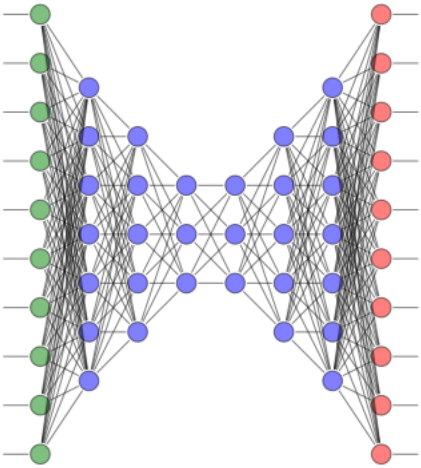
Deep Learning Notes using Julia with Flux



Hugh Murrell
hugh.murrell@gmail.com

Nando de Freitas
nandodefreitas@google.com

please cite using [12].

Contents

Notation **v**

1 Introduction **1**
1.1 Acknowledgements . . . 1
1.2 Guinea Pigs . . . 1
1.3 Machine Learning . . . 2
1.4 Introductory Notebook . . . 2
1.5 Next Chapter . . . 2

2 Linear Algebra **3**
2.1 Introductory . . . 3
2.2 Advanced . . . 4
2.3 Linear Algebra Notebook . . . 7
2.4 Next Chapter . . . 7

3 Linear Regression **8**
3.1 Outline . . . 8
3.2 Learning Data . . . 9
3.3 The learning/prediction cycle . . . 9
3.4 Minimum Loss via Hook's law . . . 10
3.5 General Linear Regression . . . 11
3.6 Optimization . . . 12
3.7 Linear Regression Notebook . . . 13
3.8 Next Chapter . . . 14

4 Maximum Likelihood **15**
4.1 Outline . . . 15
4.2 Univariate Gaussian Distribution . . . 15
4.3 Moments . . . 17

4.4 Covariance . . . 20
4.5 Multivariate Gaussian . . . 21
4.6 Likelihood . . . 22
4.7 Likelihood for linear regression . . . 23
4.8 Entropy . . . 26
4.9 Entropy of a Bernoulli random variable . . . 26
4.10 Kullback Leibler divergence . . . 27
4.11 Entropy Notebook . . . 28
4.12 Next Chapter . . . 28

5 Ridge Regression **29**
5.1 Outline . . . 29
5.2 Regularisation . . . 29
5.3 Ridge Regression via Bayes Rule . . . 33
5.4 Going non-linear using basis functions . . . 34
5.5 Kernel Regression . . . 35
5.6 Cross Validation . . . 37
5.7 Ridge Regression with Basis Functions Notebook . . . 38
5.8 Next Chapter . . . 38

6 Optimisation **39**
6.1 Outline . . . 39
6.2 The Gradient and Hessian . . . 39
6.3 Multivariate Taylor expansion . . . 40
6.4 Steepest Descent Algorithm . . . 41
6.5 Momentum . . . 42
6.6 Newton's Method . . . 43
6.7 Gradient Descent for Least Squares . . . 43
6.8 Newton CG algorithm . . . 44
6.9 Stochastic Gradient Descent . . . 44
6.10 Adaptive Gradient Descent . . . 45
6.11 Batch Normalisation . . . 47
6.12 Stochastic Gradient Descent Notebook . . . 48
6.13 Next Chapter . . . 48

7 Logistic Regression **49**
7.1 Outline . . . 49
7.2 McCulloch-Pitts Model of a Neuron . . . 49
7.3 The Sigmoid squashing function . . . 50
7.4 Separating Hyper-Plane . . . 51

7.5 Negative Log-Likelihood . . . 52
7.6 Gradient and Hessian . . . 53
7.7 Steepest Descent . . . 53
7.8 SoftMax formulation . . . 53
7.9 Loss function for the softmax formulation . . . 54
7.10 SoftMax Notebook . . . 56
7.11 Next Chapter . . . 57

8 Backpropagation **58**
8.1 Outline . . . 58
8.2 A layered network . . . 58
8.3 Layer Specification . . . 59
8.4 Reverse Differentiation and why it works . . . 61
8.5 Deep Learning layered representation . . . 61
8.6 Example Layer Implementation . . . 64
8.7 Layered Network Notebook . . . 66
8.8 Next Chapter . . . 66

9 Convolutional Neural Networks **67**
9.1 Outline . . . 67
9.2 A Convolutional Neural Network . . . 67
9.3 Correlation and Convolution . . . 68
9.4 Two dimensions . . . 69
9.5 Three dimensions . . . 70
9.6 Backpropagation . . . 72
9.7 Implementation via matrix multiplication . . . 72
9.8 Non-Linearity . . . 73
9.9 maxPooling Layer . . . 73
9.10 Fully-Connected layer . . . 74
9.11 Convolutional Neural Network Notebook . . . 75
9.12 Next Chapter . . . 75

10 Recurrent Neural Networks **76**
10.1 Credits . . . 76
10.2 Recurrent Networks . . . 76
10.3 Unrolling the recurrence . . . 77
10.4 Vanishing Gradient problem . . . 78
10.5 Long Short Term Memory . . . 79
10.6 Variants on LSTMs . . . 81
10.7 The Language Model . . . 81

10.8 LSTM Notebook 82
10.9 Next Chapter 83

11 Auto-Encoders **84**
11.1 Outline 84
11.2 The Simple Autoencoder 84
11.3 The Variational Auto-Encoder 86
11.4 A Semi-Supervised Variational Auto-Encoder 88
11.5 Autoencoder Notebooks 90

Bibliography **91**

A Julia with Flux **93**

Notation

This section provides a concise reference describing notation used throughout this document. If you are unfamiliar with any of the corresponding mathematical concepts, [6] describe most of these ideas in chapters 2–4.

Numbers and Arrays

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
A	A tensor
$\boldsymbol{I}_n$	Identity matrix with n rows and n columns
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\boldsymbol{e}^{(i)}$	Standard basis vector $[0, \dots, 0, 1, 0, \dots, 0]$ with a 1 at position i
$\text{diag}(\boldsymbol{a})$	A square, diagonal matrix with diagonal entries given by $\boldsymbol{a}$
a	A scalar random variable
$\mathbf{a}$	A vector-valued random variable
$\mathbf{A}$	A matrix-valued random variable

Sets and Graphs

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, \dots, n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$(a, b]$	The real interval excluding a but including b
$\mathbb{A} \backslash \mathbb{B}$	Set subtraction, i.e., the set containing the elements of $\mathbb{A}$ that are not in $\mathbb{B}$
$\mathcal{G}$	A graph
$Pa_{\mathcal{G}}(\mathrm{x}_i)$	The parents of x_i in $\mathcal{G}$

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
a_{-i}	All elements of vector $\boldsymbol{a}$ except for element i
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	Column i of matrix $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Element (i, j, k) of a 3-D tensor A
$\mathsf{A}_{:,:,i}$	2-D slice of a 3-D tensor
a_i	Element i of the random vector $\mathbf{a}$

Linear Algebra Operations

$\boldsymbol{A}^\top$	Transpose of matrix $\boldsymbol{A}$
$\boldsymbol{A}^+$	Moore-Penrose pseudoinverse of $\boldsymbol{A}$
$\boldsymbol{A} \odot \boldsymbol{B}$	Element-wise (Hadamard) product of $\boldsymbol{A}$ and $\boldsymbol{B}$
$\det(\boldsymbol{A})$	Determinant of $\boldsymbol{A}$

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	Matrix derivatives of y with respect to $\boldsymbol{X}$
$\nabla_{\mathbf{X}} y$	Tensor containing derivatives of y with respect to $\mathbf{X}$
$\frac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \to \mathbb{R}^m$
$\nabla_{\boldsymbol{x}}^2 f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$	The Hessian matrix of f at input point $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral over the entire domain of $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral with respect to $\boldsymbol{x}$ over the set $\mathbb{S}$

Probability and Information Theory

$\mathrm{a} \perp \mathrm{b}$	The random variables a and b are independent
$\mathrm{a} \perp \mathrm{b} \mid \mathrm{c}$	They are conditionally independent given c
$P(\mathrm{a})$	A probability distribution over a discrete variable
$p(\mathrm{a})$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$\mathrm{a} \sim P$	Random variable a has distribution P
$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$ or $\mathbb{E} f(x)$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$H(\mathrm{x})$	Shannon entropy of the random variable x
$D_{\mathrm{KL}}(P \| Q)$	Kullback-Leibler divergence of P and Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$. (Sometimes we write $f(\boldsymbol{x})$ and omit the argument $\boldsymbol{\theta}$ to lighten notation)
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $\dfrac{1}{1 + \exp(-x)}$
$\zeta(x)$	Softplus, $\log(1 + \exp(x))$
$\pi(x, y)$	Softmax, $\dfrac{\exp(x)}{\exp(x) + \exp(y)}$
$\Pi_x(y)$	Indicator, $y == x$
$\|\|\boldsymbol{x}\|\|_p$	L^p norm of $\boldsymbol{x}$
$\|\|\boldsymbol{x}\|\|$	L^2 norm of $\boldsymbol{x}$
x^+	Positive part of x, i.e., $\max(0, x)$

Sometimes we use a function f whose argument is a scalar but apply it to a vector, matrix, or tensor: $f(\boldsymbol{x})$, $f(\boldsymbol{X})$, or $f(\mathsf{X})$. This denotes the application of f to the array element-wise. For example, if $\mathsf{C} = \sigma(\mathsf{X})$, then $\mathsf{C}_{i,j,k} = \sigma(\mathsf{X}_{i,j,k})$ for all valid values of i, j and k.

Datasets and Distributions

p_{data}	The data generating distribution
$\hat{p}_{\text{data}}$	The empirical distribution defined by the training set
$\mathbb{X}$	A set of training examples
$\boldsymbol{x}^{(i)}$	The i-th example (input) from a dataset
$y^{(i)}$ or $\boldsymbol{y}^{(i)}$	The target associated with $\boldsymbol{x}^{(i)}$ for supervised learning
$\boldsymbol{X}$	The $m \times n$ matrix with input example $\boldsymbol{x}^{(i)}$ in row $\boldsymbol{X}_{i,:}$
$\mathscr{L}(\boldsymbol{\theta})$	A *Loss* function to be optimised with respect to $\boldsymbol{\theta}$.

Chapter 1

Introduction

1.1 Acknowledgements

This book was written to accompany the **Deep Learning** video lectures placed in the public domain by Nando de Freitas [3].

The book is developed using the same style as used in the Deep Learning Book, [6]. This style has been released for anyone to use freely, in order to help establish some standard notation in the deep learning community.

This version of the notes makes use of Julia as the programming environment and some sample code from the Flux package appears is referred to in the appendix of this book as does sample code from the Julia Academy website, [5].

1.2 Guinea Pigs

The author thanks the following students for participating in an experimental course on the topic of deep learning which took place at the University of KwaZulu-Natal during the second semester of 2018.

- Devin Pelser
- Gabriel de Charmoy
- Mpilo Mshengu

1.3 Machine Learning

According to Wikipedia, **Machine learning** is the scientific study of algorithms and statistical models that computer systems use to effectively perform a specific task without using explicit instructions, relying on patterns and inference instead.

Machine learning algorithms build a mathematical model of sample data, known as **training data**, in order to make predictions or decisions without being explicitly programmed to perform the task.

Machine learning deals with the problem of extracting features from data so as to solve many different predictive tasks. In his introductory lecture, [3] lists the following areas for modern machine learning.

- Forecasting (e.g. Energy demand prediction, finance)
- Imputing missing data (e.g. Netflix recommendations)
- Detecting anomalies (e.g. Security, fraud, virus mutations)
- Classifying (e.g. Credit risk assessment, cancer diagnosis)
- Ranking (e.g. Google search)
- Summarising (e.g. News zeitgeist, social media sentiment)
- Decision making (e.g. AI, robotics, compiler tuning, trading)

In this book we study **deep neural network** which is but one of many possible machine learning tools. In the next chapter we outline the linear algebra tools that are required to understand and implement deep neural networks.

1.4 Introductory Notebook

In this version of the book we implement examples using Julia with Flux [5]. In appendix A the reader will find a link to a Jupyter notebook giving a short introduction to the language features of the Julia language.

1.5 Next Chapter

In the next chapter we outline the linear algebra background required in order to grasp essential deep learning concepts.

Chapter 2

Linear Algebra

2.1 Introductory

The reader should be familiar with the following from a first year math course:

- Matrices $\boldsymbol{A} = [a_{ij}]$
- Matrix addition and subtraction $\boldsymbol{A} + \boldsymbol{B} = [a_{ij} + b_{ij}]$
- Scalar multiplication $a\boldsymbol{B} = [a\, b_{ij}]$
- Matrix multiplication $\mathbf{AB} = \left[\sum_k a_{ik} b_{kj}\right]$
- Inverse $\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}$
- Transpose $\boldsymbol{A}^\top = [a_{ji}]$
- Symmetric matrices $\boldsymbol{A} = \boldsymbol{A}^\top$
- Trace of a square matrix
  - $tr(\boldsymbol{A}) = \sum_i a_{ii}$ the sum of diagonal elements
  - Obvious: $tr(\boldsymbol{A} + \boldsymbol{B}) = tr(\boldsymbol{A}) + tr(\boldsymbol{B})$
  - Not obvious: $tr(\boldsymbol{A}\boldsymbol{B}) = tr(\boldsymbol{B}\boldsymbol{A})$
- Determinants
- Partitioned matrices

2.2 Advanced

- Linear Independence

  Let $\boldsymbol{X}$ be an $n \times p$ matrix of constants. The columns of $\boldsymbol{X}$ are said to be *linearly dependent* if there exists $\boldsymbol{v} \neq \boldsymbol{0}$ with $\boldsymbol{Xv} = \boldsymbol{0}$. We will say that the columns of $\boldsymbol{X}$ are linearly *independent* if $\boldsymbol{Xv} = \boldsymbol{0}$ implies $\boldsymbol{v} = \boldsymbol{0}$.

  For example, to show that $\boldsymbol{A}^{-1}$ exists implies that the columns of $\boldsymbol{A}$ are linearly independent.

$$\boldsymbol{Av} = \boldsymbol{0} \Rightarrow \boldsymbol{A}^{-1}\boldsymbol{Av} = \boldsymbol{A}^{-1}\boldsymbol{0} \Rightarrow \boldsymbol{v} = \boldsymbol{0}$$

- Rank
  - Row rank is the number of linearly independent rows
  - Column rank is the number of linearly independent columns
  - Rank of a matrix is the minimum of row rank and column rank
  - $rank(\boldsymbol{AB}) = \min(rank(\boldsymbol{A}), rank(\boldsymbol{B}))$

- Eigenvalues and eigenvectors

  Let $\boldsymbol{A} = [a_{i,j}]$ be an $n \times n$ matrix, then $\boldsymbol{A}$ is said to have an *eigenvalue* λ and (non-zero) *eigenvector* $\boldsymbol{x}$ corresponding to λ if

$$\boldsymbol{Ax} = \lambda\boldsymbol{x}.$$

  - Eigenvalues are the λ values that solve the determinantal equation $|\boldsymbol{A} - \lambda\boldsymbol{I}| = 0$.
  - The determinant is the product of the eigenvalues: $|\boldsymbol{A}| = \prod_{i=1}^{n} \lambda_i$

- Spectral decomposition of symmetric matrices

  every square and symmetric matrix $\boldsymbol{A} = [a_{i,j}]$ may be written

$$\boldsymbol{A} = \boldsymbol{CDC}^{\top},$$

  where the columns of $\boldsymbol{C}$ (which may also be denoted $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$) are the eigenvectors of $\boldsymbol{A}$, and the diagonal matrix $\boldsymbol{D}$ contains the corresponding eigenvalues.

$$\mathbf{D} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

The eigenvectors may be chosen to be orthonormal, so that $\boldsymbol{C}$ is an orthogonal matrix. That is, $\boldsymbol{C}\boldsymbol{C}^\top = \boldsymbol{C}^\top\boldsymbol{C} = \boldsymbol{I}$.

- Positive definite matrices

  The $n \times n$ matrix $\boldsymbol{A}$ is said to be *positive definite* if

  $$\boldsymbol{y}^\top \boldsymbol{A}\boldsymbol{y} > 0$$

  for *all* $n \times 1$ vectors $\boldsymbol{y} \neq \boldsymbol{0}$. It is called *non-negative definite* (or sometimes positive semi-definite) if $\boldsymbol{y}^\top \boldsymbol{A}\boldsymbol{y} \geq 0$.

  Example: Show $\boldsymbol{X}^\top\boldsymbol{X}$ non-negative definite:

  Let $\boldsymbol{X}$ be an $n \times p$ matrix of real constants and $\boldsymbol{y}$ be $p \times 1$. Then $\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{y}$ is $n \times 1$, and

  $$\begin{aligned} & \boldsymbol{y}^\top (\boldsymbol{X}^\top\boldsymbol{X})\, \boldsymbol{y} \\ = \; & (\boldsymbol{X}\boldsymbol{y})^\top (\boldsymbol{X}\boldsymbol{y}) \\ = \; & \boldsymbol{Z}^\top\boldsymbol{Z} \\ = \; & \sum_{i=1}^{n} Z_i^2 \geq 0 \end{aligned}$$

  For a symmetric matrix,

  Positive definite

  $\Downarrow$

  All eigenvalues positive

  $\Downarrow$

  Inverse exists $\Leftrightarrow$ Columns (rows) linearly independent

  If a real symmetric matrix is also non-negative definite, as a variance-covariance matrix *must* be, Inverse exists $\Rightarrow$ Positive definite

  For example, let $\boldsymbol{A}$ be square and symmetric as well as positive definite.

  - Spectral decomposition says $\boldsymbol{A} = \boldsymbol{C}\boldsymbol{D}\boldsymbol{C}^\top$.
  - Using $\boldsymbol{y}^\top\boldsymbol{A}\boldsymbol{y} > 0$, let $\boldsymbol{y}$ be an eigenvector, say the third one.
  - Because eigenvectors are orthonormal,

$$\begin{aligned}
\boldsymbol{y}^\top \boldsymbol{A}\boldsymbol{y} &= \boldsymbol{y}^\top \boldsymbol{C}\boldsymbol{D}\boldsymbol{C}^\top \boldsymbol{y} \\
&= \begin{pmatrix} 0 & 0 & 1 & \cdots & 0 \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} \\
&= \lambda_3 \\
&> 0
\end{aligned}$$

- Inverse of a diagonal matrix

  Suppose $\boldsymbol{D} = [d_{i,j}]$ is a diagonal matrix with non-zero diagonal elements. It is easy to verify that $\boldsymbol{D}^{-1} = [1/d_{i,j}]$

- Eigenvalues positive $\Rightarrow$ Inverse exists

  Let $\boldsymbol{A}$ be symmetric and positive definite. Then $\boldsymbol{A} = \boldsymbol{C}\boldsymbol{D}\boldsymbol{C}^\top$ and its eigenvalues are positive. Let $\boldsymbol{B} = \boldsymbol{C}\boldsymbol{D}^{-1}\boldsymbol{C}^\top$ then $\boldsymbol{B} = \boldsymbol{A}^{-1}$ and

  $$\begin{aligned}
  \boldsymbol{A}\boldsymbol{B} &= \boldsymbol{C}\boldsymbol{D}\boldsymbol{C}^\top \boldsymbol{C}\boldsymbol{D}^{-1}\boldsymbol{C}^\top = \boldsymbol{I} \\
  \boldsymbol{B}\boldsymbol{A} &= \boldsymbol{C}\boldsymbol{D}^{-1}\boldsymbol{C}^\top \boldsymbol{C}\boldsymbol{D}\boldsymbol{C}^\top = \boldsymbol{I}
  \end{aligned}$$

  So

  $$\boldsymbol{A}^{-1} = \boldsymbol{C}\boldsymbol{D}^{-1}\boldsymbol{C}^\top$$

- Square Root matrices

  For symmetric, non-negative definite, diagonal matrices, define

  $$\boldsymbol{D}^{1/2} = \begin{pmatrix} \sqrt{\lambda_1} & 0 & \cdots & 0 \\ 0 & \sqrt{\lambda_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sqrt{\lambda_n} \end{pmatrix}$$

  For a non-negative definite, symmetric matrix $\boldsymbol{A}$, define

  $$\boldsymbol{A}^{1/2} = \boldsymbol{C}\boldsymbol{D}^{1/2}\boldsymbol{C}^\top$$

  example: The square root of the inverse is the inverse of the square root

  Let $\mathbf{A}$ be symmetric and positive definite, with $\mathbf{A} = \mathbf{C}\mathbf{D}\mathbf{C}^\top$.

