iteration cycles along different axes are finished within the same piece of memory.

Compared to this implementation, the multi-axis repeat operation in NumPy is achieved by running multiple single-axis repeat, and thus is less efficient in both memory usage and execution time. The repeat operation in Julia is much slower. One reason is that this operation is implemented in pure Julia rather than the efficient C code. Another reason is that `repeat` is not a computation-intensive operation, so the optimisation techniques such as static compilation and vectorisation are of less importance than algorithm design.

The evaluation of `repeat` is similar to that of reduction operations. We use a four-dimensional ndarray of float numbers as input. All four dimensions are of the same length. We measure the speed for increasing length, the repetition times is set to 2 on all dimensions.

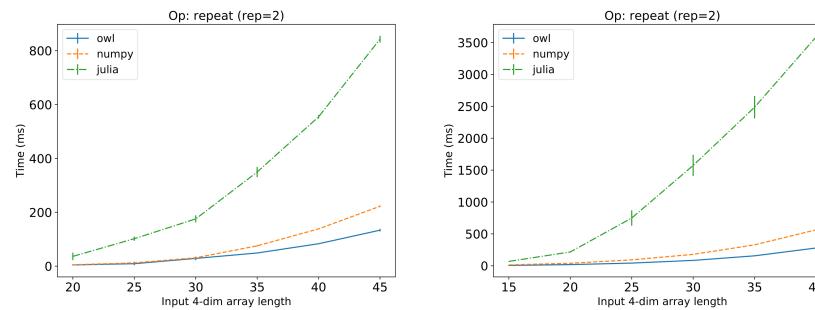


Figure 111: Repeat operation speed

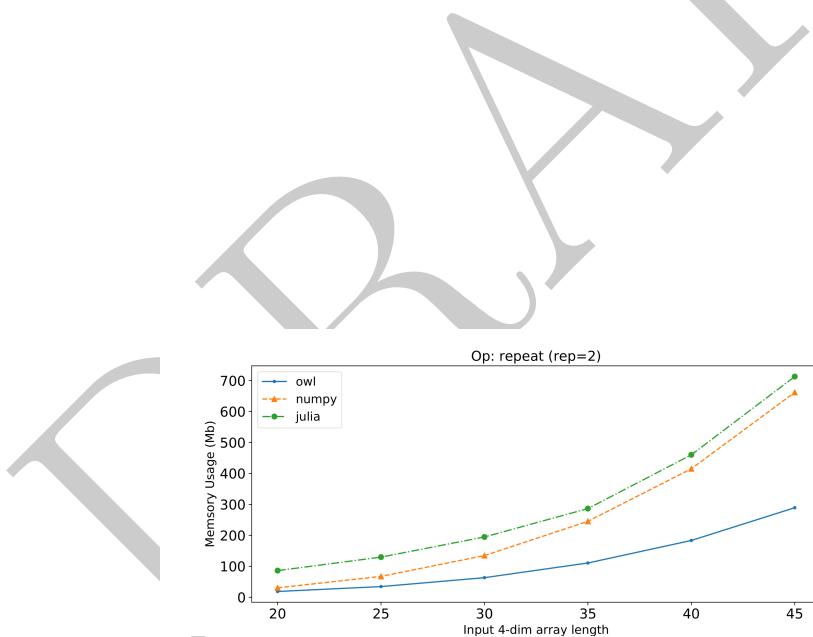


Figure 112: Repeat operation memory usage comparison

The evaluation results compared with NumPy and Julia are shown in fig. 111. We also measure the peak memory usage in fig. 112. As can be seen, my repeat operation achieves about half of that in NumPy with regard to both execution speed and memory usage. The outer repeat operation in NumPy is implemented using the single axis version, and thus is less efficient. The repeat operation in Julia is much slower. One reason is that `repeat` is not a computation-intensive operation, so the optimisation techniques such as static compilation and vectorisation are of less importance than algorithm design.

Summary

References

- Goto, Kazushige, and Robert A Geijn. 2008. “Anatomy of High-Performance Matrix Multiplication.” *ACM Transactions on Mathematical Software (TOMS)* 34 (3): 12.

Automatic Empirical Tuning

The behaviours of a software system are controlled by many of its parameters. These parameters can significantly impact the performance of the software. Assigning optimal values to the parameters to achieve the best performance is one core task for software optimisation. This chapter reveals the technical details of the parameter tuning module in Owl.

What is Parameter Tuning

Recent research work on parameter tuning mostly focus on hyper-parameter tuning, such as optimising the parameters of stochastic gradient in machine learning applications. Similarly, tuning code and parameters in low-level numerical libraries is equally important. For example, Automatically Tuned Linear Algebra Software (ATLAS) and the recent Intel Math Kernel Library (MKL) are both software libraries of optimised math routines for scientific and engineering computation. They are widely used in many popular high-level platforms such as Matlab and TensorFlow. One of the reasons these libraries can provide optimal performance is that they adopt the paradigm of *Automated Empirical Optimisation of Software* (AEOS). A technique that chooses the best method and parameters to use on a given platform to perform a required operation. One highly optimised routine may run much faster than a naively coded one. Optimised code is usually platform- and hardware-specific, but an optimised routine on one machine could perform badly on the other. Though Owl currently does not plan to improve the low-level libraries it depends on, as an initial endeavour to apply the AEOS paradigm in Owl, one ideal tuning point is the parameters of OpenMP used in Owl.

Why Parameter Tuning in Owl

Currently many computers contain shared memory multiprocessors. OpenMP is used in key operations in libraries such as Eigen and MKL. Owl has also utilised OpenMP in many mathematical operations to boost their performance by multi-threading calculation. For example, the figure below shows that when we apply the sine function on an ndarray in Owl using a 4-core CPU MacBook, the OpenMP version only takes about a third of the execution time compared with the non-OpenMP version.

However, performance improvement does not come for free. Overhead of using OpenMP comes from time spent on scheduling chunks of work to each thread, managing locks on critical sections, and startup time of creating threads, etc. Therefore, when the input ndarray is small enough, these overheads might overtake the benefit of threading. What is a suitable input size to use OpenMP then? This question would be easy to solve if there is one single suitable input size threshold for every operation, but that's not the case.

In a small experiment, I compare the performance of two operations: `abs` (calculate

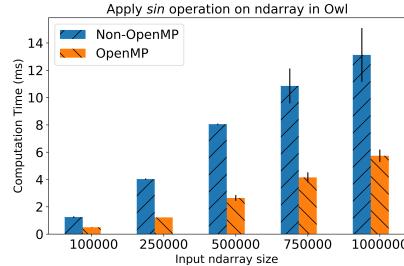


Figure 113: Compare performance of sin operations

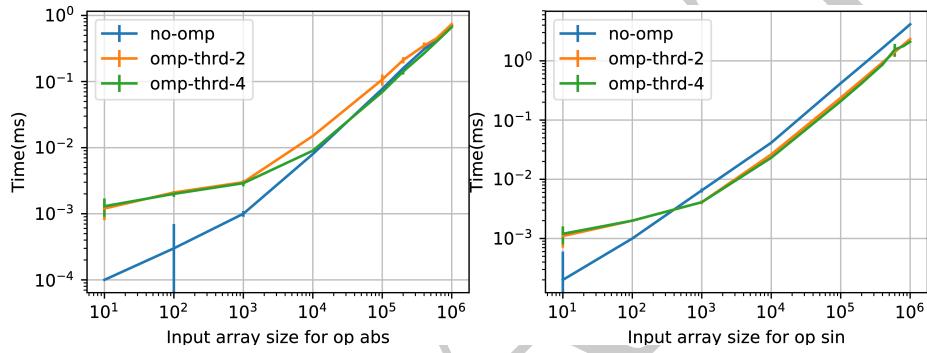


Figure 114: Observe the cross-points of OpenMP and non-OpenMP operation

absolute value) and `sin`, in three cases, including running them without using OpenMP, with 2 threads OpenMP, and with 4 threads OpenMP.

The result above shows that, with growing input size, for sine operation, the OpenMP version outperforms the non-OpenMP version at a size of less than 1000, but for `abs` operation, that cross point is at about 1,000,000. The complexity of math operations varies greatly, and the difference is even starker when we compare their performance on different machines. Note that both axes use log-scale, and that is why a small deviation when the input array size is small looks large in the figure. This issue becomes more complex when considered in real applications such as deep neural networks, where one needs to deal with operations of vastly different complexity and input sizes. Thus one fixed threshold for several operations is not an ideal solution. Considering these factors, I need a fine-grained method to decide a suitable OpenMP threshold for each operation.

How to Tune OpenMP Parameters

Towards this end, we implement the `AEOS` module in Owl. The idea is to add a tuning phase before compiling and installing Owl, so that each operation

learns a suitable threshold parameter to decide if OpenMP should be used or not, depending on input size. The key idea of parameter tuning is simple. We implement two versions of each operation, one using OpenMP and the other not. We then measure their execution time for various sizes of input. Each measurement is repeated multiple times, and to reduce the effect of outliers, only the values that are within first and third percentiles are used. After removing outliers, regression method is performed to find a suitable input size threshold. According to our initial experiment, linear regression is fit to estimate the OpenMP parameters here.

Since this tuning phase is executed before compiling Owl, the AEOS module is independent of Owl, and all necessary implementation is coded separately to ensure that future changes of Owl do not affect the AEOS module itself. The tuned parameters then need to be passed to Owl. When the OpenMP switch is turned on, the AEOS module generates a C header file which contains the definition of macros, each of which defines a threshold for one operation. When this header file is not generated, pre-defined default macro values are used instead. After that, Owl is compiled with this header file and uses these tuned parameters in its math operations. The tuning phase only needs to be performed once on each machine during installation.

The design of the AEOS module focuses on keeping tuning simple, effective, and flexible. Each operation is implemented as a single OCaml module, so that support for new operations can be easily added. The interface of such a module is shown as below.

```
module Sin : sig
  type t = {
    mutable name : string;
    mutable param : string;
    mutable value : int;
    mutable input : int array array;
    mutable y : float array
  }
  (* Tuner type definition. *)
  val make : unit -> t
  (* Create the tuner. *)
  val tune : t -> unit
  (* Tuning process. *)
  val save_data : t -> unit
  (* Save tuned data to csv file for later analysis. *)
  val to_string : t -> string
  (* Convert the tuned parameter(s) to string to be written on file *)
end
```

We expect that tuning does not have to be only about OpenMP parameters, and that different regression methods could be used in the future. For example, the Theil–Sen estimator, a robust method to fit a line to sample points, can be plugged in for parameter estimation if necessary. In each module, arbitrary tuning procedures can be plugged in as long as the interface is followed. The

AEOS module is implemented in such a way that brings little interference to the main Owl library. You only need to switch the `ENABLE_OPENMP` flag from 0 to 1 in the dune file to utilise this feature.

Make a Difference

To evaluate the performance of tuned OpenMP thresholds, we need a metric to compare them. One metric to compare two thresholds is proposed as below. First we generate a series of ndarrays, whose sizes grow by certain steps until they reach a given maximum number, e.g. 1,000,000. Note that only input sizes that fall between these two thresholds are chosen to be used. We can then calculate the performance improvement ratio of the OpenMP version function over the non-OpenMP version on these chosen ndarrays. The ratios are added up, and then amortised by the total number of ndarrays.

The table below presents the tuned threshold values of five operations on a MacBook with a 1.1GHz Intel Core m3 CPU and a Raspberry Pi 3B. We can see that they vary across different operations and different machines, depending on their computation complexity. For example, on MacBook, the tuning result is “max int”, which means that for the relatively simple square root calculation OpenMP should not be used, but that’s not the case on Raspberry Pi. Also, note that the less powerful Raspberry Pi tends to get lower thresholds.

Table 27: Tuned results using AEOS on different platforms

Platform	<i>tan</i>	<i>sqrt</i>	<i>sin</i>	<i>exp</i>	<i>sigmoid</i>
MacBook	1632	max_int	1294	123	1880
Raspberry Pi	1189	209	41	0	0

We then evaluate the performance improvement after applying AEOS. We compare each generated parameter with 30 random generated thresholds. These measured average ratios are then presented as a box plot, as shown in the figure below.

It can be observed that in general more than 20% average performance improvement can be expected on the MacBook. The result on Raspberry Pi shows a larger deviation but also a higher performance gain (about 30% on average). One reason of this difference could be that a suitable threshold on Raspberry Pi tends to be smaller, leading to a larger probability to outperform a randomly generated value. Note that we cannot proclaim that the tuned parameters are always optimal, since the figure shows that in some rare cases where the improvement percentages are minus, the randomly found values indeed perform better. Also, the result seems to suggest that AEOS can provide a certain bound, albeit a loose one, on the performance improvement, regardless of the type of operation. These interesting issues require further investigation.

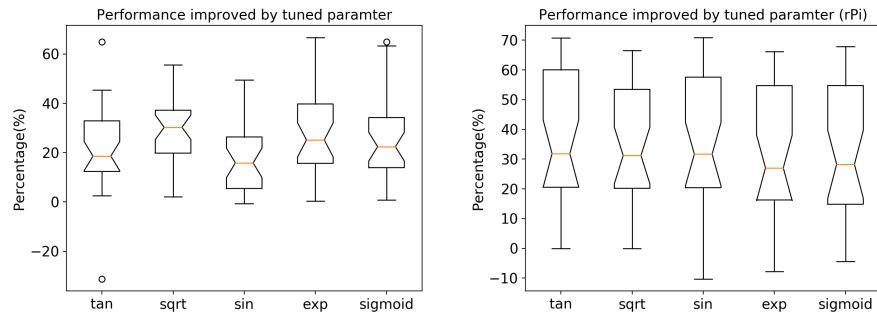


Figure 115: Evaluation of the performance improvement of AEOS

Summary

This chapter introduces the idea of automatic tuning in numerical libraries, and the usage and implementation of the tuning module in Owl. Using the OpenMP threshold for operations, we show how this module can be used to automatically improve the computation performance on different devices.

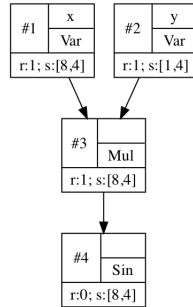


Figure 116: Computation graph of a simple function: $\sin(x * y)$

Computation Graph

This chapter first gives a bird's-eye-view on the computation graph in Owl. Then we will continue to cover the design and implementation details of the computation graph and how it is fitted into Owl's functor stack, and its implications on the architecture of numerical systems.

Introduction

What is a Computation Graph?

As a functional programmer, it is basic knowledge that a function takes an input then produces an output. The input of a function can be the output of another function which then creates dependency. If we view a function as one node in a graph, and its input and output as incoming and outgoing links respectively, as the computation continues, these functions are chained together to form a directed acyclic graph (DAG). Such a DAG is often referred to as a computation graph.

The fig. 116 shows an example graph for calculating function $\sin(x * y)$. The generated computation graph contains several pieces of information which are essential for debugging the applications. These information include node index, operation type, reference counter, and shapes of data. In the figure above, we can see the row vector y of shape [1; 4] is broadcast on the matrix x of shape [8; 4] in `Mul` operation.

From Dynamic to Static

The computation graph can be either implicitly constructed or explicitly declared in the code. Often, implicit construction is done by operator overloading while explicit declaration uses domain specific languages (DSL). The two methods lead

to two different kinds of computation graphs – *dynamic graph* and *static graph*, each has its own pros and cons.

A dynamic graph is constructed during the runtime. Due to operator overloading, its construction can be naturally blended with a language’s native constructs such as `if ... else ...` and `for` loops. This renders greatest flexibility and expressiveness. On the other hand, a static graph needs to be declared using a specific DSL (which has a steeper learning curve). Because the structure of a graph is already known during the compilation phase, there is a great space for optimisation. However, it is sometimes very difficult to use static graphs to express conditions and loops when using with native code together.

As we can see, the flexibility of a dynamic graph comes at the price of lower performance. Facebook’s PyTorch and Google’s TensorFlow are the typical examples of dynamic and static graph respectively. Many programmers need to make a choice between these two different types. A common practice is “using PyTorch at home and using TensorFlow in the company”, In other words, PyTorch is preferred for prototyping and TensorFlow is ideal for production use. (The TensorFlow 2.0 uses eager execution by default, which is easier for users to get start with.)

Owl does something slightly different from these two in order to get the best parts of both worlds. Owl achieves this by converting a dynamic graph into static one in the runtime. The motivation is based on a very important observation: in many cases, a computation graph is continuously re-evaluated after its construction. This is especially true for those iterative optimisation algorithms. We only update some inputs of the graph in each iteration.

If we know that the graph structure remains the same in every iteration, rather than re-constructing it all the time, we can convert it into a static graph before the iterative evaluation. This is exactly what Owl does. By so doing, the programmer can enjoy the flexibility offered by the dynamic graph construction with operator overloading, but still achieve the best performance from static graph.

Comparing to TensorFlow, the time overhead (for graph conversion and optimisation) is shifted to the runtime in Owl. You may worry about the performance: “Is it going to slow down my fancy DNN application?” The fact is, even for large and complex graphs, this Just-in-Time compilation (JIT) and optimisation are often quite fast. In this `lazy_lstm.ml` example, there are 15,105 nodes and 21,335 edges. Owl is able to compile the graph within 230ms then optimise it within 210ms. The optimised graph contains only 8,224 nodes, 14,444 edges and runs much faster. Remember that you only need to do it once before training. For smaller networks, it often just takes several milliseconds.

Technically, JIT is very straightforward to implement in Owl’s architecture. Given a deep neural network, Owl first runs both forward pass and backward pass. Because of the computation graph, the calculation becomes symbolic and we can obtain the complete computation graph to calculate the loss and gradients

of a neural network. We can then pass this static graph to the optimisation engine to optimise. The Neural Compiler functor is parameterised by a computation engine then compiles a DNN definition and training configuration into a device-dependent static graph.

Significance in Computing

Now that you know the basic ideas of computation graph, you may ask why it matters? Well, the computation graph makes many things a lot easier. Here is an incomplete list of potential benefits:

- simulate lazy evaluation in a language with eager evaluation;
- incremental computation (a.k.a Self-Adjusted Computation);
- reduce computation complexity by optimising the structure of a graph;
- reduce memory management overhead by pre-allocating the space;
- reduce memory footprint by reusing allocated memory space;
- natural support for parallel and distributed computing;
- natural support for heterogeneous computing;
- natural support for symbolic maths.

Some of the benefits are very obvious. Memory usage can certainly be optimised if the graph structure is fixed and the input shapes are known. One optimisation is reusing previously allocated memory, which is especially useful for those applications involving large ndarray calculations. In fact, this optimisation can also be performed by a compiler by tracking the reference number of allocated memory, a technique referred to as linear types.

Some may appear less obvious at the first glance. For example, we can decompose a computation graph into multiple independent subgraphs and each can be evaluated in parallel on different cores or even computers. Maintaining the graph structure also improves fault-tolerance, by providing natural support for rollback mechanisms.

The computation graph provides a way to abstract the flow of computations, therefore it is able to bridge the high-level applications and low-level machinery of various hardware devices. This is why we say it has natural support for heterogeneous computing.

The computation graph has more profound implications. Because the memory allocated for each node is mutable, Algodiff becomes more scalable when evaluating large and complex graphs. At the same time, mutable transformation is handled by Owl so programmers can still write safe functional code.

Examples

Before diving into the details of the design of the computation graph module, let's first shows some examples of using the CGraph modules and how the computation can be transformed into lazy evaluation.

Example 01: Basic CGraph

Let's start with a simple operation that adds up one ndarray and one scalar. Normally with Ndarray module what we do is:

```
module N = Dense.Ndarray.D
let x = N.ones [|2;2|]
let y = 2.
let g = N.add_scalar x y
```

Now, let's make it into a lazy evaluation calculation with CGraph:

```
module N = Owl_computation_cpu_engine.Make (Owl_algodiff_primal_ops.D)
```

The computation graph is designed as a functor stack. A CGraph module can be built based on a ndarray module, since in the end a lazy evaluation still requires specific computation at some point.

```
let x = N.var_arr ~shape:[|2;2|] "x"
let y = N.var_elt "y"
let g = N.add_scalar x y
```

Next we define two variables, the first `x` is a ndarray, and `y` is a scalar. At this stage, we only define these two as placeholders with no real data. Then we use the `add_scalar` function to get another lazy evaluated array `g`.

To get the value of the lazy expression `g`, we need to first assign real values to `x` and `y`:

```
let x_val = Dense.Ndarray.D.ones [|2;2|]
let y_val = 2.
let _ = N.assign_arr x x_val
let _ = N.assign_elt y y_val
```

The real values are the familiar dense ndarray and float number. Note the two different assignment method for ndarray and scalar. Finally, we can evaluate the ndarray `g`:

```
# N.eval_arr [|g|]
- : unit = ()
# N.unpack_arr g
- : Owl_algodiff_primal_ops.D.arr =
  C0 C1
R0 3 3
R1 3 3
```

The `eval_arr` returns nothing. To get the value, we need to use the `unpack_arr` or `unpack_elt` function.

Example 02: CGraph with AD

In real applications, we normally need to deal with CGraphs that are constructed in the Algorithmic Differentiation process. Here is an example of using the dense `ndarray` module to compute the gradients of a function:

```
include Owl_algodiff_generic.Make (Owl_algodiff_primal_ops.D)

let f x y = Maths.((x * sin (x + x) + ((pack_flt 1.) * sqrt x) / (pack_flt 7.)) *
    (relu y) |> sum')

let x = Dense.Ndarray.D.ones [|2;2|] |> pack_arr
let y = pack_elt 2.
let z = (grad (f x)) y |> unpack_elt
```

Obviously, it is extremely difficult for the users to manually construct the computation graph that computes the gradient of the function `f`. Instead, we use the computation graph as the base module to build the Algorithmic Differentiation module:

```
module G = Owl_computation_cpu_engine.Make (Owl_algodiff_primal_ops.D)
include Owl_algodiff_generic.Make (G)

let f x y = Maths.((x * sin (x + x) + ((pack_flt 1.) * sqrt x) / (pack_flt 7.)) *
    (relu y) |> sum')

let x = G.var_arr ~shape:[|2;2|] "x" |> pack_arr
let y = G.var_elt "y" |> pack_elt
let z = (grad (f x)) y
```

Note how the CGraph module are treated as equal to the Ndarray module in building the AD module. They decide if the AD module uses normal or lazy evaluation. Now we can evaluate `z` with the approach as before. Or we can use another approach: building a graph based on the input and output.

```
let inputs = [| unpack_arr x |> G.arr_to_node; unpack_elt y |> G_elt_to_node |]
let outputs = [| unpack_elt z |> G_elt_to_node |]
let g = G.make_graph inputs outputs "graph"
```

To build a graph, we need to specify the input and output *nodes*. Here it might be a bit confusing, since there are two layers of packing and unpacking. Currently the `x`, `y`, and `z` are both AD values of type `AD.t`, therefore we need `AD.unpack_arr` and `AD.unpack_elt` to make them CGraph lazy array and scalar values. And then, to build the explicit computation graph, we need to use the `G.arr_to_node` and

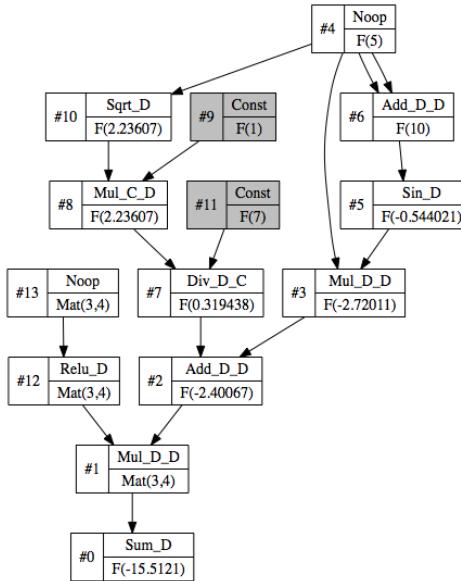


Figure 117: Computation graph of a simple math function

G_elt_to_node functions to make them into graph nodes first. Finally an explicit computation graph can be built with make_graph function.

You might be wondering why bother to build the graph if we can directly evaluate the value z. The reason is that evaluation is not always the target. For example, we often need to visualise the generated computation graph. Backward mode generates and maintains a computation graph in order to back propagate the error. The computation graph is very helpful in both debugging and understanding the characteristic of your numerical functions.

Owl provides the graph_to_dot function to facilitate you in generating computation graphs. It converts the computation graph into a dot format string. The dot file can be visualised with professional tools such as graphviz.

```

let s = G.graph_to_dot g
let _ = Owl_io.write_file "cgraph.dot" s
  
```

The generated computation graph looks like below. The Owl source code contains more examples about visualising a computation graph.

Come back to the evaluation of graph. After constructing the graph g, we can then assign real data values to the computation graph. The only difference

is that, now we need to first unpack the AD value to CGraph value before assignment:

```
let x_val = Dense.Ndarray.D.ones [|2;2|]
let y_val = 2.
let _ = G.assign_arr (unpack_arr x) x_val
let _ = G.assign_elt (unpack_elt y) y_val
```

Finally, we can evaluate the whole graph with

```
G.eval_graph g
```

Since the whole graph is evaluated, then surely the output ndarray z is also evaluated. We can first unpack it from AD value into normal CGraph ndarray and then get its value by:

```
# unpack_elt z |> G.unpack_elt
- : float = 4.20861827873129801
```

Example 03: CGraph with DNN

Since the optimisation and neural network modules are built on Algorithmic Differentiation module, they can also benefit from the power of CGraph. Suppose we have a network built of CGraph based neural network nn , we can then use the forward and backward function to get the forward inference and backward propagation computation graph from the neural network graph module, with CGraph array variable.

Actually, for ease of access, Owl has provided another functor to build the neural network module based on the CGraph module:

```
module CPU_Engine = Owl_computation_cpu_engine.Make (Owl_algodiff_primal_ops.S)
module CGCompiler = Owl_neural_compiler.Make (CPU_Engine)

open CGCompiler.Neural
open CGCompiler.Neural.Graph
open CGCompiler.Neural.Algodiff

let make_network input_shape =
  input input_shape
  |> lambda (fun x -> Maths.(x / pack_flt 256.))
  |> conv2d [|5;5;1;32|] [|1;1|] ~act_typ:Activation.Relu
  |> max_pool2d [|2;2|] [|2;2|]
  |> dropout 0.1
  |> fully_connected 1024 ~act_typ:Activation.Relu
  |> linear 10 ~act_typ:Activation.(Softmax 1)
  |> get_network ~name:"mnist"
```

The CGraph-built neural network module does not require any change of code in building the CNN except for the headers. We can then use the training function in `CGCompiler` module.

```
let pack x = CGCompiler.Engine.pack_arr x |> Algodiff.pack_arr

let train network =
  let x, _, y = Dataset.load_mnist_train_data_arr () in
  let x = pack x in
  let y = pack y in
  CGCompiler.train network x y
```

And similarly the inference can be done with `cgCompiler.model` function. You can see that to make the existing DNN programme into lazy evaluation version, all you need to do is to update the header and use packing/unpacking properly for the data.

You might be asking: the lazy evaluation version of neural network looks cool and all, but why do I need it? That brings to the large performance improvement the CGraph module can bring about to computation. To motivate you to continue to understand more about the design and optimisation of the CGraph module, you can try to run both `mnist_cnn.ml` and `lazy_mnist.ml` then compare their performance. Both Zoo scripts train the same convolution neural network to recognise the handwritten digits using MNIST datasets in 60 iterations. On a normal laptop, `mnist_cnn.ml` takes 30s to finish and consumes approximate 4GB memory, whilst `lazy_mnist.ml` only takes 5s and consumes about 0.75GB. `lazy_mnist.ml` achieves the state-of-the-art performance which you can obtain by using TensorFlow (with its recent XLA optimisation), actually Owl runs even faster on 3 out of 4 machines we have tested.

If these numbers make you interested in knowing how the magic happens, let's unveil the underlying mechanism of Owl's computation graph in the following sections.

Design Rationale

How the computation graph is designed? In the older versions, Algodiff module has some partial support of computation graph in order to perform reverse mode algorithmic differentiation (AD). The full support was only introduced in Owl 0.4.0.

Owl implements the computation graph in a very unique and interesting way. Let's first see several principles which we followed:

- Non-intrusive, the original functor stack should work as it was;
- Transparent to the programmers as much as possible;
- Support both eager and lazy evaluation;
- Flexible enough for future extension on other devices.

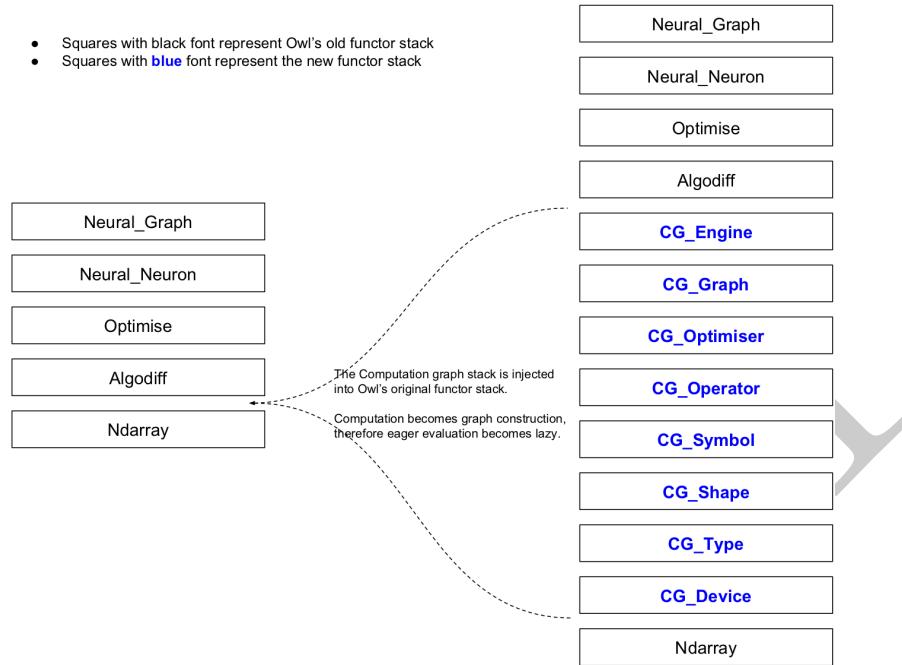


Figure 118: Computation graph functor stack in Owl

The computation graph is implemented in a very self-contained stack. I have devised a good way to “inject” it into Owl’s original functor stack. If it sounds too abstract, please have a look at the final product in the following figure.

The left figure shows part of Owl’s original functor stack, and the right one shows how the current one looks like after injection. We know the functor stack plays a central role in Owl’s architecture. In the old design, Ndarray implements a set of fundamental n-dimensional array operations, then Algodiff defines abstract mathematical operations for differentiation, finally Optimise engine glues low-level maths with high-level deep neural network applications. The whole stack is parameterised by the number type abstraction in Ndarray.

- Ndarray: provides number type abstraction and implements the fundamental numerical operations.
- Algodiff: implements algorithmic differentiation.
- Optimise: uses the derivative information to build an optimisation engine.
- Neural_Neuron: implements various kinds of neuron functions which can be optimised.
- Neural_Graph: connects neurons together to form a network so that we can train a useful model.

The functor stack of computation graph is injected between Ndarray and Algodiff. *The design principle is that the functor stack of a numerical system should be*

parameterised by both number type and device type. Number type provides data representation (real or complex, single or double, row-based or column-based layout, etc.) which decides how a maths construct should be built and operated. Device type provides hardware representation (CPU, GPU, FPGA, etc.) which decides how the computation should be performed on a specific device.

The list below summarises the functionality of each functor. The order and naming of these functors can give you a rough understanding about how it is designed.

- **Device:** device abstraction contains device-dependent types and functions.
- **Type:** type definition of various (mathematical) operations.
- **Shape:** provides the shape inference function in the graph.
- **Symbol:** provides various functions to manipulate symbols.
- **Operator:** implements maths operators (+, -, *, /, and etc.) which decide how the symbols should be connected to form a graph.
- **Optimiser:** optimises the structure of a given graph by searching and optimising various patterns.
- **Graph:** manipulates computation graphs at high level, e.g. visualisation, connecting inputs and outputs.
- **Engine:** evaluates a computation graph on a specific device.

Why the magic can happen? Simply put, the injected computation graph stack provides an abstraction layer similar to symbolic maths. The original eager evaluation becomes symbolic operation (or graph construction) therefore they can be lazily evaluated.

The shape inference functionality is able to infer the data shape of every node in a graph from its input. This allows Owl to calculate how much memory is required to evaluate the graph and pre-allocate this space. Owl can further track the reference number of each function node and reuse the allocated memory as much as possible, which reduces both memory footprint and Garbage Collector (GC) overhead, significantly improves the computation speed.

The Optimiser functor searches for various structural patterns in a graph, removes unnecessary computations and fusing computation nodes if possible. All the patterns are defined in `owl_computation_optimiser.ml`, and it is very straightforward to plug in more patterns to extend Optimiser's capability. Here are some example patterns.

Constant folding is a very basic pattern to reduce graph size. We can pre-calculate some subgraphs. For example, the inputs which node #241 depends on are all constants, so the value of #241 is already decided. We can fold all the constants to node #241 before evaluating the whole graph.

Fusing operations can effectively reduce the round trips to the memory, which saves a lot of time when operating large ndarrays. In the figure below, nodes #421, #463, and #464 are fused into one `fma` node (i.e. fused-multiply-add operation), which also improves numerical accuracy. Owl also recognises quite complicated

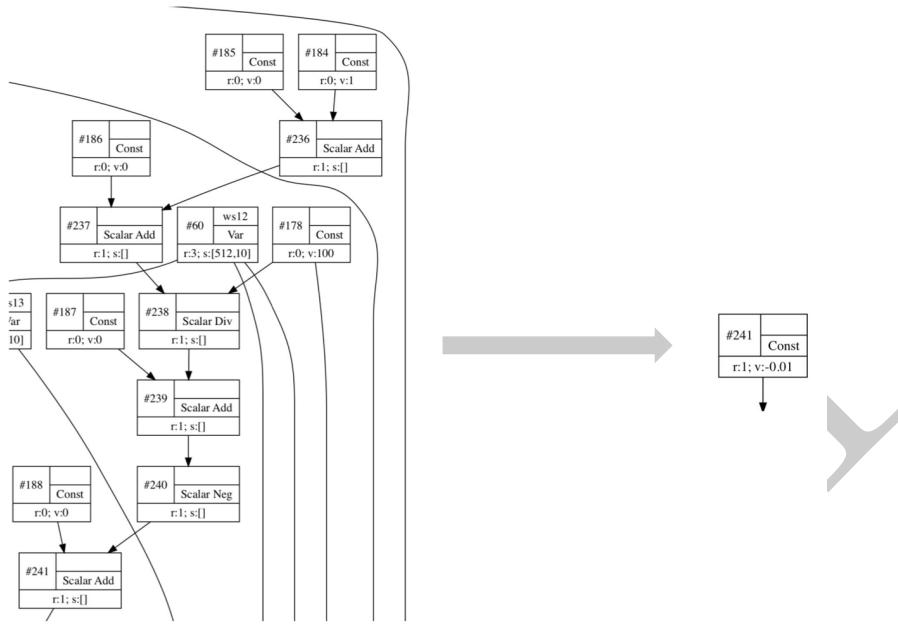


Figure 119: Optimisation techniques in computation graph: constant folding

patterns, e.g. pattern formed by nodes #511 – #515 appears a lot in DNN training that uses Adagrad (Adaptive Subgradient Methods), the Optimiser is able to fuse all these operations into one-pass calculation.

In the next example, the *Adding zero* pattern is firstly detected hence #164 and #166 are removed and others are folded. Moreover, nodes #255 for repeat operation is also removed because add operation already supports broadcasting operation. Removing #255 can save some runtime memory in the evaluation.

To understand how effective the Optimiser works, we present both the original computation graph and the optimised graph taken from `lazy_mnist.ml`. Comparing to the original network which has 201 nodes, 239 edges, the optimised one contains only 103 nodes, 140 edges.

Engine functor sits on top of the stack. This is where a computation graph finally gets executed. Engine functor contains two sub modules, one for initialising the graph and the other for evaluating graph.

Before we finish this section, we can try the following snippet in `utop`. Both snippets generate a module for DNN applications, the difference is that the first one uses the old stack whereas the second one uses the new stack with computation graph.

```
| module M =
```

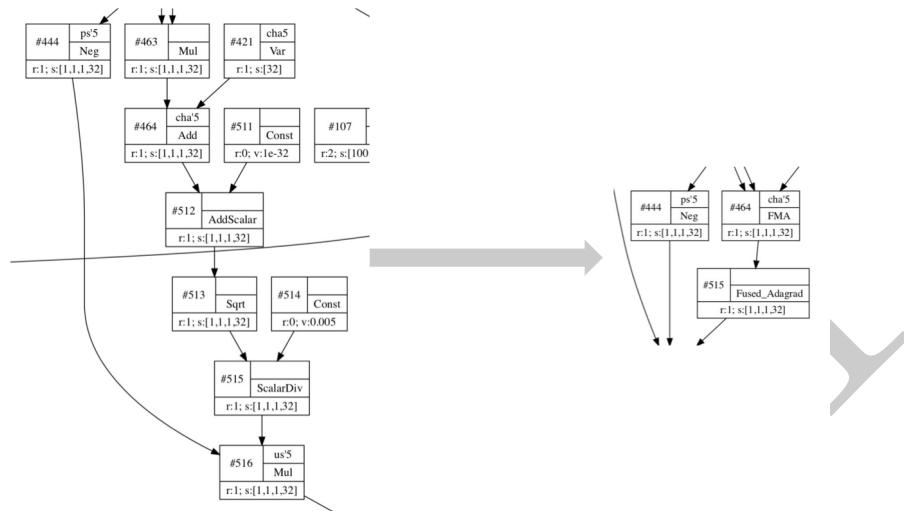


Figure 120: Optimisation techniques in computation graph: fusing operations

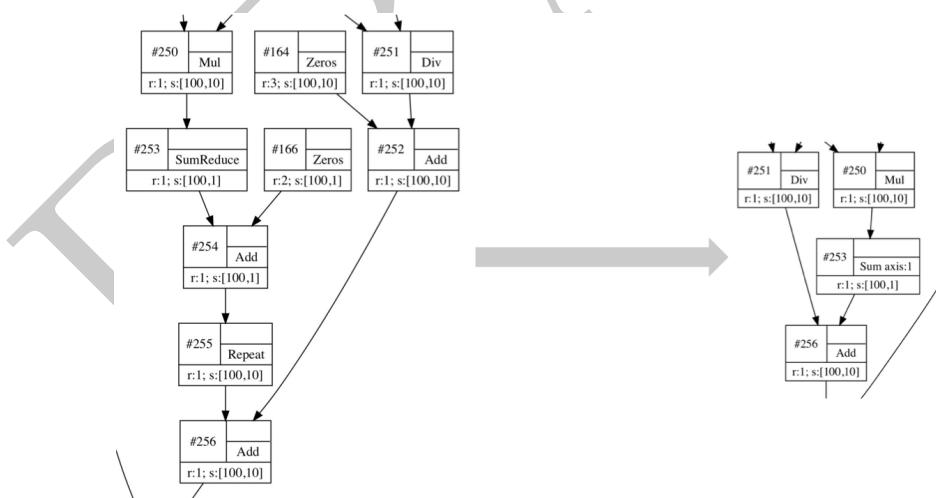


Figure 121: Optimisation techniques in computation graph: remove zero

```

Owl_neural_generic.Flatten (
  Owl_neural_graph.Make (
    Owl_neural_neuron.Make (
      Owl_optimise_generic.Make (
        Owl_algodiff_generic.Make (
          Dense.Ndarray.S)))));;

```

For the new stack, we can see it is indeed much deeper.

```

module M =
  Owl_neural_generic.Flatten (
    Owl_neural_graph.Make (
      Owl_neural_neuron.Make (
        Owl_optimise_generic.Make (
          Owl_algodiff_generic.Make (
            Owl_computation_engine.Flatten (
              Owl_computation_cpu_engine.Make_Nested (
                Owl_computation_graph.Make (
                  Owl_computation_optimiser.Make (
                    Owl_computation_operator.Make (
                      Owl_computation_symbol.Make (
                        Owl_computation_shape.Make (
                          Owl_computation_type.Make (
                            Owl_computation_cpu_device.Make (
                              Dense.Ndarray.S)))))))))))));;

```

Optimisation of CGraph

The design of Owl is often driven by real-world applications. Besides the MNIST example, we find the image segmentation another challenging application for Owl. Seeking to push the performance of this application, we manage to further optimise the design of CGraph module. This work is done by Pierre Vandenhove, and you can visit his report for more details. It starts with the MRCNN-based Object Detection application we introduce in the Case - Object Detection chapter. Please refer to this chapter for detail explanation of this application.

The first issue after constructing the network in Owl was that the memory usage, in inference mode, was huge. The network has over 400 layers and to avoid reinitialising the network for every picture, it is good to keep its input size fixed and to resize instead all the images to that size — a larger size takes more time and memory but yields more accurate results. A reasonable input size for this network is a 1024-pixel-wide square. Unfortunately, obtaining detections for one picture with this size required over 11 GB of RAM, which was too much for a laptop. As a comparison, the TensorFlow implementation only uses 1 GB. There was a big room for improvement!

This is where CGraph comes to rescue. A computation graph is always directed and acyclic. Representing the structure of a program as a computation graph has several advantages, especially for computationally-intensive code dealing with big multi-dimensional arrays. A really useful one is that prior to evaluating

the nodes, you can optimise the structure of the graph: for instance, useless calculations such as adding an array with nothing but zeros can be removed, common patterns can be merged into one node and executed more efficiently, etc. This helps a bit: thanks to these optimisations, the number of nodes of Mask R-CNN drops from 4095 to 3765. Another really important feature in this case is the ability to pre-allocate a memory space to each node, to decrease the overall memory consumption and reduce the garbage collector overhead.

Optimising memory with pebbles

To describe the problem of allocating memory in a computation graph, it is interesting to look at the *pebble game*, which was introduced in 1973 to explain register allocation.

The *pebble game* is played on a directed acyclic graph. Each node can store at most one pebble. The game begins with no pebble on any node. At each step, the player can do one of the following moves:

1. if a vertex v has no predecessor, the player can place a pebble on v .
2. if all predecessors of a vertex v are pebbled, the player can place a pebble on v or slide a pebble from one of its predecessors to v .
3. the player can remove any pebble from a vertex (and reuse that pebble later).

The goal of the game is to place a pebble at least once on some fixed output vertices of the graph.

Here is an example of an optimal pebbling strategy using the previous computation graph (gray nodes are pebbled), using moves 1 -> 2 -> 3 -> 1 -> 2 -> 2. We assume that the goal is to pebble node 5:

This relates to the memory allocation of the computation graph if we see pebbles as memory blocks used to store the output value of a node. We assume that the values of the inputs are known (move 1). We can only compute the value of a vertex if all its predecessors are simultaneously stored in memory (move 2). The *sliding* move means that the memory of a node can be overwritten by its successor during its computation (*inplace reuse*). We can always reuse a memory block from any other node (move 3). Given a graph, the idea is thus to find a strategy to pebble it using a minimum number of pebbles (in other words, using as little memory as possible).

We also want to avoid pebbling any node twice (in order to keep the execution time as low as possible, because that would mean that we compute the same node twice). Given these constraints, finding a strategy using the least amount of pebbles is unfortunately NP-complete. Since computation graphs can have a few thousand nodes, we will be looking for a fast heuristic instead of an exact algorithm.

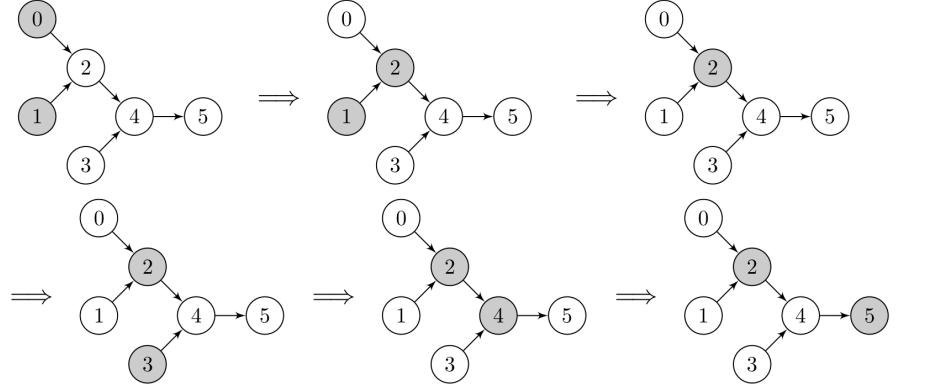


Figure 122: Modelling computation graph memory optimisation problem as a pebble game

Allocation Algorithm

The initially implemented strategy to allocate memory to a node u in Owl's computation graph module was simply to reuse the memory of a direct predecessor with same output shape as u when that is possible. This optimisation decreases the memory consumption of Mask R-CNN from 11 GB to 7 GB — much better, but still quite far from the 1 GB of the TensorFlow implementation!

We can actually make it much better by sharing memory between nodes

- that are not necessarily a parent/child pair;
- that do not have the same output size (by allocating a large block of memory once, without necessarily using all of it all the time).

To do this efficiently, we first have to fix an evaluation order (in practice, any topological order). Given this order, we can pinpoint the moment when the memory of a node becomes useless by keeping a counter of how many times it has been used. When it has been used by all its children, we can recycle its memory. Then to allocate memory to a node, we simply check which blocks are available and we select the one with the closest size (in order not to waste too much memory). If no block is available, we allocate a new one. This can be executed in $\mathcal{O}(n * \log(n))$ time, which is negligible compared to the actual cost of evaluating the graph.

Then we just have to be careful that some operations cannot overwrite their inputs while they are being computed (the *sliding* move from the pebble game is forbidden) and that some nodes cannot be overwritten for practical purposes (typically constant nodes or neural network weights). Implementing this effectively reduced the memory consumption of Mask R-CNN from 7 GB to 1 GB for a 1024x1024 picture, making it as efficient as the TensorFlow implementation!

A summary of the changes can be found in this pull request. Here are some more statistics illustrating what the computation graph with this new algorithm achieves:

Table 28: Evaluation of the effect of CGraph memory optimisation using different DNN architectures

Architecture	Time without CG (s)	Time with CG (building + evaluating) (s)	Memory without CG (MB)	Memory with CG (MB)
InceptionV3	0.565	$0.107 + 0.228 = 0.335$	625.76	230.10
ResNet50	0.793	$0.140 + 0.609 = 0.749$	1309.9	397.07
MNIST (training)	20.422	$0.144 + 10.920 = 11.064$	3685.3	895.32
Mask R-CNN	11.538	$0.363 + 8.379 = 8.742$	6483.4	870.48

InceptionV3 and ResNet50 networks are tested with a 299x299 image; Mask R-CNN is tested with a 768x768 image. The MNIST line refers to a small neural network trained to recognise hand-written digits whose implementation can be found in this code repository. The time is the average over 30 evaluations, without reusing pre-computed nodes when a computation graph is used. The graph building phase includes graph construction, optimisation and memory initialisation. The memory is the maximum resident set size of the program. This was evaluated on a laptop with an Intel i5-6300HQ and 8 GB of RAM.

For instance, when evaluated in the right order, the following computation graph, which can be used to recognise hand-written digits, needs only two different blocks of memory (each colour corresponds to a memory block, white nodes always need to be kept in memory). Part of the generated computation graph is shown in fig. 123.

You can find bigger visualisations of the allocation performed by the new algorithm in this link. You can also check this page for a demo of this Owl-powered network. If you want to apply it on videos, large images or experiment a bit more, see the GitHub repository. Pre-trained weights on 80 classes of common objects are provided, which have been converted from the TensorFlow implementation mentioned above.

As Intermediate Representations

Programming a GPU is very much like programming a computer cluster. The gain of parallel computing comes with inevitable synchronisation and communication

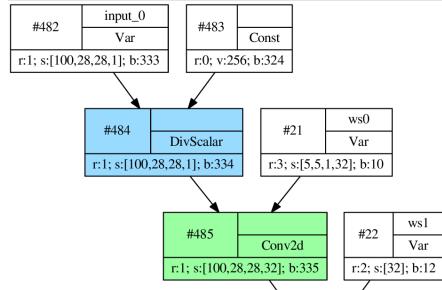


Figure 123: Optimised memory allocation

overhead. Therefore GPU computing only makes sense when the computation complexity is high enough to dwarf other overhead.

When offloading the computation to a GPU, we should avoid transmitting data back and forth between the host and the device memory, so eager evaluation is not ideal in this context because the performance will be throttled by copying. This is the gap between CPU computing and a language with eager evaluation. Computation graph essentially fills the gap between Owl and GPU computing simply because the laziness can be simulated now.

From implementation perspective, we only need to write a new engine functor for GPU devices to evaluate a graph; all the others remain the same. I am currently working on the OpenCL engine. The amount of code for implementing OpenCL engine is surprisingly small, only around 700 ~ 900 LOC. Comparing to the CPU engine, the OpenCL engine maintains the memory allocated on both host and device for each node, copying only happens whenever it is necessary, the allocated memory on the device is reused as much as possible.

Summary

In this chapter, we have introduced the core Computation Graph module in Owl. We start with the general introduction of the computation graph in numerical computing and why we build that in Owl. Then we use several examples to demonstrate how the computation graph module is used in Owl. This is followed by the internal design of this module, most importantly the CGraph stack and its position in the Owl architecture. The computation graph creates a large optimisation space, and this chapter we present one of them in detail, which is to use the pebble game to optimise the memory allocation in Owl computation.

The computation graph is a hot research topic, and there is still much we can do to improve Owl's performance based on this module. For example, the Neural Compiler still takes extra time to convert and optimise a graph. Both tasks can actually be moved into compilation phase using MetaOCaml, which will squeeze out some extra performance gain for us.

Scripting and Zoo System

In this chapter, we introduce the Zoo system, and focus on two aspects of it. The first is how to use it to make “small functions”, then distribute and share them with other users. The second is to investigate the idea of service composing and deployment based on existing script sharing function

Introduction

First, we would like to introduce the background based on which we build the Zoo system. Currently, many popular data analytics services such as machine learning applications are deployed on cloud computing infrastructures. However, they require aggregating users’ data at central server for processing. This architecture is prone to issues such as increased service response latency, communication cost, single point failure, and data privacy concerns.

Recently computation on edge and mobile devices has gained rapid growth, such as personal data analytics in home, DNN application on a tiny stick, and semantic search and recommendation on web browser. Edge computing is also boosting content distribution by supporting peering and caching. HUAWEI has identified speed and responsiveness of native AI processing on mobile devices as the key to a new era in smartphone innovation.

Many challenges arise when moving ML analytics from cloud to edge devices. One widely discussed challenge is the limited computation power and working memory of edge devices. Personalising analytics models on different edge devices is also a very interesting topic. However, one problem is not yet well defined and investigated: the deployment of data analytics services. Most existing machine learning frameworks such as TensorFlow and Caffe focus mainly on the training of analytics models. On the other, the end users, many of whom are not ML professionals, mainly use trained models to perform inference. This gap between the current ML systems and users’ requirements is growing.

Another challenge in conducting ML based data analytics on edge devices is model composition. Training a model often requires large datasets and rich computing resources, which are often not available to normal users. That’s one of the reasons that they are bounded with the models and services provided by large companies. To this end we propose the idea *Composable Service*. Its basic idea is that many services can be constructed from basic ML ones such as image recognition, speech-to-text, and recommendation to meet new application requirements. We believe that modularity and composition will be the key to increasing usage of ML-based data analytics. This idea drives us to develop the Zoo system.

Share Script with Zoo

The core functionality of the Zoo is simple: sharing OCaml scripts. It is known that we can use OCaml as a scripting language as Python (at certain performance

cost because the code is compiled into bytecode). Even though compiling into native code for production use is recommended, scripting is still useful and convenient, especially for light deployment and fast prototyping. In fact, the performance penalty in most Owl scripts is almost unnoticeable because the heaviest numerical computation part is still offloaded to Owl which runs native code.

While designing Owl, our goal is always to make the whole ecosystem open, flexible, and extensible. Programmers can make their own “small” scripts and share them with others conveniently, so they do not have to wait for such functions to be implemented in Owl’s master branch or submit something “heavy” to OPAM.

Typical Scenario

To illustrate how to use Zoo, let’s start with a simple synthetic scenario. Alice is a data analyst and uses Owl in her daily job. One day, she realised that the functions she needed had not been implemented yet in Owl. Therefore, she spent an hour in her computer and implemented these functions by herself. She thought these functions might be useful to others, e.g., her colleague Bob, she decided to share these functions using Zoo System. Now let’s see how Alice manages to do so in the following, step by step.

Create a Script

First, Alice needs to create a folder (e.g., `myscript` folder) for her shared script. What to put in the folder then? She needs at least two files in this folder. The first one is of course the file (i.e., `coolmodule.ml`) implementing the function as below. The function `sqr_magic` returns the square of a magic matrix, it is quite useless in reality but serves as an example here.

```
#!/usr/bin/env owl
open Owl
let sqr_magic n = Mat.(magic n |> sqr)
```

The second file she needs is a `#readme.md` which provides a brief description of the shared script. Note that the first line of the `#readme.md` will be used as a short description for the shared scripts. This short description will be displayed when you use `owl -list` command to list all the available Zoo code snippets on your computer.

Square of Magic Matrix

‘Coolmodule’ implements a function to generate the square of magic matrices.

Share via Gist

Second, Alice needs to distribute the files in `myscript` folder. The distribution is done via Gist, so you must have `gist` installed on your computer. E.g., if you use Mac, you can install `gist` with `brew install gist`. Owl provides a simple command line tool to upload the Zoo code snippets. Note that you need to log into your GitHub account for `gist` and `git`.

```
owl -upload myscript
```

The `owl -upload` command simply uploads all the files in `myscript` as a bundle to your Gist page. The command also prints out the URL after a successful upload. In our case, you can check the updated bundle on [this page](#).

Import in Another Script

The bundle Alice uploaded before is assigned a unique `id`, i.e. `9f0892ab2b96f81baacd7322d73a4b08`. In order to use the `sqr_magic` function, Bob only needs to use the `#zoo` directive in his script e.g. `bob.ml` in order to import the function.

```
#!/usr/bin/env owl
#zoo "9f0892ab2b96f81baacd7322d73a4b08"

let _ = Coolmodule.sqr_magic 4 |> Owl.Mat.print
```

Bob's script is very simple, but there are a couple of things worth pointing out:

- Zoo system will automatically download the bundle of a given id if it is not cached locally;
- All the `ml` files in the bundle will be imported as modules, so you need to use `Coolmodule.sqr_magic` to access the function.
- You may also want to use `chmod +x bob.ml` to make the script executable. This is obvious if you are a heavy terminal user.

Note that to use `#zoo` directive in `utop` you need to manually load the `owl-zoo` library with `#require "owl-zoo";;`. Alternatively, you can also load `owl-top` using `#require "owl-top";;` which is an OCaml toplevel wrapper of Owl. If you want to make `utop` load the library automatically by adding this line to `~/.ocamlini`.

Select a Specific Version

Alice has modified and uploaded her scripts several times. Each version of her code is assigned a unique `version id`. Different versions of code may work differently, so how could Bob specify which version to use? Good news is that, he barely needs to change his code.

```
#!/usr/bin/env owl
#zoo
"9f0892ab2b96f81baacd7322d73a4b08?vid=71261b317cd730a4dbfb0ffed02b10fcaa5948"

let _ = Coolmodule.sqr_magic 4 |> Owl.Mat.print
```

The only thing he needs to add is a version id using the parameter `vid`. The naming scheme of Zoo is designed to be similar with the field-value pair in a RESTful query. Version id can be obtained from a gist's revisions page.

Besides specifying a version, it is also quite possible that Bob prefers to use the newest version Alice provides, whatever its id may be. The problem here is that, how often does Bob need to contact the Gist server to retreat the version information? Every time he runs his code? Well, that may not be a good idea in many cases considering the communication overhead and response time. Zoo caches gists locally and tends to use the cached code and data rather than downloading them all the time.

To solve this problem, Zoo provides another parameter in the naming scheme: `tol`. It is the threshold of a gist's *tolerance* of the time it exists on the local cache. Any gist that exists on a user's local cache for longer than `tol` seconds is deemed outdated and thus requires updating the latest `vid` information from the Gist server before being used. For example:

```
#!/usr/bin/env owl
#zoo "9f0892ab2b96f81baacd7322d73a4b08?tol=300"

let _ = Coolmodule.sqr_magic 4 |> Owl.Mat.print
```

By setting the `tol` parameter to 300, Bob indicates that, if Zoo has already fetched the version information of this gist from remote server within the past 300 seconds, then keep using its local cache; otherwise contact the Gist server to check if a newer version is pushed. If so, the newest version is downloaded to local cache before being used. In the case where Bob don't want to miss every single update of Alice's gist code, he can simply set `tol` to 0, which means fetching the version information every time he executes his code.

The `vid` and `tol` parameters enable users to have fine-grained version control of Zoo gists. Of course, these two parameters should not be used together. When `vid` is set in a name, the `tol` parameter will be ignored. If both are not set, as shown in the previous code snippet, Zoo will use the latest locally cached version if it exists.

Command Line Tool

You can see that the Zoo system is not complicated at all. There will be more features to be added in future. For the time being, you can check all the available options by executing `owl`.

```
$ owl
Owl's Zoo System

Usage:
owl [utop options] [script-file] execute an Owl script
owl -upload [gist-directory] upload code snippet to gist
owl -download [gist-id] [ver-id] download code snippet from gist; download the
    latest version if ver-id not specified
owl -remove [gist-id] remove a cached gist
owl -update [gist-ids] update (all if not specified) gists
owl -run [gist-id] run a self-contained gist
owl -info [gist-ids] show the basic information of a gist
owl -list [gist-id] list all cached versions of a gist; list all the cached
    gists if gist-id not specified
owl -help print out help information
```

Note that both `run` and `info` commands accept a full gist name that can contain extra parameters, instead of only a gist id.

More Examples

Despite of its simplicity, Zoo is a very flexible and powerful tool and we have been using it heavily in our daily work. We often use Zoo to share the prototype code and small shared modules which we do not want to bother OPAM, such as those used in performance tests. Moreover, many interesting examples are also built atop of Zoo system.

- Google Inception V3 for Image Classification
- Neural Style Transfer
- Fast Neural Style Transfer

For example, you can use Zoo to perform DNN-based image classification in only 6 lines of code:

```
#!/usr/bin/env owl
#zoo "9428a62a31dbea75511882ab8218076f"

let _ =
  let image = "/path/to/your/image.png" in
  let labels = InceptionV3.infer image in
  InceptionV3.to_json ~top:5 labels
```

We include these examples as use cases in the Part III of this book, and will discuss them in detail there.

System Design

Based on these basic functionalities, we extend the Zoo system to address the composition and deployment challenges we mention at the beginning. First, we

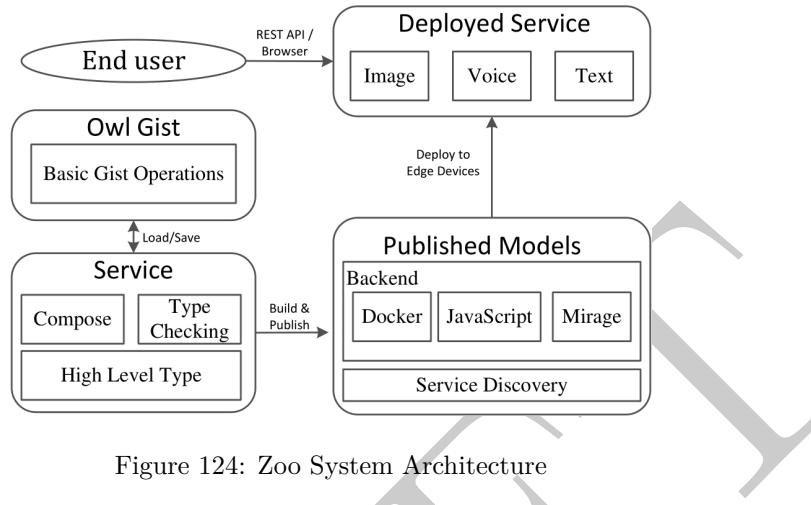


Figure 124: Zoo System Architecture

would like to briefly introduce the workflow of Zoo as shown in fig. 124.

Services

Gist is a core abstraction in Zoo. It is the centre of code sharing. However, to compose multiple analytics snippets, Gist alone is insufficient. For example, it cannot express the structure of how different pieces of code are composed together. Therefore, we introduce another abstraction: `service`.

A service consists of three parts: *Gists*, *types*, and *dependency graph*. *Gists* is the list of Gist ids this service requires. *Types* is the parameter types of this service. Any service has zero or more input parameters and one output. This design follows that of an OCaml function. *Dependency graph* is a graph structure that contains information about how the service is composed. Each node in it represents a function from a Gist, and contains the Gist's name, id, and number of parameters of this function.

Zoo provides three core operations about a service: create, compose, and publish. The `create_service` creates a dictionary of services given a Gist id. This operation reads the service configuration file from that Gist, and creates a service for each function specified in the configuration file. The `compose_service` provides a series of operations to combine multiple services into a new service. A compose operation does type checking by comparing the “types” field of two services. An error will be raised if incompatible services are composed. A composed service can be saved to a new Gist or be used for further composition. The `publish_service` makes a service’s code into such forms that can be readily used by end users. Zoo is designed to support multiple backends for these publication forms. Currently it targets Docker container, JavaScript, and MirageOS as backends.

Type Checking

One of the most important tasks of service composition is to make sure the type matches. For example, suppose there is an image analytics service that takes a PNG format image, and if we connect to it another one that produces a JPEG image, the resulting service will only generate meaningless output for data type mismatch. OCaml provides primary types such as integer, float, string, and bool. The core data structure of Owl is ndarray (or tensor as it is called in some other data analytics frameworks). However, all these types are insufficient for high level service type checking as mentioned. That motivates us to derive richer high-level types.

To support it, we use generalised algebraic data types (GADTs) in OCaml. There already exist several model collections on different platforms, e.g. Caffe and MxNet. We observe that most current popular deep learning (DL) models can generally be categorised into three fundamental types: `image`, `text`, and `voice`. Based on them, we define sub-types for each: PNG and JPEG image, French and English text and voice, i.e. `png img`, `jpeg img`, `fr text`, `en text`, `fr voice`, and `en voice` types. More can be further added easily in Zoo. Therefore type checking in OCaml ensures type-safe and meaningful composition of high level services.

Backend

Recognising the heterogeneity of edge device deployment, one key principle of Zoo is to support multiple deployment methods. Containerisation as a lightweight virtualisation technology has gained enormous traction. It is used in deployment systems such as Kubernetes. Zoo supports deploying services as Docker containers. Each container provides RESTful API for end users to query.

Another backend is JavaScript. Using JavaScript to do analytics aside from front end development begins to attract interests from academia and industry, such as Tensorflow.js and Facebook's Reason language. By exporting OCaml and Owl functions to JavaScript code, users can do complex data analytics on web browser directly without relying on any other dependencies.

Aside from these two backends, we also initially explore using MirageOS as an option. Mirage is an example of Unikernel, which builds tiny virtual machines with a specialised minimal OS that host only one target application. Deploying to Unikernel is proved to be of low memory footprint, and thus quite suitable for resource-limited edge devices.

Domain Specific Language

Based on the basic functionalities, Zoo aims to provide a minimal DSL for service composition and deployment.

Composition:

To acquire services from a Gist of id `gid`, we use `$gid` to create a dictionary, which

maps from service name strings to services. We implement the dictionary data structure using `Hashtbl` in OCaml. The `#` operator is overloaded to represent the “get item” operation. Therefore,

$$\$gid\#sname$$

can be used to get a service that is named “sname”. Now suppose we have n services: f_1, f_2, \dots, f_n . Their outputs are of type $t_{f1}, t_{f2}, \dots, t_{fn}$. Each service s accepts m_s input parameters, which have type $t_s^1, t_s^2, \dots, t_s^{m_s}$. Also, there is a service g that takes n inputs, each of them has type $t_g^1, t_g^2, \dots, t_g^n$. Its output type is t_o . Here Zoo provides the `$>` operator to compose a list of services with another:

$$[f_1, f_2, \dots, f_n] \$> g$$

This operation returns a new service that has $\sum_{s=1}^n m_s$ inputs, and is of output type t_o . This operation does type checking to make sure that $t_{fi} = t_g^i, \forall i \in 1, 2, \dots, n$.

Deployment:

Taking a service s , be it a basic or composed one, it can be deployed using the following syntax:

$$s \$@ backend$$

The `$@` operator publish services to certain backend. It returns a string of URI of the resources to be deployed.

Note that the `$>` operator leads to a tree-structure, which is in most cases sufficient for our real-world service deployment. However, a more general operation is to support graph structure. This will be our next-step work.

Service Discovery

The services require a service discovery mechanism. For simplicity’s sake, each newly published service is added to a public record hosted on a server. The record is a list of items, and each item contains the Gist id that service based on, a one-line description of this service, string representation of the input types and output type of this service, e.g. “image -> int -> string -> tex”, and service URI. For the container deployment, the URI is a DockerHub link, and for JavaScript backend, the URI is a URL link to the JavaScript file itself. The service discovery mechanism is implemented using off-the-shelf database.

Use Case

To illustrate the workflow above, let’s consider another synthetic scenario. Alice is a French data analyst. She knows how to use ML and DL models in existing platforms, but is not an expert. Her recent work is about testing the performance

of different image classification neural networks. To do that, she needs to first modify the image using the DNN-based Neural Style Transfer (NST) algorithm. The NST algorithm takes two images and outputs to a new image, which is similar to the first image in content and the second in style. This new image should be passed to an image classification DNN for inference. Finally, the classification result should be translated to French. She does not want to put academic-related information on Google's server, but she cannot find any single pre-trained model that performs this series of tasks.

Here comes the Zoo system to help. Alice finds gists that can do image recognition, NST, and translation separately. Even better, she can perform image segmentation to greatly improve the performance of NST using another Gist. All she has to provide is some simple code to generate the style images she need to use. She can then assemble these parts together easily using Zoo.

```
open Zoo
(* Image classification *)
let s_img = $ "aa36e" # "infer";;
(* Image segmentation *)
let s_seg = $ "d79e9" # "seg";;
(* Neural style transfer *)
let s_nst = $ "6f28d" # "run";;
(* Translation from English to French *)
let s_trans = $ "7f32a" # "trans";;
(* Alice's own style image generation service *)
let s_style = $ alice_Gist_id # "image_gen";;

(* Compose services *)
let s = [s_seg; s_style] $> s_nst
    $> n_img $> n_trans;;
(* Publish to a new Docker Image *)
let pub = (List.hd s) $@"
    (CONTAINER "alice/image_service:latest");;
```

Note that the Gist id used in the code is shorted from 32 digits to 5 due to column length limit. Once Alice creates the new service and published it as a container, she can then run it locally and send request with image data to the deployed machine, and get image classification results back in French.

Summary

The Zoo system was first developed as a handy tool for sharing OCaml scripts. This chapter introduces how it works with a step-by-step example. Based on its basic functionalities, we propose to use it to solve two challenges when conducting data analytics on edge: service composition and deployment. Zoo provides a simple DSL to enable easy and type-safe composition of different advanced services. We present a use case to show the expressiveness of the code. Zoo can further be combined with multiple execution backends to accommodate the heterogeneous edge deployment environment. The compiler backends will be

discussed in the next chapter. Part of the content in this chapter is adapted from our paper (Zhao et al. 2018). We refer the readers to it for more detail if you are interested.

References

- Zhao, Jianxin, Tudor Tiplea, Richard Mortier, Jon Crowcroft, and Liang Wang. 2018. “Data Analytics Service Composition and Deployment on Edge Devices.” In *Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, 27–32.

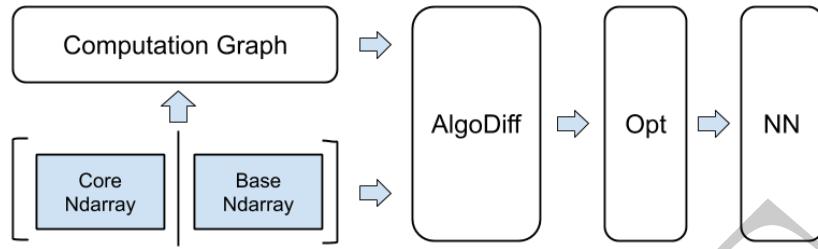


Figure 125: Core functor stack in owl

Compiler Backends

For a numerical library, it is always beneficial and a challenge to extend to multiple execution backends. We have seen how we support accelerators such as GPU by utilising symbolic representation and computation graph standard such as ONNX. In this chapter we introduce how Owl can be used on more edge-oriented backends, including JavaScript and MirageOS. We also introduce the `base` library in Owl, since this pure OCaml library is built to support these backends.

Base Library

Before we start, we need to understand how Owl enables compiling to multiple backends by providing different implementations. The Owl framework, as well as many of its external libraries, is actually divided to two parts: a Base library and a Core library. The base library is implemented with pure OCaml. For some backends such as JavaScript, we can only use the functions implemented in OCaml.

You may wonder how much we will be limited by the Base. Fortunately, the most advanced functions in Owl are often implemented in pure OCaml and they live in the Base, which includes e.g. algorithmic differentiation, optimisation, even neural networks and many others. Here is the structure of the core functor stack in Owl:

Ndarray is the core building block in Owl. As we have described in the previous chapters how we use C code to push forward the performance of Owl computation. The base library aims to implements all the necessary functions as the core library `ndarray` module. The stack is implemented in such way that the user can switch between these two different implementation without the modules of higher layer. In the Owl functor stack, `ndarray` is used to support the `CGraph` module to provide lazy evaluation functionalities.

You might be wondering: where is the `ndarray` module then? Here we use the `owl_base_algodiff_primal_ops` module, which is simply a wrapper around the base `ndarray` module. It also includes a small number of Matrix and Linear Algebra

functions. By providing this wrapper instead of using the Ndarray module directly, we can avoid mixing all the function in the ndarray module and makes it a large Goliath.

Next, the Algorithmic Differentiation can build up its computation module based on normal ndarray or its lazy version. For example, you can have an AD that relies on the normal single precision base ndarray module:

```
module AD = Owl_algodiff_generic.Make (Owl_base_algodiff_primal_ops.S)
```

Or it can be built on an double precision lazy evaluated core ndarray module:

```
module CPU_Engine = Owl_computation_cpu_engine.Make (Owl_algodiff_primal_ops.D)
module AD = Owl_algodiff_generic.Make (CPU_Engine)
```

Going even higher, we have the advanced modules Optimisation and Neural Network modules. They are both based on the AD module. For example, the code below shows how we can build a neural graph module by layers of functors from the base ndarray.

```
module G = Owl_neural_graph.Make
  (Owl_neural_neuron.Make
    (Owl_optimise_generic.Make
      (Owl_algodiff_generic.Make
        (Owl_base_algodiff_primal_ops.S))))
```

Normally the users does not have to care about how these modules are constructed layer by layer, but understanding the functor stack and typing is nevertheless beneficial, especially when you are creating new modules that relies on the base ndarray module.

These examples show that once we have built an application with the core Ndarray module, we can then seamlessly switch it to base ndarray module without changing anything else. That means that all the code and examples we have seen so far can be used directly on different backends that requires pure implementation.

The base library is still an on-going work and there is still a lot to do. Though the ndarray module is a large part in base library, there are other modules that also need to be re-implemented in OCaml, such as the Linear Algebra module. We need to add more functions such as the SVD factorisation. Even for the Ndarray itself we still cannot totally cover the core ndarray yet. Our strategy is that, we put most of the signature file in base library, and the core library signature file includes its corresponding signature file from the base library, plus functions that are currently unique to core library. The target is to total coverage so that the core and base library provide exactly the same functions.

As can be expected, the pure OCaml implementation normally performs worse than the C code implemented version. For example, for the complex convolution, without the help of optimised routines from OpenBLAS etc., we can only provide the naive implementation that includes multiple for-loops. Its performance is orders of magnitude slower than the C version. Currently our priority is to implement the functions themselves instead of caring about function optimisation, nor do we intend to out-perform C code with pure OCaml implementation.

Backend: JavaScript

At first glance, JavaScript has very little to do with high-performance scientific computing. Then why Owl cares about it? One important reason is that browser is arguably the most widely deployed technology on various edge devices, e.g. mobile phones, tablets, laptops, and etc. More functionalities are being pushed from data centers to edge for reduced latency, better privacy and security. And JavaScript applications running in a browser are getting more complicated and powerful. Moreover, JavaScript interpreters are being increasingly optimised, and even relatively complicated computational tasks can run with reasonable performance.

This chapter uses two simple examples to demonstrate how to compile Owl applications into JavaScript code so that you can deploy the analytical code into browsers, using both native OCaml code and Facebook Reason. It additionally requires the use of `dune`. As you will see, this will make the compilation to JavaScript effortless.

Use Native OCaml

We rely on the tool `js_of_ocaml` to convert native OCaml code into JavaScript. `Js_of_ocaml` is a compiler from OCaml bytecode programs to JavaScript. The process can thus be divided into two phases: first, compile the OCaml source code into bytecode executables, and then apply the `js_of_ocaml` command to it. It supports the core `Bigarray` module among most of the OCaml standard libraries. However, since the `sys` module is not fully supported, we are careful to not use functions from this module in the base library.

We have described how Algorithm Differentiation plays a core role in the ecosystem of Owl, so now we use an example of AD to demonstrate how we convert a numerical programme into JavaScript code and then get executed. The example comes from the Optimisation chapter, and is about optimise the mathematical function `sin`. The first step is writing down our application in OCaml as follows, then save it into a file `demo.ml`.

```
module AlgodiffD = Owl_algodiff_generic.Make (Owl_base_algodiff_primal_ops.D)
open AlgodiffD

let rec desc ?(eta=F 0.01) ?(eps=1e-6) f x =
```

```

let g = (diff f) x in
  if (unpack_flt g) < eps then x
  else desc ~eta ~eps f Maths.(x - eta * g)

let _ =
  let f = Maths.sin in
  let y = desc f (F 0.1) in
  Owl_log.info "argmin f(x) = %g" (unpack_flt y)

```

The code is very simple: the `desc` defines a gradient descent algorithm, and then we use `desc` to calculate the minimum value of `Maths.sin` function. In the end, we print out the result using `Owl_log` module's `info` function. Note that we pass in the base `Ndarray` module to the `AD` functor to create a corresponding `AD` module.