  Let $\mathbf{B} = \mathbf{C}\mathbf{D}^{-1/2}\mathbf{C}^\top$. What is $\mathbf{D}^{-1/2}$?

Show $\mathbf{B} = \left(\mathbf{A}^{-1}\right)^{1/2}$

$$\begin{aligned} \mathbf{BB} &= \mathbf{C}\mathbf{D}^{-1/2}\mathbf{C}^\top\mathbf{C}\mathbf{D}^{-1/2}\mathbf{C}^\top \\ &= \mathbf{C}\mathbf{D}^{-1}\mathbf{C}^\top = \mathbf{A}^{-1} \end{aligned}$$

Show $\mathbf{B} = \left(\mathbf{A}^{1/2}\right)^{-1}$

$$\begin{aligned} \mathbf{A}^{1/2}\mathbf{B} &= \mathbf{C}\mathbf{D}^{1/2}\mathbf{C}^\top\mathbf{C}\mathbf{D}^{-1/2}\mathbf{C}^\top = \mathbf{I} \\ \mathbf{B}\mathbf{A}^{1/2} &= \mathbf{C}\mathbf{D}^{-1/2}\mathbf{C}^\top\mathbf{C}\mathbf{D}^{1/2}\mathbf{C}^\top = \mathbf{I} \end{aligned}$$

Just write $\mathbf{A}^{-1/2} = \mathbf{C}\mathbf{D}^{-1/2}\mathbf{C}^\top$

2.3 Linear Algebra Notebook

Most high level programming Languages are equipped with Linear Algebra packages that provide implementations for the linear algebra computations outlined in this chapter.

In appendix A the reader will find a link to a Jupyter notebook that demonstrates how to perform the common linear algebra computations described in this chapter using the Julia language.

2.4 Next Chapter

In the next chapter of the book we study *linear regression* and show how to solve the linear regression problem, either by using an exact solution provided by linear algebra or by using a one neuron neural-network.

Chapter 3

Linear Regression

3.1 Outline

This lecture introduces us to the topic of supervised learning. Here the data consists of input-output pairs. Inputs are also often referred to as features; while outputs are known as targets or labels. The goal of the lecture is for you to:

- Understand the supervised learning setting.
- Understand linear regression, ie: least squares.
- Understand how to apply linear regression models to make predictions.
- Learn to derive the least squares estimate.

Why start with Linear supervised learning?

- Many real processes can be approximated with linear models.
- Linear regression often appears as a module of larger systems.
- Linear problems can be solved analytically.
- Linear prediction provides an introduction to many of the core concepts of machine learning.

3.2 Learning Data

We are given a training dataset of n instances of **input-output** pairs:

$$\{\boldsymbol{x}_{1:n}, y_{1:n}\}$$

Each input $\boldsymbol{x}_i$ is a row vector with d attributes, ie $\boldsymbol{x}_i \in \mathbb{R}^{1\times d}$. The output, often referred to as the target, will be assumed to be univariate for now, $y_i \in \mathbb{R}$

A typical data set with 4 instances and two attributes for the input and 4 instances for the output could look like the data set in table 3.1

	Wind	People		Energy
$\boldsymbol{x}_1$	100	2	y_1	5
$\boldsymbol{x}_2$	50	42	y_2	25
$\boldsymbol{x}_3$	45	31	y_3	22
$\boldsymbol{x}_4$	60	35	y_4	18

Table 3.1: A typical data set with 4 instances and two attributes.

3.3 The learning/prediction cycle

Given the training set $\{\boldsymbol{x}_{1:n}, y_{1:n}\}$, we would like to **learn a model** of how the inputs affect the outputs. Given this model and a new input, $\boldsymbol{x}_{n+1}$, we can use the model to make a prediction for the output, $\hat{y}(\boldsymbol{x}_{n+1})$.

Thus learning and prediction is a two stage process:

1) TRAINING (learning)

$$\begin{array}{ccccc} \text{Data} & & & & \text{Model} \\ \{\boldsymbol{x}_{1:n}, y_{1:n}\} & \rightarrow & \text{learner} & \rightarrow & \theta \end{array}$$

2) TESTING (prediction)

$$\begin{array}{ccccc} \text{Model} & & & & \text{Prediction} \\ \{\boldsymbol{x}_{n+1}, \theta\} & \rightarrow & \text{predictor} & \rightarrow & \hat{y}(\boldsymbol{x}_{n+1}) \end{array}$$

3.4 Minimum Loss via Hook's law

For linear regression, our task is to learn the model parameters, θ_i, so that $\hat{y}_i = \sum_j \theta_j x_{ij}$ is a good estimate for y_i. Gauss tells us to do this by minimising the **loss** function:

$$\mathscr{L}(\theta) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{d}\theta_j x_{ij})^2$$

For neural networks the convention is that the first column of X is all ones which we can achieve by either adding a column of ones or by dividing through each of the training data, $\{\boldsymbol{x}_i, y_i\}$ by x_{i1} in which case the loss function now reads:

$$\mathscr{L}(\theta) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n}(y_i - \theta_1 - \sum_{j=2}^{d}\theta_j x_{ij})^2$$

Here, θ_1, is called the **bias**.

In our example from table 3.1, we divide each row, i, by x_{i1} then we learn θ by minimising the energy in the springs shown in figure 3.1.

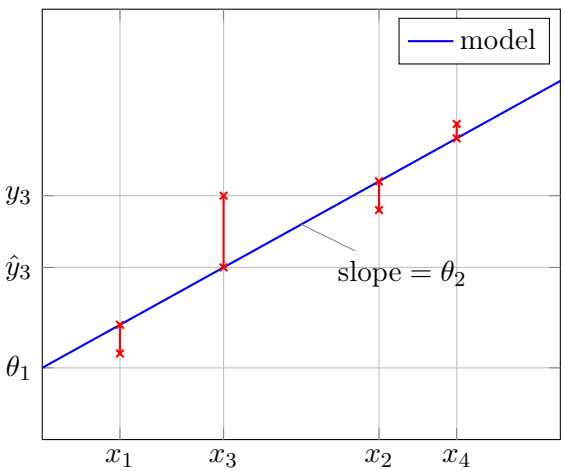


Figure 3.1: The optimal model parameters, θ are found by minimising the energy in the springs. We obtain θ by minimising $\mathscr{L}(\theta) = \sum_{i=1}^{4}(y_i - \theta_1 - \theta_2 x_{i2})^2$ and then predict the output using $\hat{y}_i = \theta_1 + x_{i2}\theta_2$. In this case θ_1 is the intercept on the y axis or the **bias** whilst θ_2 is the slope of the optimal fit.

3.5 General Linear Regression

In general the linear model is expressed as

$$\hat{y}_i = \sum_{j=1}^{d} x_{ij}\theta_j = \theta_1 + x_{i2}\theta_2 + \ldots x_{id}\theta_d$$

where we have assumed that $x_{i1} = 1$ so that θ_1 corresponds to the bias. In matrix form the linear model is

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\theta}$$

So, in our example from table 3.1, we could add a column of ones to $\boldsymbol{X}$ and then with $n = 4$ and $d = 3$ we now have the setup:

$$\boldsymbol{y} = \begin{bmatrix} 5 \\ 25 \\ 22 \\ 18 \end{bmatrix}, \boldsymbol{X} = \begin{bmatrix} 1 & 100 & 2 \\ 1 & 50 & 42 \\ 1 & 45 & 31 \\ 1 & 60 & 35 \end{bmatrix}, \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

For example, if we estimated $\boldsymbol{\theta}$ as $[1, 0, 0.5]^T$ then by multiplying $\boldsymbol{X}$ by $\boldsymbol{\theta}$ we would get the following predictions on the training set:

$$\hat{\boldsymbol{y}} = \begin{bmatrix} 1 & 100 & 2 \\ 1 & 50 & 42 \\ 1 & 45 & 31 \\ 1 & 60 & 35 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 2 \\ 22 \\ 16.5 \\ 18.5 \end{bmatrix}$$

Likewise, for a point we have never seen before, say $x = [50, 20]$, we can predict by augmenting with a one and computing the product $\hat{y} = \boldsymbol{x}\boldsymbol{\theta}$ as follows:

$$\hat{y} = \begin{bmatrix} 1 & 50 & 20 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0.5 \end{bmatrix} = 11$$

Now we have laid out the notation it remains to determine how to estimate $\boldsymbol{\theta}$ efficiently.

3.6 Optimization

Our aim is to minimise the quadratic loss between the output labels and the model predictions which we write in matrix form as:

$$\mathscr{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{d} x_{ij}\theta_j)^2 = (\boldsymbol{y} - \boldsymbol{X\theta})^T(\boldsymbol{y} - \boldsymbol{X\theta}) \tag{3.1}$$

Since the loss is quadratic in $\boldsymbol{\theta}$ we are assured of a unique minimum which we must find using steepest descent. When $d = 2$ the loss function is a **bowl** as shown in figure 3.2.

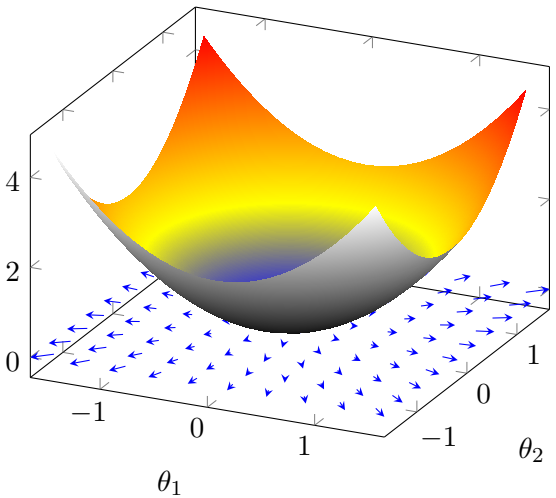


Figure 3.2: When $d = 2$ the loss function for $\boldsymbol{\theta} = [\theta_1, \theta_2]^T$ is a bowl shaped and the gradients are in the direction of steepest ascent.

We can find the minimum using steepest descent. To do this we must differentiate the loss function, $\mathscr{L}(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$, to find the gradient and then update our estimate for $\boldsymbol{\theta}$ by subtracting a portion of the gradient and moving downhill.

To find the gradient we must differentiate a function with respect to a vector. This should have been studied in 2nd year linear algebra but you can read [2] to refresh your matrix calculus. You will need the following results from Matrix differentiation.

Assuming that a matrix $\boldsymbol{A}$ is not a function of a vector $\boldsymbol{\theta}$ then we can differentiate with respect to $\boldsymbol{\theta}$ as follows

$$\frac{\partial}{\partial \boldsymbol{\theta}}(\boldsymbol{A\theta}) = \boldsymbol{A}^T \tag{3.2}$$

and if $\boldsymbol{A}$ is symmetric as well then we can establish that

$$\frac{\partial}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}^T \boldsymbol{A\theta}) = 2\boldsymbol{A}^T\boldsymbol{\theta} \tag{3.3}$$

Exercise Try to construct proofs for the above two results using [2].

The gradient of the loss function is then obtained as follows

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\theta}}\mathscr{L}(\boldsymbol{\theta}) &= \frac{\partial}{\partial \boldsymbol{\theta}}(\boldsymbol{y} - \boldsymbol{X\theta})^T(\boldsymbol{y} - \boldsymbol{X\theta}) \\ &= \frac{\partial}{\partial \boldsymbol{\theta}}(\boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{y}^T\boldsymbol{X\theta} + \boldsymbol{\theta}^T\boldsymbol{X}^T\boldsymbol{X\theta}) \\ &= -2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X\theta}\end{aligned} \tag{3.4}$$

We can use the gradient to update $\boldsymbol{\theta}$ iteratively until we reach the minimum or we can set $\frac{\partial}{\partial \boldsymbol{\theta}}\mathscr{L}(\boldsymbol{\theta}) = 0$ and solve for θ to get an exact solution as follows:

$$\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{3.5}$$

Using a linear algebra package it is simple enough to obtain the exact solution to a linear regression problem, however the exact solution can be a poisoned chalice as the matrix $\boldsymbol{X}$ is often ill-conditioned and the matrix inversion cannot be performed. So an iterative solution via steepest descent is the way to go.

3.7 Linear Regression Notebook

in appendix A the reader will find a link to a Jupyter notebook that uses the Julia language and the Flux package to demonstrate how to perform linear regression using both the exact linear algebra formulations and the optimisation techniques described in this chapter.

3.8 Next Chapter

In the next chapter, we learn to derive the linear regression estimates by *maximum likelihood* with multivariate Gaussian distributions. This leads on to a discussion on *Entropy*. Please go to the Wikipedia page for the multivariate Normal distribution beforehand or, even better, read Chapter 4 of [10] and attempt the first few exercises from [3].

Chapter 4

Maximum Likelihood

4.1 Outline

This lecture formulates the problem of linear prediction using probabilities. We also introduce the maximum likelihood estimate and show that it coincides with the least squares estimate.

The goal of the lecture is for you to understand:

- Gaussian distributions
- How to formulate the likelihood for linear regression
- Computing the maximum likelihood estimates for linear regression.
- Entropy and its relation to loss, probability and learning.

4.2 Univariate Gaussian Distribution

If x is sampled from a Normal (or Gaussian) distribution $\mathcal{N}$ with mean μ and variance σ^2 then we write

$$x \sim \mathcal{N}(\mu, \sigma^2)$$

The probability density function (PDF) (or histogram) of samples from a Gaussian distribution with mean μ and variance σ^2 is given by:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

which is a so called **bell curve** that looks like:

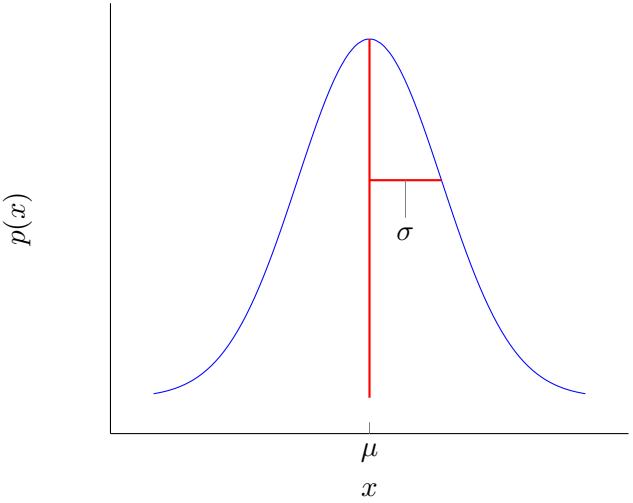


Figure 4.1: The PDF of a Gaussian distribution, $\mathcal{N}(x; \mu, \sigma^2)$

The PDF of a gaussian distribution has the property that:

$$\int_{-\infty}^{\infty} p(x)dx = 1$$

and the area under the PDF curve in the region $a \leq x \leq b$ gives the probability of x being drawn from the interval $[a, b]$

If we **accumulate** the PDF of a gaussian using integration then we obtain the Cumulative Density Function of the distribution, or CDF, as follows:

$$\text{CDF}(x) = \int_{-\infty}^{x} p(t)dt$$

The CDF(x) is the probability that the distribution will produce a draw that is less than or equal to x.

It is evident that CDF($-\infty$) $= 0$ and CDF(∞) $= 1$ and because $p(x)$ is symmetric about μ we also know that CDF(μ) $= \frac{1}{2}$. In fact the CDF of a Gaussian has a **sigmoid** shape as shown in figure 4.2.

The CDF of a distribution gives us a way to **sample** the distribution as follows.

- first generate a random number, r, in the range $[0, 1]$.
- then compute $r' = \text{CDF}^{-1}(r)$.

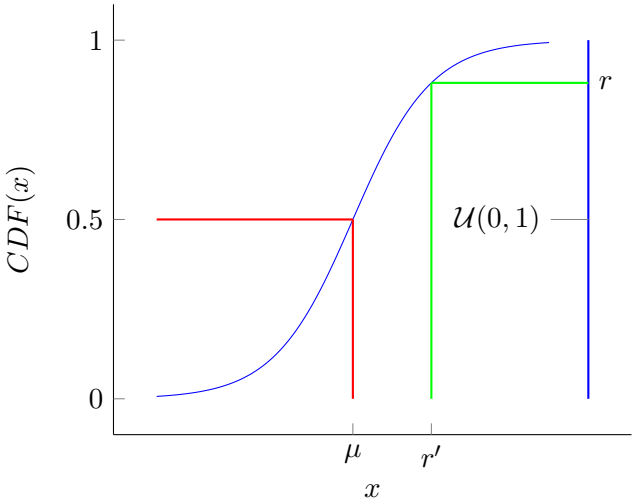


Figure 4.2: The CDF of a distribution gives us a way to sample the distribution using a random number generator $r' = \text{CDF}^{-1}(r)$.

4.3 Moments

To discuss moments of a distribution we need the concept of a **Dirac delta function**, $\delta(x)$, which was introduced by Paul Dirac at the end of the 1920's in an effort to create the mathematical tools for the development of quantum field theory. He referred to it as an **improper function** or **functional**. The usefulness of the δ function is due to a property called the sampling property.