In the second step, we need to create a `dune` file as follows. This file will instruct how the OCaml code will be first compiled into bytecode then converted into JavaScript by calling `js_of_ocaml`.

```
(executable
  (name demo)
  (modes byte js)
  (libraries owl-base))
```

With these two files in the same folder, you can then simply run the following command in the terminal.

```
dune build demo.bc && js_of_ocaml _build/default/demo.bc
```

Or even better, since `js_of_ocaml` is natively supported by `dune`, we can simply execute:

```
dune build
```

The command builds the application and generates a `demo.bc.js` in the `_build/default/` folder. Finally, we can run the JavaScript using `Node.js` (or loading into a browser using an appropriate html page).

```
node _build/default/demo.bc.js
```

As a result, you should be able to see the output result shows a value that minimise the `sin` function, and should be similar to:

```
2019-12-30 18:05:49.760 INFO : argmin f(x) = -1.5708
```

Even though we present a simple example, you should keep in mind that the base library can be used to produce more complex and interactive browser applications.

Use Facebook Reason

Facebook Reason leverages OCaml as a backend to provide type safe JavaScript. It is gaining its momentum and becoming a popular choice of developing web applications. It actually uses another tool, BuckleScript, to convert the Reason/OCaml code to JavaScript. Since Reason is basically a syntax layer built on top of OCaml, it is very straightforward to use Owl in Reason to develop advanced numerical applications.

In this example, we use reason code to manipulate multi-dimensional arrays, the core data structure in Owl. First, we save the following code into a reason file called `demo.re`. Note the suffix is `.re` now. It includes several basic math and Ndarray operations in Owl.

```
open! Owl_base;

/* calculate math functions */
let x = Owl_base_maths.sin(5.);
Owl_log.info("Result is %f", x);

/* create random ndarray then print */
let y = Owl_base_dense_ndarray.D.uniform([|3,4,5|]);
Owl_base_dense_ndarray.D.set(y,[|1,1,1|],1.0);
Owl_base_dense_ndarray.D.print(y);

/* take a slice */
let z = Owl_base_dense_ndarray.D.get_slice([[],[],[0,3]],y);
Owl_base_dense_ndarray.D.print(z);
```

The code above is simple, just creates a random ndarray, takes a slice, and then prints them out. Owl library can be seamlessly used in Reason. Next, instead of using Reason's own translation of this frontend syntax with bucklescript, we still turn to `js_of_ocaml` for help. Let's look at the `dune` file, which turns out to be the same as that in the previous example.

```
(executable
  (name demo)
  (modes js)
  (libraries owl-base))
```

As in the previous example, you can then compile and run the code with following commands.

```
dune build
node _build/default/demo.bc.js
```

As you can see, except that the code is written in different languages, the rest of the steps are identical in both example thanks to `js_of_ocaml` and `dune`.

Backend: MirageOS

MirageOS and Unikernel

Besides JavaScript, another choice of backend we aim to support is the MirageOS. It is an approach to build *unikernels*. A unikernel is a specialised, single address space machine image constructed with library operating systems. Unlike normal virtual machine, it only contains a minimal set of libraries required for one application. It can run directly on a hypervisor or hardware without relying on operating systems such as Linux and Windows. The unikernl is thus concise and secure, and extremely efficient for distributed and executed on either cloud or edge devices.

MirageOS is one solution to building unikernels. It utilises the high-level languages OCaml and a runtime to provide API for operating system functionalities. In using MirageOS, the users can think of the Xen hypervisor as a stable hardware platform, without worrying about the hardware details such as devices. Furthermore, since the Xen hypervisor is widely used in platforms such as Amazon EC2 and Rackspace Cloud, MirageOS-built unikernel can be readily deployed on these platforms. Besides, benefiting from its efficiency and security, MirageOS also aims to form a core piece of the Nymote/MISO tool stack to power the Internet of Things.

Example: Gradient Descent

Since MirageOS is based around the OCaml language, we can safely integrate the Owl library with it. To demonstrate how we use MirageOS as backend, we again use the previous Algorithm Differentiation based optimisation example. Before we start, please make sure to follow the installation instruction. Let's look at the code:

```
module A = Owl_algodiff_generic.Make (Owl_algodiff_primal_ops.S)
open A

let rec desc ?(eta=F 0.01) ?(eps=1e-6) f x =
  let g = (diff f) x in
  if (unpack_flt (Maths.abs g)) < eps then x
  else desc ~eta ~eps f Maths.(x - eta * g)

let main () =
  let f x = Maths.(pow x (F 3.) - (F 2.) * pow x (F 2.) + (F 2.)) in
  let init = Stats.uniform_rvs ~a:0. ~b:10. in
  let y = desc f (F init) in
```

```
Owl_log.info "argmin f(x) = %g" (unpack_flt y)
```

This part of code is mostly the same as before. By applying the `diff` function of the algorithmic differentiation module, we use the gradient descent method to find the value that minimises the function $x^3 - 2x^2 + 2$. Then we need to add something different:

```
module GD = struct
  let start = main (); Lwt.return_unit
end
```

Here the `start` is an entry point to the unikernel. It performs the normal OCaml function `main`, and the return a `Lwt` thread that will be evaluated to `unit`. `Lwt` is a concurrent programming library in OCaml. It provides the “promise” data type that can be determined in the future. Please refer to its document for more information if you are interested.

All the code above is written to a file called `gd_owl.ml`. To build a unikernel, next we need to define its configuration. In the same directory, we create a file called `configure.ml`:

```
open Mirage

let main =
  foreign
    ~packages:[package "owl"]
    "Gd_owl.GD" job

let () =
  register "gd_owl" [main]
```

It's not complex. First we need to open the `Mirage` module. Then we declare a value `main` (or you can name it any other name). It calls the `foreign` function to specify the configuration. First, in the `package` parameter, we declare that this unikernel requires Owl library. The next string parameter “`Gd_owl.GD`” specifies the name of the implementation file, and in that file the module `GD` that contains the `start` entry point. The third parameter `job` declares the type of devices required by a unikernel, such as network interfaces, network stacks, file systems, etc. Since here we only do the calculation, there is no extra device required, so the third parameter is a `job`. Finally, we register the unikernel entry file `gd_owl` with the `main` configuration value.

That's all it takes for coding. Now we can take a look at the compiling part. MirageOS itself supports multiple backends. The crucial choice therefore is to decide which one to use at the beginning by using `mirage configure`. In the directory that holds the previous two files, you run `mirage configure -t unix`, and it configures to build the unikernel into a Unix ELF binary that can be directly

executed. Or you can use `mirage configure -t xen`, and then the resulting unikernel will use hypervisor backend like Xen or KVM. Either way, the unikernel runs as a virtual machine after starting up. In this example we choose to use UNIX as backends. So we run:

```
| mirage configure -t unix
```

This command generates a `Makefile` based on the configuration information. It includes all the building rules. Next, to make sure all the dependencies are installed, we need to run:

```
| make depend
```

Finally, we can build the unikernels by simply running:

```
| make
```

and it calls the `mirage build` command. As a result, now your current directory contains the `_build/gd_owl.native` executable, which is the unikernel we want. Executing it yields a similar result as before:

```
| INFO : argmin f(x) = 1.33333
```

Example: Neural Network

As a more complex example we have also built a simple neural network to perform the MNIST handwritten digits recognition task:

```
module N = Owl_base_algodiff_primal_ops.S
module NN = Owl_neural_generic.Make (N)
open NN
open NN.Graph
open NN.Algodiff

let make_network input_shape =
  input input_shape
  |> lambda (fun x -> Maths.(x / F 256.))
  |> fully_connected 25 ~act_typ:Activation.Relu
  |> linear 10 ~act_typ:Activation.(Softmax 1)
  |> get_network
```

This neural network has two hidden layer, has a small weight size (146KB), and works well in testing (92% accuracy). We can right the weight into a text file.

This file is named `simple_mnist.ml`, and similar to previous example, we can add a unikernel entry point function in the file:

```
module Main = struct
  let start = infer (); Lwt.return_unit
end
```

Here the `infer` function creates a neural network, loads the weight, and then performs inference on an input image. We also need a configuration file. Again, it's mostly the same:

```
open Mirage

let main =
  foreign
    ~packages:[package "owl-base"]
    "Simple_mnist.Main" job

let () =
  register "Simple_mnist" [main]
```

All the code is included in this gist. Once compiled to MirageOS unikernel with unix backends, the generated binary is 10MB. You can also try compiling this application to JavaScript.

By these examples we show that the Owl library can be readily deployed into unikernels via MirageOS. The numerical functionalities can then greatly enhance the express ability of possible OCaml-MirageOS applications. Of course, we cannot cover all the important topics about MirageOS, please refer to the documentation of MirageOS and Xen Hypervisor for more information.

Evaluation

In the evaluation section we mainly compare the performance of different backends we use. Specifically, we observe three representative groups of operations: (1) `map` and `fold` operations on `ndarray`; (2) using gradient descent, a common numerical computing subroutine, to get *argmin* of a certain function; (3) conducting inference on complex DNNs, including SqueezeNet and a VGG-like convolution network. The evaluations are conducted on a ThinkPad T460S laptop with Ubuntu 16.04 operating system. It has an Intel Core i5-6200U CPU and 12GB RAM.

The OCaml compiler can produce two kinds of executables: bytecode and native. Native executables are compiled for specific architectures and are generally faster, while bytecode executables have the advantage of being portable.

For JavaScript, we use the `js_of_ocaml` approach as described in the previous sections. Note that for convenience we refer to the pure implementation of

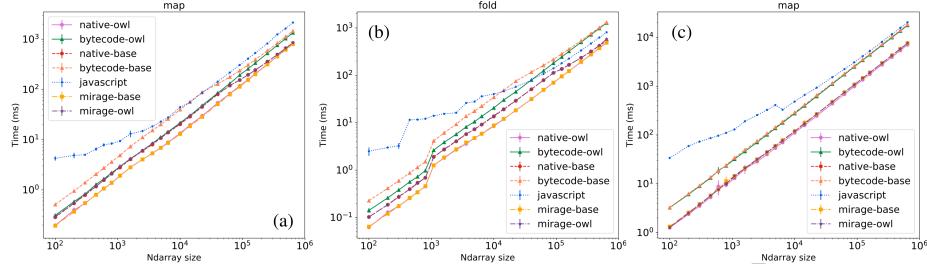


Figure 126: Performance of map and fold operations on ndarray on laptop and RaspberryPi

OCaml and the mix implementation of OCaml and C as `base-lib` and `owl-lib` separately, but they are in fact all included in the Owl library. For Mirage compilation, we use both libraries.

fig. 126(a-b) show the performance of map and fold operations on ndarray. We use simple functions such as plus and multiplication on 1-d (size < 1,000) and 2-d arrays. The log-log relationship between total size of ndarray and the time each operation takes keeps linear. For both operations, `owl-lib` is faster than `base-lib`, and native executables outperform bytecode ones. The performance of Mirage executives is close to that of native code. Generally JavaScript runs the slowest, but note how the performance gap between JavaScript and the others converges when the ndarray size grows. For fold operation, JavaScript even runs faster than bytecode when size is sufficiently large.

Note that for the fold operation, there is an obvious increase in time used at around input size of 10^3 for fold operations, while there is not such change for the map operation. That is because I change the input from one dimensional ndarray to two dimensional starting that size. This change does not affect map operation, since it treats an input of any dimension as a one dimensional vector. On the other hand, the fold operation considers the factor of dimension, and thus its performance is affected by this change.

In fig. 127, we want to investigate if the above observations still hold in more complex numerical computation. We choose to use a Gradient Descent algorithm to find the value that locally minimise a function. We choose the initial value randomly between $[0, 10]$. For both $\sin(x)$ and $x^3 - 2x^2 + 2$, we can see that JavaScript runs the slowest, but this time the `base-lib` slightly outperforms `owl-lib`.

We further compare the performance of DNN, which requires large amount of computation. We compare SqueezeNet and a VGG-like convolution network. They have different sizes of weight and networks structure complexities.

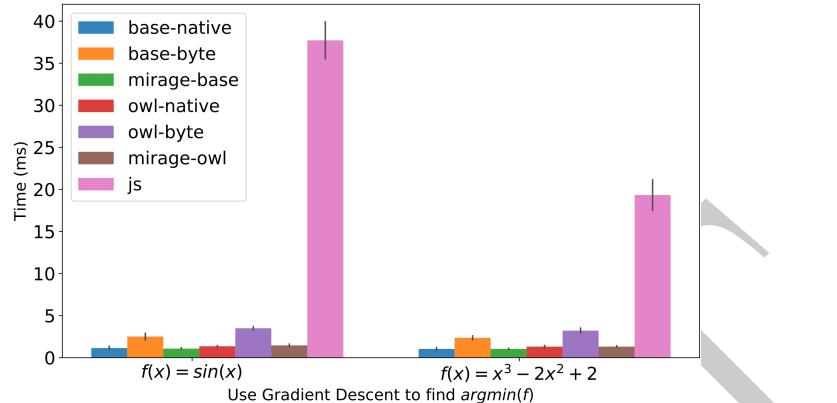
Figure 127: Performance of gradient descent on function f

Table 29: Inference Speed of Deep Neural Networks

Time (ms)	VGG	SqueezeNet
owl-native	7.96 (± 0.93)	196.26(± 1.12)
owl-byte	9.87 (± 0.74)	218.99(± 9.05)
base-native	792.56(± 19.95)	14470.97 (± 368.03)
base-byte	2783.33(± 76.08)	50294.93 (± 1315.28)
mirage-owl	8.09(± 0.08)	190.26(± 0.89)
mirage-base	743.18 (± 13.29)	13478.53 (± 13.29)
JavaScript	4325.50(± 447.22)	65545.75 (± 629.10)

tbl. 29 shows that, though the performance difference between `owl-lib` and `base-lib` is not obvious, the former is much better. So is the difference between native and bytecode for `base-lib`. JavaScript is still the slowest. The core computation required for DNN inference is the convolution operation. Its implementation efficiency is the key to these differences. Current we are working on improving its implementation in `base-lib`.

We have also conducted the same evaluation experiments on RaspberryPi 3 Model B. fig. 126(c) shows the performance of fold operation on ndarray. Besides the fact that all backends runs about one order of magnitude slower than that on the laptop, previous observations still hold. This figure also implies that, on resource-limited devices such as RaspberryPi, the key difference is between native code and bytecode, instead of `owl-lib` and `base-lib` for this operation.

Table 30: Size of executables generated by backends

Size (KB)	native	bytecode	Mirage	JavaScript
base	2,437	4,298	4,602	739
native	14,875	13,102	16,987	-

Finally, we also briefly compare the size of executables generated by different backends. We take the SqueezeNet for example, and the results are shown in tbl. 30. It can be seen that `owl-lib` executives have larger size compared to `base-lib` ones, and JavaScript code has the smallest file size. There does not exist a dominant method of deployment for all these backends. It is thus imperative to choose suitable backend according to deployment environment.

of Owl to more devices. Finally, we use several examples to demonstrate how these backends are used and their performances.

DRAFT

Distributed Computing

Background: decentralised computation

In this chapter, we will cover two topics:

1. Actor Engine
2. Barrier control, especially PSP

Refer to (Wang, Catterall, and Mortier 2017) for more detail.

Actor System

Introduction: Distributed computing engines etc.

Design

(TODO: the design of actor's functor stack; how network/barrier etc. are implemented as separated as different module. Connection with Mirage etc.)

Actor Engines

A key choice when designing systems for decentralised machine learning is the organisation of compute nodes. In the simplest case, models are trained in a centralised fashion on a single node leading to use of hardware accelerators such as GPUs and the TPU. For reasons indicated above, such as privacy and latency, decentralised machine learning is becoming more popular where data and model are spread across multiple compute nodes. Nodes compute over the data they hold, iteratively producing model updates for incorporation into the model, which is subsequently disseminated to all nodes. These compute nodes can be organised in various ways.

The Actor system has implemented core APIs in both map-reduce engine and parameter sever engine. Both map-reduce and parameter server engines need a (logical) centralised entity to coordinate all the nodes' progress. To demonstrate PSP's capability to transform an existing barrier control method into its fully distributed version, we also extended the parameter server engine to peer-to-peer (p2p) engine. The p2p engine can be used to implement both data and model parallel applications, both data and model parameters can be (although not necessarily) divided into multiple parts then distributed over different nodes.

Each engine has its own set of APIs. E.g., map-reduce engine includes `map`, `reduce`, `join`, `collect`, and etc.; whilst the peer-to-peer engine provides four major APIs: `push`, `pull`, `schedule`, `barrier`. It is worth noting there is one function shared by all the engines, i.e. `barrier` function which implements various barrier control mechanisms.

Next we will introduce these three different kinds of engines of Actor.

Map-Reduce Engine

Following MapReduce (Dean and Ghemawat 2008) programming model, nodes can be divided by tasks: either *map* or *reduce*. A map function processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function aggregates all the intermediate key/value pairs with the same key. Execution of this model can automatically be parallelized. Mappers compute in parallel while reducers receive the output from all mappers and combine to produce the accumulated result. This parameter update is then broadcast to all nodes. Details such as distributed scheduling, data divide, and communication in the cluster are mostly transparent to the programmers so that they can focus on the logic of mappers and reducers in solving a problem within a large distributed system.

We can use a simple example to demonstrate this point. (with illustration, not code)

This simple functional style can be applied to a surprisingly wide range of applications.

Interfaces in Actor:

```
val map : ('a -> 'b) -> string -> string
val reduce : ('a -> 'a -> 'a) -> string -> 'a option
val fold : ('a -> 'b -> 'a) -> 'a -> string -> 'a
val filter : ('a -> bool) -> string -> string
val shuffle : string -> string
val union : string -> string -> string
val join : string -> string -> string
val collect : string -> 'a list
```

Example of using Map-Reduce in Actor: we use the classic wordcount example.

```
module Ctx = Actor.Mapre

let print_result x = List.iter (fun (k,v) -> Printf.printf "%s : %i\n" k v) x

let stop_words = ["a";"are";"is";"in";"it";"that";"this";"and";"to";"of";"so";
"will";"can";"which";"for";"on";"in";"an";"with";"the";"-"]

let wordcount () =
  Ctx.init Sys.argv.(1) "tcp://localhost:5555";
  Ctx.load "unix://data/wordcount.data"
  |> Ctx.flatmap Str.(split (regexp "[ \t\n]"))
```

```

|> Ctx.map String.lowercase_ascii
|> Ctx.filter (fun x -> (String.length x) > 0)
|> Ctx.filter (fun x -> not (List.mem x stop_words))
|> Ctx.map (fun k -> (k,1))
|> Ctx.reduce_by_key (+)
|> Ctx.collect
|> List.flatten |> print_result;
Ctx.terminate ()

let _ = wordcount ()

```

Parameter Server Engine

The Parameter Server topology proposed by (Li et al. 2014) is similar: nodes are divided into servers holding the shared global view of the up-to-date model parameters, and workers, each holding its own view of the model and executing training. The workers and servers communicate in the format of key-value pairs. It is proposed to address of challenge of sharing large amount of parameters within a cluster. The parameter server paradigm applies an asynchronous task model to educe the overall network bandwidth, and also allows for flexible consistency, resource management, and fault tolerance.

Simple Example (distributed training) with illustration: IMAGE: Distributed training with Parameter Server (Src:(Li et al. 2014))

According to this example, we can see that the Parameter Server paradigm mainly consists of four APIs for the users.

- **schedule:** decide what model parameters should be computed to update in this step. It can be either a local decision or a central decision.
- **pull:** retrieve the updates of model parameters from somewhere then applies them to the local model. Furthermore, the local updates will be computed based on the scheduled model parameter.
- **push:** send the updates to the model plane. The updates can be sent to either a central server or to individual nodes depending on which engine is used(e.g., map-reduce, parameter server, or peer-to-peer).
- **barrier:** decide whether to advance the local step. Various synchronisation methods can be implemented. Besides the classic BSP, SSP, and ASP, we also implement the proposed PSP within this interface.

The interfaces in Actor:

```

val start : ?barrier:barrier -> string -> string -> unit
(** start running the model loop *)

val register_barrier : ps_barrier_typ -> unit
(** register user-defined barrier function at p2p server *)

```

```

val register_schedule : ('a, 'b, 'c) ps_schedule_typ -> unit
(** register user-defined scheduler *)

val register_pull : ('a, 'b, 'c) ps_pull_typ -> unit
(** register user-defined pull function executed at master *)

val register_push : ('a, 'b, 'c) ps_push_typ -> unit
(** register user-defined push function executed at worker *)

val register_stop : ps_stop_typ -> unit
(** register stopping criterion function *)

val get : 'a -> 'b * int
(** given a key, get its value and timestamp *)

val set : 'a -> 'b -> unit
(** given a key, set its value at master *)

val keys : unit -> 'a list
(** return all the keys in a parameter server *)

val worker_num : unit -> int
(** return the number of workers, only work at server side *)

```

EXPLAIN

Example of using PS in Actor :

```

module PS = Actor_param

let schedule_workers =
  let tasks = List.map (fun x ->
    let k, v = Random.int 100, Random.int 1000 in (x, [(k,v)])
  ) workers in tasks

let push id vars =
  let updates = List.map (fun (k,v) ->
    Owl_log.info "working on %i" v;
    (k,v) ) vars in
  updates

let test_context () =
  PS.register_schedule schedule;
  PS.register_push push;
  PS.start Sys.argv.(1) Actor_config.manager_addr;
  Owl_log.info "do some work at master node"

let _ = test_context ()

```

Peer-to-Peer Engine

In the above approaches the model parameter storage is managed by a set of centralised servers. In contrast, Peer-to-Peer (P2P) is a fully distributed

structure, where each node contains its own copy of the model and nodes communicate directly with each other. The benefit of this approach.

Illustrate how distributed computing can be finished with P2P model, using a figure.

To obtain the aforementioned two pieces of information, we can organise the nodes into a structured overlay (e.g., chord or kademlia), the total number of nodes can be estimated by the density of each zone (i.e., a chunk of the name space with well-defined prefixes), given the node identifiers are uniformly distributed in the name space. Using a structured overlay in the design guarantees the following sampling process is correct, i.e., random sampling.

Implementation in Actor:

```
open Actor_types

(** start running the model loop *)
val start : string -> string -> unit

(** register user-defined barrier function at p2p server *)
val register_barrier : p2p_barrier_typ -> unit

(** register user-defined pull function at p2p server *)
val register_pull : ('a, 'b) p2p_pull_typ -> unit

(** register user-defined scheduler at p2p client *)
val register_schedule : 'a p2p_schedule_typ -> unit

(** register user-defined push function at p2p client *)
val register_push : ('a, 'b) p2p_push_typ -> unit

(** register stopping criterion function at p2p client *)
val register_stop : p2p_stop_typ -> unit

(** given a key, get its value and timestamp *)
val get : 'a -> 'b * int

(** given a key, set its value at master *)
val set : 'a -> 'b -> unit
```

EXPLAIN

Example of using P2P in Actor (SGD):

```
open Owl
open Actor_types

module MX = Mat
module P2P = Actor_peer

...
```

```

let schedule _context =
  Owl_log.debug "%s: scheduling ..." !_context.master_addr;
  let n = MX.col_num !_model in
  let k = Stats.Rnd.uniform_int ~a:0 ~b:(n - 1) () in
  [ k ]

let push _context params =
  List.map (fun (k,v) ->
    Owl_log.debug "%s: working on %i ..." !_context.master_addr k;
    let y = MX.col !data_y k in
    let d = calculate_gradient 10 !data_x y v !gradfn !lossfn in
    let d = MX.(d *$ !step_t) in
    (k, d)
  ) params

let barrier _context =
  Owl_log.debug "checking barrier ...";
  true

let pull _context updates =
  Owl_log.debug "pulling updates ...";
  List.map (fun (k,v,t) ->
    let v0, _ = P2P.get k in
    let v1 = MX.(v0 - v) in
    k, v1, t
  ) updates

let stop _context = false

let start jid =
  P2P.register_barrier barrier;
  P2P.register_schedule schedule;
  P2P.register_push push;
  P2P.register_pull pull;
  P2P.register_stop stop;
  Owl_log.info "P2P: sdg algorithm starts running ...";
  P2P.start jid Actor_config.manager_addr

```

EXPLAIN

Classic Synchronise Parallel

To ensure the correctness of computation, normally we need to make sure a correct order of updates. For example, one worker can only proceed when the model has been updated with all the workers' updates from previous round. However, the iterative-convergent nature of ML programmes means that they are error-prone to a certain degree.

Most consistent system often leads to less than ideal system throughput. This error-proneness means that, the consistency can be relaxed a bit without sacrificing accuracy, and gains system performance at the same time. This trade-off is decided by the “barrier” in distributed ML. A lot of research on it, both theoretically and practically.

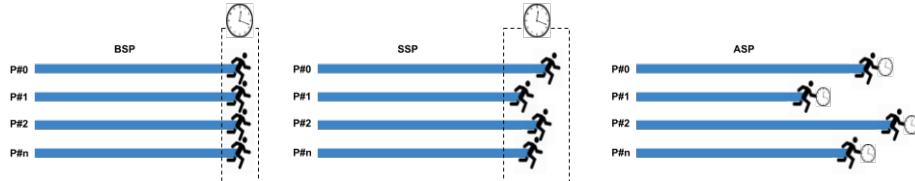


Figure 128: Barrier control methods used for synchronisation

Existing distributed processing systems operate at various points in the space of consistency/speed trade-offs. However, all effectively use one of three different synchronisation mechanisms: Bulk Synchronise Parallel (BSP), Stale Synchronise Parallel (SSP), and Asynchronous Parallel (ASP). These are depicted in fig. 128.

Bulk Synchronous Parallel

Bulk Synchronous Parallel (BSP) is a deterministic scheme where workers perform a computation phase followed by a synchronisation/communication phase where they exchange updates. The method ensures that all workers are on the same iteration of a computation by preventing any worker from proceeding to the next step until all can. Furthermore, the effects of the current computation are not made visible to other workers until the barrier has been passed. Provided the data and model of a distributed algorithm have been suitably scheduled, BSP programs are often serialisable – that is, they are equivalent to sequential computations. This means that the correctness guarantees of the serial program are often realisable making BSP the strongest barrier control method. Unfortunately, BSP does have a disadvantage. As workers must wait for others to finish, the presence of *stragglers*, workers which require more time to complete a step due to random and unpredictable factors, limit the computation efficiency to that of the slowest machine. This leads to a dramatic reduction in performance. Overall, BSP tends to offer high computation accuracy but suffers from poor efficiency in unfavourable environments.

BSP is the most strict lockstep synchronisation; all the nodes are coordinated by a central server. BSP is sensitive to stragglers so is very slow. But it is simple due to its deterministic nature, easy to write application on top of it.

Asynchronous Parallel

Asynchronous Parallel (ASP) takes the opposite approach to BSP, allowing computations to execute as fast as possible by running workers completely asynchronously. In homogeneous environments (e.g. data centers), wherein the workers have similar configurations, ASP enables fast convergence because it permits the highest iteration throughputs. Typically, P -fold speed-ups can be achieved by adding more computation/storage/bandwidth resources. However,

such asynchrony causes delayed updates: updates calculated on an old model state which should have been applied earlier but were not. Applying them introduces noise and error into the computation. Consequently, ASP suffers from decreased iteration quality and may even diverge in unfavourable environments. Overall, ASP offers excellent speed-ups in convergence but has a greater risk of diverging especially in a heterogeneous context.

ASP is the Least strict synchronisation, no communication among workers for barrier synchronisation all all. Every computer can progress as fast as it can. It is fast and scalable, but often produces noisy updates. No theoretical guarantees on consistency and algorithm's convergence.

Stale Synchronous Parallel

Stale Synchronous Parallel (SSP) is a bounded asynchronous model which can be viewed as a relaxation of BSP. Rather than requiring all workers to be on the same iteration, the system decides if a worker may proceed based on how far behind the slowest worker is, i.e. a pre-defined bounded staleness. Specifically, a worker which is more than s iterations behind the fastest worker is considered too slow. If such a worker is present, the system pauses faster workers until the straggler catches up. This s is known as the *staleness* parameter. More formally, each machine keeps an iteration counter, c , which it updates whenever it completes an iteration. Each worker also maintains a local view of the model state. After each iteration, a worker commits updates, i.e., Δ , which the system then sends to other workers, along with the worker's updated counter. The bounding of clock differences through the staleness parameter means that the local model cannot contain updates older than $c - s - 1$ iterations. This limits the potential error. Note that systems typically enforce a *read-my-writes* consistency model. The staleness parameter allows SSP to provide deterministic convergence guarantees. Note that SSP is a generalisation of BSP: setting $s = 0$ yields the BSP method, whilst setting $s = \infty$ produces ASP. Overall, SSP offers a good compromise between fully deterministic BSP and fully asynchronous ASP~[?], despite the fact that the central server still needs to maintain the global state to guarantee its determinism nature.

SSP relaxes consistency by allowing difference in iteration rate. The difference is controlled by the bounded staleness. SSP is supposed to mitigate the negative effects of stragglers. But the server still requires global state.

Probabilistic Synchronise Parallel

Existing barrier methods allow us to balance consistency against iteration rate in attempting to achieve a high rate of convergence. In particular, SSP parameterises the spectrum between ASP and BSP by introducing a staleness parameter, allowing some degree of asynchrony between nodes so long as no node lags too far behind. Figure~?? depicts this trade-off.

However, in contrast to a highly reliable and homogeneous datacentre context,

let's assume a distributed system consisting of a larger amount of heterogeneous nodes that are distributed at a much larger geographical areas. The network is unreliable since links can break down, and the bandwidth is heterogeneous. The nodes are not static, they can join and leave the system at any time. We observe that BSP and SSP are not suitable for this scenario, since both are centralised mechanisms: a single server is responsible for receiving updates from each node and declaring when the next iteration can proceed.

In contrast, ASP is fully distributed and no single node is responsible for the system making progress. We examine this trade-off in more detail, and suggest that the addition of a new *sampling* primitive exposes a second axis on which solutions can be placed, leading to greater flexibility in how barrier controls can be implemented. We call the resulting barrier control method (or, more precisely, family of barrier control methods) *Probabilistic Synchronous Parallel*, or PSP.

Basic idea: sampling

The idea of PSP is simple: in a unreliable environment, we can minimise the impact of outliers and stragglers by guaranteeing the majority of the system have synchronised boundary. The reason we can drop the results from certain portion of workers is that, practically many iterative learning algorithms can tolerate certain level of errors in the process of converging to final solutions. Given a well-defined boundary, if most of the nodes have passed it, the impact of those lagged nodes should be minimised.

Therefore, in PSP, all we need to do is to estimate what percent of nodes have passed a given synchronisation barrier. Two pieces of information is required to answer this question: - an estimate on the total number of nodes in the system; - an estimate of the distribution of current steps of the nodes.

In PSP, either a central oracle tracks the progress of each worker or the workers each hold their own local view. In a centralised system, without considering the difficulty of monitoring state of all the nodes as system grows bigger, these two pieces of information is apparently trivial to get at a central server. However, in a distributed system, where each node does not have global knowledge of other nodes, how can it get these information? In that case, a node randomly selects a subset of nodes in the system and query their individual current local step. By so doing, it obtains a sample of the current nodes' steps in the whole system. By investigating the distribution of these observed steps, it can derive an estimate of the percentage of nodes which have passed a given step. After deriving the estimate on the step distribution, a node can choose to either pass the barrier by advancing its local step if a given threshold has been reached (with certain probability) or simply holds until certain condition is satisfied. Each node only depends on several other nodes to decide its own barrier.

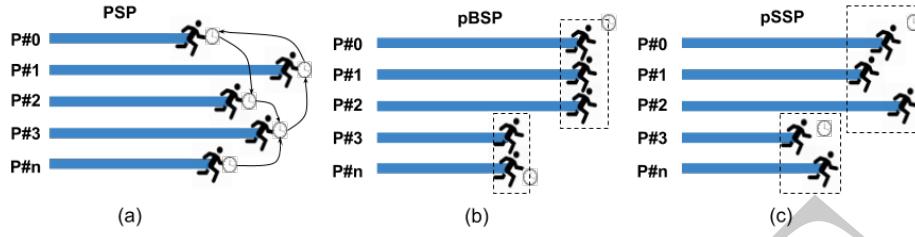


Figure 129: Probabilistic Synchronous Parallel example

Compatibility

One great advantage of PSP is its compatibility with existing synchronisation methods. For classic BSP and SSP, their barrier functions are called by the centralised server to check the synchronisation condition with the given inputs. The output of the function is a boolean decision variable on whether or not to cross the synchronisation barrier, depending on whether the criterion specified in the algorithm is met. With the proposed sampling primitive, almost nothing needs to be changed in aforementioned algorithms except that only the sampled states instead of the global states are passed into the barrier function. Therefore, we can easily derive the probabilistic version of BSP and SSP, namely *pBSP* and *pSSP*.

PSP improves on ASP by providing probabilistic guarantees about convergence with tighter bounds and less restrictive assumptions. pSSP relaxes SSP's inflexible staleness parameter by permitting some workers to fall further behind. pBSP relaxes BSP by allowing some workers to lag slightly, yielding a BSP-like method which is more resistant to stragglers but no longer deterministic.

fig. 129(a) depicts PSP showing that different subsets of the population of nodes operate in (possibly overlapping) groups, applying one or other barrier control method within their group. In this case, fig. 129(b) shows PSP using BSP within group (or pBSP), and fig. 129(c) shows PSP using SSP within group (or pSSP).

Formally, at the barrier control point, a worker samples β out of P workers without replacement. If a single one of these lags more than s steps behind the current worker then it waits. This process is pBSP (based on BSP) if $s = 0$ and pSSP (based on SSP) if $s > 0$. However, if $s = \infty$ then PSP reduces to ASP.

Barrier Trade-off Dimensions

This allows us to decouple the degree of synchronisation from the degree of distribution, introducing *completeness* as a second axis by having each node sample from the population. Within each sampled subgroup, traditional mechanisms can be applied allowing overall progress to be robust against the effect of stragglers while also ensuring a degree of consistency between iterations as the algorithm progresses. As fig. 130(a-b) depicts, the result is a larger design

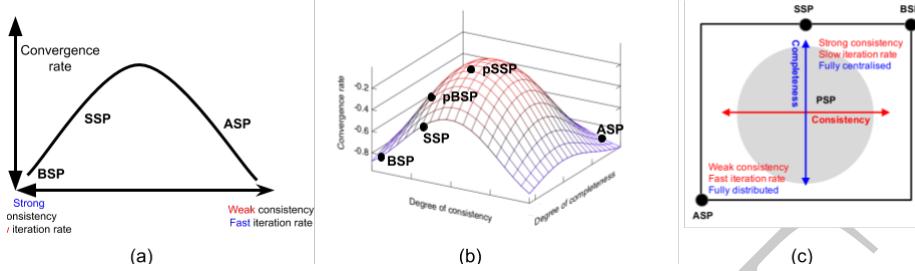


Figure 130: Extra trade-off exposed through PSP

space for synchronisation methods when operating distributed data processing at scale.

As fig. 130(c) summarises, probabilistic sampling allows us greater flexibility in designing synchronisation mechanisms in distributed processing systems at scale. When compared with BSP and SSP, we can obtain faster convergence through faster iterations and with no dependence on a single centralised server. When compared with ASP, we can obtain faster convergence with stronger guarantees by providing greater consistency between updates.

Besides its compatibility with existing synchronisation methods, it is also worth emphasising that applying sampling leads to the biggest difference between the classic synchronisation control and probabilistic control: namely the original synchronisation control requires a centralised node to hold the global state whereas the derived probabilistic ones no longer require such information thus can be executed independently on each individual node, further leading to a fully distributed solution.

Convergence

At the barrier control point, every worker samples β out of P workers without replacement. If a single one of these lags more than s steps behind the current worker then it waits. The probabilities of a node lagging r steps are drawn from a distribution with probability mass function $f(r)$ and cumulative distribution function (CDF) $F(r)$. Both r and β can be thought of as constant value.

In a distributing machine learning process, these P workers keep generating updates, and the model is updated with them continuously. In this sequence of updates, each one is indexed by t (which does not mean clock time), and the total length of this sequence is T . Ideally, in a fully deterministic barrier control system, such as BSP, the ordering of updates in this sequence should be fixed. We call it a *true sequence*. However, in reality, what we get is often a *noisy sequence*, where updates are reordered due to sporadic and random network and system delays. The difference, or lag, between the order of these two sequences, is denoted by γ_t .

Without talking too much about math in detail in this book, to prove the convergence of PSP requires to construct and idea sequence of updates, each generated by workers in the distributed learning, and compare it with the actual sequence after applying PSP. The target of proof is to show that, given sufficient time t , the difference between these two sequences γ_t is limited.

The complete proof of convergence is too long to fit into this chapter. Please refer to (Wang, Catterall, and Mortier 2017) for more detail if you are interested in the math. One key step in the proof is to show that the mean and variance of γ_t are bounded. The average of the mean is bounded by:

$$\frac{1}{T} \sum_{t=0}^T E(\gamma_t) \leq S \left(\frac{r(r+1)}{2} + \frac{a(r+2)}{(1-a)^2} \right). \quad (77)$$

The average of the variance has a similar bound:

$$\frac{1}{T} \sum_{t=0}^T E(\gamma_t^2) < S \left(\frac{r(r+1)(2r+1)}{6} + \frac{a(r^2+4)}{(1-a)^3} \right), \quad (78)$$

where

$$S = \frac{1-a}{F(r)(1-a) + a - a^{T-r+1}}. \quad (79)$$

The intuition is that, when applying PSP, the update sequence we get will not be too different from the true sequence. To demonstrate the impact of the sampling primitive on bounds quantitatively, fig. 131 shows how increasing the sampling count, β , (from 1, 5, to 100, marked with different line colours on the right) yields tighter bounds. The sampling count β is varied between 1 and 100 and marked with different line colours on the right. The staleness, r , is set to 4 with T equal to 10000. The bounds on γ_t mean that what a true sequence achieves, in time, a noisy sequence can also achieve, regardless of the order of updates. Notably, only a small number of nodes need to be sampled to yield bounds close to the optimal. This result has an important implication to justify using sampling primitive in large distributed learning systems due to its effectiveness.

A Distributed Training Example

In this section, we investigate performance of the proposed PSP in experiments. We focus on two common metrics in evaluating barrier strategies: the step progress and accuracy. We use the training of a DNN as an example, using a 9-layer structure used in the Neural Network chapter, and for the training we also use the MNIST handwritten digits dataset. The learning rate has a decay factor of 1e4. The network structure is shown below:

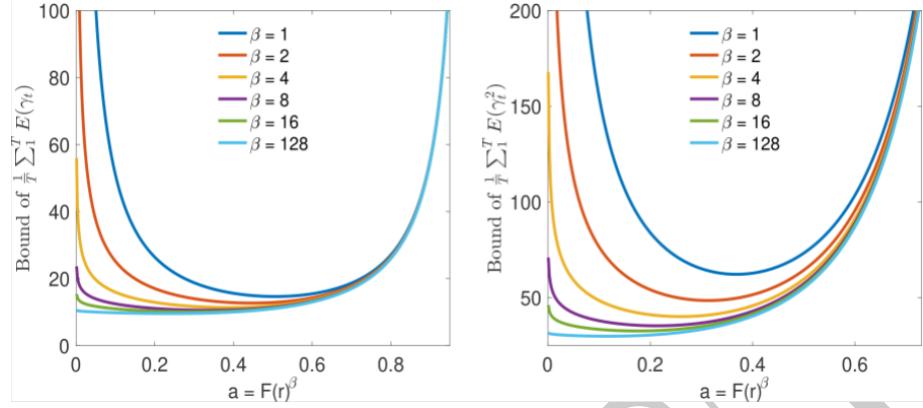


Figure 131: Plot showing the bound on the average of the means and variances of the sampling distribution.

```
let make_network () =
  input [|28;28;1|]
  |> normalisation ~decay:0.9
  |> conv2d [|5;5;1;32|] [|1;1|] ~act_typ:Activation.Relu
  |> max_pool2d [|2;2|] [|2;2|]
  |> dropout 0.1
  |> fully_connected 1024 ~act_typ:Activation.Relu
  |> linear 10 ~act_typ:Activation.(Softmax)
  |> get_network
```

We use both real-world experiments and simulations to evaluate different barrier control methods. The experiments run on 6 nodes using Actor. To extend the scale, we have also built a simulation platform. For both cases, we implement the Parameter Server framework. It consists of one server and many worker nodes. In each step, a worker takes a chunk of training data, calculates the weight value, and then aggregates these updates to the parameter server, thus updating the shared model iteratively. A worker pulls new model from server after it is updated.

Step Progress

First, we are going to investigate if PSP achieves faster iteration speed. We use 200 workers, and run the simulation for 200 simulated seconds. fig. 132 shows the distribution of all nodes' step progress when simulation is finished.

As expected, the most strict BSP leads to a tightly bounded step distribution, but at the same time, using BSP makes all the nodes progress slowly. At the end of simulation, all the nodes only proceed to about 30th step. As a comparison, using ASP leads to a much faster progress of around 100 steps. But the cost is a much loosely spread distribution, which shows no synchronisation at all among nodes. SSP allows certain staleness (4 in our experiment) and sits between BSP

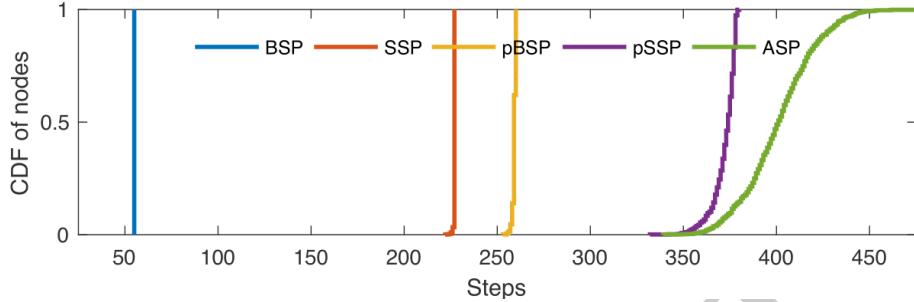


Figure 132: Progress distribution in steps

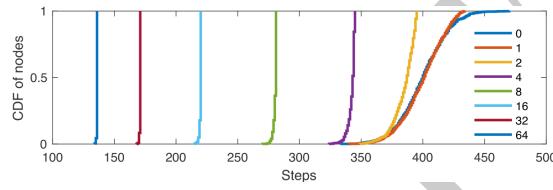


Figure 133: pBSP parameterised by different sample sizes, from 0 to 64.

and ASP.

PSP shows another dimension of performance tuning. We set sample size β to 10, i.e. a sampling ratio of only 5%. The result shows that pBSP is almost as tightly bound as BSP and also much faster than BSP itself. The same is also true when comparing pSSP and SSP. In both cases, PSP improves the iteration efficiency while limiting dispersion.

To further investigate the impact of sample size, we focus on BSP, and choose different sample sizes. In fig. 133 we vary the sample size from 0 to 64. As we increase the sample size step by step, the curves start shifting from right to left with tighter and tighter spread, indicating less variance in nodes' progress. With sample size 0, the pBSP exhibits exactly the same behaviour as that of ASP; with increased sample size, pBSP starts becoming more similar to SSP and BSP with tighter requirements on synchronisation. pBSP of sample size 4 behaves very close to SSP.

Another interesting thing we notice in the experiment is that, with a very small sample size of one or two (i.e., very small communication cost on each individual node), pBSP can already effectively synchronise most of the nodes comparing to ASP. The tail caused by stragglers can be further trimmed by using larger sample size. This observation confirms our convergence analysis, which explicitly shows that a small sample size can effectively push the probabilistic convergence guarantee to its optimum even for a large system size, which further indicates the superior scalability of the proposed solution.

Accuracy

Next, we evaluate barrier control methods in training a deep neural network using MNIST dataset. We use inference accuracy on test dataset as measurement of performance.

To run the code, we mainly implement the interfaces we mentioned before: `sche`, `push`, and `pull`. `sche` and `pull` are performed on server, and `push` is on worker.

```

let schd nodes =
  Actor_log.info "#nodes: %d" (Array.length nodes);
  if (Array.length nodes > 0) then (
    eval_server_model ()
  );

  let server_value = (get [|key_name|]).(0) in
  Array.map (fun node ->
    Actor_log.info "node: %s schd" node;
    let wid = int_of_uuid node in
    let v = make_task server_value.weight wid in
    let tasks = [|(key_name, v)|] in
    (node, tasks)
  ) nodes

t pull_kv_pairs =
  Gc.compact ();
  (* knowing that there is only one key...*)
  let key = key_name in
  Actor_log.info "pull: %s (length: %d)" key (Array.length kv_pairs);

  let s_value = (get [|key|]).(0) in
  let s_weight = ref s_value.weight in
  Array.iter (fun (_, v) ->
    s_weight := add_weight !s_weight v.weight;
    model.clocks.(v.wid) <- model.clocks.(v.wid) + 1
  ) kv_pairs;
  let value = make_task !s_weight 0 in (* wid here is meaningless *)
  [| (key, value) |]

```

And on worker:

```

let push_kv_pairs =
  Gc.compact ();
  Array.map (fun (k, v) ->
    Actor_log.info "push: %s" k;
    (* simulated communication delay *)

    (* let t = delay.(v.wid) in *)
    let t = Owl_stats.gamma_rvs ~shape:1. ~scale:1. in
    Unix.sleepf t;

    let nn = make_network () in
    Graph.init nn;

```

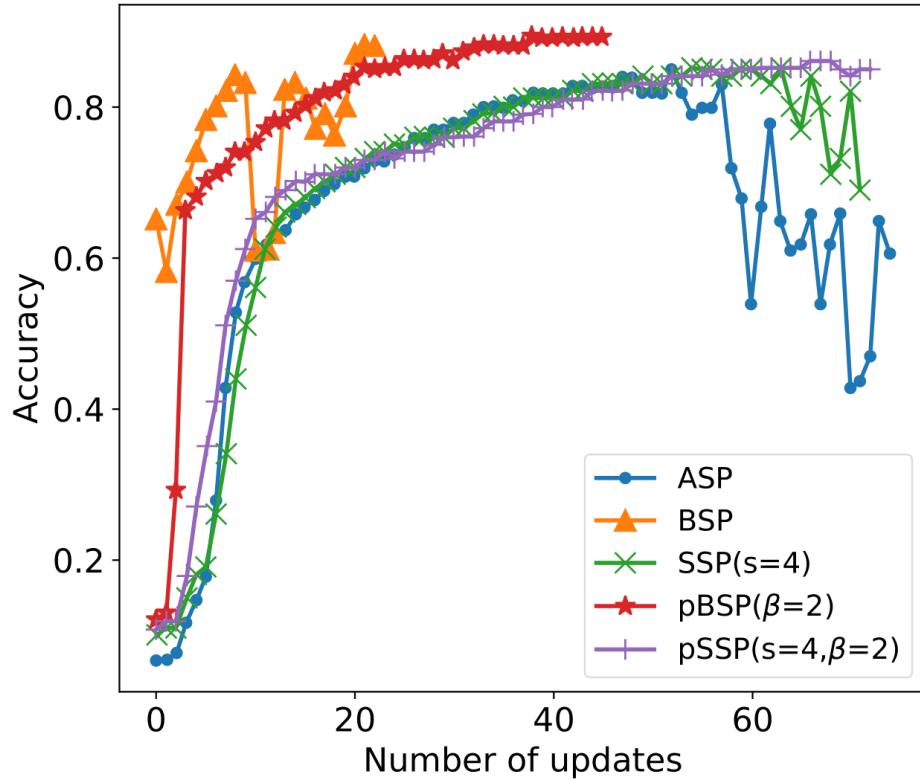


Figure 134: MNIST training using Actor

```

Graph.update nn v.weight;
let x, y = get_next_batch v.wid in
let x = pack x in
let y = pack y in
let _ = CGCompiler.train ~params ~init_model:false nn x y in
let delta = delta_nn nn v.weight in
let value = make_task delta v.wid in
(k, value)
) kv_pairs
    
```

TODO: Explain the code

In fig. 134, we conduct the real-world experiments using 6 worker nodes with the Parameter Server framework we have implemented. We run the training process for a fixed amount of time, and observe the performance of barrier methods given the same number of updates. It shows that BSP achieves the highest model accuracy with the least of number of updates, while SSP and ASP achieve lower efficiency. With training progressing, both methods show a tendency to diverge. By applying sampling, pBSP and pSSP achieve smoother accuracy progress.

Summary

References

Dean, Jeffrey, and Sanjay Ghemawat. 2008. “MapReduce: Simplified Data Processing on Large Clusters.” *Communications of the ACM* 51 (1): 107–13.

Li, Mu, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. “Scaling Distributed Machine Learning with the Parameter Server.” In *11th {Usenix} Symposium on Operating Systems Design and Implementation ({Osdi} 14)*, 583–98.

Wang, Liang, Ben Catterall, and Richard Mortier. 2017. “Probabilistic Synchronous Parallel.” *arXiv Preprint arXiv:1709.07772*.

Testing Framework

Every proper software requires testing, and so is Owl. All too often we have found that testing can help us to find potential errors we had not anticipated during coding. In this chapter, we introduce the philosophy of testing in Owl, the tool we use for conducting the unit test, and examples to demonstrate how to do that in Owl. Issues such as using functors in test, and other things to notice in writing test code for Owl etc. are also discussed in this chapter.

Unit Test

There are multiple ways to perform tests on your code. One common way is to use assertion or catching/raising errors in the code. These kinds of tests are useful, but embedded in the code itself, while we need separate test modules that check the implementation of functions against expected behaviours.

In Owl, we apply *unit test* to make sure the correctness of numerical routines as much as possible. Unit test is a software test method that checks the behaviour of individual units in the code. In our case the “unit” often means a single numerical function.

There is an approach of software development that is called Test Driven Development, where you write test code even before you implement the function to be tested itself. Though we don't enforce such approach, there are certain testing principles we follow during the development of Owl. For example, we generally don't trust code that is not tested, so in a PR it is always a good practice to accompany your implementation with unit test in the `test/` directory in the source code. Besides, try to keep the function short and simple, so that a test case can focus on a certain aspect.

We use the `alcotest` framework for testing in Owl. `alcotest` is a lightweight test framework with simple interfaces. It exposes a simple `TESTABLE` module type, a `check` function to assert test predicates and a `run` function to perform a list of `unit -> unit` test callbacks.

Example

Let's look at an example of using `alcotest` in Owl. Suppose you have implemented some functions in the linear algebra module, including the functions such as rank, determinant, inversion, etc., and try to test if before make a PR. The testing code should look something like this:

```
open Owl
open Alcotest

module M = Owl.Linalg.D

(* Section #1 *)
```

```

let approx_equal a b =
  let eps = 1e-6 in
  Stdlib.(abs_float (a -. b) < eps)

let x0 = Mat.sequential ~a:1. 1 6

(* Section #2 *)

module To_test = struct
  let rank () =
    let x = Mat.sequential 4 4 in
    M.rank x = 2

  let det () =
    let x = Mat.hadamard 4 in
    M.det x = 16.

  let vecnorm_01 () =
    let a = M.vecnorm ~p:1. x0 in
    approx_equal a 21.

  let vecnorm_02 () =
    let a = M.vecnorm ~p:2. x0 in
    approx_equal a 9.539392014169456

  let is_triu_1 () =
    let x = Mat.of_array [| 1.; 2.; 3.; 0.; 5.; 6.; 0.; 0.; 9. |] 3 3 in
    M.is_triu x = true

  let mpow () =
    let x = Mat.uniform 4 4 in
    let y = M.mpow x 3. in
    let z = Mat.(dot x (dot x x)) in
    approx_equal Mat.(y - z |> sum') 0.

  end

(* Section #3 *)

let rank () = Alcotest.(check bool) "rank" true (To_test.rank ())

let det () = Alcotest.(check bool) "det" true (To_test.det ())

let vecnorm_01 () = Alcotest.(check bool) "vecnorm_01" true (To_test.vecnorm_01 ())

let vecnorm_02 () = Alcotest.(check bool) "vecnorm_02" true (To_test.vecnorm_02 ())

let is_triu_1 () = Alcotest.(check bool) "is_triu_1" true (To_test.is_triu_1 ())

let mpow () = Alcotest.(check bool) "mpow" true (To_test.mpow ())

(* Section #4 *)

let test_set =
  [ "rank", `Slow, rank
  ; "det", `Slow, det
  
```

```
; "vecnorm_01", `Slow, vecnorm_01
; "vecnorm_02", `Slow, vecnorm_02
; "is_triu_1", `Slow, is_triu_1
; "mpow", `Slow, mpow ]
```

There are generally four sections in a test file. In the first section, you specify the required precision and some predefined input data. Here we use `1e-6` as precision threshold. Two ndarrays are deemed the same if the sum of their difference is less than `1e-6`, as shown in `mpow`. The predefined input data can also be defined in each test case, as in `is_triu_1`.

In the second section, a test module that contains a series of test functions needs to be built. The most common test function used in Owl has the type `unit -> bool`. The idea is that each test function compares a certain aspect of a function with expected results. If there are multiple test cases for the same function, such the case in `vecnorm`, we tend to build different test cases instead of using one large test function to include all the cases. The common pattern of these functions can be summarised as:

```
let test_func () =
    let expected = expected_value in
    let result = func args in
    assert (expected = result)
```

It is important to understand that the equal sign does not necessarily mean the two values have to be the same; in fact, if the float-point number is involved, which is quite often the case, we only need the two values to be approximately equal enough. If that's the case, you need to pay attention to which precision you are using: `double` or `float`. The same threshold might be enough for single precision float number, but could still be a large error for double precision computation.

In the third section wraps these functions with `alcotest` by stating the expected output. Here we expect all the test functions to return `true`, though `alcotest` does support testing returning a lot of other types such as `string`, `int`, etc. Please refer to the source file for more detail.

In the final section, we take functions from section 3 and put them into a list of test set. The test set specifies the name and mode of the test. The test mode is either `quick` or `slow`. Quick tests run on any invocations of the test suite. Slow tests are for stress tests that run only on occasion, typically before a release or after a major change.

After this step, the whole file is named `unit_linalg.ml` and put under the `test/` directory, as with all other unit test files. Now the only thing left is to add it in the `test_runner.ml`:

```
dune external-libdeps --missing @install @runitest
dune runtest -j 1 --no-buffer -p owl
Done: 1683/1685 (jobs: 1)Testing Owl.
This run has ID `A77EF9E1-8D18-4772-96BD-559DA37AA1E3` .
[OK]           linear algebra      0  rank.
[OK]           linear algebra      1  det.
[OK]           linear algebra      2  vecnorm_01.
[OK]           linear algebra      3  vecnorm_02.
[OK]           linear algebra      4  is_triu_1.
[OK]           linear algebra      5  mpow.
The full test results are available in `/Users/stark/Code/owl/_build/default/test/_build/_tests/A77EF9E1-8D18-4772-96BD-559DA37AA1E3` .
Test Successful in 0.004s. 6 tests run.
```

Figure 135: All tests passes

```
let () =
  Alcotest.run "Owl"
  [ "linear algebra", Unit_linalg.test_set;
  ... ]
```

That's all. Now you can try `make test` and check if the functions are implemented well:

What if one of the test functions does not pass? Let's intentionally make a wrong test, such as asserting the matrix in the `rank` test is 1 instead of the correct answer 2, and run the test again:

What Could Go Wrong

Who's watching the watchers?

Beware that the test code itself is still code, and thus can also be wrong. We need to be careful in implementing the testing code. There are certain cases that you may want to check.

Corner Cases

Corner cases involve situations that occur outside of normal operating parameters. That is obvious in the testing of convolution operations. As the core operation in deep neural networks, convolution is complex: it contains input, kernel, strides, padding, etc. as parameters. Therefore, special cases such as 1×1 kernel, strides of different height and width etc. are tested in various combinations, sometimes with different input data.

```
module To_test_conv2d_back_input = struct
  (* conv2D1x1Kernel *)
```

```
dune external-libdeps --missing @install @runitest
dune runtest -j 1 --no-buffer -p owl
Done: 1683/1685 (jobs: 1)Testing Owl.
This run has ID `2A19E5B0-A8EF-4578-B394-EE1A3158CAC8`.
[ERROR]           linear algebra      0  rank.
[OK]           linear algebra      1  det.
[OK]           linear algebra      2  vecnorm_01.
[OK]           linear algebra      3  vecnorm_02.
[OK]           linear algebra      4  is_triu_1.
[OK]           linear algebra      5  mpow.
-- linear algebra.000 [rank.] Failed --
in /Users/stark/Code/owl/_build/default/test/_build/_tests/2A19E5B0-A8EF-4578-B394-EE1A3158CAC8/linear algebra.000.output:
-----
ASSERT rank
-----
[failure] Error rank: expecting
true, got
false.

The full test results are available in `/Users/stark/Code/owl/_build/default/test/_build/_tests/2A19E5B0-A8EF-4578-B394-EE1A3158CAC8` .
1 error! in 0.005s. 6 tests run.
```

Figure 136: Error in tests

```

let fun00 () = ...
(* conv2D1x2KernelStride3Width5 *)
let fun01 () = ...
(* conv2D1x2KernelStride3Width6 *)
let fun02 () = ...
(* conv2D1x2KernelStride3Width7 *)
let fun03 () = ...
(* conv2D2x2KernelC1Same *)
let fun04 () = ...
...
(* conv2D2x2KernelStride2Same *)
let fun09 () = ...

```

Test Coverage

Another issue is the test coverage. It means the percentage of code for which an associated test has existed. Though we don't seek a strict 100% coverage for now, wider test coverage is always a good idea. For example, in our implementation of the repeat operation, depending on whether the given axes contains one or multiple integers, the implementation changes. Therefore in the test functions it is crucial to cover both cases.

Use Functor

Note that you can still benefit from all the powerful features OCaml such as functor. For example, in testing the convolution operation, we hope to the implementation of both that in the core library (which implemented in C), and that in the base library (in pure OCaml). Apparently there is no need to write the same unit test code twice for these two set of implementation.

To solve that problem, we have a test file `unit_conv2d_generic1.ml` that has a large module that contains all the previous four sections:

```

module Make (N : Ndarray_Algodiff with type elt = float) = struct
  (* Section #1 - #4 *)
  ...
end

```

And in the specific testing file for core implementation `unit_conv2d.ml`, it simply contains one line of code:

```

| include Unit_conv2d_generic.Make (Owl_algodiff_primal_ops.S)

```

Or in the test file for base library `unit_base_conv2d.ml`:

```
| include Unit_conv2d_generic.Make (Owl_base_algodiff_primal_ops.S)
```

Summary

In this chapter we briefly introduce how the unit tests are performed with the `alcotest` framework in the existing Owl code base. We use one example piece of test code for the linear algebra module in Owl to demonstrate the general structure of the Owl test code. We then discuss some tips we find helpful in writing tests, such as considering corner cases, test coverage, and using functors to simplify the test code. In practice, we find the unit tests come really handy in development, and we just cannot have too much of them.

Constants and Metric System

In many scientific computing problems, numbers are not abstract but reflect the realistic meanings. In other words, these numbers only make sense on top of a well-defined metric system.

What Is a Metric System

For example, when we talk about the distance between two objects, I write down a number 30 , but what does 30 mean in reality? Is it meters, kilometers, miles, or lightyears? Another example, what is the speed of light? Well, this is really depends on what metrics you are using, e.g., `km/s`, `m/s`, `mile/h` ... Things can get really messy in computation if we do not unify the metric system in a numerical library. The translation between different metrics is often important in real-world application. I do not intend to dig deep into the metric system here, so please read online articles to find out more, e.g., [Wiki: Outline of the metric system](#).

Four Metric Systems

There are four metrics adopted in Owl, and all of them are wrapped in the `Owl.Const` module.

- `Const.SI`: International System of Units
- `Const.MKS`: MKS system of units
- `Const.CGS`: Centimetre–gram–second system of units
- `Const.CGSM`: Electromagnetic System of Units

All the metrics defined in these four systems can be found in the interface file `owl_const.mli`.

In general, SI is much newer and recommended to use. International System of Units (French: Système international d'unités, SI), historically also called the MKSA system of units for metre–kilogram–second–ampere. The SI system of units extends the MKS system and has 7 base units, by expressing any measurement of physical quantities using fundamental units of Length, Mass, Time, Electric Current, Thermodynamic Temperature, Amount of substance and Luminous Intensity, which are Metre, Kilogram, Second, Ampere, Kelvin, Mole and Candela respectively. Here is a nice one-page poster from NPL to summarise what have talked about SI.

SI Prefix

As a computer scientist, you must be familiar with prefixes such as `kilo`, `mega`, `giga`. SI system includes the definition of these prefixes as well. But be careful

NPL
National Physical Laboratory

Units of measurement

		Electricity	ampere <small>(A)</small>
Time	second <small>(s)</small>		
The International System (SI) base units are realised at the National Physical Laboratory (NPL). These units are then used throughout the United Kingdom for trade, industry, science and health & safety.		The ampere is that constant current which, if maintained in two straight parallel conductors of infinite length, of negligible circular cross-section, and placed in a vacuum, would produce between these conductors a force equal to 2×10^{-7} newton per metre of length.	
Formally agreed in 1960, the SI is at the centre of all modern science and technology. The definition and realisation of the base and derived units is an active research topic for metrologists with more precise methods being introduced as they become available.		The electrical power generated in a controlled experiment is compared to mechanical power, and using an accurate measurement of resistance the ampere can be calculated ($\text{Power} = (\text{Current})^2 \times \text{Resistance}$).	
Length	metre <small>(m)</small>	Substance	mole <small>(mol)</small>
The second is the duration of 919 631 770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom.		The mole is the amount of substance of a system which contains as many elementary entities as there are atoms in 0.012 kilogramme of carbon 12.	
NPL built the world's first accurate caesium atomic clock in 1968 and the very best atomic clocks in the world today are accurate to within 150 atoms. NPL's atomic clocks help the UK run on time through dissemination of the national time scale and by contributing to Co-ordinated Universal Time.		Note: The carbon 12 atoms are unbound at rest and in their ground state.	
Mass	kilogram <small>(kg)</small>	Light	candela <small>(cd)</small>
The kilogram is the unit of mass; it is equal to the mass of the international prototype of the kilogram.		The candela is the luminous intensity, in a given direction, of a source that emits monochromatic radiation of frequency 540×10^{12} hertz and that has a radiant intensity in that direction of $1/683$ watt per steradian.	
Note: The international prototype of the kilogram is made of platinum (90%) and iridium (10%) and is kept at the International Bureau of Weights and Measures in Paris, France.		At NPL, the candela is realised using the epycopic absolute radiometer, an instrument capable of measuring optical power (in watts) in a laser beam as an absolute quantity. A beam splitter divides the beam into two paths. One path contains a photometer, a detector with a filter to mimic the spectral response of the human eye. The other path contains a tungsten lamp. The optical power is measured by a tungsten lamp (or other types of light source) with an uncertainty of 0.2%.	
The kilogram is the last remaining base unit to be defined as a physical object. All standards of mass must ultimately be traceable to this one object. The search is on in industry and science to find a way of defining the kilogram in terms of a fundamental constant; two key approaches are being pursued; building an electrical kilogram and counting atoms.		The kelvin, unit of thermodynamic temperature, is the fraction 1/273.16 of the thermodynamic temperature of the triple point of water.	
		The triple point of water is the unique temperature at which the three phases of water (solid, liquid and vapour) co-exist. It is fractionally higher than the melting point of water, being 0.01 °C or 273.16 K.	
		Visit our website for lots of measurement-related information: www.npl.co.uk	

Figure 137: Units of measurement in the SI metric system

(especially for computer science guys), the base is 10 instead of 2. These prefixes are defined in the `Const.Prefix` module.

```
Const.Prefix.peta;;
Const.Prefix.tera;;
Const.Prefix.giga;;
Const.Prefix.mega;;
Const.Prefix.kilo;;
Const.Prefix.hecto;;
```

Example: Physics and Math constants

Now we can safely talk about the distance between two objects, light of speed, and a lot of other real-world stuff with atop of a well-defined metric system in Owl. See the following examples.

```
Const.SI.light_year;; (* light year in SI system *)
Const.MKS.light_year;; (* light year in MKS system *)
Const.CGS.light_year;; (* light year in CGS system *)
Const.CGSM.light_year;; (* light year in CGSM system *)
```

How about Planck's constant?

```
Const.SI.plancks_constant_h;; (* in SI system *)
Const.MKS.plancks_constant_h;; (* in MKS system *)
Const.CGS.plancks_constant_h;; (* in CGS system *)
Const.CGSM.plancks_constant_h;; (* in CGSM system *)
```

The table below shows some physical constants that the `si` module includes:

Table 31: Physical constants

Constant name	Explanation
<code>speed_of_light</code>	speed of light in vacuum
<code>gravitational_constant</code>	Newtonian constant of gravitation
<code>plancks_constant_h</code>	Planck constant
<code>plancks_constant_hbar</code>	reduced Planck constant
<code>astronomical_unit</code>	one astronomical unit in meters
<code>light_year</code>	one light year in meters
<code>parsec</code>	one light year in meters
<code>grav_accel</code>	standard acceleration of gravity
<code>electron_volt</code>	electron volt
<code>mass_electron</code>	electron mass
<code>mass_muon</code>	muon mass
<code>mass_proton</code>	proton mass
<code>mass_neutron</code>	neutron mass
<code>rydberg</code>	Rydberg constant
<code>boltzmann</code>	Boltzmann constant
<code>molar_gas</code>	molar gas constant
<code>standard_gas_volume</code>	molar volume of ideal gas (273.15 K, 100 kPa)
<code>bohr_radius</code>	Bohr radius
<code>stefan_boltzmann_constant</code>	Stefan-Boltzmann constant
<code>thomson_cross_section</code>	Thomson cross section in square metre
<code>bohr_magneton</code>	Bohr magneton in Joules per Tesla
<code>nuclear_magneton</code>	Nuclear magneton in Joules per Tesla
<code>electron_magnetic_moment</code>	electron magnetic moment in Joules per Tesla
<code>proton_magnetic_moment</code>	proton magnetic moment in Joules per Tesla
<code>faraday</code>	Faraday constant
<code>electron_charge</code>	electron volt in Joules
<code>vacuum_permittivity</code>	vacuum electric permittivity
<code>vacuum_permeability</code>	vacuum magnetic permeability
<code>debye</code>	one debye in coulomb metre
<code>gauss</code>	one gauss in maxwell per square metre

Some basic mathematical constants are also provided in Owl, though some constants in advanced mathematics are not yet included such as the golden ratio or Euler–Mascheroni constant.

Table 32: Math constants

Constant name	Explanation
pi	Pi
e	Natural constant
euler	Euler constant

Besides these constants, we also provide some frequently used computations based on them, including:

- $\log_2 e$ ($\log_2 e$)
- $\log_{10} e$ ($\log_{10} e$)
- $\log_e 2$
- $\log_e 10$
- $\log_e \pi$
- $\pi_2 (2\pi)$
- $\pi_4 (4\pi)$
- $\pi_{-2} (\pi/2)$
- $\pi_{-4} (\pi/4)$
- $\sqrt{1/2}$
- $\sqrt{2}$
- $\sqrt{3}$
- $\sqrt{\pi}$

International System of Units

Now that you know how to use constants, we will use the International System of Units (SI) module as an example to show the constants we include in Owl. These units are all derived from the seven basic units we have mentioned, and can be categorised according to different application fields.

Time

The base SI unit for time measurement is second.

Table 33: Time units

Constant name	Explanation
minute	one minute in seconds
hour	one hour in seconds
day	one day in seconds
week	one week in seconds

Length

The base SI unit for length measurement is metre.

Table 34: Length units

Constant name	Explanation
inch	one inch in metres
foot	one foot in metres
yard	one yard in metres
mile	one mile in metres
mil	one mil in metres
fathom	one fathom in metres
point	one point in metres
micron	one micron in metres
angstrom	one angstrom in metres
nautical_mile	one nautical mile in metres

Area

Measuring area and volume still relies on SI base unit metre.

Table 35: Area units

Constant name	Explanation
hectare	one hectare in square meters
acre	one acre in square meters
barn	one barn in square meters

Volume

Table 36: Volume units

Constant name	Explanation
liter	one liter in cubic meters
us_gallon	one gallon (US) in cubic meters
uk_gallon	one gallon (UK) in cubic meters
canadian_gallon	one Canadian gallon in cubic meters
quart	one quart (US) in cubic meters
cup	one cup (US) in cubic meters
pint	one pint in cubic meters
fluid_ounce	one fluid ounce (US) in cubic meters
tablespoon	one tablespoon in cubic meters

Speed

The base units for speed are that of time and length.

Table 37: Speed units

Constant name	Explanation
<code>miles_per_hour</code>	miles per hour in metres per second
<code>kilometers_per_hour</code>	kilometres per hour in metres per second
<code>knot</code>	one knot in metres per second

Mass

The base unit for presenting mass is kilogram (kg).

Table 38: Mass units

Constant name	Explanation
<code>pound_mass</code>	one pound (avoirdupois) in kg
<code>ounce_mass</code>	one ounce in kg
<code>metric_ton</code>	1000 kg
<code>ton</code>	one short ton in kg
<code>uk_ton</code>	one long ton in kg
<code>troy_ounce</code>	one Troy ounce in kg
<code>carat</code>	one carat in kg
<code>unified_atomic_mass</code>	atomic mass constant
<code>solar_mass</code>	one solar mass in kg

Force

Measuring force relies on the SI derived unit: `newton`, and one newton equals to 1 kilogram metre per squared second.

Table 39: Force units

Constant name	Explanation
<code>newton</code>	SI derived unit ($kg \cdot m \cdot s^{-2}$)
<code>gram_force</code>	one gram force in newtons
<code>kilogram_force</code>	one kilogram force in newtons
<code>pound_force</code>	one pound force in newtons
<code>poundal</code>	one poundal in newtons
<code>dyne</code>	one dyne in newtons

Energy

The unit of measuring energy level is joule, which equals to one kilogram square metre per square second.

Table 40: Energy units

Constant name	Explanation
joule	SI base unit
calorie	one calorie (thermochemical) in Joules
btu	one British thermal unit (International Steam Table) in Joules
therm	one therm (US) in Joules
erg	one erg in Joules

Power

The unit of power is watts, a SI derived unit. One watts equals to one kilogram square metre per cubic second, or one Joule per second.

Table 41: Power units

Constant name	Explanation
horsepower	one horsepower in watts

Pressure

To measure pressure we often use pascal as a standard unit. One pascal equals to a kilogram per metre per square second, or a newton per square metre.

Table 42: Pressure units

Constant name	Explanation
bar	one bar in pascals
std_atmosphere	standard atmosphere in pascals
torr	one torr (mmHg) in pascals
meter_of_mercury	one metre of mercury in pascals
inch_of_mercury	one inch of mercury in pascals
inch_of_water	one inch of water in pascals
psi	one psi in pascals

Viscosity

The poise is a unit in dynamic viscosity and the stokes is for kinematic viscosity. They are actually included in the CGS-based system for electrostatic units.

Table 43: Viscosity units

Constant name	Explanation
poise	base unit
stokes	base unit

Luminance

Candela is the base unit for luminance, and both lumen and lux are derived units.

Table 44: Luminance units

Constant name	Explanation
stilb	Candela per square metre
lumen	luminous flux, Candela square radian, SI derived unit
phot	base unit
lux	one lux in photos, SI derived unit
footcandle	one footcandle in photos
lambert	base unit
footlambert	one footlambert in lambert

Radioactivity

The SI unit of radioactivity is becquerel, named in honour of the scientist Henri Becquerel, defined as one transformation (or decay or disintegration) per second. The other base units such as ampere, second, and kilogram are also used.

Table 45: Radioactivity units

Constant name	Explanation
curie	one curie in becquerel
roentgen	one ampere second per kilogram
rad	erg per gram

Internal Utility Modules

During development of Owl, we find some utility modules are immensely handy. In this chapter, we share some of them. These are not the main feature of Owl, and perhaps you can implement your own version very quickly. But we hope to present how these features are used in Owl.

Dataset Module

The dataset modules provide easy access to various datasets to be used in Owl, mainly the MNIST and CIFAR10 datasets. You can get all these data in Owl by executing: `Dataset.download_all ()`. The data are downloaded in the home directory, for example, `~/.owl/dataset` on Linux.

MNIST

The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. Each example is of size 28 x 28. It is a good starting point for deep neural network related tasks. You can get MNIST data via these Owl functions:

- `Dataset.load_mnist_train_data ()`: returns a triplet `x, y, y'`.
 - `x` is a [60000, 784] ndarray (`Owl_dense_ndarray.S.mat`) where each row represents a [28, 28] image.
 - `y` is a [60000, 1] label ndarray. Each row is an integer ranging from 0 to 9, indicating the digit on each image.
 - `y'` is a [60000, 10] label ndarray. Each one-hot row vector corresponds to one label.
- `Dataset.load_mnist_test_data ()`: returns a triplet. Similar to `load_mnist_train_data`, only that it returns the test set, so the example size is 10,000 instead of 60,000.
- `Dataset.load_mnist_train_data_arr ()`: similar to `load_mnist_train_data`, but returns `x` as [60000, 28, 28, 1] ndarray
- `Dataset.load_mnist_test_data_arr ()`: similar to `load_mnist_train_data_arr`, but it returns the test set, so the example size is 10, 000 instead of 60, 000.
- `Dataset.draw_samples x y n` draws `n` random examples from images ndarray `x` and label ndarray `y`.