$\delta(x - a)$ can be thought of as a gaussian centred at point a with negligible variance that has the following properties:

$$\text{mass:} \quad \int_{-\infty}^{\infty} \delta(x - a)dx = 1 \tag{4.1}$$

$$\text{sampling: } \int_{-\infty}^{\infty} f(x)\delta(x-a)dx = f(a) \tag{4.2}$$

The sampling property applies whenever $f(x)$ is a function continuous in a neighbourhood of a. Furthermore, If $f(x)$ is any function with continuous derivatives up to the n^{th} order in some neighbourhood of a, then integration by parts gives us:

$$\int_{-\infty}^{\infty} f(x)\delta^{(n)}(x-a)dx = (-1)^n f^{(n)}(a) \tag{4.3}$$

Here, $\delta^{(n)}$ is the generalised n^{th} order derivative of δ. This derivative defines a functional, $\delta^{(n)}(x-a)$, which assigns the value $(-1)^n f^{(n)}(a)$ to $f(x)$. You can imagine these derivatives by considering derivatives of Gaussians as their variance vanishes, see figure 4.3

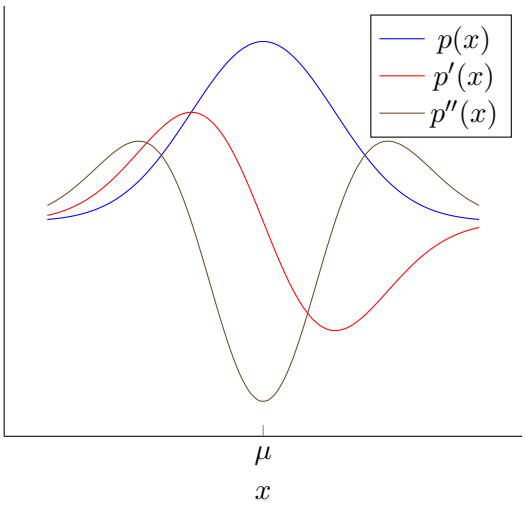


Figure 4.3: Derivatives of Gaussian distributions.

We can make use of δ functions to build an estimator for a PDF as shown in figure 4.4.

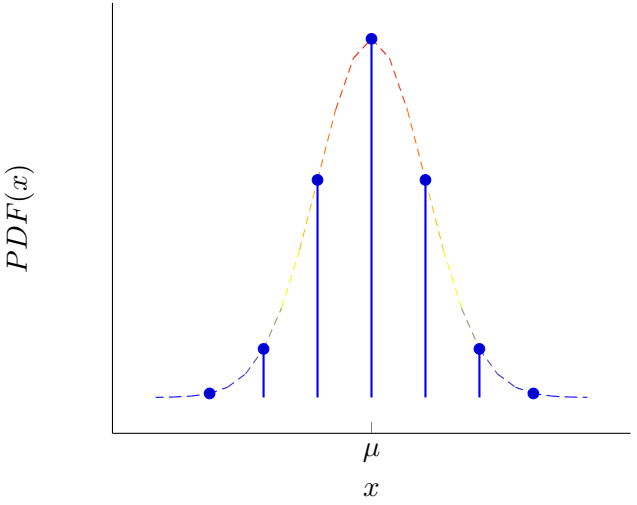


Figure 4.4: Estimating a PDF using a number of δ functions.

If we have N samples, $x^{(i)}$, from our distribution then an estimate for it's PDF is:

$$p(x) \approx \frac{1}{N} \sum_{i=1}^{N} \delta(x - x^{(i)}) \tag{4.4}$$

which we will make use of to derive estimates for the so called **moments** of a PDF.

4.3.1 Expectation (first moment)

The first moment, or **expectation**, of a random variable, x with probability density function, $p(x)$, is defined by:

$$\mathbb{E}(x) = \int_{-\infty}^{\infty} xp(x)dx \tag{4.5}$$

which using expression 4.4 can be estimated from N samples as:

$$\mathbb{E}(x) \approx \int_{-\infty}^{\infty} x \frac{1}{N} \sum_{i=1}^{N} \delta(x - x^{(i)})dx$$

$$= \frac{1}{N}\sum_{i=1}^{N}\int_{-\infty}^{\infty} x\delta(x - x^{(i)})dx$$
$$= \frac{1}{N}\sum_{i=1}^{N} x^{(i)} \tag{4.6}$$

In the case of a gaussian distribution, the first moment evaluates exactly to the parameter, μ. (prove this as an exercise)

4.3.2 Variance (second moment)

The second moment, or **variance**, of a random variable, x with probability density function, $p(x)$ and first moment, μ, is defined by:

$$\mathrm{Var}(x) = \int_{-\infty}^{\infty}(x-\mu)^2 p(x)dx \tag{4.7}$$

which again, using expression 4.4 can be estimated from N samples as:

$$\mathrm{Var}(x) \approx \int_{-\infty}^{\infty}(x-\mu)^2\frac{1}{N}\sum_{i=1}^{N}\delta(x - x^{(i)})dx$$
$$= \frac{1}{N}\sum_{i=1}^{N}\int_{-\infty}^{\infty}(x-\mu)^2\delta(x - x^{(i)})dx$$
$$= \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \mu)^2 \tag{4.8}$$

In the case of a gaussian distribution, the second moment evaluates exactly to the parameter, σ^2. (prove this as an exercise)

4.4 Covariance

The **covariance** between two random variables x and y measures the degree to which x and y are linearly related:

$$\mathrm{Cov[x, y]} = \mathbb{E}[(\mathrm{x} - \mathbb{E}[\mathrm{x}])(\mathrm{y} - \mathbb{E}[\mathrm{y}])] = \mathbb{E}[\mathrm{xy}] - \mathbb{E}[\mathrm{x}]\mathbb{E}[\mathrm{y}] \tag{4.9}$$

If $\mathbf{x}$ is a d dimensional random vector then its covariance matrix is defined to be the following symmetric positive definite matrix:

$$\begin{aligned}\text{Cov}[\mathbf{x}] &= \mathbb{E}[(\mathbf{x} - \mathbb{E}[\mathbf{x}])(\mathbf{x} - \mathbb{E}[\mathbf{x}])^T] \\ &= \begin{bmatrix} \text{Var}[\mathbf{x}_1] & \text{Cov}[\mathbf{x}_1, \mathbf{x}_2] & \text{Cov}[\mathbf{x}_1, \mathbf{x}_3] & \dots & \text{Cov}[\mathbf{x}_1, \mathbf{x}_d] \\ \text{Cov}[\mathbf{x}_2, \mathbf{x}_1] & \text{Var}[\mathbf{x}_2] & \text{Cov}[\mathbf{x}_2, \mathbf{x}_3] & \dots & \text{Cov}[\mathbf{x}_2, \mathbf{x}_d] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \text{Cov}[\mathbf{x}_d, \mathbf{x}_1] & \text{Cov}[\mathbf{x}_d, \mathbf{x}_2] & \text{Cov}[\mathbf{x}_d, \mathbf{x}_3] & \dots & \text{Var}[\mathbf{x}_d] \end{bmatrix}\end{aligned} \tag{4.10}$$

Using matrices we write the above as

$$\boldsymbol{\Sigma} = \text{Cov}[\boldsymbol{X}] \tag{4.11}$$

4.5 Multivariate Gaussian

We are now in a position to write down the probability density function, $p(\boldsymbol{x})$, for a distribution $\mathbf{x} \sim \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ over a d dimensional random vector $\boldsymbol{x}$.

$$p(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^d \, |\boldsymbol{\Sigma}|}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}-\boldsymbol{\mu})} \tag{4.12}$$

When $d = 2$ the gaussian distribution could be as shown in figure 4.5.

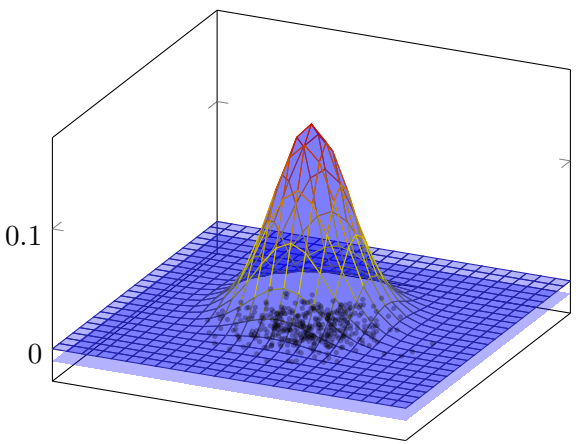


Figure 4.5: A 2 dimensional Gaussian distribution

4.6 Likelihood

Suppose we have drawn n data points from a Gaussian distribution with unknown mean, θ and known variance, 1.

$$y_i \sim \mathcal{N}(\theta, 1) = \theta + \mathcal{N}(0, 1)$$

For example if $n = 3$ we might have drawn, $\{y_1 = 1, y_2 = 0.5, y_3 = 1.5\}$.

The **likelihood** of our data for any θ is then given by the product:

$$P(y_1|\theta)P(y_2|\theta)P(y_3|\theta)$$

Now consider two guesses for the mean, $\theta = 1$ or $\theta = 2.5$. Which has the highest **likelihood**, or probability of generating the three observations?

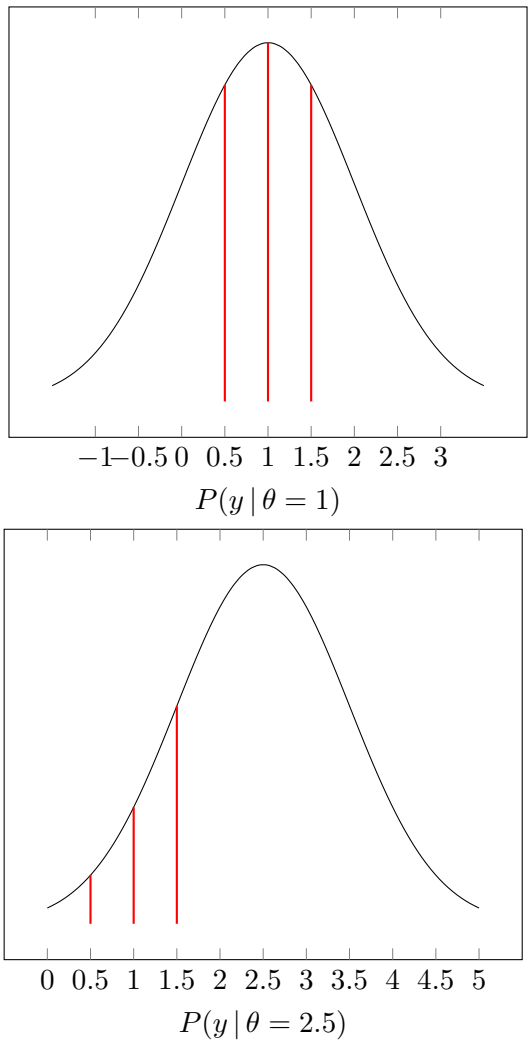


Figure 4.6: The θ that maximises the likelihood is found by shifting the gaussian.

Figure 4.6 demonstrates that finding the θ that maximises the likelihood is equivalent to moving the Gaussian until the product of the 3 likelihoods is maximised.

4.7 Likelihood for linear regression

Let us assume that each label y_i is Gaussian distributed with mean $\boldsymbol{x}_i^T\boldsymbol{\theta}$ and variance σ^2 which we can write as:

$$y_i \sim \mathcal{N}(\boldsymbol{x}_i\boldsymbol{\theta}, \sigma^2) = \boldsymbol{x}_i\boldsymbol{\theta} + \mathcal{N}(0, \sigma^2)$$

For any particular choice of parameters we can compute the likelihood of the data using:

$$\begin{aligned} P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}, \sigma) &= \prod_{i=1}^{n} p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}, \sigma) \\ &= \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{1}{2\sigma^2}(y_i - \boldsymbol{x}_i^T\boldsymbol{\theta})^2} \\ &= (\frac{1}{\sqrt{2\pi\sigma^2}})^n e^{\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \boldsymbol{x}_i^T\boldsymbol{\theta})^2} \\ &= (\frac{1}{\sqrt{2\pi\sigma^2}})^n e^{\frac{1}{2\sigma^2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})} \end{aligned}$$

The **maximum likelihood estimate** (MLE) of $\boldsymbol{\theta}$ is found by taking the derivative of the log-likelihood, $\log p(\boldsymbol{y}, \boldsymbol{X}, \boldsymbol{\theta}, \sigma)$ with respect to $\boldsymbol{\theta}$, setting the result to zero and solving for θ.

The goal is to maximise the likelihood of seeing the training data, $\boldsymbol{y}$, by modifying the model parameters, $\boldsymbol{\theta}$ and σ. However, instead of maximising the log-likelihood we choose to minimise the **negative** of the log-likelihood which is given by:

$$-\log(P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}, \sigma)) = \frac{n}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}) \tag{4.13}$$

Differentiating with respect to $\boldsymbol{\theta}$ we obtain a gradient for a steepest descent algorithm

$$\begin{aligned} \frac{\partial}{\partial\boldsymbol{\theta}}[-\log(P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}, \sigma))] &= \frac{1}{2\sigma^2}\frac{\partial}{\partial\boldsymbol{\theta}}(\boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{y}^T\boldsymbol{X}\boldsymbol{\theta} + \boldsymbol{\theta}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta}) \\ &= \frac{1}{2\sigma^2}(-2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta}) \end{aligned}$$

We could set this gradient to zero and solve for $\boldsymbol{\theta}$ to find that the MLE for $\boldsymbol{\theta}$ is identical to the least square estimate:

$$\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{4.14}$$

The MLE for σ is obtained using:

$$\begin{aligned}\frac{\partial}{\partial\sigma}[-\log(P(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{\theta},\sigma))] &= \frac{\partial}{\partial\sigma}[\frac{n}{2}\log(2\pi\sigma^2)+\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})] \\ &= \frac{n}{\sigma}-\frac{1}{\sigma^3}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})\end{aligned}$$

Setting this expression to zero and solving for σ we find that

$$\sigma^2 = \frac{1}{n}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(y_i-\boldsymbol{x}_i^T\boldsymbol{\theta})^2 \tag{4.15}$$

as expected.

Having estimated the maximum likelihood parameters, $\boldsymbol{\theta}$ and σ^2, we can predict a new output, $\hat{y}$ from a new input, $\boldsymbol{x}_*$ by sampling

$$\hat{y} \sim \mathcal{N}(y|\boldsymbol{x}_*^T\boldsymbol{\theta},\sigma^2)$$

The one dimensional prediction procedure is shown in figure 4.7.

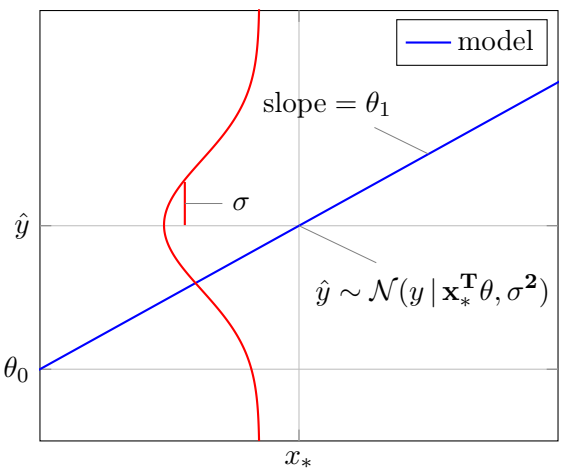


Figure 4.7: Predicting new output from a trained model.

4.8 Entropy

In information theory, **entropy** H is a measure of the uncertainty associated with a random variable, x. It is defined in terms of the probability distribution function of the random variable as:

$$H(\mathrm{x}) = -\sum_{x} p(x|\theta) \log p(x|\theta) \tag{4.16}$$

Note that the log is usually taken in base 2 for an entropy measured in bits.

4.9 Entropy of a Bernoulli random variable

As an example, consider a **Bernoulli** discrete random variable, x, which takes on discrete values from the set $\{0, 1\}$ with a probability distribution function dependent on a parameter, θ given by:

$$p(x|\theta) = \theta^x (1-\theta)^{(1-x)} = \begin{cases} \theta & \text{if } x = 1 \\ 1-\theta & \text{if } x = 0 \end{cases} \tag{4.17}$$

In this case the entropy is given by:

$$\begin{aligned} H(\theta) &= -\sum_{x \in \{0,1\}} \theta^x (1-\theta)^{(1-x)} \log(\theta^x (1-\theta)^{(1-x)}) \\ &= -[(1-\theta)\log(1-\theta) + \theta \log \theta] \end{aligned} \tag{4.18}$$

and the dependence of entropy (or uncertainty) on the Bernoulli parameter θ is shown in figure 4.8.

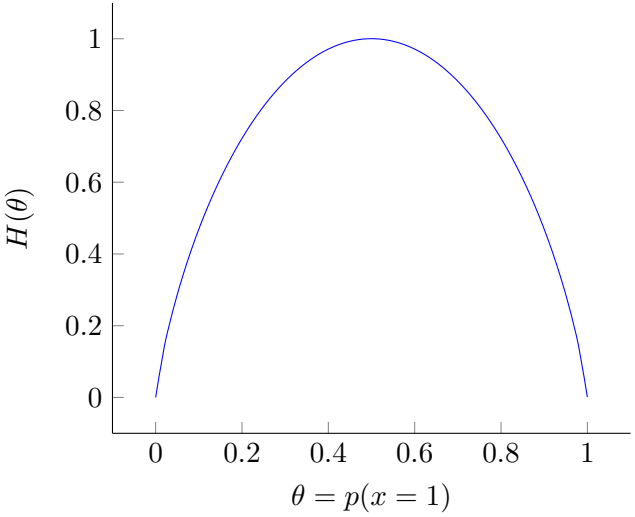


Figure 4.8: Uncertainty in a biased dice with bias, θ.

As an exercise, show that the entropy of a Gaussian in d dimensions is given by:

$$H(\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})) = \frac{1}{2} \log[(2\pi e)^d |\boldsymbol{\Sigma}|]$$

4.10 Kullback Leibler divergence

Theorem

For independent identically distributed data from $p(\boldsymbol{x}, \boldsymbol{\theta}_0)$ the MLE minimises the **Kullback Leibler divergence** between the real world with parameters $\boldsymbol{\theta}_0$ and the model with parameters $\boldsymbol{\theta}$.

Proof

$$\begin{aligned}\hat{\boldsymbol{\theta}} &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} p(\boldsymbol{x}_i|\boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p(\boldsymbol{x}_i|\boldsymbol{\theta})\end{aligned}$$

$$
\begin{aligned}
&= \arg\max_{\boldsymbol{\theta}} \left[\frac{1}{N} \sum_{i=1}^{N} \log p(\boldsymbol{x}_i|\boldsymbol{\theta}) - \frac{1}{N} \sum_{i=1}^{N} \log p(\boldsymbol{x}_i|\boldsymbol{\theta}_0) \right] \\
&= \arg\max_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \log \frac{p(\boldsymbol{x}_i|\boldsymbol{\theta})}{p(\boldsymbol{x}_i|\boldsymbol{\theta}_0)} \\
&= \arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \log \frac{p(\boldsymbol{x}_i|\boldsymbol{\theta}_0)}{p(\boldsymbol{x}_i|\boldsymbol{\theta})} \\
&\to \arg\min_{\boldsymbol{\theta}} \int p(\boldsymbol{x}|\boldsymbol{\theta}_0) \log \frac{p(\boldsymbol{x}|\boldsymbol{\theta}_0)}{p(\boldsymbol{x}|\boldsymbol{\theta})} d\boldsymbol{x} \text{ as } N \to \infty \\
&= \arg\min_{\boldsymbol{\theta}} \left[\int p(\boldsymbol{x}|\boldsymbol{\theta}_0) \log p(\boldsymbol{x}|\boldsymbol{\theta}_0) d\boldsymbol{x} - \int p(\boldsymbol{x}|\boldsymbol{\theta}_0) \log p(\boldsymbol{x}|\boldsymbol{\theta}) d\boldsymbol{x} \right] \\
&= \arg\min_{\boldsymbol{\theta}} \left[\begin{pmatrix} \text{information} \\ \text{from the world} \end{pmatrix} - \begin{pmatrix} \text{information} \\ \text{from the model} \end{pmatrix} \right]
\end{aligned}
$$

4.11 Entropy Notebook

in appendix A the reader will find a link to a Jupyter notebook that uses the Julia language to explore some of the Entropy concepts discussed in this chapter.