Here is what the dataset looks like when loaded into Owl:

```
# let x, _, y = Dataset.load_mnist_train_data_arr ();;
val x : Owl_dense_ndarray.S.arr =
```

```
C0
R[0,0,0] 0
```

```

R[0,0,1] 0
R[0,0,2] 0
R[0,0,3] 0
R[0,0,4] 0
...
R[59999,27,23] 0
R[59999,27,24] 0
R[59999,27,25] 0
R[59999,27,26] 0
R[59999,27,27] 0

val y : Owl_dense_matrix.S.mat =
  C0 C1 C2 C3 C4 C5 C6 C7 C8 C9
  R0 0 0 0 0 0 1 0 0 0 0
  R1 1 0 0 0 0 0 0 0 0 0
  R2 0 0 0 0 1 0 0 0 0 0
  R3 0 1 0 0 0 0 0 0 0 0
  R4 0 0 0 0 0 0 0 0 0 1
  ...
  R59995 0 0 0 0 0 0 0 0 1 0
  R59996 0 0 0 1 0 0 0 0 0 0
  R59997 0 0 0 0 0 0 1 0 0 0
  R59998 0 0 0 0 0 0 1 0 0 0
  R59999 0 0 0 0 0 0 0 0 1 0

```

You can find the MNIST dataset used in training and testing a DNN in Owl:

```

let train network =
  let x, _, y = Dataset.load_mnist_train_data_arr () in
  Graph.train network x y |> ignore;
  network

let test network =
  let imgs, _, labels = Dataset.load_mnist_test_data () in
  let m = Dense.Matrix.S.row_num imgs in
  let imgs = Dense.Ndarray.S.reshape imgs [|m;28;28;1|] in

  let mat2num x = Dense.Matrix.S.of_array (
    x |> Dense.Matrix.Generic.max_rows
    |> Array.map (fun (_,_,num) -> float_of_int num)
  ) 1 m
  in

  let pred = mat2num (Graph.model network imgs) in
  let fact = mat2num labels in
  let accu = Dense.Matrix.S.(elt_equal pred fact |> sum') in
  Owl_log.info "Accuracy on test set: %f" (accu /. (float_of_int m))

```

CIFAR-10

The CIFAR-10 dataset include small scale color images for more realistic complex image classification tasks. It includes 10 classes of images: aeroplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. It consists of 60,000 32 x 32 colour

images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images.

Due to the limit of file size on GitHub, the training set is cut into 5 smaller batches. You can get CIFAR-10 data using `owl`:

- `Dataset.load_cifar_train_data batch`: returns a triplet `x, y, y'`.
 - The input parameter `batch` can range from 1 to 5, indicating which training set batch to choose.
 - `x` is an `[10000, 32, 32, 3]` ndarray (`owl_dense_ndarray.S.arr`). The last dimension indicates color channels (first Red, then Green, finally Blue).
 - `y` is an `[10000, 1]` label ndarray, each number representing an image class.
 - `y'` is the corresponding `[10000, 10]` one-hot label ndarray.
- `Dataset.load_cifar_test_data ()`: similar to `load_cifar_train_data`, only that it loads test data.
- `Dataset.draw_samples_cifar x y n` draws `n` random examples from images ndarray `x` and label ndarray `y`.

Note that all elements in the loaded matrices and ndarrays are of `float32` format. The CIFAR10 dataset can be loaded in a similar way as MNIST:

```
let train network =
  let x, _, y = Dataset.load_cifar_train_data 1 in
  Graph.train network x y
```

Graph Module

The Graph module in Owl provides a general data structure to manipulate graphs. It is defined as:

```
type 'a node =
  { mutable id : int
  ; (* unique identifier *)
    mutable name : string
  ; (* name of the node *)
    mutable prev : 'a node array
  ; (* parents of the node *)
    mutable next : 'a node array
  ; (* children of the node *)
    mutable attr : 'a (* indicate the validity *)}
```

The attribution here is generic so that you can define your own graph where each node contains an integer, a string, or any data type you define. This makes the graph module extremely flexible.

Graph module provides a rich set of APIs. First, you can build a Graph using these methods:

- `node ~id ~name ~prev ~next attr` creates a node with given id and name string. The created node is also connected to parents in `prev` and children in `next`. The `attr` will be saved in `attr` field.
- `connect parents children` connects a set of parents to a set of children. The created links are the Cartesian product of parents and children. In other words, they are bidirectional links between parents and children. Note that this function does not eliminate any duplicates in the array.
- `connect_descendants parents children` connects parents to their children. This function creates unidirectional links from parents to children. In other words, this function saves `children` to `parent.next` field.
- `connect_ancestors parents children` connects children to their parents. This function creates unidirectional links from children to parents. In other words, this function saves `parents` to `child.prev` field.
- `remove_node x` removes node `x` from the graph by disconnecting itself from all its parent nodes and child nodes.
- `remove_edge src dst` removes a link `src -> dst` from the graph. Namely, the corresponding entry of `dst` in `src.next` and `src` in `dst.prev` will be removed. Note that it does not remove `[dst -> src]` if there exists one.
- `replace_child x y` replaces `x` with `y` in `x` parents. Namely, `x` parents now make link to `y` rather than `x` in `next` field. Note that the function does not make link from `y` to `x` children. Namely, the `next` field of `y` remains intact.

Then, to obtain and update properties of a graph using these functions:

```

val id : 'a node -> int
(** ``id x`` returns the id of node ``x``. *)

val name : 'a node -> string
(** ``name x`` returns the name string of node ``x``. *)

val set_name : 'a node -> string -> unit
(** ``set_name x s`` sets the name string of node ``x`` to ``s``. *)

val parents : 'a node -> 'a node array
(** ``parents x`` returns the parents of node ``x``. *)

val set_parents : 'a node -> 'a node array -> unit
(** ``set_parents x parents`` set ``x`` parents to ``parents``. *)

val children : 'a node -> 'a node array
(** ``children x`` returns the children of node ``x``. *)

val set_children : 'a node -> 'a node array -> unit
(** ``set_children x children`` sets ``x`` children to ``children``. *)

```

```

val attr : 'a node -> 'a
(** ``attr x`` returns the ``attr`` field of node ``x``. *)

val set_attr : 'a node -> 'a -> unit
(** ``set_attr x`` sets the ``attr`` field of node ``x``. *)

```

Similarly, you can get other properties of a graph use the other functions:

- indegree x returns the in-degree of node
- outdegree x returns the out-degree of node
- degree x returns the total number of links of x
- num_ancestor x returns the number of ancestors of x
- num_descendant x returns the number of descendants of x
- length x returns the total number of ancestors and descendants of x

Finally, we provide functions for traversing the graph in either Breadth-First order or Depth-First order. You can also choose to iterate the descendants or ancestors of a given node.

```

val iter_ancestors
  : ?order:order
  -> ?traversal:traversal
  -> ('a node -> unit)
  -> 'a node array
  -> unit
(** Iterate the ancestors of a given node. *)

val iter_descendants
  : ?order:order
  -> ?traversal:traversal
  -> ('a node -> unit)
  -> 'a node array
  -> unit
(** Iterate the descendants of a given node. *)

val iter_in_edges : ?order:order -> ('a node -> 'a node -> unit) -> 'a node array
  -> unit
(** Iterate all the in-edges of a given node. *)

val iter_out_edges : ?order:order -> ('a node -> 'a node -> unit) -> 'a node array
  -> unit
(** Iterate all the out-edges of a given node. *)

val topo_sort : 'a node array -> 'a node array
(** Topological sort of a given graph using a DFS order. Assumes that the graph is
    acyclic.*)

```

You can also use functions: `filter_ancestors`, `filter_descendants`, `fold_ancestors`, `fold_descendants`, `fold_in_edges`, and `fold_out_edges` to perform fold or filter operations when iterating the graph. Within Owl, the Graph module is heavily used to facilitate other modules such as the Computation Graph module to provide

the basic graph structuring functionalities. We will discuss the Computation Graph in detail in later chapter in the book.

Stack and Heap Modules

Both *Stack* and *Heap* are two common abstract data types for collection of elements. They are also used in Owl code. Similar to graph, they use generic types so that any data type can be plugged in. Here is the definition of a stack:

```
type 'a t =
{ mutable used : int
; mutable size : int
; mutable data : 'a array
}
```

The stack and heap modules support four standards operations:

- `push`: push element into a stack/heap
- `pop`: pops the top element in stack/heap. It returns `None` if the container is empty
- `peek`: returns the value of top element in stack/heap but it does not remove the element from the stack. `None` is returned if the container is empty.
- `is_empty`: returns true if the stack/heap is empty

The stack data structure is used in many places in Owl: the `filteri` operation in `ndarray` module, the topological sort in `graph`, the reading file IO operation for keeping the content from all the lines, the data frame... The heap structure is used in key functions such as `search_top_elements` in the `Ndarray` module, which searches the indices of top values in input `ndarray` according to the comparison function.

Count-Min Sketch

Count-Min Sketch is a probabilistic data structure for computing approximate counts. It is particularly ideal for use when the space is limited and exact results are not required. Imagine that we want to count how frequent certain elements are in a realtime stream, what would you do? An intuitive answer is that you can create a hash table, with the element as key and its count as value. The problem with this solution is that the stream could have millions and billions of elements. Even if you somehow manage to cut the long tail (such as the unique elements), the storage requirement is still terribly large.

Now that you think about it, you don't really care about the precise count of an element from the stream. What you really need is an estimation that is not very far away from the true. That leaves space for optimising the solution. First, apply a hashing function and use $h(e)$ as the key, instead of the element e itself. Besides, the total number of key-value pairs can be limited. Towards the end, this approach can be summarised as three steps:

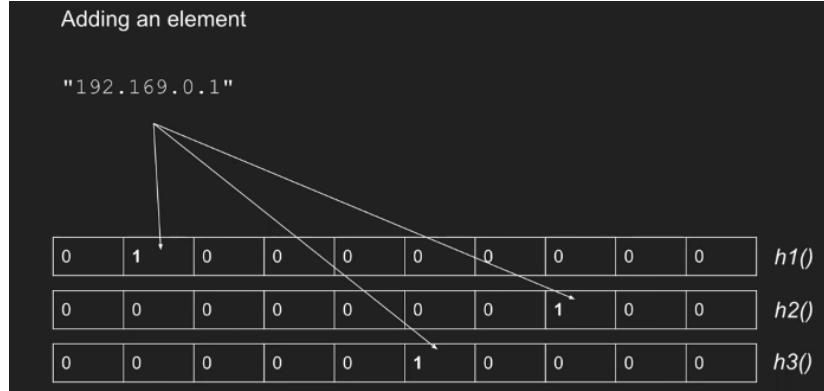


Figure 138: Use Count-Min Sketch method for counting

1. initialise an array of n elements, each set to 0;
2. when processing one element e , increase the count of the hashed index:
 $\text{count}[h(e)] += 1;$
3. when querying the count for certain element, just return $\text{count}[h(e)]$.

Obviously, this approach tends to give an overestimated answer because of the inevitable collision in hash table. Here the *Count-Min Sketch* method comes to help. Its basic idea is simple: follow the process stated above, but the only difference is that now instead of maintaining a vector of length n , we now need to maintain a matrix of shape $d \times n$, i.e. d rows and n columns. Each row is assigned with a different hash function, and when processing one element e , apply h_0, h_1, \dots, h_d to it, and make $\text{count}[i][h_i(e)] += 1$, for each $i = 0, 1, 2, \dots, d$. At any time if you want to know the count of an element e , you again apply the same set of hash functions, retrieve the d counts of this element from all the rows, and choose the smallest count to return. This process is shown in fig. 138 (Src).

In this way, the effect of collision is reduced in the counting. The reason is simple: if these different hashing functions are independent, then the probability that the same element leads to collision in multiple lines can be exponentially reduced with more hash function used.

Even though this method looks like just a heuristic, it actually provides a theoretical guarantee of its counting error. Specifically, we have two error bounds parameter: failure probability σ , and the approximation ratio ϵ , and let s be the sum of all counts stored in the data structure. It can be proved that with a probability of $1 - \sigma$, the error between the estimated count and the true count is ϵs at most. The detailed proof can be seen in the original paper (Cormode and Muthukrishnan 2005). Note that this guarantee implies that elements that appear more frequently will have more accurate counts, since the maximum possible error in a count is linear in the total number of counts in the data

structure.

Owl has provided this probabilistic data structure. It is implemented by Pratap Singh. Owl provides these interfaces for use:

```
module type Sig = sig
  type 'a sketch
  (** The type of Count-Min sketches *)

  (** {6 Core functions} *)

  val init : epsilon:float -> delta:float -> 'a sketch
  (** ``init epsilon delta`` initializes a sketch with approximation ratio
  `` $(1 + \text{epsilon})$ `` and failure probability ``delta``.
  *)

  val incr : 'a sketch -> 'a -> unit
  (** ``incr s x`` increments the frequency count of ``x`` in sketch ``s``
  in-place. *)

  val count : 'a sketch -> 'a -> int
  (** ``count s x`` returns the estimated frequency of element ``x`` in ``s``. *)

  val init_from : 'a sketch -> 'a sketch
  (** ``init_from s`` initializes a new empty sketch with the same parameters as
  ``s``, which
  can later be merged with ``s``.
  *)

  val merge : 'a sketch -> 'a sketch -> 'a sketch
  (** ``merge s1 s2`` returns a new sketch whose counts are the sum of those in ``s1``
  and ``s2``.
  Raises ``INVALID_ARGUMENT`` if the parameters of ``s1`` and ``s2`` do not match.
  *)
end
```

Owl provides two different implementations of the underlying table of counts, one based on the OCaml native array and one based on the Owl ndarray. These are exported as `Owl_base.Countmin_sketch.Native` and `Owl_base.Countmin_sketch.Owl` respectively. In our testing, we have found the OCaml native array to be about 30% faster.

As an example, we can use the count-min sketch to calculate the frequencies of some words in a large corpus. The code below builds a count-min sketch and fills it with text data from here, a corpus of online news articles of about 61 million words. It then tests for the frequencies of some common words and one that doesn't appear. WARNING: this code will download the file `news.txt.gz` (96.5MB) onto your machine and expand it into `news.txt` (340.3MB).

```

module CM = Owl_base.Countmin_sketch.Native

let get_corpus () =
  let fn = "news.txt" in
  if not (Sys.file_exists (Owl_dataset.local_data_path () ^ fn)) then
    Owl_dataset.download_data (fn ^ ".gz");
  open_in (Owl_dataset.local_data_path () ^ fn)

let get_line_words inch =
  let regexp = Str.regexp "[^A-Za-z]+" in
  try
    Some ((input_line inch) |> Str.split regexp)
  with
    End_of_file -> None

let fill_sketch inch epsilon delta =
  let c = CM.init ~epsilon ~delta in
  let rec aux () =
    match get_line_words inch with
    | Some lst -> List.iter (CM.incr c) lst; aux ()
    | None -> c in
  aux ()

let _ =
  let inch = get_corpus () in
  let c = fill_sketch inch 0.001 0.001 in
  let words = ["the"; "and"; "of"; "said"; "floccinaucinihilipilification"] in
  List.iter (fun word -> Printf.printf "%s: %d\n" word (CM.count c word)) words

```

The example output is shown below. It shows that the common words appear with accurate counts, but the word which does not appear in the text gets a positive count.

```

the: 3378663
and: 1289949
of: 1404742
said: 463257
floccinaucinihilipilification: 15540

```

The count-min sketch is a useful data structure when we are interested in approximate counts of important objects in a data set. One such application is to find *heavy hitters*—for example, finding out the most popular web pages given a very long website access log. Formally, the k -heavy-hitters of a dataset are those elements that occur with relative frequency at least $1/k$. So the 100-heavy-hitters are the elements which each appear at least 1% of the time in the dataset.

We can use the count-min sketch, combined with a min-heap, to find the k -heavy-hitters in a particular dataset. The general idea is to maintain in the heap all the current heavy hitters, with the lowest-count heavy hitter at the top. Whenever we get a new element, we add it to the count-min sketch, and then get its count from the sketch. If the relative frequency of that element is greater than $1/k$, we add it to the heap. Then, we check if the current minimum

element in the heap has gone below the relative frequency threshold of $1/k$, and if so remove it from the heap. We repeat this process to remove all heavy hitters whose relative frequency is below $1/k$. So the heap always contains only the elements which have relative frequency at least $1/k$. To get the heavy hitters and their counts, we just get all the elements currently in the heap.

Owl implements this data structure on top of the count-min sketch. The interface is as follows:

```
module type Sig = sig
  type 'a t
  (** Core functions *)
  val init : k:float -> epsilon:float -> delta:float -> 'a t
    (** `init k epsilon delta` initializes a sketch with threshold k, approximation factor epsilon, and failure probability delta.
    *)
  val add : 'a t -> 'a -> unit
    (** `add h x` adds value `x` to sketch `h` in-place. *)
  val get : 'a t -> ('a * int) list
    (** `get h` returns a list of all heavy-hitters in sketch `h`, as a (value, frequency) pair, sorted in decreasing order of frequency.
    *)
end
```

Owl provides two implementations of the heavy-hitters data structure, as `Owl_base.HeavyHitters_sketch.Native` and `Owl_base.HeavyHitters_sketch.Owl`, using the two types of count-min sketch table. As described above, we have found the Native implementation to be faster. An example use of this data structure to find the heavy hitters in the `news.txt` corpus can be found in the Owl examples repository.

Summary

In this chapter, we introduce several internal modules in Owl. First we have the dataset module that provides access to the commonly used datasets such as MNIST and CIFAR10. Then we also have modules that can be used to build common data structures such as graph, stack, and heap. One less-known but interesting data structure is the Count-Min Sketch, which is a probabilistic data structure for computing approximate counts. It is also discussed in detail in this chapter. These modules are not complex and may even not quite necessary in all

numerical libraries, but they may come surprisingly handy when you need them.

Cormode, Graham, and Shan Muthukrishnan. 2005. “An Improved Data Stream Summary: The Count-Min Sketch and Its Applications.” *Journal of Algorithms* 55 (1): 58–75.

DRAFT

Case Studies

DRAFT

Case - Image Recognition

How can a computer take an image and answer questions like “what is in this picture? A cat, dog, or something else?” In the recent few years the machine learning community has been making tremendous progress on tackling this difficult problem. In particular, Deep Neural Network (DNN) is able to achieve reasonable performance on visual recognition tasks – matching or even exceeding human performance in some domains.

We have introduced the neural network module in the previous chapter. In this chapter, we will show one specific example that is built on the neural network module: using the InceptionV3 architecture to perform the image classification task.

Background

InceptionV3 is a widely-used DNN architecture for image classification that can attain significant accuracy with small amount of parameters. It is not invented out of thin air. The development using DNN to perform image recognition is a stream that dates back to more than 20 years ago. During this period, the research in this area is pushed forward again and again in various work. In this chapter, we first introduce how image classification architectures are developed up until Inception. Surveying these related work will help us to understand how Inception architectures are built.

LeNet

In the regression chapter, we have seen a simple neural network that contains three layers, and use it to recognise the simple handwritten numbers from the MNSIT dataset. Here, each pixel acts as an input feature. Recall that each image can be seen as a ndarray. For a black and white image such as the MNSIT image, its bitmap are interpreted as a matrix. Every pixel in the bitmap has a value between 0 and 255. For a colour image, it can be interpreted as a 3-dimension array with three channels, each corresponding to the blue, green, and red layer.

However, we cannot rely on adding more fully connected layers to do real world high precision image classification. One important improvement that the Convolution Neural Network makes is that it uses filters in the convolution operation. As a result, instead of using the whole image as an array of features, the image is divided into a number of tiles. They will then serve as the basic feature of the network’s prediction.

The next building block is the pooling layer. Recall from the neural network chapter that, both average pooling and max pooling can aggregate information from multiple pixels into one and “blur” the input image or feature. So why is it so important? By reducing the size of input, pooling helps to reduce the number of parameters and the amount of computation required. Besides, blurring the features is a way to avoid over-fitting training data. In the end, only connect

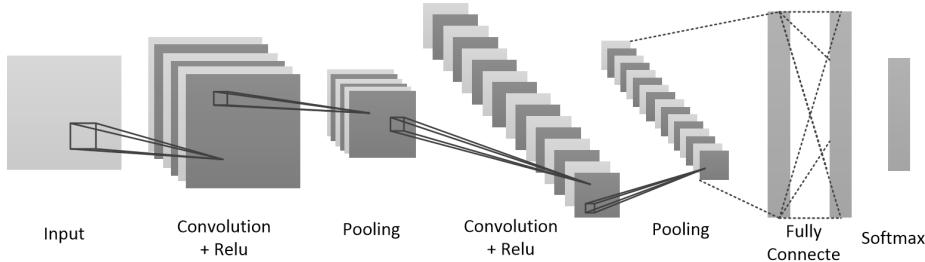


Figure 139: Workflow of image classification

high-level features with fully connection. This is the structure proposed in (LeCun et al. 1998). This whole process can be shown in fig. 139.

In general, layers of convolution retrieve information from detailed to more abstract gradually. In a DNN, The lower layers of neurons retrieve information about simple shapes such as edges and points. Going higher, the neurons can capture complex structures, such as the tire of a car, the face of a cat, etc. Close to the top layers, the neurons can retrieve and abstract complex ideas, such as “car”, “cat”, etc. And then finally generates the classification results.

AlexNet

Next breakthrough comes from the AlexNet proposed in (Krizhevsky, Sutskever, and Hinton 2012). The authors introduce better *non-linearity* in the network with the ReLU activation. Operations such as convolution include mainly linear operations such as matrix multiplication and add. But that's not how the real world data looks like. Recall that from the previous Regression chapter, even though linear regression can be effective, for most of the real world application we need more complex method such as polynomial regression. The same can be applied here. We need to increase the non-linearity to accommodate real world data such as image.

There are multiple activation choices, such as `tanh` and `sigmoid`. Compared to the previous activation functions such as `tanh` ($f(x) = \tanh(x)$), or `sigmoid` ($f(x) = (1 + e^{-x})^{-1}$), the computation of ReLU is simple: $f(x) = \max(0, x)$. In terms of training time with gradient descent, ReLU is much faster than the aforementioned two computational expensive functions, and thus more frequently used. The accuracy loss in gradient computation is small, and it also makes the training faster. For example, (Krizhevsky, Sutskever, and Hinton 2012) shows that, a four-layer convolutional neural network with ReLUs reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with the `tanh` neurons. Besides, in practice, networks with ReLU tend to show better convergence performance than `sigmoid`.

Another thing that AlexNet proposes is to use the `dropout` layer. It is mainly

used to solve the over-fitting problem. This operation only works during training phase. It randomly selects some elements in the input ndarray and set their values to zero, thus “deactivates” the knowledge that can be learnt from these points. In this way, the network intentionally drops certain part of training examples and avoids the over-fitting problem. It is similar to the regularisation method we use in the linear regression.

The one more thing that we need to take a note from AlexNet is that by going deeper make the network “longer”, we achieve better accuracy. So instead of just convolution and pooling, we now build convolution followed by convolution and then pooling, and repeat this process again.... A deeper network captures finer features, and this would be a trend that is followed by successive architectures.

VGG

The VGG network proposed in (Simonyan and Zisserman 2014) is the next step after AlexNet. The most notable change introduced by VGG is that it uses small kernel sizes such as 3×3 instead of the 11×11 with a large stride of 4 in AlexNet. Using multiple small kernels is much more flexible than only using a large one. For example, for an input image, by applying two 3×3 kernels with slide size of 1, that equals to using a 5×5 kernel. If stacking three 3×3 , it equals using one 7×7 convolution.

By replacing large kernels with multiple small kernels, the number of parameter is visibly reduced. In the previous two examples, replace one 5×5 with two 3×3 , we reduce the parameter by $1 - 2 * 3 * 3 / (5 * 5) = 28\%$. Replace the 7×7 kernel, and we save parameters by $1 - 3 * 3 * 3 / (7 * 7) = 45\%$.

Therefore, with this reduction of parameter size, we can now build network with more layers, which tends to yield better performance. The VGG networks come with two variates, VGG16 and VGG19, which are the same in structure, and the only difference is that VGG19 is deeper. The code to build a VGG16 network with Owl is shown in (Zhao and Wang 2019d).

One extra benefit is that using small kernels increases non-linearity. Imagine an extreme case where the kernel is as large as the input image, then the whole convolution is just one big matrix multiplication, a complete linear operation. As we have just explained in the previous section, we hope to reduce the linearity in training CNN to accommodate more real-world problems.

ResNet

We keep saying that building deeper network is the trend. However, going deeper has its limit. The deeper you go, the more you will experience the “vanishing gradient” problem. This problem is that, in a very deep network, during the back-propagation phase, the repeated multiplication operations will make the gradients very small, and thus the performance affected.

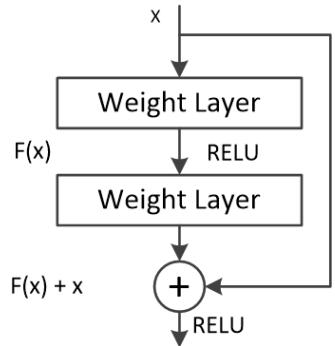


Figure 140: Residual block in the ResNet

The ResNet in (He et al. 2016) proposes an “identity shortcut connection” that skips one or more layers and combines with predecessor layers. It is called a residual block, as shown in fig. 140.

We can see that there is the element-wise addition that combines the information of the current output and its predecessors two layers before. It solves the gradient problem in stacking layers, since now the error can be backpropagated through multiple paths. The authors show that during training the deeper layers do not produce an error higher than its predecessor in lower layer.

Also note that the residual block aggregate features from different level of layers, instead of purely stacking them. This pattern proves to be useful and will also be used in the Inception architecture. The ResNet can also be constructed by stacking for different layers, so that we have ResNet50, ResNet101, ResNet152, etc. The code to build a ResNet50 network with Owl is shown in (Zhao and Wang 2019b).

SqueezeNet

All these architectures, including Inception, mainly aim to push the classification accuracy forward. However, at some point we are confronted with the tradeoff between accuracy and model size. We have seen that sometimes reducing the parameter size can help the network to go deeper, and thus tends to give better accuracy. However, with the growing trend of edge computing, there is requirement for extremely small deep neural networks so that it can be easily distributed and deployed on less powerful devices. For that, sacrificing a bit of accuracy is acceptable.

There are more and more efforts towards that direction, and the SqueezeNet (Iandola et al. 2016) is one of them. It claims to have the AlexNet level of accuracy, but with 50 times fewer parameters.

There are mainly three design principles for the SqueezeNet. The first one is what

we are already familiar with: using a lot of 1x1 convolutions. The second one also sounds familiar. The 1x1 convolutions are used to reduce the output channels before feeding the result to the more complex 3x3 convolutions. Therefore the 3x3 convolution can have smaller input channels and thus smaller parameter size. These two principles are incorporated into a basic building block called “fire module”.

Now that we can have a much smaller network, what about the accuracy? We don’t really want to discard the accuracy requirement totally. The third design principle in SqueezeNet is to delay the down-sampling to later in the network. Recall that the down-sampling layers such as maxpooling “blur” the information intentionally. The intuition is that, if we only use them occasionally and in the deeper layer of the network, we can have large activation maps that preserve more information and make the detection result more accurate. The code to build a SqueezeNet network with Owl is shown in (Zhao and Wang 2019c).

Capsule Network

The research on image detection network structures is still on-going. Besides the parameter size and detection accuracy, more requirements are proposed. For example, there is the problem of recognising an object, e.g. a car, from different perspective. And the “Picasso problem” in image recognition where some feature in an object is intentionally distorted or misplaced. These problems show one deficiency in the existing image classification approach: the lack of connection between features. It may recognise a “nose” feature, and an “eye” feature, and then the object is recognised as a human face, even though the nose is perhaps above the eyes. The “Capsule network” is proposed to address this problem. Instead of using only scalar to represent feature, it uses a vector that includes more information such as orientation and object size etc. The “capsule” utilises these information to capture the relative relationship between features.

There are many more networks that we cannot cover them one by one here, but hopefully you can see that there are some common themes in the development of image recognition architectures. Next, we will come to the main topic of this chapter: how InceptionV3 is designed based on these previous work.

Building InceptionV3 Network

Proposed by Christian Szegedy et. al., InceptionV3 is one of Google’s latest efforts to perform image recognition. It is trained for the ImageNet Large Visual Recognition Challenge. This is a standard task in computer vision, where models try to classify entire images into 1000 classes, like “Zebra”, “Dalmatian”, and “Dishwasher”, etc. Compared with previous DNN models, InceptionV3 is one of the most complex neural network architectures in computer vision.

The design of image recognition networks is about the trade-off between computation cost, memory usage, and accuracy. Just increasing model size and computation cost tend to increase the accuracy, but the benefit will diminish

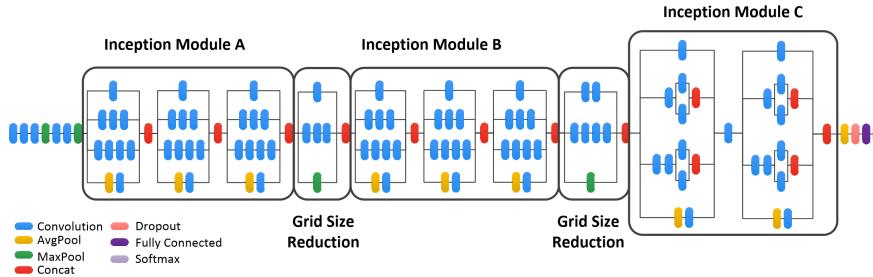


Figure 141: Network Architecture of InceptionV3

soon. To solve this problem, compared to previous similar networks, the Inception architecture aims to perform well with strict constraints on memory and computational budget. This design follows several principles, such as balancing the width and depth of the network, and performing spatial aggregation over lower dimensional embeddings which may lead to small loss in representational power of networks. The resulting Inception network architectures have high performance and a relatively modest computation cost compared to simpler, more monolithic architectures. fig. 141 shows the overall architecture of this network:

We can see that the whole network can be divided into several parts, and the inception module A, B, and C are both repeated based on one structure. That's where the name "Inception" comes from: like dreams, you can have stacked these basic units layer by layer.

InceptionV1 and InceptionV2

The reason we say "InceptionV3" is because it is developed based on two previous similar architectures. To understand InceptionV3, we first need to know the characteristics of its predecessors.

The first version of Inception, GoogLeNet (Szegedy et al. 2015), proposes to combine convolutions with different filter sizes on the same input, and then concatenate the resulting features together. Think about an image of a bird. If you stick with using a normal square filter, then perhaps the features such as "feather" is a bit difficult to capture, but easier to do if you use a "thin" filter with a size of e.g. 1×7 . By aggregating information from applying different features, we can extract features from multi-level at each step.

Of course, adding extra filters increases computation complexity. To remedy this effect, the Inception network proposes to utilise the 1×1 convolution to reduce the dimensions of feature maps. For example, we want to apply a 3×3 filter to input ndarray of size $[1; 300; 300; 768]$ and the output channel should be 320. Instead of applying a convolution layer of $[3; 3; 768; 320]$ directly, we first reduce the dimension to, say, 192 by using a small convolution layer $[1; 1; 768;$

`[192]`, and then apply a `[13; 3; 192; 320]` convolution layer to get the final result. By reducing input dimension before the more complex computation with large kernels, the computation complexity is reduced.

For those who are confused about the meaning of this array: recall from previous chapter that the format of an image as ndarray is `[batch; column; row; channel]`, and that the format of a convolution operation is mainly represented as `[kernel_column; kernel_row; input_channel; output_channel]`. Here we ignore the other parameters such as slides and padding since here we focus only on the change of channels of feature map.

In an updated version of GoogLeNet, the InceptionV2 (or BN-inception), utilises the “Batch Normalisation” layer. We have seen how normalising input data plays a vital role in improving the efficiency of gradient descent in the Optimisation chapter. Batch normalisation follows a similar path, only at it now works between each layer instead of just at the input data. This layer rescales each mini-batch with the mean and variance of this mini-batch.

Imagine that we train a network to recognise horse, but most of the training data are actually black or blown horse. Then the network’s performance on white horse might not be quite ideal. This again leads us back to the over-fitting problem. The batch normalisation layer adds noise to input by scaling. As a result, the content at deeper layer is less sensitive to content in lower layers. Overall, the batch normalisation layer greatly improves the efficiency of training.

Factorisation

Now that we understand how the image recognition architectures are developed, finally it’s time to see who these factors are utilised into the InceptionV3 structure.

```
open Owl
open Owl_types
open Neural.S
open Neural.S.Graph

let conv2d_bn ?(padding=SAME) kernel stride nn =
  conv2d ~padding kernel stride nn
  |> normalisation ~training:false ~axis:3
  |> activation Activation.Relu
```

Here the `conv2d_bn` is a basic building block used in this network, consisting of a convolution layer, a normalisation layer, and a relu activation layer. We have already introduced how these different types of layer work. You can think of `conv2d_bn` as an enhanced convolution layer.

Based on this basic block, the aim in building Inception network is still to go deeper, but here the authors introduces the three types of *Inception Modules* as a unit of stacking layers. Each module factorise large kernels into smaller ones. Let’s look at them one by one.

```

let mix_typ1 in_shape bp_size nn =
  let branch1x1 = conv2d_bn [|1;1;in_shape;64|] [|1;1|] nn in
  let branch5x5 = nn
    |> conv2d_bn [|1;1;in_shape;48|] [|1;1|]
    |> conv2d_bn [|5;5;48;64|] [|1;1|]
  in
  let branch3x3dbl = nn
    |> conv2d_bn [|1;1;in_shape;64|] [|1;1|]
    |> conv2d_bn [|3;3;64;96|] [|1;1|]
    |> conv2d_bn [|3;3;96;96|] [|1;1|]
  in
  let branch_pool = nn
    |> avg_pool2d [|3;3|] [|1;1|]
    |> conv2d_bn [|1;1;in_shape; bp_size |] [|1;1|]
  in
  concatenate 3 [|branch1x1; branch5x5; branch3x3dbl; branch_pool|]

```

The `mix_typ1` structure implement the first type of inception module. In `branch3x3dbl` branch, it replaces a 5×5 kernel convolution layer with two 3×3 convolution layers. It follows the design in the VGG network. Of course, as we have explained, both branches uses the 1×1 to reduce dimensions before complex convolution computation.

In some implementation the 3×3 convolutions can also be further factorised into 3×1 and then 1×3 convolutions. There are more than one way to do the factorisation. You might be thinking that it is also a good idea to replace the 3×3 convolution with two 2×2 s. Well, we could do that but it saves only about 11% parameters, compared to the 33% save of current practice.

```

let mix_typ4 size nn =
  let branch1x1 = conv2d_bn [|1;1;768;192|] [|1;1|] nn in
  let branch7x7 = nn
    |> conv2d_bn [|1;1;768;size|] [|1;1|]
    |> conv2d_bn [|1;7;size;size|] [|1;1|]
    |> conv2d_bn [|7;1;size;192|] [|1;1|]
  in
  let branch7x7dbl = nn
    |> conv2d_bn [|1;1;768;size|] [|1;1|]
    |> conv2d_bn [|7;1;size;size|] [|1;1|]
    |> conv2d_bn [|1;7;size;size|] [|1;1|]
    |> conv2d_bn [|7;1;size;size|] [|1;1|]
    |> conv2d_bn [|1;7;size;192|] [|1;1|]
  in
  let branch_pool = nn
    |> avg_pool2d [|3;3|] [|1;1|]
    |> conv2d_bn [|1;1; 768; 192|] [|1;1|]
  in
  concatenate 3 [|branch1x1; branch7x7; branch7x7dbl; branch_pool|]

```

As shown in the code above, `mix_typ4` shows the Type B inception module, another basic unit. It still separate into three branches and then concatenate

them together. The special about this this type of branch is that it factorise a 7×7 convolution into the combination of a 7×1 and then a 1×7 convolution. Again, this change saves $(49 - 14)/49 = 71.4\%$ parameters. If you have doubt about this replacement, you can do a simple experiment:

```
open Neural.S
open Neural.S.Graph

let network_01 =
  input [|28;28;1|]
  |> conv2d ~padding:VALID [|7;1;1|] [|1;1|]
  |> get_network

let network_02 =
  input [|28;28;1|]
  |> conv2d ~padding:VALID [|7;1;1|] [|1;1|]
  |> conv2d ~padding:VALID [|1;7;1|] [|1;1|]
  |> get_network
```

Checking the output log, you can find out that both networks give the same output shape. This factorisation is intuitive: convolution of size does not change the output shape at that dimension. Then the two convolutions actually performs feature extraction along each dimension (height and width) respectively.

```
let mix_typ9 input nn =
  let branch1x1 = conv2d_bn [|1;1;input;320|] [|1;1|] nn in
  let branch3x3 = conv2d_bn [|1;1;input;384|] [|1;1|] nn in
  let branch3x3_1 = branch3x3 |> conv2d_bn [|1;3;384;384|] [|1;1|] in
  let branch3x3_2 = branch3x3 |> conv2d_bn [|3;1;384;384|] [|1;1|] in
  let branch3x3 = concatenate 3 [|branch3x3_1; branch3x3_2|] in
  let branch3x3dbl = nn |> conv2d_bn [|1;1;input;448|] [|1;1|] |> conv2d_bn
    [|3;3;448;384|] [|1;1|] in
  let branch3x3dbl_1 = branch3x3dbl |> conv2d_bn [|1;3;384;384|] [|1;1|] in
  let branch3x3dbl_2 = branch3x3dbl |> conv2d_bn [|3;1;384;384|] [|1;1|] in
  let branch3x3dbl = concatenate 3 [|branch3x3dbl_1; branch3x3dbl_2|] in
  let branch_pool = nn |> avg_pool2d [|3;3|] [|1;1|] |> conv2d_bn
    [|1;1;input;192|] [|1;1|] in
  concatenate 3 [|branch1x1; branch3x3; branch3x3dbl; branch_pool|]
```

The final inception module is a bit more complex, but by now you should be able to understand its construction. It aggregate information from four branches. The 1×1 convolutions are used to reduce the dimension and computation complexity. Note that in both the `branch3x3` and `branch3x3dbl` branches, both 3×1 and 1×3 convolutions are used, not as replacement of 3×3 , but as separate branches. This module is for promoting high dimensional representations.

Together, these three modules make up the core part of the InceptionV3 architecture. By applying different techniques and designs, these modules take less memory and less prone to over-fitting problem. Thus they can be stacked together to make the whole network go deeper.

Grid Size Reduction

For the very beginning of the design of CNN, we need to reduce the size of feature maps as well as increasing the number or channel of the feature map. We have explained how it is done using the combination of pooling layer and convolution layer. However, the reduction solution constructed this way is either too greedy or two computationally expensive.

Inception architecture proposes a grid size reduction module. It put the same input feature map into two set of pipelines, one of them uses the pooling operation, and the other uses only convolution layers. These two types of layers are then not stacked together but concatenated vertically, as shown in the next part of code.

```
let mix_typ3 nn =
  let branch3x3 = conv2d_bn [|3;3;288;384|] [|2;2|] ~padding:VALID nn in
  let branch3x3dbl = nn
    |> conv2d_bn [|1;1;288;64|] [|1;1|]
    |> conv2d_bn [|3;3;64;96|] [|1;1|]
    |> conv2d_bn [|3;3;96;96|] [|2;2|] ~padding:VALID
  in
  let branch_pool = max_pool2d [|3;3|] [|2;2|] ~padding:VALID nn in
  concatenate 3 [|branch3x3; branch3x3dbl; branch_pool|]
```

`mix_typ3` builds the first grid size reduction module. This replacement strategy can perform efficient reduction without losing too much information. Similarly, an extended version of this grid size reduction module is also included.

```
let mix_typ8 nn =
  let branch3x3 = nn
    |> conv2d_bn [|1;1;768;192|] [|1;1|]
    |> conv2d_bn [|3;3;192;320|] [|2;2|] ~padding:VALID
  in
  let branch7x7x3 = nn
    |> conv2d_bn [|1;1;768;192|] [|1;1|]
    |> conv2d_bn [|1;7;192;192|] [|1;1|]
    |> conv2d_bn [|7;1;192;192|] [|1;1|]
    |> conv2d_bn [|3;3;192;192|] [|2;2|] ~padding:VALID
  in
  let branch_pool = max_pool2d [|3;3|] [|2;2|] ~padding:VALID nn in
  concatenate 3 [|branch3x3; branch7x7x3; branch_pool|]
```

`mix_typ8` is the second grid size reduction module in the deeper part of the network. It uses three branches instead of two, and each convolution branch is more complex. The 1×1 convolutions are again used. But in general it still follows the principle of performing reduction in parallel and then concatenates them together, performing an efficient feature map reduction.

InceptionV3 Architecture

After introducing the separate units, finally we can construct them together into the whole network in code.

```
let make_network img_size =
  input [|img_size;img_size;3|]
  |> conv2d_bn [|3;3;3;32|] [|2;2|] ~padding:VALID
  |> conv2d_bn [|3;3;32;32|] [|1;1|] ~padding:VALID
  |> conv2d_bn [|3;3;32;64|] [|1;1|]
  |> max_pool2d [|3;3|] [|2;2|] ~padding:VALID
  |> conv2d_bn [|1;1;64;80|] [|1;1|] ~padding:VALID
  |> conv2d_bn [|3;3;80;192|] [|1;1|] ~padding:VALID
  |> max_pool2d [|3;3|] [|2;2|] ~padding:VALID
  |> mix_typ1 192 32 |> mix_typ1 256 64 |> mix_typ1 288 64
  |> mix_typ3
  |> mix_typ4 128 |> mix_typ4 160 |> mix_typ4 160 |> mix_typ4 192
  |> mix_typ8
  |> mix_typ9 1280 |> mix_typ9 2048
  |> global_avg_pool2d
  |> linear 1000 ~act_typ:Activation.(Softmax 1)
  |> get_network
```

There is only one last thing we need to mention: the *global pooling*. Global Average/Max Pooling calculates the average/max output of each feature map in the previous layer. For example, if you have an $[|1;10;10;64|]$ feature map, then this operation can make it to be $[|1;1;1;64|]$. This operation seems very simple, but it works at the very end of a network, and can be used to replace the fully connection layer. The parameter size of the fully connection layer has always been a problem. Now that it is replaced with a non-trainable operation, the parameter size is greatly reduced without the performance. Besides, the global pooling layer is more robust to spatial translations in the data and mitigates the over-fitting problem in fully connection, according to (Lin, Chen, and Yan 2013).

The full code is listed in (Zhao and Wang 2019a). Even if you are not quite familiar with Owl or OCaml, it must still be quite surprising to see the netwrok that contains 313 neuron nodes can be constructed using only about 150 lines of code. And we are talking about one of the most complex neural networks for computer vision.

Besides InceptionV3, you can also easily construct other popular image recognition networks, such as ResNet50, VGG16, SqueezeNet etc. with elegant Owl code. We have already mentioned most of them in previous sections.

Preparing Weights

Only building a network structure is not enough. Another important aspect is proper weights of a neural network. It can be achieved by training on GBs of image data for days or longer on powerful machine clusters.

The training is usually done via supervised learning using a large set of labelled images. Although Inception v3 can be trained from many different labelled image sets, ImageNet is a common dataset of choice. In this image dataset, the basic unit is called “synset”, which is a concept that is described by multiple words or word phrases. ImageNet aims to provide about 1,000 images for each of the 100,000 synset in WordNet. In training the InceptionV3 model, it takes more than 1 million images from ImageNet. The training of this model can take hundreds of hours of training on multiple high-performance GPUs.

However, not everyone has access to such large resource. Another option is more viable: importing weights from existed pre-trained TensorFlow models, which are currently widely available in model collections such as this one.

The essence of weights is list of ndarrays, which is implemented using Bigarray in OCaml. So the basic idea is to find a intermediate representation so that we can exchange the ndarray in NumPy and Bigarray in OCaml. In our implementation, we choose to use the HDF5 as this intermediate data exchange format. In Python, we use the h5py library, and in OCaml we use hdf5-ocaml.

The method to save or load hdf5 data files are fixed, but the methods to retrieve data from model files vary depending on the type of original files. For example, if we choose to import weight form a TensorFlow model, we do something like this to achieve the weight data of each layer:

```
...
reader = tf.train.NewCheckpointReader(checkpoint_file)
for key in layer_names:
    data=reader.get_tensor(key).tolist()
...

```

In a keras, it's a bit more straightforward:

```
...
for layer in model.layers:
    weights = layer.get_weights()
...

```

In the OCaml side, we first create a Hashtbl and read all the HDF5 key-value pairs into it. Each value is saved as a double precision Owl ndarray.

```
...
let h = Hashtbl.create 50 in
let f = H5.open_ronly h5file in
for i = 0 to (Array.length layers - 1) do
  let w = H5.read_float_genarray f layers.(i) C_layout in
  Hashtbl.add h layers.(i) (Dense.Ndarray.Generic.cast_d2s w)
done;
...
```

And then we can use the mechanism in the Neural Network model to load these values from the hashtable to networks:

```

...
let wb = Neuron.mkpar n.neuron in
Printf.printf "%s\n" n.name;
wb.(0) <- NeuronArr (Hashtbl.find layers n.name);
Neuron.update n.neuron wb
...
```

It is very important to make clear the difference in naming of each layer in different platforms, since the creator of the original model may choose any name for each layer. Other differences have also to be taken care of. For example, the `beta` and `gamma` weights in the batch normalisation layer are represented as two different values in TensorFlow model, but they belong to the same layer in Owl. Also, some times the dimensions have to be swapped in an ndarray during this weight conversion.

Note that this is one-off work. Once you successfully update the network with weights, the weights can be saved using `Graph.save_weights`, without having to repeat all these steps again. We have already prepared the weights for the InceptionV3 model and other similar models, and the users don't have to worry about these trivial model exchanging details.

Processing Image

Image processing is challenging, since OCaml does not provide powerful functions to manipulate images. Though there are image processing libraries such as `CamlImages`, but we don't want to add extra liabilities to Owl itself.

To this end, we choose the non-compressed image format PPM. A PPM file is a 24-bit color image formatted using a text format. It stores each pixel with a number from 0 to 65536, which specifies the color of the pixel. Therefore, we can just use `ndarray` in Owl and convert that directly to PPM image without using any external libraries. We only need to take care of header information during this process.

For example, here is the code for converting a 3-dimensional array in Owl `img` into a ppm file. We first need to get the content from each of the three colour channels using slicing, such as the blue channel:

```

let b = N.get_slice [[[]];[];[0]] img in
let b = N.reshape b [|h; w|] in
...
```

Here `h` and `w` are the height and width of the image. Then we need to merge all three matrix into a large matrix that of size `[|h; 3*w|]`.

```

let img_mat = Dense.Matrix.S.zeros h (3 * w) in
Dense.Matrix.S.set_slice [[];[0;-1;3]] img_mat r;
...

```

Then, after rotating this matrix by 90 degrees, we need to rewrite this matrix to a large byte variable. Note that though the method is straightforward, you need to be careful about the index of each element during the conversion.

```

let img_str = Bytes.make (w * h * 3) ' ' in
let ww = 3 * w in
for i = 0 to ww - 1 do
  for j = 0 to h - 1 do
    let ch = img_arr.(i).(j) |> int_of_float |> char_of_int in
    Bytes.set img_str ((h - 1 - j) * ww + i) ch;
  done
done;

```

Finally we build another byte string that contains the metadata such as height and width, according to the specification of PPM format. Concatenating the metadata and data together, we then write the bytes data into a file.

Similarly, to read a PPM image file into ndarray in Owl, we treat the ppm file line by line with `input_line` function. The metadata such as version and comments are ignored. We get the metadata such as width and height from the header.

```

...
let w_h_line = input_line fp in
let w, h = Scanf.sscanf w_h_line "%d %d" (fun w h -> w, h) in
...

```

Then according this information, we read the rest of data into large bytes with `Pervasives.really_input`. Note that in 32bit OCaml, this will only work when reading strings up to about 16 megabytes.

```

...
let img = Bytes.make (w * h * 3) ' ' in
really_input fp img 0 (w * h * 3);
close_in fp;
...

```

Then we need to re-arrange the data in the bytes into a matrix.

```

let imf_o = Array.make_matrix (w * 3) h 0.0 in
let ww = 3 * w in
for i = 0 to ww - 1 do

```

```

for j = 0 to h - 1 do
    imf_o.(i).(j) <- float_of_int (int_of_char (Bytes.get img ((h - 1 - j ) * ww +
                                                i)));
done
done;

```

This matrix can then be further processed into an ndarray with proper slicing.

```

...
let m = Dense.Matrix.S.of_arrays img in
let m = Dense.Matrix.S.rotate m 270 in
let r = N.get_slice [[];[0;-1;3]] m in
let g = N.get_slice [[];[1;-1;3]] m in
let b = N.get_slice [[];[2;-1;3]] m in
...

```

There are other functions such as reading in ndarray from PPM file. The full image processing code can be viewed in this gist.

Of course, most of the time we have to deal with image of other more common format such as PNG and JPEG. For the conversion from these formats to PPM or the other way around, we use the tool ImageMagick. ImageMagick is a free and open-source tool suite for image related tasks, such as converting, modifying, and editing images. It can read and write over 200 image file formats, including the PPM format. Therefore, we preprocess the images by converting its format to PPM with the command `convert`. Also, the computation time of is related with the input size, and we often hope to limit the size of images. That can also be done with the command `convert -resize`.

Another important preprocessing is to normalise the input. Instead of processing the input ndarray whose elements ranges from 0 to 255, we need to simply preprocess it so that all the elements fall into the range [-1, 1], as shown in the code below.

```

let normalise img =
  let img = Arr.div_scalar img 255. in
  let img = Arr.sub_scalar img 0.5 in
  let img = Arr.mul_scalar img 2. in
  img

```

Running Inference

Now that we have built the network, and loaded the proper weights, it's time to perform the most exciting and actually the easiest part: inferencing. Actually the most important part is just one function: `Graph.model`. In this section we hope to show how to build a service around this function so that a user can perform all the inference steps, from input image to output classification result, with as simple command as possible.

First let's look at the code:

```
#!/usr/bin/env owl
open Owl

#zoo "9428a62a31dbea75511882ab8218076f"

let _ =
  let img = "panda.png" in
  let labels = InceptionV3.infer img in
  let top = 5 in
  let labels_json = InceptionV3.to_json ~top labels in
  let labels_tuples = InceptionV3.to_tuples labels in

  Printf.printf "\nTop %d Predictions:\n" top;
  Array.iteri (fun i x ->
    let cls, prop = x in
    Printf.printf "Prediction #d (%.2f%%) : %s\n" i (prop *. 100.) cls;
  ) labels_tuples
```

The code itself is quite simple. First, we need to build the whole Inception network structure and loading weight etc., which is what we have already done in this gist. All we need is to loading with the `#zoo` module in Owl. The `InceptionV3` module provides three APIs:

- `infer`: Service that performs image recognition tasks over client images. It accepts a string that specifies the location of a local image. Its return value is a 1×1000 N-dimension array, each element is a float number between 0 and 1, indicating the possibility that the image belongs to one of the 1000 classes from ImageNet.
- `to_json`: Convert the inferred result to a raw JSON string. Parameter `top`: an int value to specify the top-N likeliest labels to return. Default value is 5.
- `to_tuples`: Convert the inferred result to an array of tuples, each tuple contains label name ("class", string) and the probability ("prop", float, between 0 and 1) of target image being in that class.

After loading the inception model with `#zoo` primitive, the user needs to designate an absolute path of the local input image. Here we use the `extend_zoo_path` utility function to automatically find the "panda.png" image contained in the Gist itself. But surely a user can have her own choice. It can be of any common image format (jpg, png, gif, etc.) and size. We provide this panda picture just in case you running short of images currently.

With these work done, we can run inference with the neural network model and the input image by calling the `infer` API from `InceptionV3` module, which wraps around the `Graph.model` function.

Then we need to decode the result, getting top-N (N defaults to 5) predictions



Figure 142: Panda image that is used for image recognition task

in human-readable format. The output is an array of tuple. Each tuple consists of a string for classified type description, and a float number ranging from 0 to 100 to represent the percentage probability of the input image actually being this type. Finally, if you want, you can pretty-print the result.

Now that the whole script is ready, we can wrap it into a zoo gist. Then all it takes for the user is just one line:

```
owl -run 6dfed11c521fb2cd286f2519fb88d3bf
```

That's it. This one-liner is all you need to do to see an image classification example in action. Here is the output (assume using the panda image previously mentioned):

```
Top 5 Predictions:
Prediction #0 (96.20%) : giant panda, panda, panda bear, coon bear, Ailuropoda
    melanoleuca
Prediction #1 (0.12%) : lesser panda, red panda, panda, bear cat, cat bear,
    Ailurus fulgens
Prediction #2 (0.06%) : space shuttle
Prediction #3 (0.04%) : soccer ball
Prediction #4 (0.03%) : indri, indris, Indri indri, Indri brevicaudatus
```

If you are not interested in installing anything, here is a web-based demo of this image classification application powered by Owl. Please feel free to play with it! And the server won't store your image. Actually, if you are so keen to protect your personal data privacy, then you definitely should try to pull the code here and fast build a local image processing service without worrying your images being seen by anybody else.

Applications

Building an image classification application is not the end by itself. It can be used in a wide range of applications.

- Face recognition. If you have a Facebook account, it must be very familiar to you how your friends are tagged in your uploaded photo.
- Photo organisation. Take Eden Photo, a Mac application, for an example. It automatically provides tags and keywords to a photo, so as to provide powerful photo categorisation and enhanced user experience.
- Self-driving cars. The image recognition is no doubt the core part in self-driving technology, by identifying objects on the road.
- Medical imaging. This technology can be used to enhance cancer detection and MRIs etc. by automatically detect signs diseases in medical images.

That's just a small number of examples of the usage of image recognition. Actually, image recognition is often one fundamental step for many computer vision tasks, including the instance segmentation and image style transfer we will talk about in the next chapters. That's why Google and Microsoft both provide their cloud-based Vision APIs.

Summary

In this chapter, we first review how the existing state-of-art DNN architectures are developed step by step. This case demonstrates that, with a typical architecture InceptionV3, the full cycle of performing image classification using this architectures. It includes how the network is constructed, how the model weights are imported, the image processing in OCaml, and how the inference is performed. We also show how this process can be greatly simplified by using the Zoo system we have previously introduced. Finally, some prominent application field of image classification is briefly discussed.

References

- He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. “Deep Residual Learning for Image Recognition.” In *Proceedings of the Ieee Conference on Computer Vision and Pattern Recognition*, 770–78.
- Iandola, Forrest N, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. “SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters And < 0.5 Mb Model Size.” *arXiv Preprint arXiv:1602.07360*.
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton. 2012. “Imagenet Classification with Deep Convolutional Neural Networks.” In *Advances in Neural Information Processing Systems*, 1097–1105.
- LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. “Gradient-Based Learning Applied to Document Recognition.” *Proceedings of the IEEE* 86 (11): 2278–2324.

- Lin, Min, Qiang Chen, and Shuicheng Yan. 2013. “Network in Network.” *arXiv Preprint arXiv:1312.4400*.
- Simonyan, Karen, and Andrew Zisserman. 2014. “Very Deep Convolutional Networks for Large-Scale Image Recognition.” *arXiv Preprint arXiv:1409.1556*.
- Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. “Going Deeper with Convolutions.” In *Proceedings of the Ieee Conference on Computer Vision and Pattern Recognition*, 1–9.
- Zhao, Jianxin, and Liang Wang. 2019a. “InceptionV3 network architecture defined in Owl.” <https://gist.github.com/jzstark/9428a62a31dbe75511882ab8218076f/>.
- . 2019b. “ResNet50 network architecture defined in Owl.” <https://gist.github.com/pvdhove/a05bf0dbe62361b9c2aff89d26d09ba1/>.
- . 2019c. “SqueezeNet network architecture defined in Owl.” <https://gist.github.com/jzstark/c424e1d1454d58cfb9b0284ba1925a48/>.
- . 2019d. “VGG16 network architecture defined in Owl.” <https://gist.github.com/jzstark/f5409c44d6444921a8ceec00e33c42c4/>.

Case - Instance Segmentation

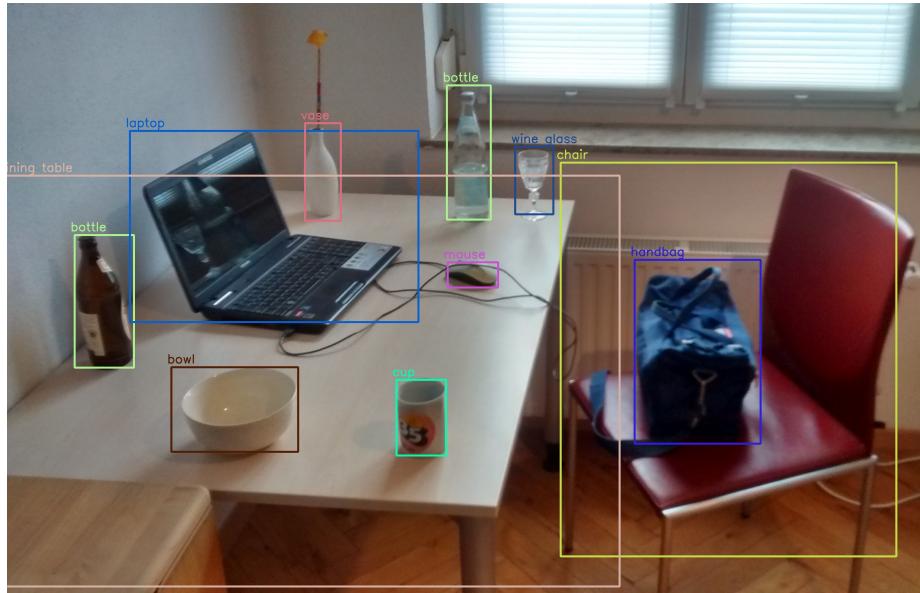
Computer vision is a field dealing with many different automated tasks whose goal are to give high-level descriptions of images and videos. It has been applied to a wide variety of domains ranging from highly technical ones, such as automatic tagging of satellite images, analysis of medical images etc., to more mundane applications, including categorising pictures in your phone, making your face into an emoji, etc. This field has seen tremendous progress since 2012, when A. Krizhevsky et al. used the first deep learning approach to computer vision, crushing all their opponents in the ImageNet challenge. We have already discussed this image recognition task in the previous chapter.

In this chapter, we are going to introduce another classical computer vision task: the *Instance Segmentation*, which is to give different labels for separate instances of objects in an image. We will discuss its connection with other similar applications, how the deep neural network is constructed in Owl, and how the network, loaded with pre-trained weights, can be used to process users' input images. We have also included an online demo for this application. Let's begin!

Introduction

In the previous chapter about the image recognition task, we have introduced how the DNN can be applied to classify the one single object in an image. However, this technique can easily get confused if it is applied on an image with multiple objects, which frequently happens in the real world. That's why we need other methods.

For example, *Object Detection* is another classical computer vision task. Given an image that contains multiple objects, an object detection application aims to classify individual objects and localize each one using a bounding box, as shown in the fig. ??.



Example of object detection (src)

Similarly, the *Semantic Segmentation* task requires to classify the pixels on an image into different categories. Each segment is recognised by a “mask” that follows cover the whole object. This can be illustrated in the [#@#fig:case-obj-detect:example_seg].



Example of semantic segmentation (src)

In 2017, the *Mask R-CNN* (Mask Region-based Convolutional Neural Network) architecture was published. With sufficient training, it can solve these problems at once: it can detect objects on an image, label each of them, and provide a



Figure 143: Example: Street view

binary mask to tell which pixels belong to the objects. This task is called the *Instance Segmentation*. As a preliminary example and visual motivation, the examples below show what this task generates:

In these two examples, normal pictures are processed by the pre-trained Mask R-CNN (MRCNN) network, and the objects (people, sheep, bag, car, bus, umbrella, etc.) are segmented from the picture and recognised with a percentage of confidence, represented by a number between 0 and 1.

Image segmentation have massive application scenarios in the industry, such as medical imaging (locating tumours, detecting cancer cells ...), traffic control systems, locate objects in satellite images, etc. The Mask R-CNN network has been implemented in Owl. In the next sections, we will explain how this network is constructed.

Mask R-CNN Network

This section will briefly outline the main parts of architecture of Mask R-CNN and how it stands out from its predecessors. You can get more detailed and technical explanations in the original paper (He et al. 2017). The Owl implementation of the inference mode is available in this repository. The code was mostly ported from the Keras/TensorFlow implementation. Work in this chapter is conducted by Pierre Vandenhove during his internship in the OCamlLabs.



Figure 144: Example: Sheep

MRCNN network is extended based on the Faster R-CNN network (Ren et al. 2015), which itself extends the Fast R-CNN (Girshick 2015). In Fast R-CNN, the authors propose a network that accepts input images and regions of interest (RoI). For each region, features are extracted by several fully-connected layers, and the features are fed into a branch. One output of this branch contains the output classification (together with possibility of that classification) of the object in that region, and the other specifies the rectangle location of the object.

In Faster R-CNN, the authors point out that, there is no need to find RoIs using other methods than the network itself. They propose a Region Proposal Network (RPN) that share the same feature extraction backbone as that in Fast R-CNN. RPN is a small convolutional network that scans the feature maps quickly, produces a set of rectangular possible object region proposals, each associated with a number that could be called the *objectness* of that region. The RoI feature extraction part of Fast R-CNN is kept unchanged here. In this way, a single Faster R-CNN network can be trained and then perform the object detection task without extra help from other region proposal methods.

To perform the task of not just objection detection, but also semantic segmentation, Mask R-CNN keeps the architecture of Faster R-CNN, only adding an extra branch in the final stage of its RoI feature layer. Where previously the outputs includes object classification and location, now a third branch contains information about the mask of object in the RoI. Therefore, for any RoI, the Mask R-CNN can retrieve its rectangle bound, classification results, classification possibility, and the mask of that object, all information in one pass.

Building Mask R-CNN

After a quick introduction of the MRCNN and how it is developed in theory, let's look at the code to understand how it is constructed in Owl, one piece at a

time. Please feel free to jump this part if you are eager to try to run the code and process one of your images.

```
open Owl

module N = Dense.Ndarray.S

open CGraph
open Graph
open AD

module RPN = RegionProposalNetwork
module PL = ProposalLayer
module FPN = FeaturePyramidNetwork
module DL = DetectionLayer
module C = Configuration

let image_shape = C.get_image_shape () in
if image_shape.(0) mod 64 > 0 || image_shape.(1) mod 64 > 0 then
invalid_arg "Image height and width must be divisible by 64";

let inputs = inputs
~names:[|"input_image"; "input_image_meta"; "input_anchors"|]
[|image_shape; [|C.image_meta_size|]; [|num_anchors; 4|]|] in
let input_image = inputs.(0)
and input_image_meta = inputs.(1)
and input_anchors = inputs.(2) in
```

The network accepts three inputs, representing images, meta data, and number of anchors (the rectangular regions). The `Configuration` module contains a list of constants that will be used in building the network.

Feature Extractor

The picture is first fed to a convolutional neural network in order to extract features of the image. The first few layers detect low-level features of an image, such as edges and basic shapes. But as you go deeper into the network, these features are assembled to detect higher level features such as “people”, “cars” (which, some argue, works in the same way as the brain). Five of these layers (called “feature maps”) of various sizes, both high- and low-level, are then passed on to the next parts. This implementation chooses Microsoft’s ResNet101 as a feature extractor.

```
let tdps = C.top_down_pyramid_size in
let str = [|1; 1|] in
let p5 = conv2d [|1; 1; 2048; tdps|] str ~padding:VALID ~name:"fpn_c5p5" c5 in
let p4 =
add ~name:"fpn_p4add"
[|upsampling2d [|2; 2|] ~name:"fpn_p5upsampled" p5;
conv2d [|1; 1; 1024; tdps|] str ~padding:VALID ~name:"fpn_c4p4" c4|] in
let p3 =
```

```

add ~name:"fpn_p3add"
[|upsampling2d [|2; 2|] ~name:"fpn_p4upsampled" p4;
 conv2d [|1; 1; 512; tdps|] str ~padding:VALID ~name:"fpn_c3p3" c3|] in
let p2 =
add ~name:"fpn_p2add"
[|upsampling2d [|2; 2|] ~name:"fpn_p3upsampled" p3;
 conv2d [|1; 1; 256; tdps|] str ~padding:VALID ~name:"fpn_c2p2" c2|] in

let conv_args = [|3; 3; tdps; tdps|] in
let p2 = conv2d conv_args str ~padding:SAME ~name:"fpn_p2" p2 in
let p3 = conv2d conv_args str ~padding:SAME ~name:"fpn_p3" p3 in
let p4 = conv2d conv_args str ~padding:SAME ~name:"fpn_p4" p4 in
let p5 = conv2d conv_args str ~padding:SAME ~name:"fpn_p5" p5 in
let p6 = max_pool2d [|1; 1|] [|2; 2|] ~padding:VALID ~name:"fpn_p6" p5 in

let rpn_feature_maps = [|p2; p3; p4; p5; p6|] in
let mrcnn_feature_maps = [|p2; p3; p4; p5|]

```

The features are extracted combining both ResNet101 and the Feature Pyramid Network. ResNet extracts features of the image (the first layers extract low-level features, the last layers extract high-level features).

Feature Pyramid Network creates a second pyramid of feature maps from top to bottom so that every map has access to high and low level features. This combination proves to achieve excellent gains in both accuracy and speed.

Proposal Generation

To try to locate the objects, about 250,000 overlapping rectangular regions or anchors are generated.

```

let nb_ratios = Array.length C.rpn_anchor_ratios in
let rpns = Array.init 5 (fun i =>
  RPN.rpn_graph rpn_feature_maps.(i)
  nb_ratios C.rpn_anchor_stride
  ("_p" ^ string_of_int (i + 2))) in
let rpn_class = concatenate 1 ~name:"rpn_class"
  (Array.init 5 (fun i => rpns.(i).(0))) in
let rpn_bbox = concatenate 1 ~name:"rpn_bbox"
  (Array.init 5 (fun i => rpns.(i).(1)))

```

Single RPN graphs are applied on different features in `rpn_features_maps`, and the results from these networks are then concatenated together. For each bounding box on the image, the RPN returns the likelihood that it contains an object and a refinement for the anchor, both are represented by rank-3 ndarrays.

Next, in the proposal layer, the 1000 best anchors are then selected according to their objectness (higher is better). Anchors that overlap too much with each other are eliminated, to avoid detecting the same object multiple times. Each selected anchor is also refined in case it was not perfectly centred around the object.

```
let rpn_rois =
  let prop_f = PL.proposal_layer C.post_nms_rois C.rpn_nms_threshold in
  MrcnnUtil.delay_lambda_array [|C.post_nms_rois; 4|] prop_f ~name:"ROI"
  [|rpn_class; rpn_bbox; input_anchors|] in
```

The proposal layer picks the top anchors from the RPN output, based on non maximum suppression and anchor scores. It then applies the deltas to the anchors.

Classification

All anchor proposals from the previous layer are resized to a fixed size and fed into a 10-layer neural network. The network assigns each of them the probability that it belongs to each class. The network is pre-trained on fixed classes; changing the set of classes requires to re-train the whole network. Note that this step does not take as much time for each anchor as a full-fledged image classifier such as Inception, since it reuses the pre-computed feature maps from the Feature Pyramid Network. Therefore there is no need to go back to the original picture. The class with the highest probability is chosen for each proposal, and, thanks to the class predictions, the anchor proposals are even more refined. Proposals classified in the background class are deleted. Eventually, only the proposals with an objectness over some threshold are kept, and we have our final detections, each having a bounding box and a label.

```
let mrcnn_class, mrcnn_bbox =
FPN.fpn_classifier_graph rpn_rois mrcnn_feature_maps input_image_meta
  C.pool_size C.num_classes C.fpn_classif_fc_layers_size in

let detections = MrcnnUtil.delay_lambda_array [|C.detection_max_instances; 6|]
  (DL.detection_layer ()) ~name:"mrcnn_detection"
  [|rpn_rois; mrcnn_class; mrcnn_bbox; input_image_meta|] in
let detection_boxes = lambda_array [|C.detection_max_instances; 4|]
  (fun t -> Maths.get_slice [[]; []; [0;3]] t.(0))
  [|detections|]
```

A Feature Pyramid Network classifier associates a class to each proposal and refines the bounding box for that class even more. The only thing left to do is to generate a binary mask on each object. This is handled by a small convolutional neural network which produces a small square of values between 0 and 1 for each detected bounding box. This square is resized to the original size of the bounding box with bilinear interpolation, and pixels with a value over 0.5 are tagged as being part of the object.

```
let mrcnn_mask = FPN.build_fpn_mask_graph detection_boxes mrcnn_feature_maps
  input_image_meta C.mask_pool_size C.num_classes
```

And finally, the output contains detection results and masks from the previous steps.

```
outputs ~name:C.name [|detections; mrcnn_mask|]
```

Run the Code

After getting to know the internal mechanism of the MRCNN architecture, we can then try run the code to see how it works. One example of using the MRCNN code is in this example script. The core part is listed below:

```
open Mrcnn

let src = "image.png" in
let fun_detect = Model.detect () in
let Model.({rois; class_ids; scores; masks}) = fun_detect src in
let img_arr = Image.img_to_ndarray src in
let filename = Filename.basename src in
let format = Images.guess_format src in
let out_loc = out ^ filename in
Visualise.display_masks img_arr rois masks class_ids;
let camlimg = Image.img_of_ndarray img_arr in
Visualise.display_labels camlimg rois class_ids scores;
Image.save out_loc format camlimg;
Visualise.print_results class_ids rois scores
```

The most important step is to apply the `Model.detect` function on the input images, which returns the region of interests, the classification result ID of the object in that region, the classification certainty scores, and a mask that shows the outline of that object in the region. With this information, the `visualise` module runs for three passes on the original image: the first for adding bounding boxes and object masks, the second for adding the numbers close to the bounding box, and finally for printing out the resulting images from the previous two steps.

In this example, the pre-trained weights on 80 classes of common objects are provided, which have been converted from the TensorFlow implementation mentioned above. As to the execution speed, processing one image with a size of 1024x1024 pixels takes between 10 and 15 seconds on a moderate laptop. You can try a demo of the network without installing anything.

Summary

This chapter introduces Instance Segmentation, another powerful computer vision task that is wildly used in various fields. It encompasses several different tasks, including image recognition, semantic segmentation, and object detection. This task can be performed with the Mask R-CNN network architecture. We explain in detail how it is constructed using the Owl code. Besides, we also provide example code and online demo so that the readers can try to play with this powerful application conveniently.

References

- Girshick, Ross. 2015. “Fast R-Cnn.” In *Proceedings of the Ieee International Conference on Computer Vision*, 1440–8.
- He, Kaiming, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. “Mask R-Cnn.” In *Proceedings of the Ieee International Conference on Computer Vision*, 2961–9.
- Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. 2015. “Faster R-Cnn: Towards Real-Time Object Detection with Region Proposal Networks.” In *Advances in Neural Information Processing Systems*, 91–99.



Figure 145: Example of applying neural style transfer on a street view picture

Case - Neural Style Transfer

What is Neural Style Transfer (NST)? It is “the process of using DNN to migrate the semantic content of one image to different styles”, a pretty cool application of the deep neural network. The idea is actually very simple. As the fig. 145 shows, this application takes two images A and B as input. Let’s say A is a normal street view, and B is “The Starry Night” of Vincent van Gogh. We then specify A as the “content image” and B as the “style image”. What a NST application can produce is a new street view image with the style of Van Gogh. If you want another style, just replace image B and run the application again. Impressionism, abstractionism, classical art, you name it. Or you can also apply the same “Starry Sky” style to any other images.

Content and Style

The paper (Gatys, Ecker, and Bethge 2015) first proposes to use DNN to let programmes to create artistic images of high perceptual quality. In this section, we will first introduce the intuition about how the neural style transfer algorithm works. For more formal and detailed introduction, please visit the original paper.

The basic idea is plain: we want to get an image whose content is similar to one image and its artistic style close to the other image. Of course, to make the algorithm work, first we need to express this sentence in mathematical form so that computers can understand it. Let’s assume for a moment we have already known that, then style transfer can be formalised as an optimisation problem. Given a content image c and a style image s , our target is to get an output image x so that it minimises:

$$g(x) = \text{content_distance}(x, c) + \text{style_distance}(x, s)$$

Here the “distance” between two feature maps is calculated by the Euclidean distance between the two ndarrays.

You may remember from the regression or neural network chapters that training process is also an optimisation process. However, do not mistake the optimisation

in NST as regression or DNN training. For the latter one, there is the function f_w that contains parameter w and the training process optimises w to minimise $f_w(x)$. The optimisation in NST is more like the traditional optimisation problem: we have a function f , and we start with an initial input x_0 and update it iteratively until we have satisfying x that minimise the target function.

Now we can come back to the key problem. While we human beings can kind of feel the style of paint and visually recognise the contents in a picture, how can we mathematically express the “content” and the “style” of an image? That’s where the convolution network comes to help. DNNs, especially the ones that are used for computer vision tasks, are found to be a convenient tool to capture the characteristics of an image. We have demonstrate in the previous chapter how CNNs are good at spotting the “features” in an image layer by layer. Therefore, in the next two sub-sections, we will explain how it can be used to express the content and style feature of an image.

We have introduced several CNN architectures to perform image detection task in the previous chapter. We choose to use the VGG19 since it follows a simple linear stack structure and is proved to have good performance. We have built the VGG19 network structure in this gist. It contains 38 layers in total, and we prepared pre-trained weights for it.

Content Reconstruction

From the image detection case, we know that the CNN extract features layer by layer until the features are so abstract that it can give an answer such as “this is a car” “this is an apple” etc. Therefore, we can use the feature map to reconstruct content of an image.

But which layer’s output should we use as a suitable indication of the image content? Let’s perform a simplified version of NST: we only care about reconstructing the content of the input image, so our target is to minimise:

$$f(x) = \text{content_distance}(x, c)$$

As an example, we use fig. 146 as the target content. (This image “Tourists in Nanzen-Ji Hojo” by blieusong is licensed under CC BY-SA 2.0.)

Suppose we choose the output of the `idx` layer as the chosen feature map to represent the content. First, we need to compute the target feature map:

```
let fill_content_targets x net =
  let selected_topo = Array.sub nn.topo 0 (idx + 1) in
  run' selected_topo x
```

The function `fill_content_targets` takes the content image and the VGG network as input, and returns the target feature map as output. We only need to compute the feature map of the target content image once.