4.12 Next Chapter

In the next chapter, we introduce ridge regression, bases functions and look at the issue of controlling complexity.

Chapter 5

Ridge Regression

5.1 Outline

This lecture will show how to fit nonlinear functions by using bases functions and how to control model complexity. The goal is:

- Learn how to derive ridge regression.
- Understand the trade-off of fitting the data and regularising it.
- Learn polynomial regression.
- Understand that, if basis functions are given, the problem of learning the parameters is still linear.
- Learn cross-validation.
- Understand model complexity and generalisation.

5.2 Regularisation

So far all our estimates for the parameters, $\boldsymbol{\theta}$, are of the form:

$$\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

This requires the inversion of $\boldsymbol{X}^T\boldsymbol{X}$ which can lead to numerical problems if the matrix is poorly conditioned. The solution is to add a **small** element to the diagonal:

$$\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X} + \delta^2\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

This is called the **ridge regression** estimate. It is the $\boldsymbol{\theta}$ that minimises the **regularised** quadratic cost function:

$$J(\boldsymbol{\theta}) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}) + \delta^2\boldsymbol{\theta}^T\boldsymbol{\theta} \tag{5.1}$$

since if we differentiate with respect to $\boldsymbol{\theta}$ we obtain:

$$\begin{aligned}\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} &= \frac{\partial}{\partial \boldsymbol{\theta}}\left(\boldsymbol{\theta}^T\boldsymbol{X}^t\boldsymbol{X}\boldsymbol{\theta} - 2\boldsymbol{y}^T\boldsymbol{X}\boldsymbol{\theta} + \boldsymbol{y}^T\boldsymbol{y} + \delta^2\boldsymbol{\theta}^T\boldsymbol{\theta}\right)\\ &= 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} - 2\boldsymbol{X}^T\boldsymbol{y} + 2\delta^2\boldsymbol{I}\boldsymbol{\theta}\\ &= 2(\boldsymbol{X}^T\boldsymbol{X} + \delta^2\boldsymbol{I})\boldsymbol{\theta} - 2\boldsymbol{X}^T\boldsymbol{y}\end{aligned}$$

and on equating to zero we find:

$$\boldsymbol{\theta}_{\text{ridge}} = (\boldsymbol{X}^T\boldsymbol{X} + \delta^2\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{5.2}$$

5.2.1 Geometric Interpretation

In figure 5.1 we give a geometric interpretation for the minimisation of the regularised quadratic cost function of equation 5.1.

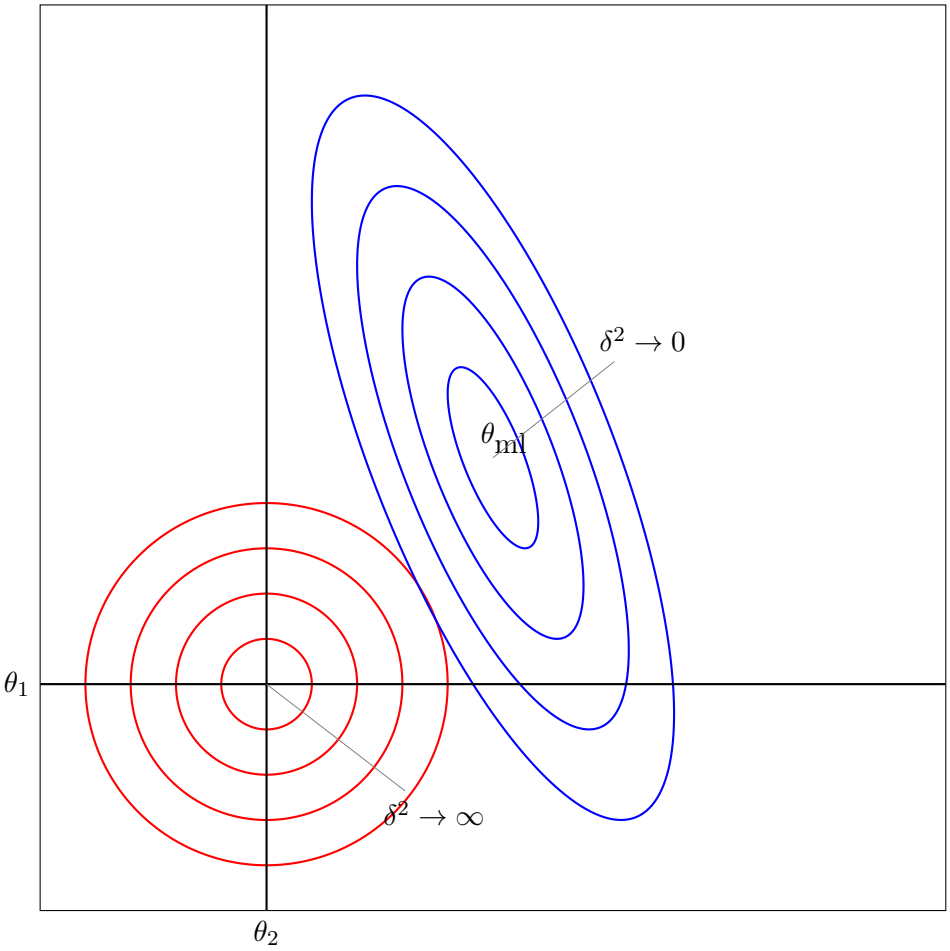


Figure 5.1: A geometric interpretation for a regulariser using the L_2 norm.

The level curves of the quadratic cost are depicted in red whilst those of the regulariser are shown in blue. The minimum quadratic cost subject to the restriction $\boldsymbol{\theta}^T\boldsymbol{\theta} = \delta^2$ is found at the point where a quadratic cost level curve is tangential to the regulariser's level curve.

At $\delta = 0$ there is no regularisation and the solution reverts to the least squares optimum. As $\delta \to \infty$ the regularisation becomes overbearing and $\boldsymbol{\theta} \to 0$. The line drawn in purple shows the path taken by the regulariser for differing values of δ

In effect we have recast the ridge regression problem as a constrained optimisation problem employing the L_2 norm for $\boldsymbol{\theta}^T\boldsymbol{\theta}$ vis $\|\boldsymbol{\theta}\|_2^2 = \sum \theta_i^2$.

$$\underset{\boldsymbol{\theta}:\boldsymbol{\theta}^T\boldsymbol{\theta}\leq\delta^2}{\arg\min}\,(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}) \tag{5.3}$$

We can also regularise using an L_1 norm, $\|\boldsymbol{\theta}\|_1 = \sum|\theta_i|$, in which case the regularisation circles are replaced by regularisation squares as shown in figure 5.2. This kind of regularisation is known as the **lasso** and as the red path shows it has the effect of assigning zero weights to less important attributes as δ increases.

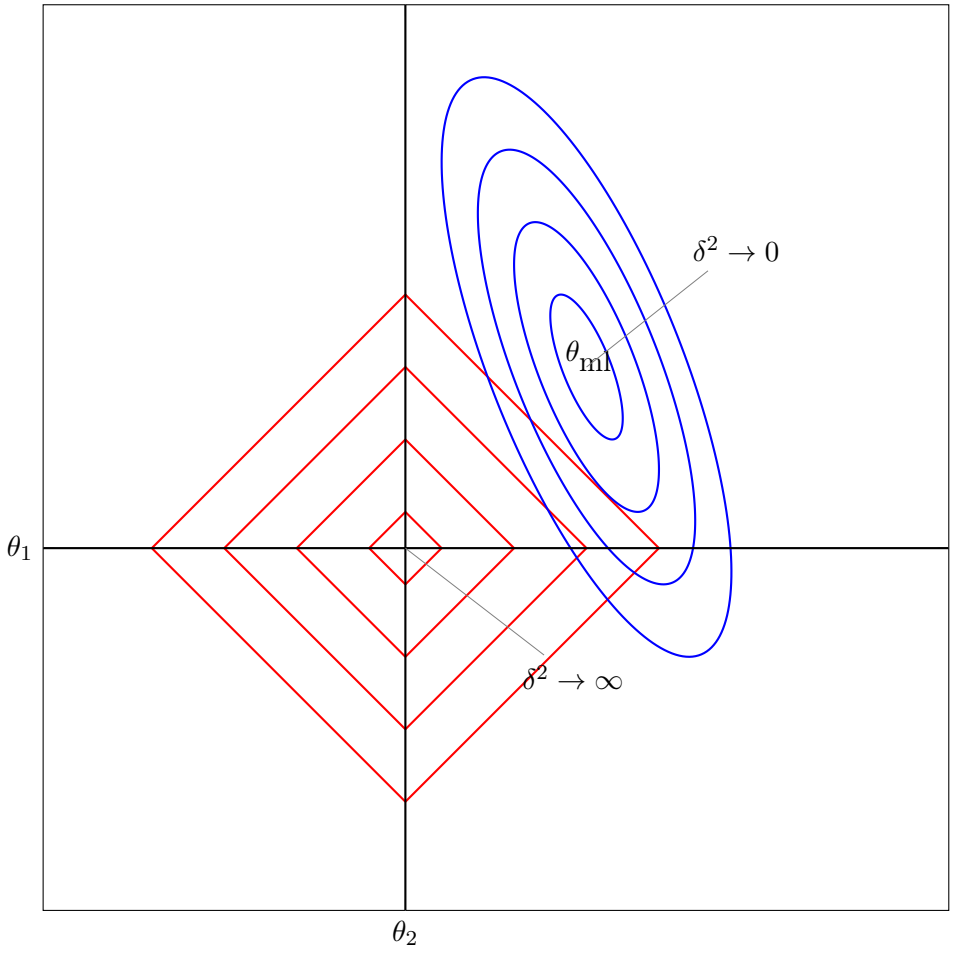


Figure 5.2: A geometric interpretation for the **lasso**, which employs the L_1 norm

5.3 Ridge Regression via Bayes Rule

Bayes Rule allows us to obtain a maximum likelihood estimate for the model parameters, $\boldsymbol{\theta}$. The rule is usually written as follows:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

In our setting we can employ Bayes as follows:

$$P(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y}) = \frac{P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta})P(\boldsymbol{\theta})}{P(\boldsymbol{y}|\boldsymbol{X})}$$

The term in the denominator is independent of $\boldsymbol{\theta}$. We could calculate it by integrating over all possible models as $P(\boldsymbol{y}|\boldsymbol{X}) = \int P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta})P(\boldsymbol{\theta})d\boldsymbol{\theta}$. We see that $\boldsymbol{\theta}$ has been marginalised out so we can treat this term as a constant and then try to find the $\boldsymbol{\theta}$ that maximises the proportionality relation:

$$P(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y}) \propto P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta})P(\boldsymbol{\theta})$$

Now assuming that the data is normally distributed about the model predictions with variance σ^2 and that the model parameters come from a gaussian **prior** with zero mean and variance τ^2 we can use Bayes to write the likelihood for a model as:

$$\begin{aligned} P(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y}) &\propto \left(\sigma^2 2\pi^{-\frac{n}{2}} e^{-\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})}\right) \times \left(\tau^2 2\pi^{-\frac{d}{2}} e^{-\frac{1}{2\tau^2}\boldsymbol{\theta}^T\boldsymbol{\theta}}\right) \\ &= e^{-\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}) - \frac{1}{2\tau^2}\boldsymbol{\theta}^T\boldsymbol{\theta}} \times \sigma^2 2\pi^{-\frac{n}{2}} \times \tau^2 2\pi^{-\frac{d}{2}} \end{aligned}$$

Given that we are looking to maximise this with respect to $\boldsymbol{\theta}$, we can ignore the terms that don't depend on $\boldsymbol{\theta}$ and try to find:

$$\begin{aligned} &\arg\max_{\boldsymbol{\theta}} \left(e^{-\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}) - \frac{1}{2\tau^2}\boldsymbol{\theta}^T\boldsymbol{\theta}}\right) && \text{max likelihood} \\ =&\arg\max_{\boldsymbol{\theta}} \left(-\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}) - \frac{1}{2\tau^2}\boldsymbol{\theta}^T\boldsymbol{\theta}\right) && \text{max log-likelihood} \\ =&\arg\max_{\boldsymbol{\theta}} -1\left((\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}) + \frac{2\sigma^2}{2\tau^2}\boldsymbol{\theta}^T\boldsymbol{\theta}\right) && \text{rearranging} \\ =&\arg\min_{\boldsymbol{\theta}} \left((\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}) + \frac{\sigma^2}{\tau^2}\boldsymbol{\theta}^T\boldsymbol{\theta}\right) && \max(\text{f}) = \min(\text{-f}) \end{aligned}$$

which, if we set $\delta^2 = \frac{\sigma^2}{\tau^2}$, is equivalent to maximising the regularised cost, $J(\boldsymbol{\theta})$, from equation 5.1. So, a lower variance on the prior for the weights, $\boldsymbol{\theta}$ (i.e. a more constrained prior) is equivalent to a higher δ^2 value in the ridge regression solution.

5.4 Going non-linear using basis functions

We introduce basis functions, $\boldsymbol{\phi}(\boldsymbol{x})$, to deal with nonlinearity. For example if $\boldsymbol{\phi}(\boldsymbol{x}) = [\mathbf{1}, \boldsymbol{x}, \boldsymbol{x}^2]$ then $\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{x})\boldsymbol{\theta} + \epsilon$ and we have a model that can fit a parabola to data as shown in figurs 5.3.

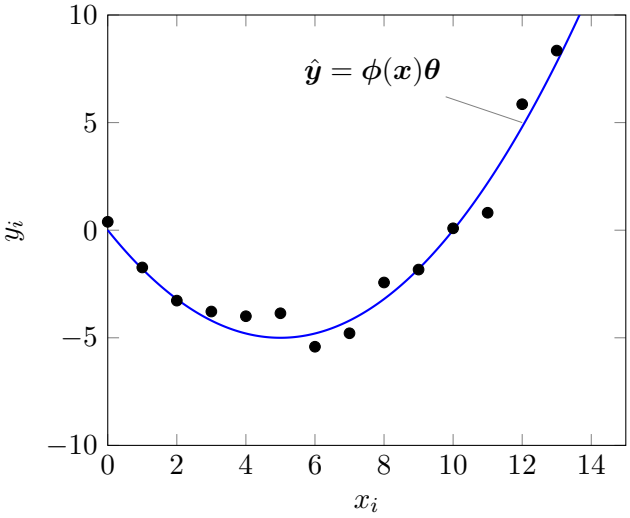


Figure 5.3: Basis functions for a parabolic fit

And finding the optimal model parameters reduces to linear regression with a regularised solution:

$$\boldsymbol{\theta} = (\boldsymbol{\phi}^T\boldsymbol{\phi} + \delta^2\boldsymbol{I})^{-1}\phi^T\boldsymbol{y}$$

By introducing higher degree polynomial basis functions it is a simple matter to adapt this procedure for higher dimensional models as shown in figure 5.4.

$$\boldsymbol{\phi}(\mathbf{x}) = [1, x_1, x_2] \qquad \boldsymbol{\phi}(\mathbf{x}) = [1, x_1, x_2, x_1^2, x_2^2]$$

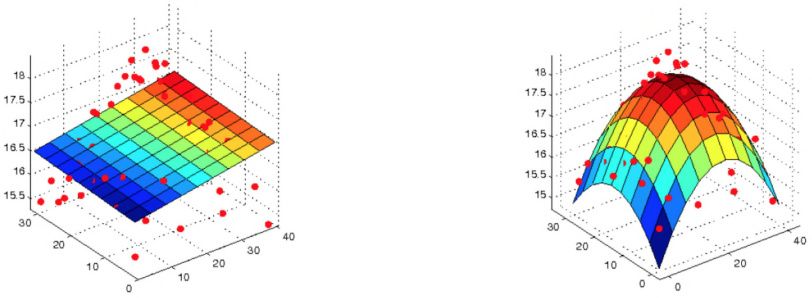


Figure 5.4: Basis functions for a quadratic fits in higher dimensions, linear on the left and quadratic on the right.

5.5 Kernel Regression

We can use so-called kernels as the basis vectors in our regression problem.
For example if the kernels are **radial basis functions** or RBFs then we have

$$\boldsymbol{\phi}(\boldsymbol{x}) = [k(\boldsymbol{x}, \boldsymbol{\mu}_1, \lambda), \ldots, k(\boldsymbol{x}, \boldsymbol{\mu}_d, \lambda] \text{ where } k(\boldsymbol{x}, \boldsymbol{\mu}_i, \lambda) = e^{-\frac{1}{\lambda}\|\boldsymbol{x}-\boldsymbol{\mu}_i\|^2}$$

with an example depected in figure 5.5.

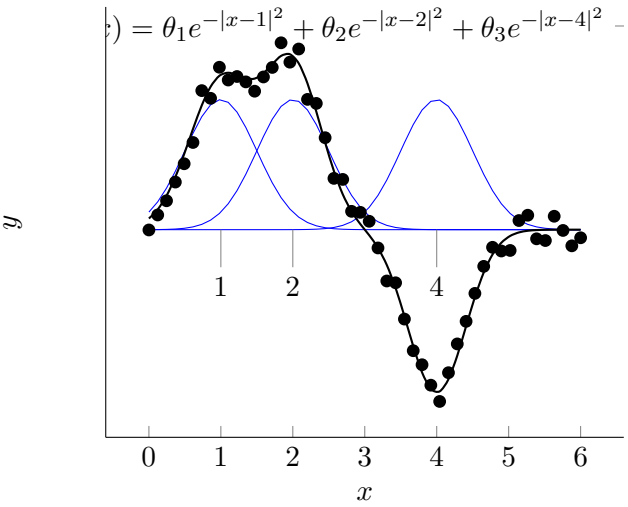


Figure 5.5: Radial Basis Functions for kernel regression

In the example depicted in figure 5.5

$$\boldsymbol{\phi}(x_i) = [1, k(x_i, 1, \lambda), k(x_i, 2, \lambda), k(x_i, 4, \lambda)]$$

and $\boldsymbol{\Phi} = \boldsymbol{\phi}(\boldsymbol{x})$ is an n by 4 matrix which, given the model parameters, $\boldsymbol{\theta}$, we can use to predict output according to $\boldsymbol{y} = \boldsymbol{\Phi}\boldsymbol{\theta}$. Again we can obtain $\boldsymbol{\theta}$ using least squares or ridge regression via one of:

$$\begin{aligned}\boldsymbol{\theta}_{\text{ls}} &= (\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T\boldsymbol{y} \\ \boldsymbol{\theta}_{\text{ridge}} &= (\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \delta^2\boldsymbol{I})^{-1}\boldsymbol{\Phi}^T\boldsymbol{y}\end{aligned}$$

In the appendix, A, we give a link to a Julia script for ridge regression using radial basis functions as the kernels.

In figure 5.6 we show possible output of this script for three different settings for the kernel width.