Figure 146: Example content image in neural style transfer

Here the `run'` function is implemented by accumulating the inference result along the selected part of the network, from the network input until the chosen layer, instead of processing the whole network:

```
let run' topo x =
  let last_node_output = ref (F 0.) in
  Array.iteri (fun i n -
    let input = if i = 0 then x else !last_node_output in
    let output = run [|input|] n.neuron in
    last_node_output := output;
  ) topo;
  !last_node_output
```

Then we can start optimising the input image `x`. Let's set the initial `x` to be a “white noise” image that only contains random pixels. This image has the same shape as the content image.

```
let input_shape = Dense.Ndarray.S.shape content_img in
Dense.Ndarray.S.(gaussian input_shape |> scalar_mul 0.256)
```

The feature map of the input image `x` is still calculated using the same process shown in the function `fill_content_targets`. We call the resulting feature map `response`, then the loss value can be calculated with the L2Norm of the difference between two feature maps, and then normalised with the feature map size.

```
let c_loss response target =
  let loss = Maths.((pow (response - target) (F 2.)) |> sum') in
  let _, h, w, feature = get_shape target in
  let c = float_of_int (feature * h * w) in
  Maths.(loss / (F c))
```



Figure 147: Contents reconstruction from different layers

Once the loss value is calculated, we can apply optimisers. Here we use the `minimise_fun` from `Optimise` module. The target function can be described as:

```
let g x =
  fill_losses x;
  c_loss response target
```

All it performs is what we just described: first calculating the feature map response of input image at a certain layer, and then computing the distance between it and the target content feature map as the loss value.

Finally, we can perform the optimisation iterations:

```
let state, img = Optimise.minimise_fun params g (Arr input_img) in
let x' = ref img in
while Checkpoint.(state.current_batch < state.batches) do
  Checkpoint.(state.stop <- false);
  let a, img = Optimise.minimise_fun ~state params g !x' in
  x' := img
done;
```

We keep updating the image x for a fixed number of iterations. Particularly, we use the Adam adaptive learning rate method, for it proves to be quite effective in style transfer optimisation:

```
let params = Params.config
  ~learning_rate:(Learning_Rate.Adam (learning_rate, 0.9, 0.999))
  ~checkpoint:(Checkpoint.Custom chkpt)
  iter
```

Using the process above, we return to the problem of choosing a suitable layer as the indication of the image content. In this 38-layer VGG network, some frequently used practical choices are these layers: 2, 7, 12, 21, 30. Then we can compare the optimisation results to see the effect of image reconstruction. Each one is generated after 100 iterations.



Figure 148: Example style image in neural style transfer

It is shown in fig. 147 that, the content information is kept accurate at the low layers. Along the processing hierarchy of the network, feature map produced by the lower layer cares more about the small features that at the pixel level, while the higher layer gives more abstract information but less details to help with content reconstruction.

Style Recreation

Then similarly, we explore the other end of this problem. Now we only care about recreating an image with only the style of an input image. That is to say, we optimise the input image with this target to minimise:

$$h(x) = \text{style_distance}(x, s)$$

As an example, we will use the famous “The Great Wave of Kanagawa” by Hokusai as our target style image:

The basic approach is the same as before: first compute the style representation of target image using the output from one or more layers, and then compute the style representation of the input image following the same method. The normalised distance between these two ndarrays are used as the optimisation target.

However, the difference is that, unlike the content representation, we cannot directly take one filter map from certain layer as the style representation. Instead, we need to compute the correlations between different filters from the output of a layer. This correlation can be represented by the *Gram matrix*, which intuitively captures the “distribution of features” of feature maps from a certain layer. The (i, j) -th element of a Gram matrix is computed by element-wisely multiplying the i -th and j -th channels in the feature maps and summing across both width and height. This process can be simplified as a matrix multiplication.

The result is normalised with the size of the feature map. The code is shown below.

```
let gram x =
  let _, h, w, feature = get_shape x in
  let new_shape = [|h * w; feature|] in
  let ff = Maths.(reshape x new_shape) in
  let size = F (float_of_int (feature * h * w)) in
  Maths.((transpose ff) *@ ff / size)
```

Now that we have a method to represent the “style” of an image, we can proceed to calculate the loss value during optimisation. It is very similar to that of content recreation, and the only difference is that we use the distance between Gram matrices instead of the feature maps from a certain layer as loss value.

```
let s_loss response_gram target_gram =
  let loss = Maths.((pow (response_gram - target_gram) (F 2.)) |> sum') in
  let s = Algodiff.shape target_gram in
  let c = float_of_int (s.(0) * s.(1)) in
  Maths.(loss / (F c))
```

However, note that for the optimisation, instead of using output from one layer, we usually utilises the loss values from multiple layers and the optimisation target for style reconstruction:

```
let h x =
  fill_losses x;
  Array.fold_left Maths.(+) (F 0.) style_losses
```

Here the `fill_losses` function compute style losses at different layers, and store them into the `style_losses` array. Then they are added up as the optimisation target. The rest process is the same as in the content reconstruction.

In this example, we choose the same five layers from the VGG19 network: layer 2, 7, 12, 21, and 30. Then we compute the aggregated loss value of the first layer, the first two layers, the first three layers, the first four layers, and all layers, as five different optimisation target. Optimising these five different target, the resulting images are shown in fig. 149.

As the result shows, features from the beginning tends to contain low level information such as pixels, so reconstructing styles according to them results in a fragmented white-noise-like representation, which really does not show any obvious style. Only by adding more deep layer features can the style be gradually reconstructed. The fifth generated image shows a quite obvious wave-like style.

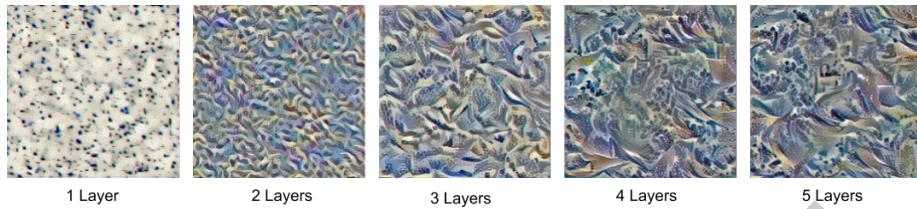


Figure 149: Style reconstruction from different layers

Combining Content and Style

Now that we have seen these two extremes: only recreating content and only recreating style, it's straightforward to understand the theory of style transfer: to synthesise an image that has similar content with one image and style close to the other. The code would be mostly similar to what we have seen, and the only difference now is simply adding the loss value of content and styles as the final optimisation target.

One thing we need to note during combining contents and style is the proportion of each part, and the choice of layers as representation. This problem is actually more artistic than technique, so here we only follow the current practice about parameter configuration. Please refer to the original paper about the effect of parameter tuning.

As suggested by previous experiment results, we use the feature maps from 23rd layer for content recreation, and combine the output of layer 2, 7, 12, 21, and 30 in VGG19 to represent the style feature of an image. When combining the loss values, we multiply the style loss with a weight number, and then add it to the content loss. Practice shows that a weight number of 20 shows good performance.

You might also be wondering: why not choose the 2nd layer if it show the best content reconstruction result? The intuition is that we don't want the synthesised image to be too close to the content image in content, because that would mean less style. Therefore we use a layer from the middle of CNN which shows to keep most of the information for content reconstruction.

Combining all these factors together, fig. 150 shows the result of running our code and creating an artistic view based on the original image.

All the code (about 180 lines) is included in this gist. The pre-trained weight file for VGG19 is also included. As with the image detection applications, it also relies on the tool `ImageMagick` to manipulate image format conversion and resizing. We only list part of it above, and many implementation details such as garbage collection are omitted to focus on the theory of the application itself. We therefore suggest you to play with the code itself with images or parameters of your choice.



Figure 150: Combining content and style reconstruction to perform NST

Running NST

To make the code above more suitable to use, this NST application provides a simple interfaces to use. Here is an example showing how to use it with two lines of code:

```
#zoo "6f28d54e69d1a19c1819f52c5b16c1a1"
Neural_transfer.run
~ckpt:50
~src:"path/to/content_img.jpg"
~style:"path/to/style_img.jpg"
~dst:"path/to/output_img.png" 250.;;
```

Similar to the image detection application, the command can be simplified using the Zoo system in owl. The first line downloads gist files and imported this gist as an OCaml module, and the second line uses the `run` function to produce an output image to your designated path. Its syntax is quite straightforward, and you may only need to note the final parameter. It specifies how many iterations the optimisation algorithm runs. Normally 100 ~ 500 iterations is good enough.

This module also supports saving the intermediate images to the same directory as output image every N iterations (e.g. `path/to/output_img_N.png`). N is specified by the `ckpt` parameter, and its default value is 50 iterations. If users are already happy with the intermediate results, they can terminate the program without waiting for the final output image.

That's all. Now you can try the code easily. If you don't have suitable input images at hand, the gist already contains exemplar content and style images to get you started. More examples can be seen on our online demo page.

Extending NST

The neural style transfer has attracted a lot of attention since its publication. It is the core technology of many successful industrial applications, most notably

photo rendering applications. For example, the Prisma Photo Editor features transforming your photos into paintings of hundreds of styles.

There are also many research work that aim to extend this work. One of these work is the *Deep Photo Style Transfer* proposed in (Luan et al. 2017). The idea is simple: instead of using an art image, can I use another normal image as style reference? For example, we have a normal daylight street view in New York as a content image, and then we want to use the night view of London as reference, to synthesise an image of the night view of New York.

The authors identify two key challenges in this problem. The first is that, unlike in NST, we hope to only change to colours of the style image, and keep the content un-distorted, so as to create a “real” image as much as possible. For this challenge, the authors propose to add a regularisation item to our existing optimisation target “content distance + style distance”. This item, depending on only input and output images, penalises image distortion and seeks an image transform that is locally affine in colour space. The second challenge is that, we don’t want the styles to be applied globally. For example, we only want to apply the style of an sunset sky to a blue sky, not a building. For this problem, the authors propose to coarsely segment input images into several parts before apply style transfer separately. If you are interested to check the original paper, the resulting photos are indeed beautifully and realistically rendered.

Another similar application is the “image-to-image translation”. This computer vision broadly involves translating an input image into certain output image. The style transfer or image colourisation can be seen as examples of it. There are also applications that change the lighting/weather in a photo. These can also be counted as examples of image to image translation.

In (Isola et al. 2017) the authors propose to use the Generative Adversarial Networks (GANs) to provide general framework for this task. In GAN, there are two important component: the generator, and the discriminator. During training, the generator synthesises images based on existing parameters, and the discriminator tries its best to separate the generated data from true data. This process is iterated until the discriminator can no longer tell the difference between these two. The work in (Isola et al. 2017) utilises convolution neural network to construct the generator and discriminator in the GAN. This approach is successfully applied in many applications, such as Pix2Pix, face ageing, increase photo resolution, etc.

Another variant is called the Fast Style Transfer. Instead of iteratively updating image, it proposes to use one pre-trained feed-forward network to do the style transfer, and therefore improve the speed of rendering by orders of magnitude. That’s what we will be talking about in the rest of this chapter.

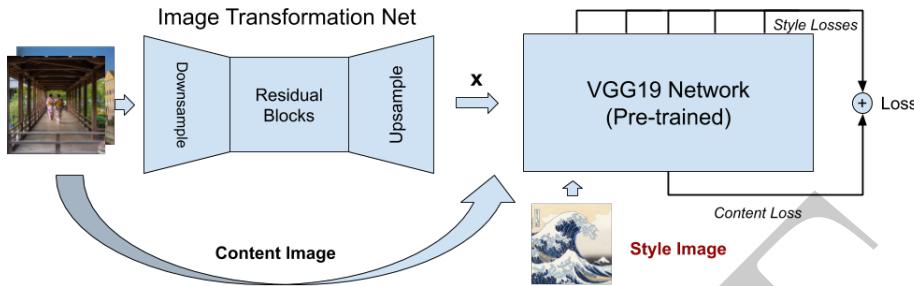


Figure 151: System overview of the image transformation network and its training.

Fast Style Transfer

One disadvantage of NST is that it could take a very long time to rendering an image, and if you want to change to another content or style image, then you have to wait a long time for the training again. If you want to render some of your best (or worst) selfies fast and send to your friends, NST is perhaps not a perfect choice.

This problem then leads to another application: Fast Neural Style Transfer (FST). FST sacrifice certain degrees of flexibility, which is that you cannot choose style images at will. But as a result, you only need to feed your content image to a DNN, finish an inference pass, and then the output will be the rendered styled image as you expected. The best part is that, one inference pass is much faster than keep running a training phase.

Building FST Network

The Fast Style Transfer network is proposed in (Johnson, Alahi, and Fei-Fei 2016). The authors propose to build and train an *image transformation network*. Image transformation is not a totally new idea. It takes some input image and transforms it into a certain output image. One way to do that is to train a feed-forward CNN. This method is applied in different applications such as colourising grayscale photos or image segmentation. In this work the author use a similar approach to solve the style transfer problem.

fig. 151 shows a system overview of the image transformation network and its training. It can be divided into two parts. The first part includes the image transformation network architecture. To synthesise an image of the same size as input image, it first uses down-sampling layers, and then the up-sampling layers. One benefit of first down-sampling images is to reduce the computation, which enables building a deeper network. We have already seen this design principle in the image detection case chapter.

Instead of using the normal pooling or upsampling layer in CNN, here the

convolution layers are used for down/up-sampling. We want to keep the image information as much as possible during the whole transformation process. Specifically, we use the transpose convolution for upsampling. This operation goes the opposite direction of a normal convolution, from small feature size to larger one, and still maintains the connectivity pattern in convolution.

```
open Owl
open Neural.S
open Neural.S.Graph
open Neural.S.Algodiff
module N = Dense.Ndarray.S

let conv2d_layer ?(relu=true) kernel stride nn =
  let result =
    conv2d ~padding: SAME kernel stride nn
    |> normalisation ~decay:0. ~training:true ~axis:3
  in
  match relu with
  | true -> (result |> activation Activation.Relu)
  | _ -> result

let conv2d_trans_layer kernel stride nn =
  transpose_conv2d ~padding: SAME kernel stride nn
  |> normalisation ~decay:0. ~training:true ~axis:3
  |> activation Activation.Relu
```

Here, combined with batch normalisation and Relu activation layers, we build two building blocks: the `conv2d_layer` and the `conv2d_trans_layer`. Think of them as enhanced convolution and transpose convolution layers. The benefit of adding these two types of layers is discussed in previous chapter.

What connect these two parts are multiple residual blocks, which is proposed in the ResNet architecture. The authors claim that using residual connections makes it easier to keep the structure between output and input. It is an especially attractive property for an style transfer neural networks. Specifically, the authors use the residual structure proposed in the ResNet. All the convolution layers use the common 3x3 kernel size. This residual block can be implemented with the `conv2d_layer` unit we have built.

```
let residual_block wh nn =
  let tmp = conv2d_layer [|wh; wh; 128; 128|] [|1;1|] nn
  |> conv2d_layer ~relu:false [|wh; wh; 128; 128|] [|1;1|]
  in
  add [|nn; tmp|]
```

Here in the code the `wh` normally takes a value of 3. The residual block, as with in the ResNet, is repeatedly stacked for several times. With these three different parts ready, finally we can piece them together. Note how the output channel of each convolution increases, stays the same, and then decreases symmetrically.

Before the final output, we use the `tanh` activation layer to ensure all the values are between [0, 255] for the output image.

```
let make_network h w =
  input [|h;w;3|]
  |> conv2d_layer [|9;9;3;32|] [|1;1|]
  |> conv2d_layer [|3;3;32;64|] [|2;2|]
  |> conv2d_layer [|3;3;64;128|] [|2;2|]
  |> residual_block 3
  |> conv2d_trans_layer [|3;3;128;64|] [|2;2|]
  |> conv2d_trans_layer [|3;3;64;32|] [|2;2|]
  |> conv2d_layer ~relu:false [|9;9;32;3|] [|1;1|]
  |> lambda (fun x -> Maths.((tanh x) * (F 150.) + (F 127.5)))
  |> get_network
```

After constructing the image transformation network, let's look at the training process. In previous work, when training a image transformation network, normally the output will be compared with the ground-truth image pixel-wisely as the loss value. That is not an ideal approach here since we cannot know what is a "correct" style-transferred image in advance. Instead, the authors are inspired by the NST work. They use the same training process with a pre-trained VGG19 network to compute the loss (they call it the *perceptual loss* against the per-pixel loss, since high level perceptual information is contained in this loss).

We should be familiar with the training process now. The output image x from image transformation network is the one to be optimised. The input image itself is content image, and we provide another fixed style image. We can then proceed to calculate the final loss by computing the distance between image x and the input with regard to content and styles. All of these are the same as in the NST. The only difference is that, where we train an image before, now we train the weights of the image transformation network during back-propagation. Note that this process means that we can only train one set of weight for only one style. Considering that the artistic styles are relatively fixed compared to the unlimited number of content image, and the orders of magnitude of computation speed improved, fixing the styles is an acceptable trade-off.

Even better, this training phase is one-off. We can train the network once and the reuse it in the inference phase again and again. We refer you to the original paper if you want to know more details about the training phase. In our implementation, we directly convert and import weights from a TensorFlow implementation. Next we will show how to use it to perform the fast style transfer.



Figure 152: Artistic Styles used in fast style transfer

Running FST

Like NST and image classification, we have wrapped all things up in a gist, and provide a simple user interface to users. Here is an example:

```
#zoo "f937ce439c8adcaea23d42753f487299"
FST.list_styles (); (* show all supported styles *)
FST.run ~style:1 "path/to/content_img.png" "path/to/output_img.jpg"
```

The `run` function mainly takes one content image and output to a new image file, the name of which is designated by the user. The image could be of any popular formats: jpeg, png, etc. This gist contains exemplar content images for you to use. A set of trained weight for the FST DNN represents a unique artistic style. We have already included six different weight files for using, and the users just need to pick one of them and load them into the DNN, without worrying about how to train these weights.

Current we support six art styles: “Udnie” by Francis Picabia, “The Great Wave off Kanagawa” by Hokusai, “Rain Princess” by Leonid Afremov, “La Muse” by Picasso, “The Scream” by Edvard Munch, and “The shipwreck of the Minotaur” by J. M. W. Turner. These style images are shown in fig. 152.

Maybe six styles are not enough for you. But think about it, you can now render any of your image to a nice art style fast, maybe about half a minute. It would be even faster if you are using GPU or other accelerators. As an example, we use the Willis Tower in Chicago as an input image:



Figure 153: Example input image: Willis tower of Chicago



Figure 154: Fast style transfer examples

We then apply FST on this input image with the styles shown above. The rendered city view with different styles are shown in fig. 154. Each of them rendered in only seconds on a moderate CPU desktop.

Moreover, based these code, we have built a demo website for the FST application. You can choose a style, upload an image, get yourself a cup of coffee, and then checkout the rendered image. To push things even further, we apply FST to some videos frame-by-frame, and put them together to get some artistic videos, as shown in this Youtube list. You are welcome to try this application with images of your own.

Summary

During the development of Owl, we have built the style transfer application to benchmark the expressiveness and performance of Owl. In this chapter we introduce this use case in detail, including the theory behind style transfer application, our implementation detail, and examples. The NST application is extended in many ways, one of which is the fast style transfer. We then introduce how this application works with example. Theories aside, we quite enjoy ourselves in building and using these style transfer applications. Hope this chapter can also help you to better understand this state-of-art DNN application.

References

- Gatys, Leon A, Alexander S Ecker, and Matthias Bethge. 2015. “A Neural Algorithm of Artistic Style.” *arXiv Preprint arXiv:1508.06576*.
- Isola, Phillip, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. 2017. “Image-to-Image Translation with Conditional Adversarial Networks.” In *Proceedings of the Ieee Conference on Computer Vision and Pattern Recognition*, 1125–34.
- Johnson, Justin, Alexandre Alahi, and Li Fei-Fei. 2016. “Perceptual Losses for Real-Time Style Transfer and Super-Resolution.” In *European Conference on Computer Vision*.
- Luan, Fujun, Sylvain Paris, Eli Shechtman, and Kavita Bala. 2017. “Deep Photo Style Transfer.” In *Proceedings of the Ieee Conference on Computer Vision and Pattern Recognition*, 4990–8.

Case - Recommender System

Introduction

Our daily life heavily relies on recommendations, intelligent content provision aims to match a user's profile of interests to the best candidates in a large repository of options. There are several parallel efforts in integrating intelligent content provision and recommendation in web browsing. They differentiate between each other by the main technique used to achieve the goal.

The initial effort relies on the semantic web stack, which requires adding explicit ontology information to all web pages so that ontology-based applications (e.g., Piggy bank) can utilise ontology reasoning to interconnect content semantically. Though semantic web has a well-defined architecture, it suffers from the fact that most web pages are unstructured or semi-structured HTML files, and content providers lack of motivation to adopt this technology to their websites. Therefore, even though the relevant research still remains active in academia, the actual progress of adopting ontology-based methods in real-life applications has stalled in these years.

Collaborative Filtering (CF), which was first coined in 1992, is a thriving research area and also the second alternative solution. Recommenders built on top of CF exploit the similarities in users' rankings to predict one user's preference on a specific content. CF attracts more research interest these years due to the popularity of online shopping (e.g., Amazon, eBay, Taobao, etc.) and video services (e.g., YouTube, Vimeo, Dailymotion, etc.). However, recommender systems need user behaviour rather than content itself as explicit input to bootstrap the service, and are usually constrained within a single domain. Cross-domain recommenders have made progress lately, but the complexity and scalability need further investigation.

Search engines can be considered as the third alternative though a user needs explicitly extract the keywords from the page then launch another search. The ranking of the search results is based on multiple ranking signals such as link analysis on the underlying graph structure of interconnected pages such as PageRank. Such graph-based link analysis is based on the assumption that those web pages of related topics tend to link to each other, and the importance of a page often positively correlates to its degree. The indexing process is modelled as a random walk atop of the graph derived from the linked pages and needs to be pre-compiled offline.

The fourth alternative is to utilise information retrieval (IR) technique. In general, a text corpus is transformed to the suitable representation depending on the specific mathematical models (e.g., set-theoretic, algebraic, or probabilistic models), based on which a numeric score is calculated for ranking. Different from the previous CF and link analysis, the underlying assumption of IR is that the text (or information in a broader sense) contained in a document can very well indicate its (latent) topics. The relatedness of any two given documents can be

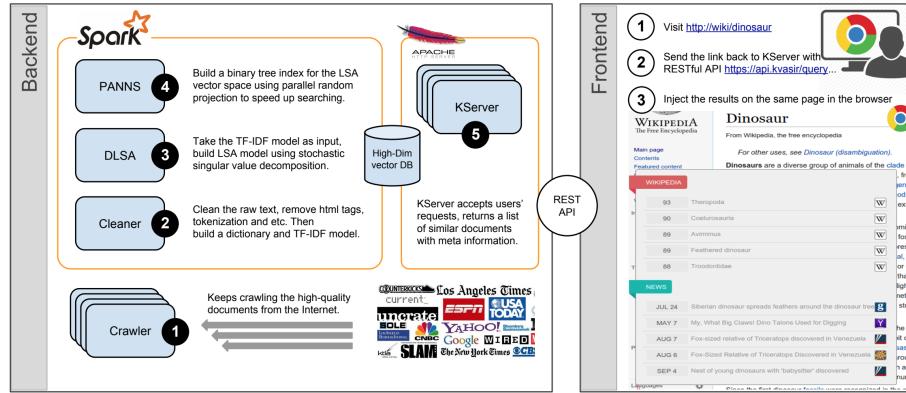


Figure 155: Kvasir architecture with components numbered based on their order in the workflow

calculated with a well-defined metric function atop of these topics. Since topics can have a direct connection to context, context awareness therefore becomes the most significant advantage in IR, which has been integrated into Hummingbird, Google's new search algorithm.

In the rest of this chapter, we will introduce **Kvasir**, a system built on top of latent semantic analysis. Kvasir automatically looks for the similar articles when a user is browsing a web page and injects the search results in an easily accessible panel within the browser view for seamless integration. Kvasir belongs to the content-based filtering and emphasises the semantics contained in the unstructured web text. This chapter is based on the papers in (Wang et al. 2016) and (Hyvönen et al. 2016), and you will find that many basic theory are already covered previously in the NLP chapter in Part I. Henceforth we will assume you are familiar with this part.

Architecture

At the core, Kvasir implements an LSA-based index and search service, and its architecture can be divided into two subsystems as **frontend** and **backend**. Figure fig. 155 illustrates the general workflow and internal design of the system. The frontend is currently implemented as a lightweight extension in Chrome browser. The browser extension only sends the page URL back to the KServer whenever a new tab/window is created. The KServer running at the backend retrieves the content of the given URL then responds with the most relevant documents in a database. The results are formatted into JSON strings. The extension presents the results in a friendly way on the page being browsed. From user perspective, a user only interacts with the frontend by checking the list of recommendations that may interest him.

To connect to the frontend, the backend exposes one simple *RESTful API* as

below, which gives great flexibility to all possible frontend implementations. By loosely coupling with the backend, it becomes easy to mash-up new services on top of Kvasir. In the code below, Line 1 and 2 give an example request to Kvasir service. `type=0` indicates that `info` contains a URL, otherwise `info` contains a piece of text if `type=1`. Line 4-9 present an example response from the server, which contains the meta-info of a list of similar articles. Note that the frontend can refine or rearrange the results based on the meta-info (e.g., similarity or timestamp).

```
POST https://api.kvasir/query?type=0&info=url
{
  "results": [
    {"title": document title,
     "similarity": similarity metric,
     "page_url": link to the document,
     "timestamp": document create date}
  ]
}
```

The backend system implements indexing and searching functionality which consist of five components: *Crawler*, *Cleaner*, *DLSA*, *PANNS* and *KServer*. Three components (i.e., Cleaner, DLSA and PANNS) are wrapped into one library since all are implemented on top of Apache Spark. The library covers three phases as text cleaning, database building, and indexing. We briefly present the main tasks in each component as below.

Crawler collects raw documents from the web and then compiles them into two data sets. One is the English Wikipedia dump, and another is compiled from over 300 news feeds of the high-quality content providers such as BBC, Guardian, Times, Yahoo News, MSNBC, and etc. Table 46 summarises the basic statistics of the data sets. Multiple instances of the Crawler run in parallel on different machines. Simple fault-tolerant mechanisms like periodical backup have been implemented to improve the robustness of crawling process. In addition to the text body, the Crawler also records the timestamp, URL and title of the retrieved news as meta information, which can be further utilised to refine the search results.

Table 46: Two data sets are used in Kvasir evaluation

Data set	# of entries	Raw text size	Article length
Wikipedia	3.9×10^6	47.0 GB	Avg. 782 words
News	4.6×10^5	1.62 GB	Avg. 648 words

Cleaner cleans the unstructured text corpus and converts the corpus into term frequency-inverse document frequency (TF-IDF) model. In the preprocessing phase, we clean the text by removing HTML tags and stop words, de-accenting, tokenisation, etc. The dictionary refers to the vocabulary of a language model.

Its quality directly impacts the model performance. To build the dictionary, we exclude both extremely rare and extremely common terms, and keep 10^5 most popular ones as features. More precisely, a term is considered as rare if it appears in less than 20 documents, while a term is considered as common if it appears in more than 40% of documents.

DLSA builds up an LSA-based model from the previously constructed TF-IDF model. Technically, the TF-IDF itself is already a vector space language model. The reason we seldom use TF-IDF directly is because the model contains too much noise and the dimensionality is too high to process efficiently even on a modern computer. To convert a TF-IDF to an LSA model, DLSA's algebraic operations involve large matrix multiplications and time-consuming SVD. We initially tried to use MLlib to implement DLSA. However, MLlib is unable to perform SVD on a data set of 10^5 features with limited RAM, we have to implement our own stochastic SVD on Apache Spark using rank-revealing technique. The DLSA will be discussed in detail in later chapter.

PANNS builds the search index to enable fast k -NN search in high dimensional LSA vector spaces. Though dimensionality has been significantly reduced from TF-IDF (10^5 features) to LSA (10^3 features), k -NN search in a 10^3 -dimension space is still a great challenge especially when we try to provide responsive services. A naive linear search using one CPU takes over 6 seconds to finish in a database of 4 million entries, which is unacceptably long for any realistic services. PANNS implements a parallel RP-tree algorithm which makes a reasonable tradeoff between accuracy and efficiency. PANNS is the core component in the backend system and we will present its algorithm in detail in later chapter. PANNS is becoming a popular choice of Python-based approximate k-NN library for application developers. According to the PyPI's statistics, PANNS has achieved over 27,000 downloads since it was first published in October 2014.

KServer runs within a web server, processes the users requests and replies with a list of similar documents. KServer uses the index built by PANNS to perform fast search in the database. The ranking of the search results is based on the cosine similarity metric. A key performance metric for KServer is the service time. We wrapped KServer into a Docker image and deployed multiple KServer instances on different machines to achieve better performance. We also implemented a simple round-robin mechanism to balance the request loads among the multiple KServers.

Kvasir architecture provides a great potential and flexibility for developers to build various interesting applications on different devices, e.g., semantic search engine, intelligent Twitter bots, context-aware content provision, and etc. We provide the live demo videos of the seamless integration of Kvasir into web browsing at the official website. Kvasir is also available as browser extension on Chrome and Firefox.

Build Topic Models

As has been explained in the previous section, the crawler and cleaner performs data collection and processing to build vocabulary and TF-IDF model. We have already talked about this part in detail in the NLP chapter. DLSA and PANNS are the two core components responsible for building language models and indexing the high dimensional data sets in Kvasir. In this section, we first sketch out the key ideas in DLSA.

First, a recap of LSA from the NLP chapter. The vector space model belongs to algebraic language models, where each document is represented with a row vector. Each element in the vector represents the weight of a term in the dictionary calculated in a specific way. E.g., it can be simply calculated as the frequency of a term in a document, or slightly more complicated TF-IDF. The length of the vector is determined by the size of the dictionary (i.e., number of features). A text corpus containing m documents and a dictionary of n terms will be converted to an $A = m \times n$ row-based matrix. Informally, we say that A grows taller if the number of documents (i.e., m) increases, and grows fatter if we add more terms (i.e., n) in the dictionary.

The core operation in LSA is to perform SVD. For that we need to calculate the covariance matrix $C = A^T \times A$, which is a $n \times n$ matrix and is usually much smaller than A . This operation poses as ad bottleneck in computing: the m can be very large (a lot of documents) or the n can be very large (a lot of features for each document). For the first, we can easily parallelise the calculation of C by dividing A into k smaller chunks of size $[\frac{m}{k}] \times n$, so that the final result can be obtained by aggregating the partial results as $C = \sum_{i=1}^k A_i^T \times A_i$.

However, a more serious problem is posed by the second issue. The SVD function in MLlib is only able to handle tall and thin matrices up to some hundreds of features. For most of the language models, there are often hundreds of thousands features (e.g., 10^5 in our case). The covariance matrix C becomes too big to fit into the physical memory, hence the native SVD operation in MLlib of Spark fails as the first subfigure of Figure fig. 156 shows.

In linear algebra, a matrix can be approximated by another matrix of lower rank while still retaining approximately properties of the matrix that are important for the problem at hand. In other words, we can use another thinner matrix B to approximate the original fat A . The corresponding technique is referred to as rank-revealing QR estimation. We won't talk about this method in detail, but the basic idea is that, the columns are sparse and quite likely linearly dependent. If we can find the rank r of a matrix A and find suitable r columns to replace the original matrix, we can then approximate it. A TF-IDF model having 10^5 features often contains a lot of redundant information. Therefore, we can effectively thin the matrix A then fit C into the memory. Figure fig. 156 illustrates the algorithmic logic in DLSA, which is essentially a distributed stochastic SVD implementation.

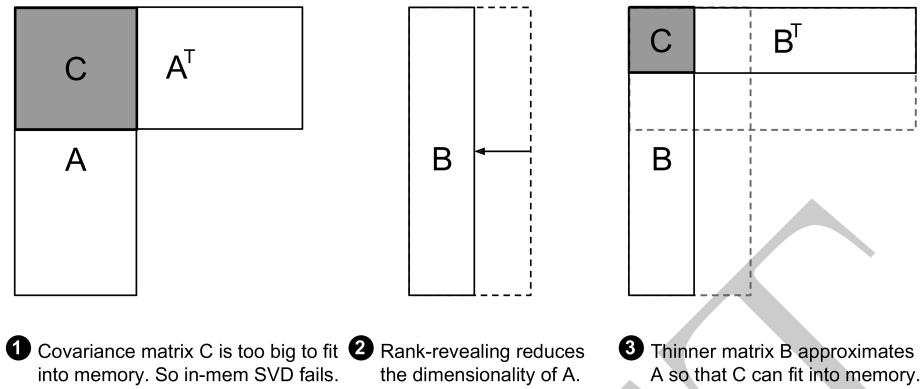


Figure 156: Rank-revealing reduces dimensionality to perform in-memory SVD

To sum up, we propose to reduce the size of TF-IDF model matrix to fit it into the memory, so that we can get a LSA model, where we know the document-topic and topic-word probability distribution.

Index Text Corpus

With a LSA model at hand, finding the most relevant document is equivalent to finding the nearest neighbours for a given point in the derived vector space, which is often referred to as k-NN problem. The distance is usually measured with the cosine similarity of two vectors. In the NLP chapter we have seen how to use linear search in the LSA model. However, neither naive linear search nor conventional k-d tree is capable of performing efficient search in such high dimensional space even though the dimensionality has been significantly reduced from 10^5 to 10^3 by LSA.

The key observation is that, we need not locate the exact nearest neighbours in practice. In most cases, slight numerical error (reflected in the language context) is not noticeable at all, i.e., the returned documents still look relevant from the user's perspective. By sacrificing some accuracy, we can obtain a significant gain in searching speed.

Random Projection

To optimise the search, the basic idea is that, instead of searching in all the existing vectors, we can pre-cluster the vectors according to their distances, each cluster with only a small number of vectors. For an incoming query, as long as we can put this vector into a suitable cluster, we can then search for close vectors only in that cluster.

fig. 157 gives a naive example on a 2-dimension vector space. First, a random vector x is drawn and all the points are projected onto x . Then we divide the

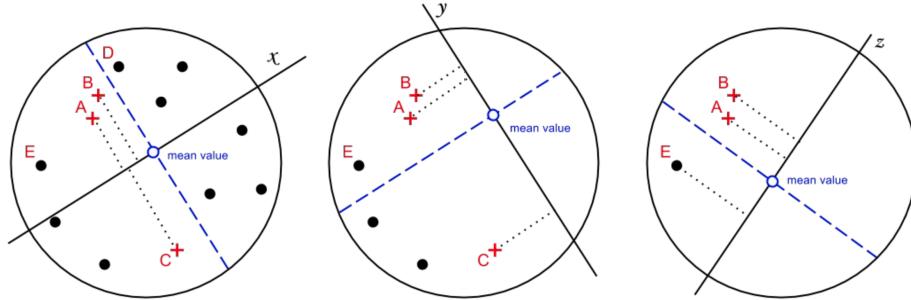


Figure 157: Projection on different random lines

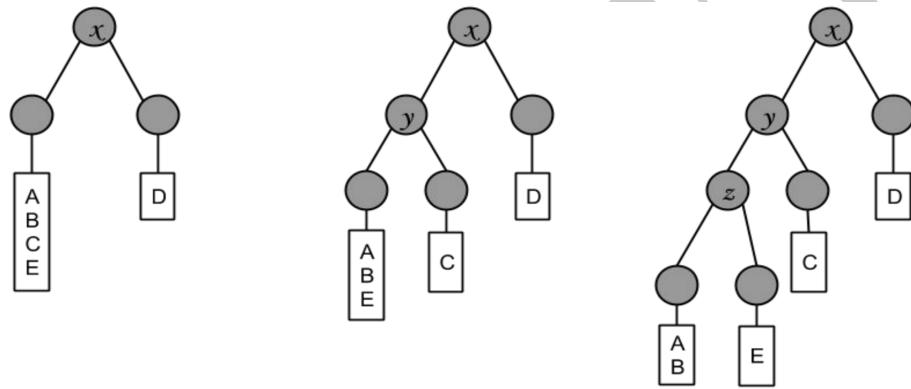


Figure 158: Construct a binary search tree from the random projection

whole space into half at the mean value of all projections (i.e., the blue circle on x) to reduce the problem size. For each new subspace, we draw another random vector for projection, and this process continues recursively until the number of points in the space reaches the predefined threshold on cluster size.

In the implementation, we can construct a binary tree to facilitate the search. Technically, this can be achieved by any tree-based algorithms. Given a tree built from a database, we answer a nearest neighbour query q in an efficient way, by moving q down the tree to its appropriate leaf cell, and then return the nearest neighbour in that cell. In Kvasir, we use the Randomised Partition tree (RP-tree) introduced in (Dasgupta and Sinha 2013) to do it. The general idea of RP-tree algorithm used here is clustering the points by partitioning the space into smaller subspaces recursively.