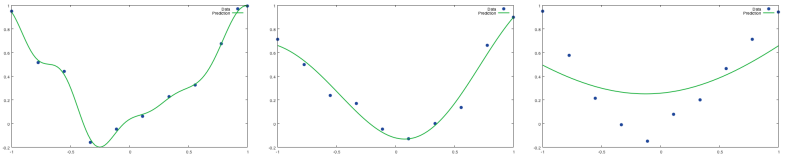


Figure 5.6: RBF regression depends on kernel width, on the left the kernel width is too small and we are overfitting and on the right the kernel width is too large.

The problem of choosing a suitable kernel width, λ, and regularisation parameter, δ^2 is partially solved through the use of **cross validation**.

5.6 Cross Validation

The idea of **cross validation** is simple. We split the training data into K folds; then, for each fold k we train on all the folds but the k^{th}, and test on the k^{th}, in a round-robin fashion. It is common to use $K = 5$; this is called 5-fold CV, see figure 5.7.

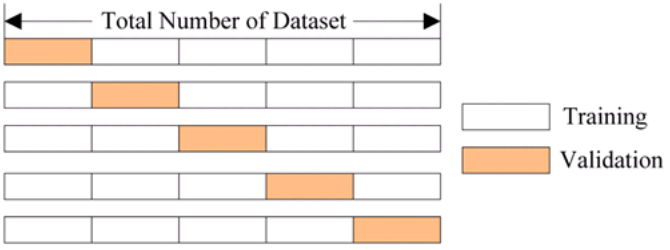


Figure 5.7: 5-fold cross validation

We then average the prediction errors in the test folds and obtain a CV-error versus the parameter we trying to optimise. For example if we were trying to optimise the regularisation parameter we might plot $\log \text{MSE}$ as δ^2 increases and then select the δ^2 that minimises the cross validation error, see figure 5.8.

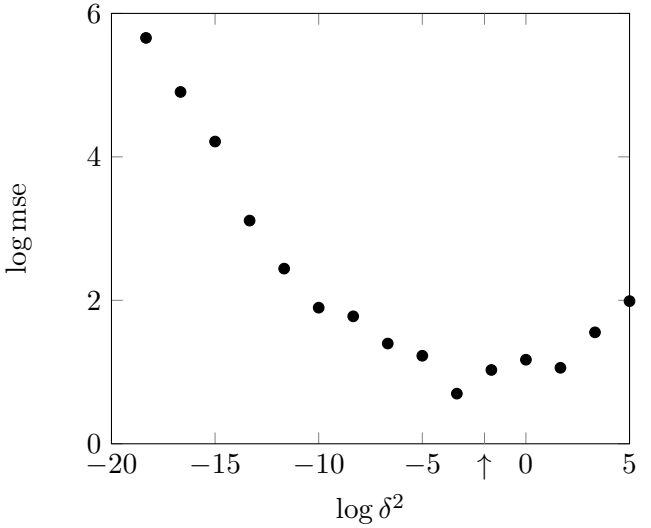


Figure 5.8: Cross Validation Mean Square Error as a function of $\log \delta^2$.

5.7 Ridge Regression with Basis Functions Notebook

in appendix A the reader will find a link to a Jupyter notebook that uses the Julia language and the Linear Algebra package to perform regularised ridge regression by fitting a set of radial basis functions to non-linear data.

5.8 Next Chapter

In the next chapter, we delve into the world of optimisation. Please revise your multivariable calculus and in particular the definition of gradient.

Chapter 6

Optimisation

6.1 Outline

Many machine learning problems can be cast as optimisation problems. This lecture introduces optimisation. The objective is for you to learn:

- The definitions of gradient and Hessian.
- The gradient descent algorithm.
- Newton's algorithm.
- Stochastic gradient descent (SGD) for online learning.
- Popular variants, such as AdaGrad and Asynchronous SGD.
- Improvements such as momentum and Polyak averaging.
- How to apply all these algorithms to linear regression.

6.2 The Gradient and Hessian

A differentiable function $f(\boldsymbol{\theta})$ such that $f : \mathbb{R}^d \to \mathbb{R}$ has a vector field called the **gradient**, $\nabla f(\boldsymbol{\theta})$, at each point $\boldsymbol{\theta}$ in $\mathbb{R}^d$ which is defined as follows:

$$\nabla f(\boldsymbol{\theta}) = \left[\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \ldots, \frac{\partial f}{\partial \theta_d} \right]^T \tag{6.1}$$

For example, in figure 6.1 we show the gradient vector field for a function f mapping $\mathbb{R}^2$ to $\mathbb{R}$.

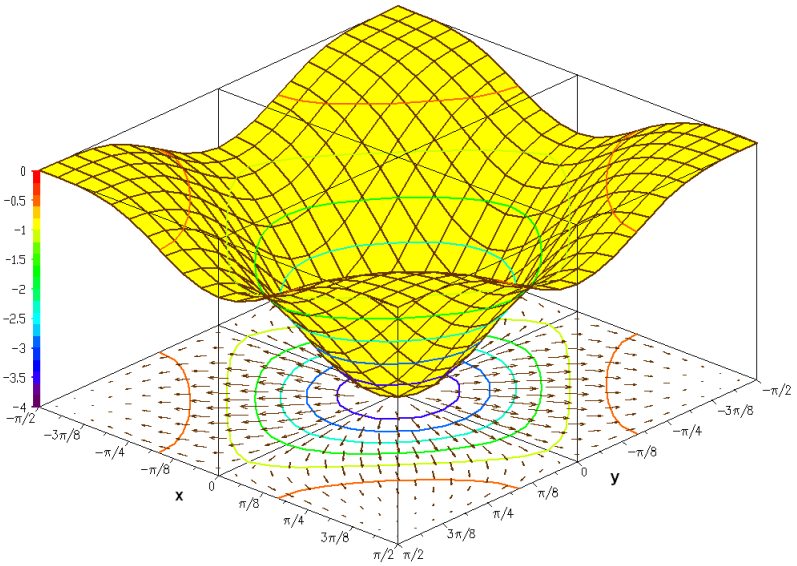


Figure 6.1: The gradient vector field for $f(x, y) = -((\cos x)^2 + (\cos y)^2)^2$.

If $f(\boldsymbol{\theta})$ is twice differentiable in each of its variables then $f : \mathbb{R}^d \to \mathbb{R}$ has a tensor field called the **Hessian** at each point $\boldsymbol{\theta}$ in $\mathbb{R}^d$, The Hessian is sometimes written as the matrix, $\boldsymbol{H}$, and sometimes as the gradient of the gradient, $\nabla^2 f(\boldsymbol{\theta})$, and it is defined as follows:

$$\boldsymbol{H} = \nabla^2 f(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial^2 f}{\partial \theta_1^2} & \frac{\partial^2 f}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 f}{\partial \theta_1 \partial \theta_d} \\ \frac{\partial^2 f}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 f}{\partial \theta_2^2} & \cdots & \frac{\partial^2 f}{\partial \theta_2 \partial \theta_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial \theta_d \partial \theta_1} & \frac{\partial^2 f}{\partial \theta_d \partial \theta_2} & \cdots & \frac{\partial^2 f}{\partial \theta_d^2} \end{bmatrix} \tag{6.2}$$

6.3 Multivariate Taylor expansion

Recall Taylor's expansion for a function of one variable, θ, about some point, θ_k say:

$$f(\theta) = f(\theta_k) + (\theta - \theta_k)f'(\theta_k) + \frac{(\theta - \theta_k)^2}{2!}f''(\theta_k) + \dots \tag{6.3}$$

The multivariate version of Taylor's expansion reads:

$$\begin{aligned} f(\boldsymbol{\theta}) &= f(\boldsymbol{\theta}_k) + (\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \nabla f(\boldsymbol{\theta}_k) + \frac{1}{2!}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \nabla^2 f(\boldsymbol{\theta}_k)(\boldsymbol{\theta} - \boldsymbol{\theta}_k) + \dots \\ &= f(\boldsymbol{\theta}_k) + (\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \boldsymbol{g}_k + \frac{1}{2!}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \boldsymbol{H}_k(\boldsymbol{\theta} - \boldsymbol{\theta}_k) + \dots \end{aligned} \tag{6.4}$$

where we have abbreviated $\nabla f(\boldsymbol{\theta}_k)$ to $\boldsymbol{g}_k$ and $\nabla^2 f(\boldsymbol{\theta}_k)$ to $\boldsymbol{H}_k$.

6.4 Steepest Descent Algorithm

One of the simplest optimisation algorithms is called gradient descent or steepest descent. In the minimisation of $f(\boldsymbol{\theta})$ we follow the path of steepest descent.
The algorithm for choosing the next $\boldsymbol{\theta}$ is written as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \boldsymbol{g}_k \tag{6.5}$$

where k indexes steps of the algorithm and $\eta_k > 0$ is called the **learning rate** or **step size**, see figure 6.2 which was taken from an excellent *Nextjournal* article by Robert Luciani where he used Julia and Flux to animate steepest descent paths in the loss landscape, see [9].

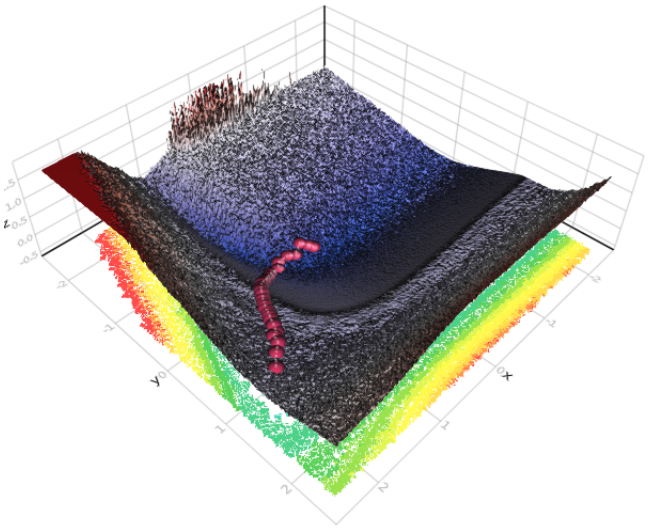


Figure 6.2: A sequence of steps in Steepest Descent path in search of minimum loss, taken from [9] .

6.5 Momentum

We can speed up steepest descent by including a **momentum** parameter that weights the next update step with a term that depends on the previous update step.

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1}) + (1-\alpha)[-\eta_k \boldsymbol{g}_k] \tag{6.6}$$

where α is the momentum parameter with $\alpha = 0$ for no momentum. Usually we set $\alpha = \frac{1}{2}$.

6.6 Newton's Method

Steepest Descent is a first order algorithm. We can speed up the process using a second order process named after Newton where updates to $\boldsymbol{\theta}$ are of the form:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \boldsymbol{H}_k^{-1}\boldsymbol{g}_k \tag{6.7}$$

To derive Newton's method, 6.7, we differentiate the second order multivariate Taylor expansion, 6.4, with respect to $\boldsymbol{\theta}$, treating $\boldsymbol{\theta}_k$ as a constant, and we set the result to zero because we expect zero gradient at an extremum. We then solve for $\boldsymbol{\theta} = \boldsymbol{\theta}_{k+1}$ as follows:

$$\begin{aligned} 0 = \nabla f(\boldsymbol{\theta}) &= 0 + \boldsymbol{g}_k + \boldsymbol{H}_k(\boldsymbol{\theta} - \boldsymbol{\theta}_k) \\ \implies \boldsymbol{\theta} = \boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k - \boldsymbol{H}_k^{-1}\boldsymbol{g}_k \end{aligned} \tag{6.8}$$

6.7 Gradient Descent for Least Squares

When the function to be minimised is the least squares loss function, $J(\boldsymbol{\theta})$, the gradient and Hessian can be computed algebraically:

$$\begin{aligned} J(\boldsymbol{\theta}) &= (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}) \\ \boldsymbol{g}_{\boldsymbol{\theta}} &= \nabla J(\boldsymbol{\theta}) = -2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta} \\ \boldsymbol{H}_{\boldsymbol{\theta}} &= \nabla^2 J(\boldsymbol{\theta}) = 2\boldsymbol{X}^T\boldsymbol{X} \end{aligned} \tag{6.9}$$

and thus steepest descent reduces to

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \left[-2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta}_k\right] \tag{6.10}$$

whilst Newton's method becomes:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \left(2\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\left[-2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta}_k\right]$$

$$= (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{6.11}$$

and we see that in the least squares case, Newton's method reduces to a **one step** method that jumps immediately to the exact solution.

6.8 Newton CG algorithm

Rather than computing $\boldsymbol{d}_k = \boldsymbol{H}_k^{-1}\boldsymbol{g}_k$ directly, we can solve the linear system of equations $\boldsymbol{H}_k\boldsymbol{d}_k = \boldsymbol{g}_k$ for $\boldsymbol{d}_k$. One popular way to do this, especially if $\boldsymbol{H}$ is sparse, is to use a **conjugate gradient** method to solve the linear system:

- initialise $\boldsymbol{\theta}_0$
- for $k = 0, 1, 2, \ldots$ do
  - evaluate $\boldsymbol{g}_k = \nabla f(\boldsymbol{\theta}_k)$
  - evaluate $\boldsymbol{H}_k = \nabla^2 f(\boldsymbol{\theta}_k)$
  - solve $\boldsymbol{H}_k\boldsymbol{d}_k = \boldsymbol{g}_k$ for $\boldsymbol{d}_k$.
  - use a line search to find best η_k along d_k
  - update $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k\boldsymbol{d}_k$

6.9 Stochastic Gradient Descent

As the name implies, **Stochastic Gradient Descent**, or SGD, is an update method that uses n random samples of the data set to estimate the gradient for the descent algorithm to follow.

The derivation of the method has its roots in a probability argument. We estimate the gradient at $\boldsymbol{\theta}$ using:

$$\begin{aligned}\nabla f(\boldsymbol{\theta}) &= \int \nabla f(\boldsymbol{X}, \boldsymbol{\theta}) P(\boldsymbol{X}) d\boldsymbol{X} \\ &= \mathbb{E}\left[\nabla f(\boldsymbol{X}, \boldsymbol{\theta})\right] \\ &\approx \frac{1}{n}\sum_{i=1}^{n} \nabla f(\boldsymbol{X}_{i,:}, \boldsymbol{\theta})\end{aligned} \tag{6.12}$$

and with this approximation to the gradient our update for steepest descent now reads:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \frac{1}{n} \sum_{i=1}^{n} \nabla f(\boldsymbol{X}_{i,:}, \boldsymbol{\theta}_k) \tag{6.13}$$

In the case of least squares where we are trying to optimise

$$J(\boldsymbol{\theta}) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})$$

with gradient

$$\nabla J(\boldsymbol{\theta}) = -2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta}$$

the SGD update for **batch learning** from n randomly chosen samples will read:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{X}_{i,:}^T (y_i - \boldsymbol{X}_{(i)}\boldsymbol{\theta}_k) \tag{6.14}$$

When n is small, $n = 20$ say, this is called **mini-batch learning** and if $n = 1$ it is called **online learning**.

6.10 Adaptive Gradient Descent

Consider the standard steepest descent update:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \boldsymbol{g}_k \tag{6.15}$$

Now the gradient, $\boldsymbol{g}_k$, is a vector with one component for each component of $\boldsymbol{\theta}_k$. Each of these components have the same step size, η, in the evaluation of the update.

In **Adaptive Gradient Descent**, or **AdaGrad**, we weight the i^{th} component of the gradient, $\boldsymbol{g}_k^{(i)}$,as follows;

$$\boldsymbol{\theta}_{k+1}^{(i)} = \boldsymbol{\theta}_k^{(i)} - \frac{\eta_k}{\sqrt{\sum_{t=1}^{k} (\boldsymbol{g}_t^{(i)})^2 + \epsilon}} \boldsymbol{g}_k^{(i)} \tag{6.16}$$

Here, ϵ is a smoothing term that avoids division by zero and the denominator gives rise to a scaling factor for the learning rate that applies to a single parameter $\boldsymbol{g}^{(i)}$. Since the denominator in this scaling factor is the L_2 norm of previous gradient components, extreme parameter updates get dampened, while parameters that get few or small updates receive higher learning rates.

One of AdaGrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of $\eta = 0.01$ and leave it at that.

AdaGrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. There are many attempts in the literature to produce algorithms that aim to resolve this flaw. Some include, **Adadelta**, **RMSprop**, **Adam**, **AdaMax**, **Nadam** and **AMSGrad**.

The reader is referred to an online article by [14], where the different variants are discussed and an animation of paths taken by the different optimisation scheemes is produced, see a still from the animation in figure 6.3.

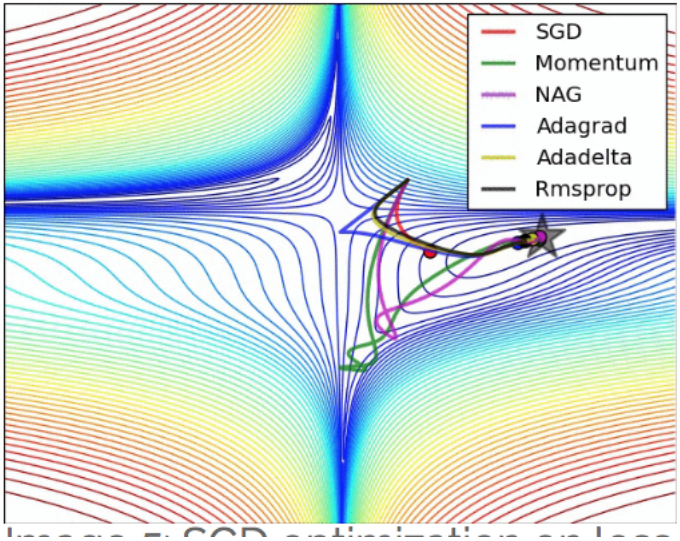


Figure 6.3: Paths taken by different adaptions of Steepest Descent.

6.11 Batch Normalisation

Some of the ideas for this section come from a blog by Jeremy Jordan, see [8].

By normalising all of our inputs to a standard scale, we allow the network to more quickly learn the optimal parameters for each node.

Additionally, it's useful to ensure that our inputs are roughly in the range of -1 to 1 to avoid errors associated with floating point number precision. If your inputs and target outputs are on a completely different scale than the typical -1 to 1 range, the default parameters for your neural network (ie. learning rates) will likely be ill-suited for your data.

It is common practice to scale your data inputs to have zero mean and unit variance. Normalising the input of your network is a well-established technique for improving the convergence properties of a network.

Batch normalisation was proposed to extend the improved loss function topology to more parameters of the network. By ensuring the activations of each layer are normalised, we can simplify the overall loss function topology. This is especially helpful for the hidden layers of our network, since the distribution of unnormalised activations from previous layers will change as the network evolves and learns more optimal parameters. By normalising each layer, we introduce a level of orthogonality between layers - which generally makes for an easier learning process.