The fig. 158 illustrates how binary search can be built according to the dividing steps shown above. You can see the five nodes in the vector space are put into five clusters/leaves step by step. The information of the random vectors such as x , y , and z are also saved. Once we have this tree, given another query vector,

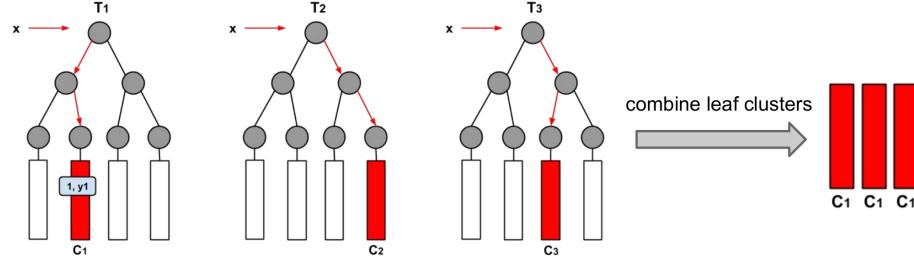


Figure 159: Aggregate clustering result from multipel RP-trees

we can put it into one of the clusters along the tree to find the cluster of vectors that are close to it.

Of course, we have already said that this efficiency is traded-off with search accuracy. One type of common misclassification is that it is possible that we can separate close vectors into different clusters. As we can see in the first subfigure of fig. 157, though the projections of A , B , and C seem close to each other on x , C is actually quite distant from A and B . The reverse can also be true: two nearby points are unluckily divided into different subspaces, e.g., points B and D in the left panel of fig. 157.

It has been shown that such misclassifications become arbitrarily rare as the iterative procedure continues by drawing more random vectors and performing corresponding splits. In the implementation, we follow this path and build multiple RP-trees. We expect that the randomness in tree construction will introduce extra variability in the neighbours that are returned by several RP-trees for a given query point. This can be taken as an advantage in order to mitigate the second kind of misclassification while searching for the nearest neighbours of a query point in the combined search set. As shown in fig. 159, given an input query vector x , we find its neighbour in three different RP-trees, and the final set of neighbour candidates comes from the union of these three different sets.

Optimising Vector Storage

You may have noticed that, in this method, we need to store all the random vectors that are generated in the non-leaf nodes of the tree. That means storing a large number of random vectors at every node of the tree, each with a large number features. It introduces significant storage overhead. For a corpus of 4 million documents, if we use 10^5 random vectors (i.e., a cluster size of $\frac{4 \times 10^6}{2 \times 10^5} = 20$ on average), and each vector is a 10^3 -dimension real vector (32-bit float number), the induced storage overhead is about 381.5~MB for each RP-tree. Therefore, such a solution leads to a huge index of 47.7~GB given 128 RP-trees are included, or 95.4~GB given 256 RP-trees.

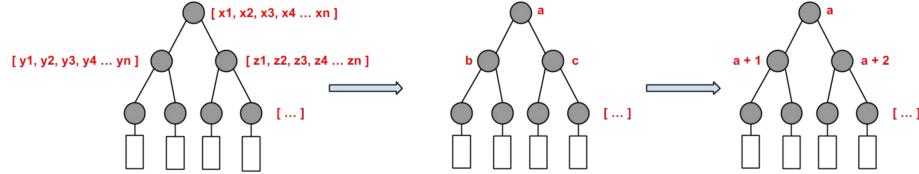


Figure 160: Use a random seed to generate on the fly

The huge index size not only consumes a significant amount of storage resources, but also prevents the system from scaling up after more and more documents are collected. One possible solution to reduce the index size is reusing the random vectors. Namely, we can generate a pool of random vectors once, and then randomly choose one from the pool each time when one is needed. However, the immediate challenge emerges when we try to parallelise the tree building on multiple nodes, because we need to broadcast the pool of vectors onto every node, which causes significant network traffic.

To address this challenge, we propose to use a pseudo random seed in building and storing search index. Instead of maintaining a pool of random vectors, we just need a random seed for each RP-tree. As shown in fig. 160, in a leaf cluster, instead of storing all the vectors, only the indices of vectors in the original data set are stored. The computation node can build all the random vectors on the fly from the given seed according to the random seed.

From the model building perspective, we can easily broadcast several random seeds with negligible traffic overhead instead of a large matrix in the network. In this way we improve the computation efficiency. From the storage perspective, we only need to store one 4-byte random seed for each RP-tree. In such a way, we are able to successfully reduce the storage overhead from 47.7~GB to 512~B for a search index consisting of 128 RP-trees (with cluster size 20), or from 95.4~GB to only 1~KB if 256 RP-trees are used.

Optimise Data Structure

Let's consider a bit more about using multiple RP-trees. Regarding the design of PANNS, we have two design options in order to improve the searching accuracy. Namely, given the size of the aggregated cluster which is taken as the union of all the target clusters from every tree, we can either use fewer trees with larger leaf clusters, or use more trees with smaller leaf clusters. Increasing cluster size is intuitive: if we increase it to so large that includes all the vectors, then it is totally accurate.

On the other hand, we expect that when using more trees the probability of a query point to fall very close to a splitting hyperplane should be reduced, thus it should be less likely for its nearest neighbours to lie in a different cluster. By reducing such misclassifications, the searching accuracy is supposed to be

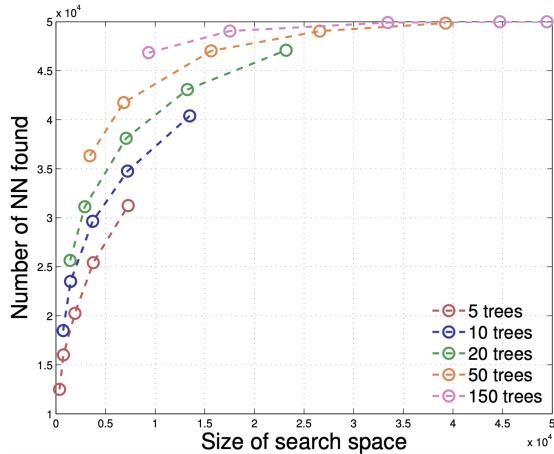


Figure 161: The number of true nearest neighbours found for different number of trees

improved. Based on our knowledge, although there are no previous theoretical results that may justify such a hypothesis in the field of nearest neighbour search algorithms, this concept could be considered as a combination strategy similar to those appeared in ensemble clustering, a very well established field of research. Similar to our case, ensemble clustering algorithms improve clustering solutions by fusing information from several data partitions.

To experimentally investigate this hypothesis we employ a subset of the Wikipedia database for further analysis. In what follows, the data set contains 500,000 points and we always search for the 50 nearest neighbours of a query point. Then we measure the searching accuracy by calculating the amount of actual nearest neighbours found.

We query 1,000 points in each experiment. The results presented in fig. 161 correspond to the mean values of the aggregated nearest neighbours of the 1,000 query points discovered by PANNS out of 100 experiment runs. Note that x -axis represents the “size of search space” which is defined by the number of unique points within the union of all the leaf clusters that the query point falls in. Therefore, given the same search space size, using more trees indicates that the leaf clusters become smaller. As we can see in fig. 161, for a given x value, the curves move upwards as we use more and more trees, indicating that the accuracy improves. As shown in the case of 50 trees, almost 80% of the actual nearest neighbours are found by performing a search over the 10% of the data set.

Our empirical results clearly show *the benefits of using more trees instead of using larger clusters for improving search accuracy*. Moreover, regarding the searching performance, since searching can be easily parallelised, using more

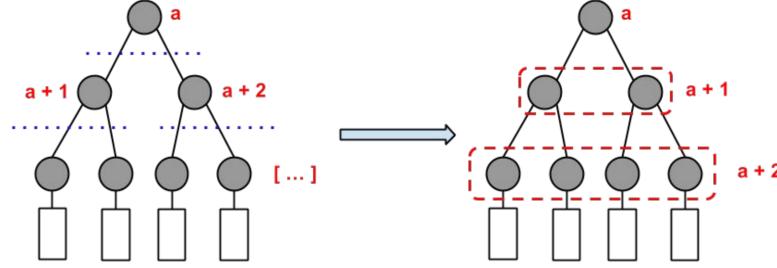


Figure 162: Illustration of parallelising the computation.

trees will not impact the searching time.

Optimise Index Algorithm

In classic RP trees we have introduced above, a different random vector is used at each inner node of a tree. In this approach, the computations in the child-branches cannot proceed without finishing the computation in the parent node, as show in the left figure of fig. 162. Here the blue dotted lines are critical boundaries. Instead, we propose to use the same random vector for all the sibling nodes of a tree. This choice does not affect the accuracy at all because a query point is routed down each of the trees only once; hence, the query point is projected onto a random vector r_i sampled from the same distribution at each level of a tree. This means that we don't need all the inner non-leaf node to be independent random vectors. Instead, the query point is projected onto only l i.i.d. random vectors r_1, \dots, r_l . An RP-tree has $2^l - 1$ inner nodes. Therefore, if each node of a tree had a different random vector as in classic RP-trees, $2^l - 1$ different random vectors would be required for one tree. However, when a single vector is used on each level, only l vectors are required. This reduces the amount of memory required by the random vectors from exponential to linear with respect to the depth of the trees.

Besides, another extra benefit of using one random vector for one layer is that it speeds up the index construction significantly, since we can vectorise the computation. Let's first look at the projection of vector a on b . The projected length on b can be expressed as:

$$\|a\| \cos \theta = a \cdot \frac{b}{\|b\|}. \quad (80)$$

Here $\|a\|$ means the length of vector \mathbf{a} . If we requires that all the random vectors \mathbf{b} has to be normalised, eq. 80 becomes $a \cdot b$, the vector dot. Now we can perform the projection at this layer by computing: Xb_l . Here X is the dataset, and each row is a document and each column is a feature; b_l is a random vector that we

use for this layer. In this way, we don't have to wait for the left tree to finish to start cutting the right tree.

Now here is the tricky bit: we don't even have to wait for the upper layer to start cutting the lower layer! The reason is that, at each layer, we do random projection of *all the nodes* in the dataset on one single random vector b . We don't really care the random clustering result from the previous layer. Therefore, we can perform Xb_1, Xb_2, \dots, Xb_l at the same time. That means, the projected data set P can be computed directly from the dataset X and a random matrix B as $P = XB$ with only one pass of matrix multiplication. Here each column of B is just the random vector we use at a layer.

In this approach there is no boundary, and all the projections can be done in just one matrix multiplication. While some of the observed speed-up is explained by a decreased amount of the random vectors that have to be generated, mostly it is due to enabling efficient computation of all the projections. Although the total amount of computation stays the same, in practice this speeds up the index construction significantly due to the cache effects and low-level parallelisation through vectorisation. The matrix multiplication is a basic linear algebra operation and many low level numerical libraries, such as OpenBLAS and MKL, provide extremely high-performance implementation of it.

Search Articles

By using RP-tree we have already limit the search range from the whole text corpus to only a cluster of small number of documents (vectors), where we can do a linear searching. We have also introduced several optimisations on the RP-tree itself, including using multiple trees, using random seed to remove the storage of random vectors, improving computation efficiency etc. But we don't stop here: can we further improve the linear searching itself? It turns out, we can.

To select the best candidates from a cluster of points, we need to use the coordinates in the original space to calculate their relative distance to the query point. This however, first increases the storage overhead since we need to keep the original high dimensional data set which is usually huge; second increases the query overhead since we need to access such data set. The performance becomes more severely degraded if the original data set is too big to load into the physical memory. Moreover, computing the distance between two points in the high dimensional space per se is very time-consuming.

Nonetheless, we will show that it is possible to completely get rid of the original data set while keeping the accuracy at a satisfying level. The core idea of is simple. Let's look at the second subfigure in fig. 157. Imagine that we add a new point to search for similar vectors. The normal approach is that we compute the distance between this node and A, B, C etc. But if you look at it close, all the existing nodes are already projected on the vector y , and we can also project the incoming query vector on y , and check to which of these points it is close

to. Instead of computing the distances of two vectors, now we only compute the absolute value of subtraction of two numbers (since we can always project a vector onto another one and get a real number as result) as the distance. By replacing the original space with the projected one, we are able to achieve a significant reduction in storage and non-trivial gains in searching performance.

Of course, it is not always an accurate estimation. In the first subfigure of fig. 157, a node can be physically close to \mathbf{a} or \mathbf{b} , but its projection could be closest to that of \mathbf{c} . That again requires us to consider using multiple RP-trees. But instead of the actual vector content, in the leaf node of the trees we store only `(index, projected value)`. Now for the input query vector, we run it in the N RP-trees and get N set of `(index, value)` pairs. Here each value is the absolute value of the difference of projected values between the vector in the tree and the query vector itself. Each vector of course is label by a unique index.

For each index, we propose to use this metric: $\sum \frac{d_i}{c_i}$ to measure how close it is to the query vector. Here d_i is the distance between node i and query node on projected space, and c_i is the count of total number of node i in all the candidate sets from all the RP-trees. Smaller measurement means closer distance. The intuition is that, if distance value of a node on the projected space is small, then it is possibly close to the query node; or, if a node appears many times from the candidate sets of different RP-trees, it is also quite likely a possible close neighbour.

As a further improvement, we update this metric to $\frac{\sum d_i}{(\sum c_i)^3}$. By so doing, we give much more weight on the points which have multiple occurrences from different RP-trees by assuming that such points are more likely to be the true k-NN. Experiment results confirm that by using this metric it is feasible to use only the projected space in the actual system implementation. Please refer to the original paper if you are interested with more detail.

Code Implementation

What we have introduced is the main theory behind the Kvasir, a smart content discovery tool to help you manage this rising information flood. In this chapter, we will show some naive code implementation in OCaml and Owl to help you better understand what we have introduced so far.

First, we show the simple random projection along a RP-tree.

```
let make_projection_matrix seed m n =
  Owl_stats_prng.init seed;
  Mat.gaussian m n |> Mat.to_arrays

let make_projected_matrix m n =
  Array.init m (fun _ -> Array.make n 0.)
```

These two functions make projection matrix and the matrix to save projected results, both return as row vectors.

```
let project i j s projection projected =
  let r = ref 0. in
  Array.iter (fun (w, a) ->
    r := !r +. a *. projection.(w).(j));
  ) s;
  projected.(j).(i) <- !r
```

Based on these two matrices, the `project` function processes document `i` on the level `j` in the tree. The document vector is `s`. The projection is basically a dot multiplication between `s` and matrix `projection`.

```
let random seed cluster tfidf =
  let num_doc = Nlp.Tfidf.length tfidf in
  let vocab_len = Nlp.Tfidf.vocab_len tfidf in
  let level = Maths.log2 (float_of_int num_doc /. cluster) |> ceil |> int_of_float
  in

  let projection = make_projection_matrix seed vocab_len level in
  let projected = make_projected_matrix level num_doc in

  Nlp.Tfidf.iteri (fun i s ->
    for j = 0 to level - 1 do
      project i j s projection projected;
    done;
  ) tfidf;

  vocab_len, level, projected
```

The `random` function performs a random projection of sparse data set, based a built TF-IDF model. Technically, a better way is to use LSA model as the vectorised representation of documents as we have introduced above, since a LSA model acquired based on TF-IDF represents more abstract idea of topics and has less features. However, here it suffices to use the TF-IDF model to show the random projection process. This function projects all the document vectors in the model to the projected matrix, level by level. Recall that the result only contains the projected value instead of the whole vector.

As we have explained in the “Search Articles” section, this process can be accelerated to use matrix multiplication. The code below shows this implementation for the random projection function. It also returns the shape of projection and the projected result.

```
let make_projection_matrix seed m n =
  Owl_stats_prng.init seed;
  Mat.gaussian m n
```

```

let random seed cluster data =
  let m = Mat.row_num data in
  let n = Mat.col_num data in
  let level = Maths.log2 (float_of_int m /. cluster) |> ceil |> int_of_float in

  let projection = make_projection_matrix seed n level in
  let projected = Mat.dot data projection |> Mat.transpose in

  n, level, projected, projection

```

After getting the projection result, we need to build a RP-tree accordingly. The following is about how to build the index in the form of a binary search tree. The tree is defined as:

```

type t =
| Node of float * t * t (* intermediate nodes: split, left, right *)
| Leaf of int array (* leaves only contains doc_id *)

```

An intermediate node includes three parts: split, left, right, and the leaves only contain document index.

```

let split_space_median space =
  let space_size = Array.length space in
  let size_of_l = space_size / 2 in
  let size_of_r = space_size - size_of_l in
  (* sort into increasing order for median value *)
  Array.sort (fun x y -> Pervasives.compare (snd x) (snd y)) space;
  let median =
    match size_of_l < size_of_r with
    | true -> snd space.(size_of_l)
    | false -> (snd space.(size_of_l-1) +. snd space.(size_of_l)) /. 2.
  in
  let l_subspace = Array.sub space 0 size_of_l in
  let r_subspace = Array.sub space size_of_l size_of_r in
  median, l_subspace, r_subspace

```

The `split_space_median` function divides the projected space into subspaces to assign left and right subtrees. The passed in `space` is the projected values on a specific level. The criterion of division is the median value. The `Array.sort` function sorts the space into increasing order for median value.

```

let filter_projected_space level projected subspace =
  let plevel = projected.(level) in
  Array.map (fun (doc_id, _) -> doc_id, plevel.(doc_id)) subspace

```

Based on the document id of the points in the subspace, `filter_projected_space` function filters the projected space. The purpose of this function is to update the projected value using a specified level so the recursion can continue. Both the space and the returned result are of the same format: `(doc_id, projected value)`.

```

let rec make_subtree level projected subspace =
  let num_levels = Array.length projected in
  match level = num_levels with
  | true -> (
    let leaf = Array.map fst subspace in
    Leaf leaf
  )
  | false -> (
    let median, l_space, r_space = split_space_median subspace in
    let l_space = match level < num_levels - 1 with
      | true -> filter_projected_space (level+1) projected l_space
      | false -> l_space
    in
    let r_space = match level < num_levels - 1 with
      | true -> filter_projected_space (level+1) projected r_space
      | false -> r_space
    in
    let l_subtree = make_subtree (level+1) projected l_space in
    let r_subtree = make_subtree (level+1) projected r_space in
    Node (median, l_subtree, r_subtree)
  )

```

Based on these functions, the `make_subtree` recursively grows the binary subtree to make a whole tree. The `projected` is the projected points we get from the first step. It is of shape `(level, document_number)`. The `subspace` is a vector of shape `(1, document_number)`.

```

let grow projected =
  let subspace = Array.mapi (fun doc_id x -> (doc_id, x)) projected.(0) in
  let tree_root = make_subtree 0 projected subspace in
  tree_root

```

The `grow` function calls `make_subtree` to build the binary search tree. It initialises the first subspace at level 0, and then start recursively making the subtrees from level 0. Currently everything is done in memory for efficiency consideration.

```

let rec traverse node level x =
  match node with
  | Leaf n -> n
  | Node (s, l, r) -> (
    match x.(level) < s with
    | true -> traverse l (level+1) x
    | false -> traverse r (level+1) x
  )

```

Now that the tree is built, we can perform search on it. The recursive `traverse` function traverses the whole tree to locate the cluster for a projected vector `x` starting from a given level.

```

let rec iter_leaves f node =
  match node with
  | Leaf n -> f n
  | Node (s, l, r) -> iter_leaves f l; iter_leaves f r

let search_leaves node id =
  let leaf = ref [] in
  (
    try iter_leaves (fun l ->
      if Array.mem id l = true then (
        leaf := l;
        failwith "found";
      )
    ) node
    with exn -> ()
  );
  Array.copy !leaf

```

Finally, `search_leaves` returns the leaves/clusters which have the given `id` inside it. It mainly depends on the `iter_iterate` function which iterates all the leaves in a tree and applies function, to perform this search.

All these code above is executed on one tree. When we collect the k-NN candidates from all the trees, instead of calculating the vector similarity, we utilise the frequency/count of the vectors in the union of all the candidate sets from all the RP-trees.

```

let count_votes nn =
  let h = Hashtbl.create 128 in
  Owl_utils.aarr_iter (fun x ->
    match Hashtbl.mem h x with
    | true -> (
      let c = Hashtbl.find h x in
      Hashtbl.replace h x (c + 1)
    )
    | false -> Hashtbl.add h x 1
  ) nn;
  let r = Array.make (Hashtbl.length h) (0,0) in
  let l = ref 0 in
  Hashtbl.iter (fun doc_id votes ->
    r.(!l) <- (doc_id, votes);
    l := !l + 1;
  ) h;
  Array.sort (fun x y -> Pervasives.compare (snd y) (snd x)) r;
  r

```

The `count_votes` function takes in an array of array `nn` as input. Each inner array contains the indexes of candidate nodes from one RP-tree. These nodes are collected into a hash table, using index as key and the count as value. Then the results are sorted according to the count number.

Make It Live

We provide a live demo of Kvasir. Here we briefly introduce the implementation of the demo with OCaml. This demo mainly relies on Lwt. The Lwt library implements cooperative threads. It is often used as web server in OCaml.

This demo takes in document in the form of web query API and returns similar documents in the text corpus already included in our backend. First, we need to do some simple preprocessing using regular expression. This of course needs some fine tuning in the final product, but needs to be simple and fast.

```
let simple_preprocess_query_string s =
  let regex = Str.regexp "[=+%0-9]+"
  Str.global_replace regex " " s
```

The next function `extract_query_params` parse the web query, and retrieves parameters.

```
let extract_query_params s =
  let regex = Str.regexp "num=\\"([0-9]+\\\")" in
  let _ = Str.search_forward regex s 0 in
  let num = Str.matched_group 1 s |> int_of_string in

  let regex = Str.regexp "mode=\\"([a-z]+\\\")" in
  let _ = Str.search_forward regex s 0 in
  let mode = Str.matched_group 1 s in

  let regex = Str.regexp "doc=\\"(.+\\\")" in
  let _ = Str.search_forward regex s 0 in
  let doc = Str.matched_group 1 s in

  (num, mode, doc)
```

Finally, `start_service` function includes the core query service that keeps running. It preprocesses the input document and processed with similar document searching according to different search mode. We won't cover the details of web server implementation details using Lwt. Please refer to its documentation for more details.

```
let start_service lda idx =
  let num_query = ref 0 in
  let callback _conn req body =
    body |> Cohttp_lwt_body.to_string >|= (fun body ->
      let query_len = String.length body in
      match query_len > 1 with
      | true -> (
        try (
          let num, mode, doc = extract_query_params body in
          Log.info "process query #%i ... %i words" !num_query query_len;
          num_query := !num_query + 1;
```

```

let doc = simple_preprocess_query_string doc in
match mode with
| "linear" -> query_linear_search ~k:num lda doc
| "kvasir" -> query_kvasir_idx ~k:num idx lda doc
| _ -> failwith "kvasir:unknown search mode"
)
with exn -> "something bad happened :("
)
| false -> (
  Log.warn "ignore an empty query";
  ""
)
>>= (fun body -> Server.respond_string ~status:`OK ~body ())
in
Server.create ~mode:(`TCP (`Port 8000)) (Server.make ~callback ())

```

Summary

In this chapter, we presented Kvasir which provides seamless integration of LSA-based content provision into web browsing. To build Kvasir as a scalable Internet service, we addressed various technical challenges in the system implementation. Specifically, we proposed a parallel RP-tree algorithm and implemented stochastic SVD on Spark to tackle the scalability challenges in index building and searching. We have introduced the basic algorithm and how it can optimised step by step, from storage to computation. These optimisations include aggregating results from multiple trees, replacing random variable with a single random seed, removing the projection computation boundary between different layers, using count to approximate vector distance, etc. Thanks to its novel design, Kvasir can easily achieve millisecond query speed for a 14 million document repository. Kvasir is an open-source project and is currently under active development. The key components of Kvasir are implemented as an Apache Spark library, and all the source code are publicly accessible on GitHub.

References

- Dasgupta, Sanjoy, and Kaushik Sinha. 2013. “Randomized Partition Trees for Exact Nearest Neighbor Search.” In *Conference on Learning Theory*, 317–37.
- Hyvönen, V., T. Pitkänen, S. Tasoulis, E. Jääsaari, R. Tuomainen, L. Wang, J. Corander, and T. Roos. 2016. “Fast Nearest Neighbor Search Through Sparse Random Projections and Voting.” In *2016 Ieee International Conference on Big Data (Big Data)*, 881–88. <https://doi.org/10.1109/BigData.2016.7840682>.
- Wang, L., S. Tasoulis, T. Roos, and J. Kangasharju. 2016. “Kvasir: Scalable Provision of Semantically Relevant Web Content on Big Data Framework.” *IEEE Transactions on Big Data* 2 (3): 219–33. <https://doi.org/10.1109/TBDA.2016.2557348>.

Case - Applications in Finance

Introduction

Introduce the importance of numerical computing in Finance. Introduce what companies use what functional languages in their applications.

Bond Pricing

Net present value ...

Black-Scholes Model

Mathematical Model

Option Pricing

Portfolio Optimisation

Introduce Modern portfolio theory (MPT), Markowitz model.

Mathematical Model

Efficient Frontier

Maximise Sharpe Ratio

Appendix

DRAFT

Acknowledgement

Owl is built on top of an enormous amount of previous work. Without the efforts of these projects and the intellectual contribution of these people, it will be very difficult for me to continue developing Owl.

We thank our sponsors all the contributors, Owl can grow and help many people in solving various real world problems because of their generous support.

OCaml Labs has been providing various types of support including computation resources, internship and studentship revolving around Owl's research and development.

Theses

Probabilistic Synchronous Parallel

By Benjamin P. W. Catterall | Part III | June 2017

The synchronisation scheme used to manage parallel updates of a distributed machine learning model can dramatically impact performance. System and algorithm designers need methods which allow them to make trade-offs between fully asynchronous and fully deterministic schemes. Barrier control methods represent one possible solution. In this report, I present Probabilistic Synchronous Parallel (PSP), a barrier control method for distributed machine learning. I provide analytical proofs of convergence and carry out an experimental verification of the method using a bespoke simulator. I find that PSP improves the convergence speed and iteration throughput of more traditional barrier control methods. Furthermore, I demonstrate that PSP provides stronger convergence guarantees than a fully asynchronous design whilst maintaining the general characteristics of stronger methods.

Supporting Browser-based Machine Learning

By Tudor Petru Tiplea | Part III | June 2018

Because it can tell so much about nature and people, digital data is collected and analysed in immeasurable quantities. Processing this data often requires collections of resources typically organised in massive data centres, a paradigm known as cloud computing. However, this paradigm has certain limitations. Apart from the often prohibitive costs, cloud computing requires data centralisation, which could slow down real-time applications, or require exorbitant storage. Edge computing—a solution aiming to move computation to the network’s edge—is regarded as a promising alternative, especially when tailored for Internet-of-Things deployment.

Aiming for more large-scale adoption, this project provides a proof of concept for edge computing support on an ubiquitous platform—the web-browser. This work is framed within an emerging OCaml ecosystem for data processing and machine learning applications. We explored options for OCaml-to-JavaScript compilation, and extended Owl, the main library in the ecosystem, guided by those findings. Next, we researched solutions for efficient data transmissions between browsers, then based on that, implemented a browser-compatible communication system analogous to TCP/IP network sockets. This system was later used to modify Actor, Owl’s distributed computing engine, making it deployable in the browser.

We demonstrated our work on Owl was successful, exemplifying the browser-deployed localised computing capabilities. The performance limitations of this part were analysed, and we suggest directions for optimisations based on empirical results. We also illustrated the accomplishment of browser-based distributed computing, again identifying limitations that must be overcome in the future for

a complete solution.

Adaptable Asynchrony in Distributed Learning

By De Sheng Royson Lee | M.Phil | June 2018

Distributed training of deep learning models is typically trained using stochastic optimisation in an asynchronous or synchronous environment. Increasing asynchrony is known to add noise introduced from stale gradient updates, whereas relying on synchrony may be inefficient due to stragglers. Although there has been a wide range of approaches to mitigate or even negate these weaknesses, little has been done to improve asynchronous adaptive stochastic gradient descent (SGD) optimisation. In this report, I survey these approaches and propose a technique to better train these models. In addition, I empirically show that the technique works well with delay-tolerance adaptive SGD optimisation algorithms, improving the rate of convergence, stability, and test accuracy. I also demonstrate that my approach performs consistently well in a dynamic environment in which the number of workers changes uniformly at random.

Applications of Linear Types

By Dhruv C. Makwana | Part III | June 2018

In this thesis, I argue that linear types are an appropriate, type-based formalism for expressing aliasing, read/write permissions, memory allocation, re-use and deallocation, first, in the context of the APIs of linear algebra libraries and then in the context of matrix expression compilation. I show that framing the problem using linear types can reduce bugs by making precise and explicit, the informal, ad-hoc practices typically employed by experts and matrix expression compilers and automate checking them.

As evidence for this argument, I show non-trivial, yet readable, linear algebra programs, that are safe and explicit (with respect to aliasing, read/write permissions, memory allocation, re-use and deallocation) which (1) are more memory-efficient than equivalent programs written using high-level linear algebra libraries and (2) perform just as predictably as equivalent programs written using low-level linear algebra libraries. I also argue the experience of writing such programs with linear types is qualitatively better in key respects. In addition to all of this, I show that it is possible to provide such features as a library on top of existing programming languages and linear algebra libraries.

Composing Data Analytical Services

By Jianxin Zhao | PhD | June 2018

Data analytics on the cloud is known to have issues such as increased response latency, communication cost, single point failure, and data privacy concerns. While moving analytics from cloud to edge devices has recently gained rapid

growth in both academia and industry, this topic still faces many challenges such as limited computation resource on the edge. In this report, we further identify two main challenges: the composition and deployment of data analytics services on edge devices. Initially, the Zoo system is designed to make it convenient for developers to share and execute their OCaml code snippets, with fine-grained version control mechanism. We then extend it to address those two challenges. On one hand, Zoo provides simple domain-specific language and high-level types to enable easy and type-safe composition of different data analytics services. On the other hand, it utilises multiple deployment backends, including Docker container, JavaScript, and MirageOS, to accommodate the heterogeneous edge deployment environment. We demonstrate the expressiveness of Zoo with a use case, and thoroughly compare the performance of different deployment backends in evaluation.

Computer Vision in OCaml

By Pierre Vandenhove | MSc | October 2018

Computer vision tasks are known to be highly computationally-heavy, both performance-wise and memory-wise. They are thus especially relevant to put a numerical framework such as Owl to the test. The first part of this project focuses on the implementation of several computer vision applications using Owl's neural network library. The first such application is Microsoft's 'ResNet' network to perform simple image classification (paper 1512.03385, Resnet implementation in Owl). The second, more extensive one, is 'Mask R-CNN', which is one of the leading networks to perform object detection, segmentation and classification (paper 1703.06870, MRCNN implementation). This allowed exemplifying some use cases to improve Owl's flexibility and ease of use, as well as add some necessary operations.

These applications are valuable benchmarking tools to identify bottlenecks and guide the optimisation of different subcomponents of Owl. A crucial step in this process is to apply Owl's computation graph to them, which is the key to obtaining state-of-the-art performance and memory usage. With the new applications as examples, it was possible to make it more robust, efficient and user-friendly.

Automatic Parameter Tuning for OpenMP

By Jianxin Zhao | PhD | November 2018

Automatic Empirical Optimisation of Software (AEOS) is crucial for high performance computing software. It is a methodology to generate optimised software using empirically tuned parameters. As an initial attempt to improve the performance of Owl with it, we build the AEOS module to tune the OpenMP parameters in Owl. OpenMP is an application programming interface that supports multi-platform shared memory multiprocessing programming. It is

used in Owl to boost performance of basic operations. However, using OpenMP brings certain overhead, so that when the size of input data is small, or the operation is simple, the non-OpenMP version operation might be faster. Thus an optimal threshold varies for different operations and machines. In the AEOS module, each operation is abstracted as a stand-alone module, and uses linear regression to find this optimal threshold. Compared with the previous practice of set a single threshold for all OpenMP operations, using AEOS module further improves their performance. The AEOS module is designed in such way that extending it to accommodate more parameters or operations should be easy.

Run Your Owl Computation on TensorFlow

By Jianxin Zhao | PhD | February 2019

In this project we are looking at computation interoperability of Owl with existing libraries such as TensorFlow. Our target is to have the best of both worlds. On one hand, we can define “how to compute” on Owl with its elegant and powerful syntax; on the other hand, we can execute the computation efficiently across various hardware devices, such as GPU and TPU, that TensorFlow supports. One crucial decision to make is to find the correct intermediate representation in exchanging computation between different platforms. Unlike many existing systems and tools, we decide that computation graph, rather than neural network graph, should be the fundamental abstraction. Based on this decision, we build an experimental converter system. It aims to export CGraph defined in Owl and execute it in TensorFlow. This system centres around the abstraction of TensorFlow computation graph, and how to map Owl computation graph to it. Our system utilises the Save and Restore mechanism in TensorFlow to provide a concise workflow. Currently we are actively developing the system. Though still quite limited at the initial phase, the system has shown its potential in real-world examples, including deep neural network inference and algorithmic differentiation. In our next step, it would be interesting to see how our system can be extended and combined with related topics such as GPU and XLA.

Ordinary Differential Equation Solver

By Ta-Chu Kao and Marcello Seri | July 2019

Owl Ode is a lightweight package for solving ordinary differential equations. Built on top of Owl’s numerical library, Owl Ode was designed with extensibility and ease of use in mind and includes a number of classic ode solvers (e.g. Euler and Runge-Kutta, in both adaptive and fixed-step variants) and symplectic solvers (e.g. Leapfrog), with more to come. Taking full advantage of Owl’s automatic differentiation library, we plan on supporting a number of fully differentiable solvers, which can be used, for example, for training Neural Odes.

Currently, Owl Ode includes separately-released thin wrappers around Sundials Cvode (via sundialsml’s own wrapper) and ODEPACK, native ocaml contact

variational integrators, and exposes a fully native ocaml module compatible with js_of_ocaml (owl-ode-base). Going forward, we aim to expose more functions in Sundials, make the API even more flexible and configurable, and provide bindings for other battle-tested ODE solvers.

DRAFT

Resources

list some useful links here

DRAFT

Epilogue

Back in the summer of 2019, Liang and I were considering the maintenance of Owl's documentation. We were glad that the documentation serve us well and was growing day by day. Then somehow it occurred to us that, now that we have such a large documentation at hand, as well as paper drafts and blogs etc., why don't we put these different pieces together into something like a tutorial book. That's basically the root of this book.

We then discussed the possible ways to organise the book. One obvious and convenient way is to extending the existing materials according to the different modules of Owl. However, while the architectures of Owl may change easily in the future, the topic of scientific computing does not. In the end, we decided to more or less follow the outlines of the traditional textbooks in scientific computing and numerical computing etc. However, we do not intend to make it yet another data science/machine learning/numerical computing algorithm book, and you may find this book a little bit different.

The whole book is divided into three parts. The first part "numerical techniques" covers a wide range of topics. The Statistics and Linear Algebra are traditional mathematical topics; manipulation and slicing of ndarray belongs to the introductory part of a language; visualisation and dataframes are often placed in the first chapter of a data science book; differential equation, signal processing, and optimisation are among the core topics of numerical methods; regression, DNN, and NLP are surely covered in numerous Machine Learning books. There is also AD, the topic that is less mentioned but nevertheless extremely important. The reason we mix all these topics together is that, instead of letting the content of book to follow the structure of Owl, we want Owl to follow as many topics as possible that are related to scientific computing, and check our coverage of them. We are glad to find out that Owl can be of help in most of these listed topics. Also, it provides a good guidance for our next step development.

The second part is about system architecture of Owl. Most of the topics discussed here, such as distributed computing, low level optimisation, computation graph, etc. are not often seen in other books. Maybe they are too academic for general readers. However, as the creator and developer of Owl, what we can provide is not just "how to use it", but also "how it is built".

The third part demonstrates several cases. While the foundation of scientific computing is solid and concrete, its applications are springing out at an astounding speed. Each chapter shows one sophisticated real world use case of Owl. Currently we have three computer vision applications: image recognition, instance segmentation, and style transfer. They are widely used in the state-of-art technologies. The recommender system is also based on a product Kvasir that's deployed in the production environment. This part is meant to be extendable. We are very interested to add more cases contributed by experts from other fields and disciplines.

So that's a brief explanation about the organisation of this book. I feel obliged to explain to those readers who are confused about the theme of this book. No doubt that there are still a lot to be improved about this book. Almost all the chapters in the first part can be extended to at least one whole book, and there is no way we can claim ourselves master of each one of them. We will keep fixing and updating this book in the second and third editions. Also, while the core of scientific computing keeps stable for the last several decades, its applications are changing day by day. More state-of-art use cases should be included in the third part.

Writing this book surely feels like a long journey for me, and what a journey it is. I joined the Owl project at about 2017, and worked on two parts: optimising the low level operations, and building high level DNN applications to verify and drive the design of Owl architecture. Working on these two extremes gives me a great opportunity to get to know the Owl stack. To debug a neural network application, I have to understand how the modules work with each other, how the computation graph is built, and how it is affected by the performance of single operations, etc. Some of the chapters are actually based on the learning notes of myself. For example, the AD chapter starts with the definition of algorithmic differentiation, builds a toy implementation based on one example, and then keeps revising the solution with more and more details. That's exactly how I learned this topic in Owl. You can also find this learn-and-write style in several other chapters. If you can learn something by following the way we did it, I would call this book a great success. *"For here, I hope, begins our lasting joy."*