Given a vector of linear combinations from the previous layer $\boldsymbol{z}_i$ for each observation i in a dataset, we can calculate the mean and variance as:

$$\begin{aligned} \boldsymbol{\mu} &= \frac{1}{m}\sum_i \boldsymbol{z}_i \\ \sigma^2 &= \frac{1}{m}\sum_i |\boldsymbol{z}_i - \boldsymbol{\mu}|^2 \end{aligned}$$

Using these values, we can normalise the vectors $\boldsymbol{z}_i$ as follows:

$$\bar{v}z_i = \frac{\boldsymbol{z}_i - \boldsymbol{\mu}}{\sqrt{\sigma^2 + \epsilon}}$$

We add a very small number ϵ to prevent the chance of a divide by zero error.

6.12 Stochastic Gradient Descent Notebook

in appendix A the reader will find a link to a Jupyter notebook that uses the Julia language to perform stochastic gradient descent to find the optimum parameters for minimising a quadratic loss function.

6.13 Next Chapter

In the next chapter, we apply these optimisation techniques to learn the parameters of a neural network with a single neuron (logistic regression).

Chapter 7

Logistic Regression

7.1 Outline

This lecture describes the construction of binary classifiers using a technique called **Logistic Regression**. The objective is for you to learn:

- How to apply logistic regression to discriminate between two classes.
- How to formulate the logistic regression likelihood.
- How to derive the gradient and Hessian of logistic regression.
- How to incorporate the gradient vector and Hessian matrix into Newton's optimisation algorithm so as to come up with an algorithm for logistic regression, which we call IRLS.
- How to do logistic regression with the **softmax** link.

7.2 McCulloch-Pitts Model of a Neuron

Figure 7.1 show how a Neuron is modelled as a network of two functional compositions. The sum of weighted inputs is *squashed* to decide whether or not the neuron output fires.

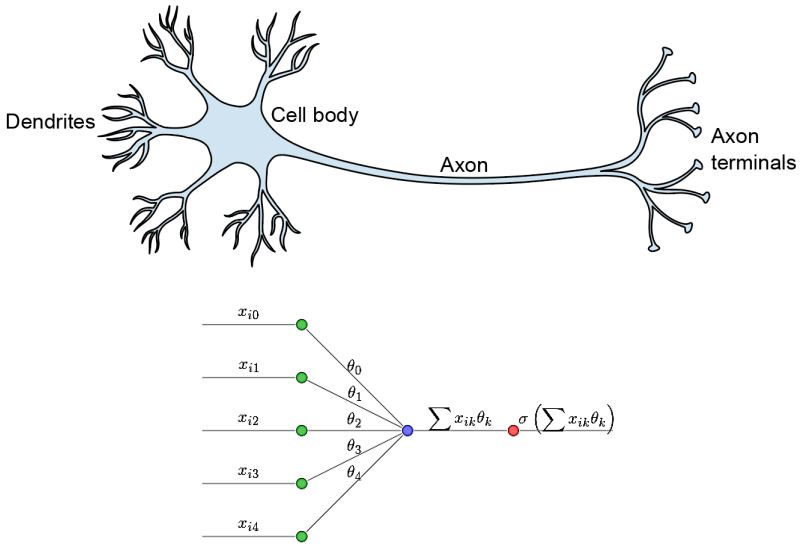


Figure 7.1: The biological single neuron (top) with the McCulloch-Pitts model (bottom).

7.3 The Sigmoid squashing function

The squashing function, σ, for the McCulloch-Pitts neuron model is usually implemented as a `sigmoid` function, also known as the `logistic` or `logit` function. The sigmoid function maps the whole Real line onto the interval $[0, 1]$ and thus can be interpreted as producing a probability of whether the input belongs to a particular class, see figure 7.2.

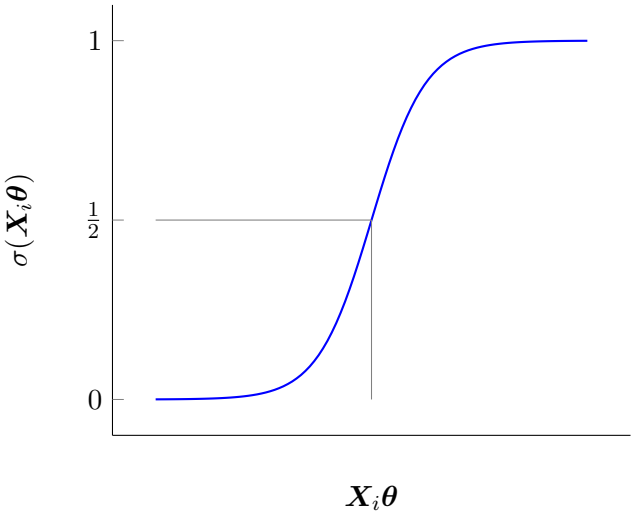


Figure 7.2: The `sigmoid` squashing function, $\sigma(\boldsymbol{X}_i\boldsymbol{\theta})$.

When the input is $\boldsymbol{X}_i\boldsymbol{\theta}$ the *squashing* and hence the class assignment is accomplished by evaluating:

$$\sigma_i = P(y_i = 1|\boldsymbol{X}_i, \boldsymbol{\theta}) = \sigma(\boldsymbol{X}_i\boldsymbol{\theta}) = \frac{1}{1 + e^{-\boldsymbol{X}_i\boldsymbol{\theta}}} \tag{7.1}$$

7.4 Separating Hyper-Plane

Consider two dimensional data points that are labeled in one of two classes, $\{0 \text{ blue}, 1 \text{ red}\}$. With this setup, once we have learnt the parameters $\boldsymbol{\theta}$, equation 7.1 will yield a **separating plane** precisely when $\boldsymbol{X}_i\boldsymbol{\theta} = 0$, see figure 7.3.

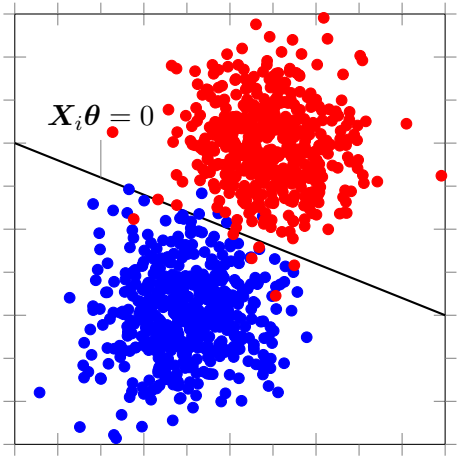


Figure 7.3: σ as a separator, $\sigma_i = p(\boldsymbol{y}_i = 1|\boldsymbol{X}_i, \boldsymbol{\theta}) = \Pi(\boldsymbol{X}_i\boldsymbol{\theta}) = \frac{1}{2}$ when $\boldsymbol{X}_i\boldsymbol{\theta} = 0$.

7.5 Negative Log-Likelihood

In order to learn $\boldsymbol{\theta}$ we must define an objective function to optimise. To do this we will assume that the distribution of $y_i \in \{0, 1\}$ behaves like a discrete Bernoulli random variable given input π_i. Thus for the full data set $\boldsymbol{X}$ we have using 4.17 that the likelihood is

$$\begin{aligned} p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) &= \prod_{i=1}^{n} \text{Ber}(y_i|\pi_i) \\ &= \prod_{i=1}^{n} \pi_i^{y_i}(1-\pi_i)^{1-y_i} \end{aligned} \tag{7.2}$$

and we must optimise the negative log-likelihood (NLL) loss function which using equation 4.18 now reads:

$$\begin{aligned} \mathscr{L}(\boldsymbol{\theta}) &= -\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) \\ &= -\sum_{i=1}^{n} y_i \log \pi_i + (1-y_i)\log(1-\pi_i) \end{aligned} \tag{7.3}$$

7.6 Gradient and Hessian

The gradient and the Hessian of this loss function can be computed via:

$$\boldsymbol{g}(\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}}\mathscr{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} y_i \boldsymbol{X}_i^T(\pi_i - y_i) = \boldsymbol{X}^T(\boldsymbol{\pi} - \boldsymbol{y}) \tag{7.4}$$

$$\boldsymbol{H}(\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}}\boldsymbol{g}(\boldsymbol{\theta})^T = \sum_{i=1}^{n} \pi_i(1-\pi_i)\boldsymbol{x}_i\boldsymbol{x}_i^T = \boldsymbol{X}^T\boldsymbol{S}\boldsymbol{X} \tag{7.5}$$

where S is a $d \times d$ diagonal matrix with i^{th} entry $\pi_i(1-\pi_i)$
Deriving equations 7.4 and 7.5 is mechanical and left to the reader as an exercise.

7.7 Steepest Descent

This time Newton's method won't yield a one-step method, because the loss function is no longer quadratic, but we can still iterate downhill using Newton's method:

$$\begin{aligned}
\boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k - \boldsymbol{H}_k^{-1}\boldsymbol{g}_k \\
&= \boldsymbol{\theta}_k + (\boldsymbol{X}^T S_k \boldsymbol{X})^{-1}\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{\pi}_k) \\
&= (\boldsymbol{X}^T S_k \boldsymbol{X})^{-1}[(\boldsymbol{X}^T S_k \boldsymbol{X})\boldsymbol{\theta}_k + \boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{\pi}_k) \\
&= (\boldsymbol{X}^T S_k \boldsymbol{X})^{-1}\boldsymbol{X}^T[\boldsymbol{S}_k\boldsymbol{X}\boldsymbol{\theta}_k + \boldsymbol{y} + \boldsymbol{\theta}_k]
\end{aligned} \tag{7.6}$$

7.8 SoftMax formulation

In figure 7.4 below, we show the so-called **softmax** formulation for logistic regression. Instead of feeding $\boldsymbol{X}_i\boldsymbol{\theta}$ though one `sigmoid` function, we maintain **two** sets of parameters and feed, $\boldsymbol{X}_i\boldsymbol{\theta}_1$ and $\boldsymbol{X}_i\boldsymbol{\theta}_2$ to their own `softmax` function in order to generate a **probability** for each of the two classes.

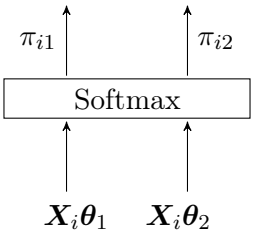


Figure 7.4: The `softmax` formulation for **two** classes which allows generalisation to more than two classes .

$$\pi_{i1} = \frac{e^{\boldsymbol{X}_i\boldsymbol{\theta}_1}}{e^{\boldsymbol{X}_i\boldsymbol{\theta}_1} + e^{\boldsymbol{X}_i\boldsymbol{\theta}_2}} = P(y_i = 0|\boldsymbol{X}, \boldsymbol{\theta})$$
$$\pi_{i2} = \frac{e^{\boldsymbol{X}_i\boldsymbol{\theta}_2}}{e^{\boldsymbol{X}_i\boldsymbol{\theta}_1} + e^{\boldsymbol{X}_i\boldsymbol{\theta}_2}} = P(y_i = 1|\boldsymbol{X}, \boldsymbol{\theta})$$

7.9 Loss function for the softmax formulation

We compute a likelihood for the softmax formulation as follows

$$P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) = \prod_{i=1}^{n} \pi_{i1}^{\Pi_0(y_i)} \pi_{i2}^{\Pi_1(y_i)} \tag{7.7}$$

where $\Pi_c(y_i)$ is the **indicator function** that takes on the value 1 if $y_i = c$ and 0 otherwise.

The loss function to optimise is then provided by the negative log likelihood thus:

$$\mathscr{L}(\boldsymbol{\theta}) = -\log P(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) = -\sum_{i=1}^{n} \Pi_0(y_i) \log \pi_{i1} + \Pi_1(y_i) \log \pi_{i2} \tag{7.8}$$

The expression $\log \pi_{ic}$ gives rise to the `LogSoftMax` layer as shown in figure 7.5.

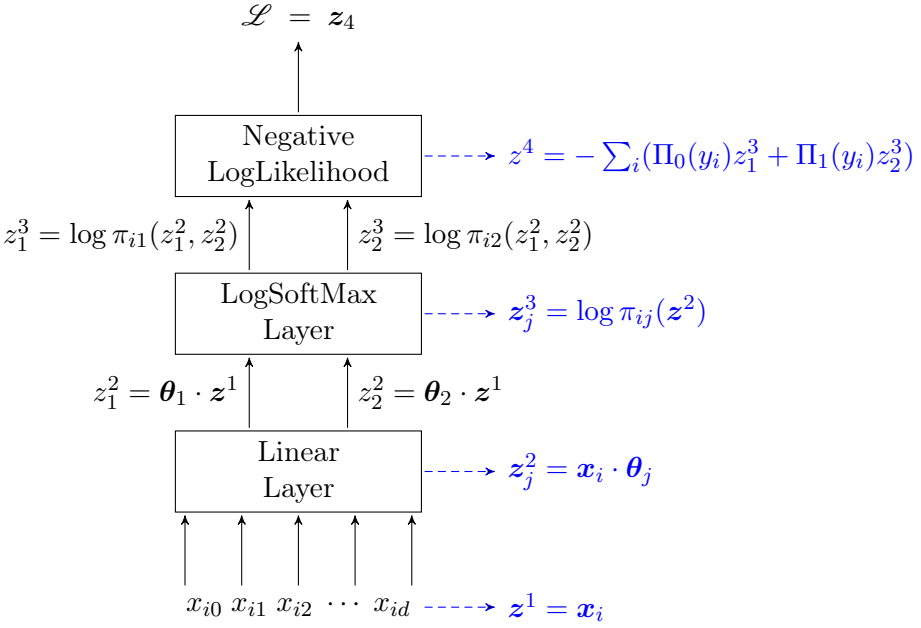


Figure 7.5: The neural network representation of loss due to a `logsoftmax` formulation.

To implement this network we must be able to compute an expression for the gradient of the loss function. We can obtain an expression for the gradient due to the differentiable nature of the `logsoftmax` function. For example to obtain the gradient with respect to $\boldsymbol{\theta}_2$ we note that:

$$\begin{aligned} \frac{\partial}{\partial \boldsymbol{\theta}_2} \log \pi_{i1} &= -\boldsymbol{X}_i \pi_{i2} \\ \frac{\partial}{\partial \boldsymbol{\theta}_2} \log \pi_{i2} &= \boldsymbol{X}_i (1 - \pi_{i2}) \end{aligned} \tag{7.9}$$

the proof of which is left as an exercise to the reader. Using equations 7.9 we can obtain an expression for the gradient of the loss function with respect to $\boldsymbol{\theta}_2$ say as follows:

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\theta}_2}\mathscr{L}(\boldsymbol{\theta}) &= -\sum_{i=1}^{n} \Pi_0(y_i)(-\boldsymbol{X}_i\pi_{i2}) + \Pi_1(y_i)\boldsymbol{X}_i(1-\pi_{i2}) \\ &= -\sum_{i=1}^{n} (1-y_i)(-\boldsymbol{X}_i\pi_{i2}) + y_i\boldsymbol{X}_i(1-\pi_{i2})\end{aligned}$$

which reduces to:

$$\frac{\partial}{\partial \boldsymbol{\theta}_2}\mathscr{L}(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \boldsymbol{X}_i(y_i - \pi_{i2}) \tag{7.10}$$

The equivalent expression for the gradient with respect to $\boldsymbol{\theta}_1$ can be deduced from the symmetry of the softmax formulation as:

$$\frac{\partial}{\partial \boldsymbol{\theta}_1}\mathscr{L}(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \boldsymbol{X}_i(y_i - \pi_{i1}) \tag{7.11}$$

and one can now see how to generalise the forward computation of the loss function for logsoftmax network as well as the backward computation of the gradients with respect to parameters for $m > 2$ classes.

$$\begin{aligned}\mathscr{L}(\boldsymbol{\theta}) &= -\sum_{i=1}^{n}\sum_{j=1}^{m} \Pi_j(y_i)\log \pi_{ij} \\ \frac{\partial}{\partial \boldsymbol{\theta}_j}\mathscr{L}(\boldsymbol{\theta}) &= -\sum_{i=1}^{n} \boldsymbol{X}_i(y_i - \pi_{ij})\end{aligned} \tag{7.12}$$

7.10 SoftMax Notebook

in appendix A the reader will find a link to a Jupyter notebook that uses the Julia language together with the Flux package to introduce the *softmax* formulation that allows us to build a three class classifier for images of fruit.

7.11 Next Chapter

In the next chapter, we develop a layer-wise approach to computing all the necessary derivatives known as back-propagation. This is the approach used in most deep learning packages.

Chapter 8

Backpropagation

8.1 Outline

This lecture describes modular ways of formulating and learning distributed representations of data. The objective is for you to learn:

- How to specify models such as logistic regression in layers.
- How to formulate layers and loss criterions.
- How well formulated local rules results in correct global rules.
- How back-propagation works.
- How this manifests itself in Torch.

8.2 A layered network

To demonstrate how back propagation is constructed we will use the example from the previous lecture as was shown in figure 7.5.

In that example, the loss function $\mathscr{L}(\boldsymbol{\theta})$ can be expressed as a function of the two sets of parameters, $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, using functional composition:

$$\mathscr{L}(\boldsymbol{\theta}) = z^4\left[z_1^3\left[z_1^2(\boldsymbol{\theta}_1, z^1), z_2^2(\boldsymbol{\theta}_2, z^1)\right], z_2^3\left[z_1^2(\boldsymbol{\theta}_1, z^1), z_2^2(\boldsymbol{\theta}_2, z^1)\right]\right] \tag{8.1}$$

Using this functional composition one can find the gradient of the loss function in parameter space via the chain rule. For example:

$$\frac{\partial \mathscr{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} = \frac{\partial z^4}{\partial z_1^3}\frac{\partial z_1^3}{\partial z_1^2}\frac{\partial z_1^2}{\partial \boldsymbol{\theta}_1} + \frac{\partial z^4}{\partial z_1^3}\frac{\partial z_1^3}{\partial z_2^2}\frac{\partial z_2^2}{\partial \boldsymbol{\theta}_1} + \frac{\partial z^4}{\partial z_2^3}\frac{\partial z_2^3}{\partial z_1^2}\frac{\partial z_1^2}{\partial \boldsymbol{\theta}_1} + \frac{\partial z^4}{\partial z_2^3}\frac{\partial z_2^3}{\partial z_2^2}\frac{\partial z_2^2}{\partial \boldsymbol{\theta}_1} \tag{8.2}$$

and a similar expression can be written down for $\frac{\partial \mathscr{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_2}$. However, this decomposition of the gradient is not efficient as many of the partial derivatives are repeated. Instead we turn to a layered representation for the network.

8.3 Layer Specification

In figure 8.1 we focus on the functional composition structure of *forward/backward* propagation and outline a 3-brick representation of the neural network from figure 7.5 of the previous chapter.

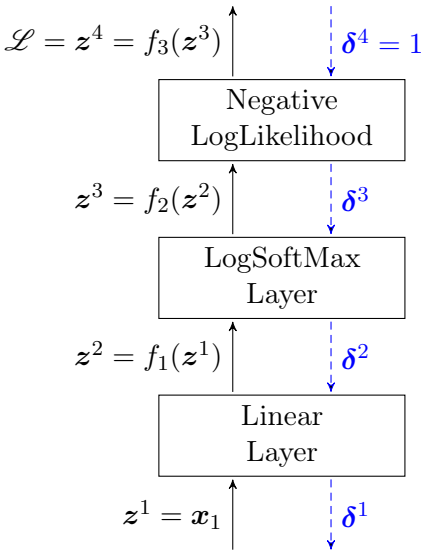


Figure 8.1: our example network consisting of 3 layers each consisting of one *brick* .

In figure 8.2 we give a *brick* schematic for a single layer that takes into account any parameters involved in that layer.

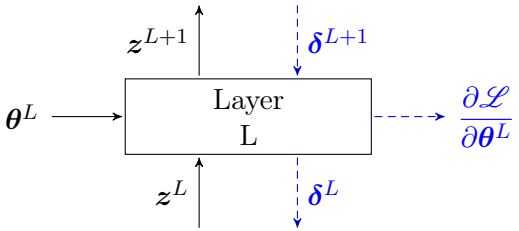


Figure 8.2: A schematic for a single layer L with parameters $\boldsymbol{\theta}^L$.

In these diagram we have abbreviated the notation, z^L and δ^L may have many components. The forward pass through the network to evaluate the loss now reads:

$$z_i^{L+1} = f_L(z^L) \tag{8.3}$$

whilst the backward pass to evaluate the gradient reads

$$\begin{aligned} \delta_i^L &= \frac{\partial \mathscr{L}}{\partial z_i^L} = \sum_j \frac{\partial \mathscr{L}}{\partial z_j^{L+1}} \frac{\partial z_j^{L+1}}{\partial z_i^L} = \sum_j \delta_j^{L+1} \frac{\partial z_j^{L+1}}{\partial z_i^L} \\ \frac{\partial \mathscr{L}}{\partial \boldsymbol{\theta}^L} &= \sum_j \frac{\partial \mathscr{L}}{\partial z_j^{L+1}} \frac{\partial z_j^{L+1}}{\partial \boldsymbol{\theta}^L} = \sum_j \delta_j^{L+1} \frac{\partial z_j^{L+1}}{\partial \boldsymbol{\theta}_i^L} \end{aligned} \tag{8.4}$$

Note that the *brick* by *brick* algorithm delivers the expected gradient for the full network, for example:

$$\begin{aligned} \frac{\partial \mathscr{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} &= \sum_{j=1}^{2} \frac{\partial \mathscr{L}}{\partial z_j^2} \frac{\partial z_j^2}{\partial \boldsymbol{\theta}_1} \\ &= \sum_{j=1}^{2} \left(\sum_{k=1}^{2} \frac{\partial \mathscr{L}}{\partial z_k^3} \frac{\partial z_k^3}{\partial z_j^2} \right) \frac{\partial z_j^2}{\partial \boldsymbol{\theta}_1} \\ &= \sum_{j=1}^{2} \left(\sum_{k=1}^{2} \left(\sum_{l=1}^{1} \frac{\partial \mathscr{L}}{\partial z_l^4} \frac{\partial z_l^4}{\partial z_k^3} \right) \frac{\partial z_k^3}{\partial z_j^2} \right) \frac{\partial z_j^2}{\partial \boldsymbol{\theta}_1} \\ &= \sum_{j=1}^{2} \sum_{k=1}^{2} 1 \frac{\partial z^4}{\partial z_k^3} \frac{\partial z_k^3}{\partial z_j^2} \frac{\partial z_j^2}{\partial \boldsymbol{\theta}_1} \end{aligned}$$

$$= \text{ same expression as before} \tag{8.5}$$

8.4 Reverse Differentiation and why it works

On the forward pass loss output from the network is a nested composition of functions from each layer:

$$\text{Loss} = \mathscr{L} = f^L(\boldsymbol{f}^{L-1}(\dots \boldsymbol{f}^2(\boldsymbol{f}^1(\boldsymbol{x}))\dots))$$

where we have bundled $\boldsymbol{\theta}$ in with $\boldsymbol{x}$ to shorten the notation.

Now we could choose to evaluate the gradient of the loss function with respect to the parameters as follows

$$\frac{\partial f^L(\boldsymbol{f}^{L-1}(\dots \boldsymbol{f}^2(\boldsymbol{f}^1(\boldsymbol{x}))\dots))}{\partial \boldsymbol{x}} = \frac{\partial f^L}{\partial \boldsymbol{f}^{L-1}}\frac{\partial \boldsymbol{f}^{L-1}}{\partial \boldsymbol{f}^{L-2}}\cdots\frac{\partial \boldsymbol{f}^2}{\partial \boldsymbol{f}^1}\frac{\partial \boldsymbol{f}^1}{\partial \boldsymbol{x}}$$

However the expression on the right hand side requires storage for $L-1$ matrices and one vector which could be prohibitive if there are many parameters. Instead we **accumulate** the gradient on the backward pass by evaluating it in a nested fashion as shown below

$$\frac{\partial f^L(\boldsymbol{f}^{L-1}(\dots \boldsymbol{f}^2(\boldsymbol{f}^1(\boldsymbol{x}))\dots))}{\partial \boldsymbol{x}} = \left(\left(\left(\cdots\left(\frac{\partial f^L}{\partial \boldsymbol{f}^{L-1}}\frac{\partial \boldsymbol{f}^{L-1}}{\partial \boldsymbol{f}^{L-2}}\right)\cdots\right)\frac{\partial \boldsymbol{f}^2}{\partial \boldsymbol{f}^1}\right)\frac{\partial \boldsymbol{f}^1}{\partial \boldsymbol{x}}\right)$$

This nested computation only requires storage for one vector and one matrix at a time.

8.5 Deep Learning layered representation

We can now generalise our layered representation for a complete network consisting of L layers with the last layer being a **loss** layer as shown in figure 8.3

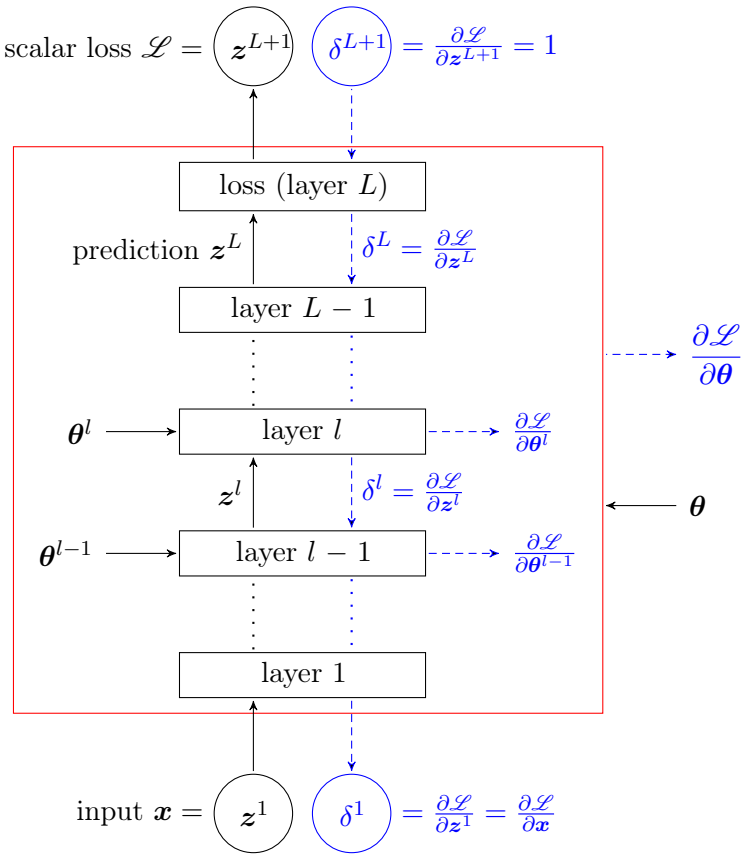


Figure 8.3: Deep Learning back propagation.

We emphasise the nested layered structure by enclosing the whole network using a single layer schematic. Consider a single layer as shown in the schematic figure 8.4.

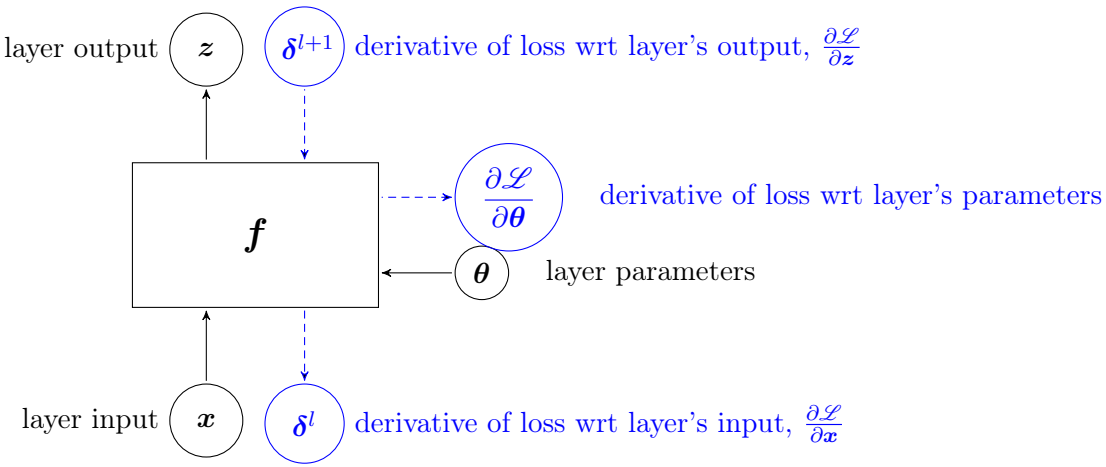


Figure 8.4: Schematic for a single layer of deep learning back propagation.

To implement such a layer we must implement 3 computations:

$$\begin{aligned} \boldsymbol{z}_j &= f_j(\boldsymbol{x}, \boldsymbol{\theta}_j) \\ \delta_i^l &= \sum_j \delta_j^{l+1} \frac{\partial f_j(\boldsymbol{x}; \boldsymbol{\theta}_j)}{\partial x_i} \\ \frac{\partial \mathscr{L}}{\partial \theta_{ij}} &= \sum_j \delta_j^{l+1} \frac{\partial f_j(\boldsymbol{x}; \boldsymbol{\theta}_j)}{\partial \theta_{ij}} \end{aligned} \tag{8.6}$$

where i is an index running over the input to the layer and j is an index running over the output.

The vector valued function, $\boldsymbol{f}$, has components j that are each multivariate functions of inputs i. Each connection from input to output may involve a learnable parameter $\boldsymbol{\theta}_{ij}$

In what follows we will construct units with associated **layer equations** for each of the networks we have seen thus far.

8.6 Example Layer Implementation

8.6.1 Linear units

For the **linear layer**, we have $f_j(\boldsymbol{x}, \boldsymbol{\theta}_j) = \sum_i x_i \theta_{ij}$, and thus the layer equations 8.6 for a linear layer are implemented as

$$\begin{aligned} \boldsymbol{z}_j &= f_j(\boldsymbol{x}, \boldsymbol{\theta}_j) = \sum_i x_i \theta_{ij} \\ \delta_i^l &= \sum_j \delta_j^{l+1} \frac{\partial f_j(\boldsymbol{x}; \boldsymbol{\theta}_j)}{\partial x_i} = \sum_j \delta_j^{l+1} \theta_{ij} \\ \frac{\partial \mathscr{L}}{\partial \theta_{ij}} &= \sum_j \delta_j^{l+1} \frac{\partial f_j(\boldsymbol{x}; \boldsymbol{\theta}_j)}{\partial \theta_{ij}} = \delta_j^{l+1} x_i \end{aligned} \tag{8.7}$$

8.6.2 Squashing units

For a **sigmoid layer** $z = \frac{1}{1+e^{-x}}$, or a **tanh** layer $z = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, with the number of outputs equal to the number of inputs, each output is computed as a sigmoid of the corresponding input. There are no parameters and we only need one index.

$$\begin{aligned} & \text{sigmoid} & & \text{tanh} \\ z_j &= \frac{1}{1+e^{-x_j}} & z_j &= \frac{e^{x_j} - e^{-x_j}}{e^{x_j} + e^{-x_j}} \\ \delta_i^l &= \delta_i^{l+1} z_i (1 - z_i) & \delta_i^l &= \delta_i^{l+1} (1 - z_i^2) \end{aligned} \tag{8.8}$$

As an exercise, try to derive the back propagation gradient updates presented in equations 8.8.

8.6.3 Rectified Linear Unit

In the 1980's Prof. Fukushima from Japan introduced the **Rectified Linear Unit** (or ReLU) as a simple alternative to the sigmoid function, see figure 8.5

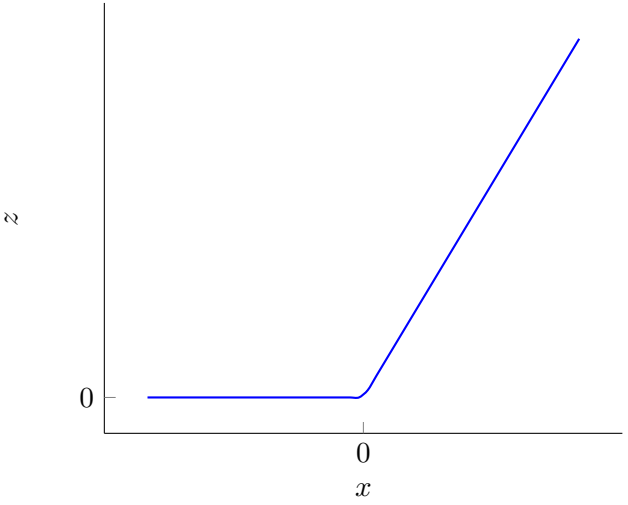


Figure 8.5: The **Rectified linear unit**, $z(x) = \max(0, x)$.

The ReLU can be implemented as follows:

$$\begin{aligned} & \text{relu} \\ z_j &= \max(0, x_j) \\ \delta_i^l &= \delta_i^{l+1} \Pi(x_i > 0) \end{aligned} \tag{8.9}$$

In the 1980's ReLUs were seldom used because networks were shallow but they have become popular again with the advent of deep learning.

8.6.4 SoftMax unit

The layer equations for a SoftMax unit, $z_j = \frac{e^{x_j}}{\sum_i e^{x_i}}$, are more difficult to derive. Although there are no parameters and we again have as many outputs as there are inputs, this time each output depends on **all** the inputs.

$$\begin{aligned} & \text{softmax} \\ z_j &= \frac{e^{x_j}}{\sum_i e^{x_i}} \\ \delta_i^l &= z_i \left(\delta_i^{l+1} - (\boldsymbol{z} \cdot \boldsymbol{\delta}^{L+1}) \right) \end{aligned} \tag{8.10}$$

As an exercise, try to derive the back propagation gradient updates for the softmax layer as presented in equation 8.10.

8.7 Layered Network Notebook

in appendix A the reader will find a link to a Jupyter notebook that uses the Julia language together with the Flux package to introduce the *layered* structure of Flux models that provide the user with forward function evaluations back-propagation of gradients so that parameters for the model may be learnt via gradient descent. As an example a digit recogniser is built for the MNIST digit dataset.

8.8 Next Chapter

In the next chapter, we will look at a successful type of neural network that is very popular in speech and object recognition, known as a **convolutional neural network**.

Chapter 9

Convolutional Neural Networks

9.1 Outline

This lecture introduces you to convolutional neural networks. These models have revolutionised speech and object recognition. The goal is for you to learn

- Convnets for object recognition and language
- How to design convolutional layers
- How to design pooling layers
- How to build convnets in torch

9.2 A Convolutional Neural Network

Before defining the **convolution** operation we start by outlining the architecture for a convolutional neural network as shown in figure 9.1.

In the ConvNet model, we start with a **convolution layer** where the input data are **images**, two dimensional arrays of pixels, and the parameters for the layer are **filters**, small two dimensional arrays.

The network also employs **pooling layers** where the image data is sub-sampled so as to reduce the number of pixels in the data.

After a series of convolutions followed by pooling the data is passed to a Multi Layer Perceptron for final classification.

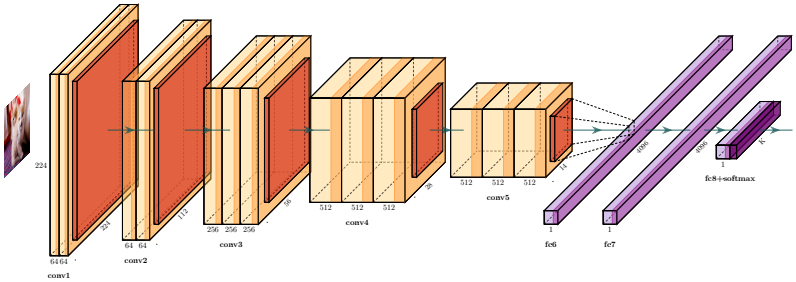


Figure 9.1: A convolutional neural net for labeled images, the architecture is from [15].

9.3 Correlation and Convolution

Given a 1D signal, $\boldsymbol{x}$, we can **correlate** it with a smaller filter, $\boldsymbol{w}$, of odd length indexed by $-k \leq j \leq k$ by computing a new signal $\boldsymbol{z}$ as follows

$$z_i = \sum_{j=-k}^{k} x_{i+j} w_j \tag{9.1}$$

which we write using vector notation as

$$\boldsymbol{z} = \boldsymbol{x} \otimes \boldsymbol{w} \tag{9.2}$$

There are issues at the boundary of the input signal, $\boldsymbol{x}$, but they can be resolved using **zero padding**. Correlation can be thought of as a **matching** operation. When the input signal matches the filter then the output signal is high.

Convolution is similar to correlation except that we **flip** the filter about its midpoint before computing the output. So for convolution, if $\overline{\boldsymbol{w}} = \text{flip}(\boldsymbol{w})$, we compute:

$$z_i = \sum_{j=-k}^{k} x_{i+j} w_{-j} = \sum_{j=-k}^{k} x_{i+j} \overline{w}_j \tag{9.3}$$

which we can write using vector notation as

$$\boldsymbol{z} = \boldsymbol{x} \otimes \overline{\boldsymbol{w}} \tag{9.4}$$

Note that when the filter is symmetric about its mid-point, convolution and correlation are identical.

9.4 Two dimensions

Convolution can be carried out in more than one dimension. In two dimensions the input and output signals are **images** and the filters are small two dimensional arrays with a central element. The flip operation is carried out in both dimensions. and the convolution operation becomes:

$$Z_{ij} = \sum_{k_1=-K}^{K} \sum_{k_2=-K}^{K} X_{i+k_1,j+k_2} \overline{W}_{k_1,k_2} \tag{9.5}$$

which we write using matrix notation as

$$\boldsymbol{Z} = \boldsymbol{X} \otimes \overline{\boldsymbol{W}} \tag{9.6}$$

We can think of convolution as the sliding of the flipped filter over the input image and at each pixel in the input image taking the dot product of the filter with that piece of the input image currently under the sliding filter. The process is depicted in figure 9.2.

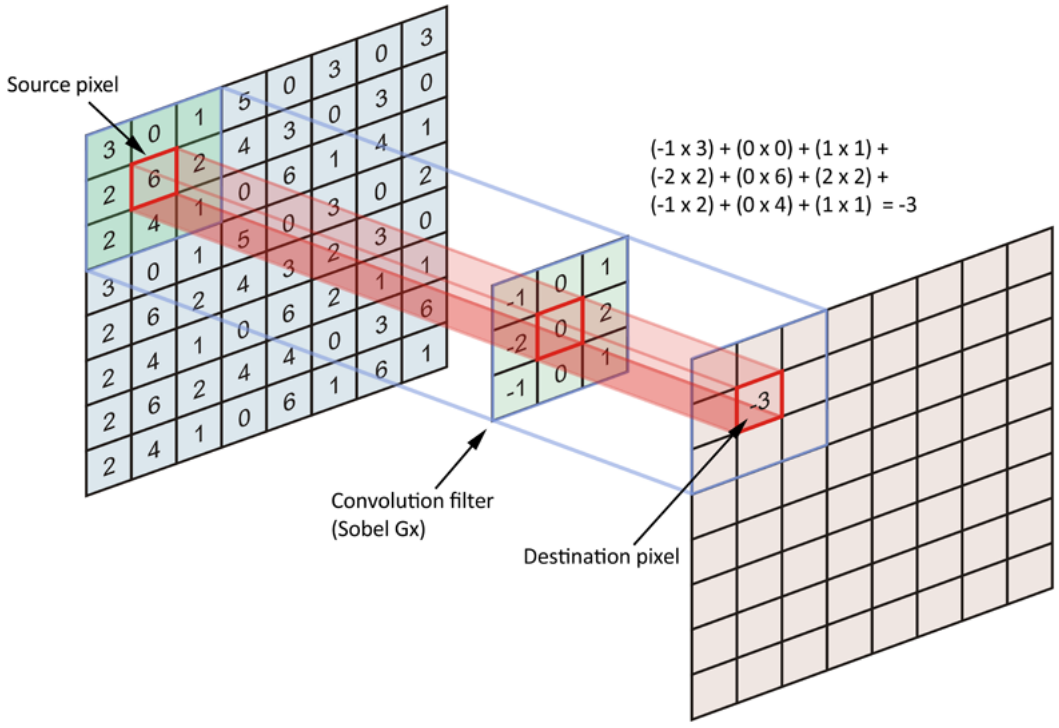


Figure 9.2: The convolution operation.

9.5 Three dimensions

Input to a convolutional layer of a ConvNet are usually not images but **volumes** For example an RBG image can be thought of as a 3 dimensional volume with with width, W, height, H, and depth, 3. The first layer of the network might perform convolutions with say F different filters that are also volumes each of size $K \times K \times 3$ The depth of the filters is the same as the depth of the input volume. The filters are translated over the input volume in the width and height dimensions and 3-dimensional dot products are computed in the overlap region to end up with a $W \times H$ output image per filter. So we end up with a $W \times H \times F$ output volume, see figure 9.3 for a pictorial representation.

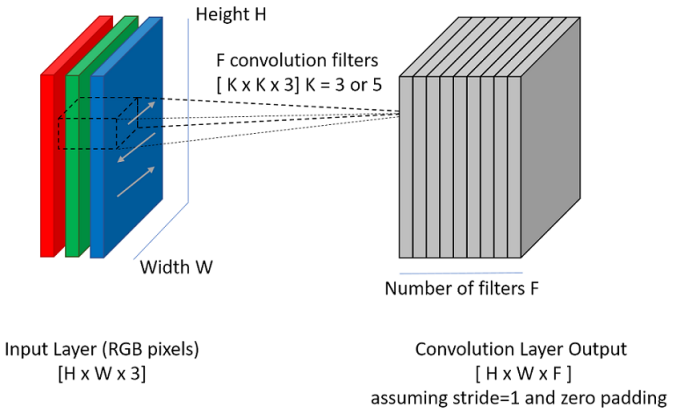


Figure 9.3: The convolution operation for volumes.

Sometimes designers include a **stride** peramater which allows the filters to skip positions as they translate. The default is a stride of 1. Anything higher decreases the size of the output volume by a factor of stride^2.

To summarise, the Convolutional Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyper-parameters:

  Number of filters F,
  their spatial extent K,
  the stride S,
  the amount of zero padding P.

- Produces a volume of size $W_2 \times H_2 \times D_2$ where:

  $W_2 = (W_1 - K + 2P)/S + 1$
  $H_2 = (H_1 - K + 2P)/S + 1$
  $D_2 = F$

- With **parameter sharing** we introduce $K \times K \times D_1$ new parameters per filter, for a total of $K \times K \times D_1 \times F$ parameters and F biases per convolution layer.
- In the output volume, the d^{th} depth slice (of size $W_2 \times H_2$) is the result of performing a convolution of the d^{th} filter over the input volume with a stride of S, and then offset by the d^{th} bias.

9.6 Backpropagation

To complete the convolutional layer it remains to back propagate the gradients.

Now the input is an image volume, $\boldsymbol{X}_{ijd}$, and the parameters are filter volumes, $\boldsymbol{\Theta}_{ijdf}$. The output is an image volume, $\boldsymbol{Z}_{ijf}$.

The gradient is back propagated as a four dimensional array, $\boldsymbol{\delta}_{ijdf}$, one gradient coefficient for each filter parameter. It turns out that the three layer equations all reduce to convolutions as follows:

$$\begin{aligned} \boldsymbol{Z} &= \boldsymbol{X} \otimes \overline{\boldsymbol{\Theta}} \\ \boldsymbol{\delta}^l &= \overline{\boldsymbol{\Theta}} \otimes \boldsymbol{\delta}^{l+1} \\ \frac{\partial E}{\partial \boldsymbol{\Theta}} &= \boldsymbol{X} \otimes \boldsymbol{\delta}^{l+1} \end{aligned} \tag{9.7}$$

A complete derivation of this result can be found in [16].

9.7 Implementation via matrix multiplication

The convolution operation essentially performs dot products between the filters and local regions of the input. A common implementation pattern of the ConvNet layer is to take advantage of this fact and formulate the forward pass of a convolutional layer as one big matrix multiply as follows:

The local regions in the input image are stretched out into columns in an operation commonly called `im2col`. For example, if the input is $227 \times 227 \times 3$ and it is to be convolved with $11 \times 11 \times 3$ filters at stride 4, then we would take $11 \times 11 \times 3$ blocks of pixels in the input and stretch each block into a column vector of size $11 \times 11 \times 3 = 363$.

Iterating this process in the input at stride of 4 gives $(227-11)/4+1 = 55$ locations along both width and height, leading to an output matrix `Xcol` of im2col of size [363 x 3025], where every column is a stretched out receptive field and there are $55 \times 55 = 3025$ of them in total. Note that since the receptive fields overlap, every number in the input volume may be duplicated in multiple distinct columns.

The weights of the ConvNet layer are similarly stretched out into rows. For example, if there are 96 filters of size $11 \times 11 \times 3$ this would give a matrix `Trow` of size 96×363.

The result of a convolution is now equivalent to performing one large matrix

multiply `Trow * Xcol`, which evaluates the dot product between every filter and every receptive field location. In our example, the output of this operation would be 96×3025, giving the output of the dot product of each filter at each location.

The result must finally be reshaped back to its proper output dimension $55 \times 55 \times 96$.

This approach has the downside that it can use a lot of memory, since some values in the input volume are replicated multiple times in `Xcol`. However, the benefit is that there are many very efficient implementations of Matrix Multiplication that we can take advantage of. Moreover, the same `im2col` idea can be reused to perform the pooling operation, which we discuss next.

9.8 Non-Linearity

It is common for convolutional layers to pass their output through a non-linear squashing function before the data reaches the next convolutional layer. Common choices are, `tanh` and relu units. These non-linear units do not have parameters but do affect the back-propagated gradients, see chapter 8.

9.9 maxPooling Layer

It is common to periodically insert a **maxPooling** layer in-between successive convolutional layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the number of parameters in the network, and hence to also control overfitting.

The maxPooling Layer operates independently on every depth slice of the input and resizes it spatially, using the `max` operation. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2 down-samples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every `max` operation would in this case be taking a maximum over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged.

In summary, the pooling layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires two hyper-parameters:

  the spatial extent, K, of the pool

the stride S

- Produces a volume of size $W_2 \times H_2 \times D_2$ where:

$$W_2 = (W_1 - K)/S + 1$$
$$H_2 = (H_1 - K)/S + 1$$
$$D_2 = D_1$$

- Introduces zero new parameters since it computes a fixed function of the input

It is worth noting that there are only two commonly seen variations of the max-Pooling layer found in practice: A pooling layer with $K = 3$ and $S = 2$ (also called overlapping pooling), and more commonly $K = 2$ and $S = 2$, see figure 9.4. Pooling sizes with larger receptive fields are too destructive.

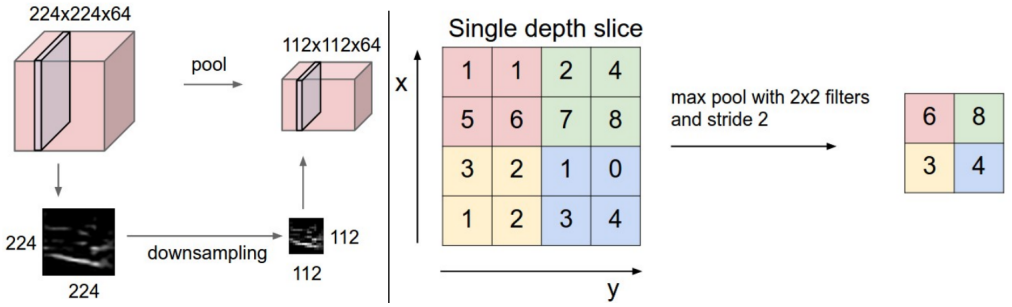


Figure 9.4: The pooling operation to down-sample the images.

9.10 Fully-Connected layer

Finally, after several convolutional and maxPooling layers, the high-level reasoning in the neural network is done via a fully connected layer. A fully connected layer takes all neurons in the previous layer (be it fully connected, pooling, or convolutional) and connects it to every single neuron it has. Fully connected layers are not spatially located anymore (you can visualise them as one-dimensional), so there can be no convolutional layers after a fully connected layer. Usually the last layer has one neuron for each class we are hoping to learn and `softMax` is implemented.

9.11 Convolutional Neural Network Notebook

in appendix A the reader will find a link to a Jupyter notebook that uses the Julia language together with the Flux package to build a digit recogniser for the MNIST dataset adain, this time using a Convolutional Neural Network.

The ConvNet consists of **three** rounds of `conv` plus `maxPool` layers, followed by a fully connected `dense` linear layer to a 10 class `softMax` layer.

9.12 Next Chapter

In the next chapter, we will look at *Recurrent Neural Networks* and how to overcome the vanishing gradient problem through the introduction of the *Long Short Term Memory* unit.

Chapter 10

Recurrent Neural Networks

10.1 Credits

For this chapter we break from Nando de Freitas lecture video series and follow the **Justin Johnson** lecture on RNNs from the Stanford Engineering course, In particular, lecture 10 of 2017, [7].

Material for the section on LSTMs has also been sourced from an article by Christopher Olah, see [13].

10.2 Recurrent Networks

A recurrent neural network (RNN) is a class of artificial neural network where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behaviour. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to tasks such as handwriting recognition or speech recognition.

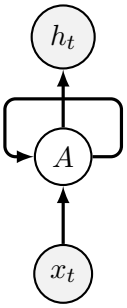


Figure 10.1: Recurrent Neural Network

In figure 10.1, a chunk of neural network looks at some input $\boldsymbol{x}$ and outputs a vector $\boldsymbol{y}$. A loop allows information to be passed from one step of the network to the next.

10.3 Unrolling the recurrence

A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:

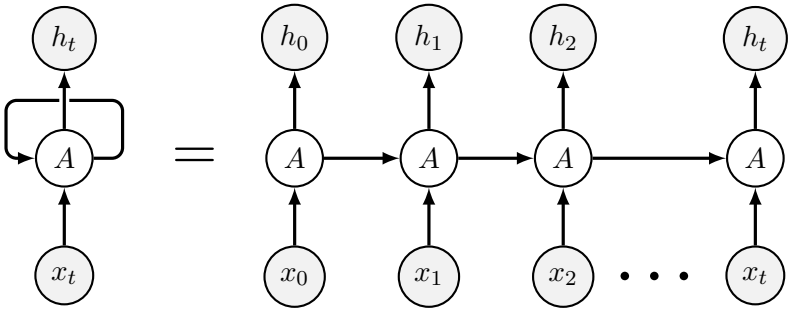


Figure 10.2: Unrolling the RNN as a list of hidden layers.

We can not have RNNs of enormous length because we have to store in memory the hidden state at each time step to be able to back-propagate.

The same function and the same set of parameters are used at very time step! The parameters may be in two parts, one part applies to the previous state and the other to the current input. For example we could have a recursive linear part feeding a squashing function to produce output:

$$\begin{aligned} \boldsymbol{h}_t &= \tanh(\boldsymbol{\Theta}_h \boldsymbol{h}_{t-1} + \boldsymbol{\Theta}_x \boldsymbol{x}_t + \boldsymbol{b}) \\ \boldsymbol{y}_t &= \boldsymbol{\Theta}_y \boldsymbol{h}_t \end{aligned} \tag{10.1}$$

Note that here we use parameter **matrices**:

If $\boldsymbol{h}_{t-1}$ has n elements and $\boldsymbol{x}_t$ has m elements then

$\boldsymbol{\Theta}_h$ is $n \times n$ and $\boldsymbol{\Theta}_x$ is $n \times m$ and $\boldsymbol{b}$ is $n \times 1$

and after passing through the activation function, $\boldsymbol{h}_t$ again has n elements.

also $\boldsymbol{\Theta}_y$ would be $m \times n$ so that input to the next layer has m elements.

Normally we will have RNNs of max length. To process bigger sequences we could divide the sequence in chunks of max length and the last hidden state of a chunk is the initial hidden state of the next chunk.

Many flavours are possible, one to many, many to many etc. We can also stack RNNs together to produce a deeper RNN. It still works the same as before but now we have a weight matrix for each row of hidden layers.

10.4 Vanishing Gradient problem

Traditional activation functions such as `tanh` have gradients in the range $(0, 1)$, and backpropagation computes gradients by the chain rule. With an RNN with long memory this has the effect of multiplying many of these small numbers together to compute gradients over time. This meaning that the gradient decreases exponentially with length of recall.

Mathematically the problem arises because

$$\begin{aligned} \frac{\partial E_t}{\partial \boldsymbol{\Theta}_h} &= \frac{\partial E_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} \frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{h}_{t-2}} \cdots \frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{h}_0} \frac{\partial \boldsymbol{h}_0}{\partial \boldsymbol{\Theta}_h} \\ &= \frac{\partial E_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} \left(\prod_{i=0}^{t-1} \frac{\partial \boldsymbol{h}_{i+1}}{\partial \boldsymbol{h}_i} \right) \frac{\partial \boldsymbol{h}_0}{\partial \boldsymbol{\Theta}_h} \\ &= \frac{\partial E_t}{\partial \boldsymbol{y}_t} \frac{\partial \boldsymbol{y}_t}{\partial \boldsymbol{h}_t} \left(\prod_{i=0}^{t-1} \boldsymbol{\Theta}^T \eta_i \right) \frac{\partial \boldsymbol{h}_0}{\partial \boldsymbol{\Theta}_h} \end{aligned}$$

$$\leq \text{const} \times \eta^t \text{ (since each } \eta_i \text{ is a derivative of tanh and hence } < 1) \text{ .} \tag{10.2}$$

which is very small if $\eta < 1$ and t is large. Note that it does not help to replace the `tanh` activation function with one that has derivatives greater than 1 since then we end up with an **exploding** gradient problem.

The solution is provided by the so called **Long Short Term Memory** unit which we describe in the next section.

10.5 Long Short Term Memory

Normally we will not use Vanilla RNNs, instead, current researchers use **Long Short Term Memory** units or LSTMs. They are similar to RNNs in that they still take into account the input and the last state but now we store two vectors at each time step, the hidden state, $\boldsymbol{h}_t$ and a so called **cell state**, $\boldsymbol{c}_t$.

Moreover in a vanilla RNN the repeating module only has one layer with a `tanh` activation function. whereas with LSTM we have **four** interacting layers with `sigmoid` and `tanh` activations, see figure 10.3.

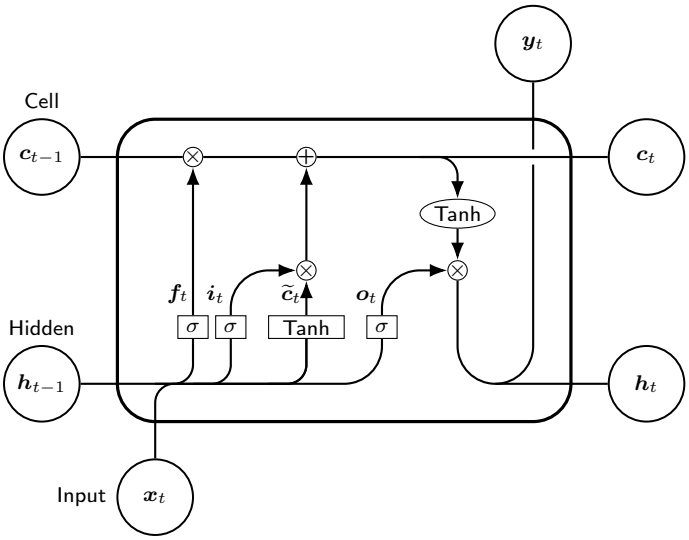


Figure 10.3: LSTM cell

The key to LSTMs is the **cell state**, $\boldsymbol{c}_t$, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight

down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. This cell state, $\boldsymbol{c}_t$, is used to decide which elements of the hidden state to update for output to the next unit.

The LSTM has the ability to remove or add information to the cell state, carefully regulated by structures called **gates**.

Think of gates as boolean variables. As a result of a sigmoid they become differentiable. They allow us to reset and add to the cell state, as well as to choose what elements in the hidden state should be updated. If the cell state was an array of counters we would want to do two operations on them:

- reset them with $\boldsymbol{f}_t$
- add -1 or 1 with $\boldsymbol{i}_t \odot \widetilde{c}_t$

Once the counters were modified we would want to use some of them them to update the hidden state:

- output $\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)$

Step by step

First Decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the **forget gate** layer. It looks at $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$, and outputs a number between 0 and 1 for each number in the cell state $\boldsymbol{c}_{t-1}$.

$$\boldsymbol{f}_t = \sigma(\boldsymbol{\Theta}_f \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f)$$

In the forget gate a 1 means *completely keep this* while a 0 means *completely get rid of this.*

Second Decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the **input gate** layer decides which values we'll update. Next, a `tanh` layer creates a vector of new candidate values, $\widetilde{\boldsymbol{c}}_t$, that could be added to the cell state.

$$\begin{aligned}\boldsymbol{i}_t &= \sigma(\boldsymbol{\Theta}_i \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i) \\ \widetilde{\boldsymbol{c}}_t &= \tanh(\boldsymbol{\Theta}_c \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_c)\end{aligned}$$

Third Update the old cell state, $\boldsymbol{c}_{t-1}$, into the new cell state $\boldsymbol{c}_t$. The previous steps already decided what to do, we just need to actually do it. We multiply the old state by $\boldsymbol{f}_t$, forgetting the things we decided to forget earlier. Then we add $\boldsymbol{i}_t \odot \widetilde{\boldsymbol{c}}_t$. These are the new candidate values, scaled by how much we decided to update each state value.

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \widetilde{\boldsymbol{c}}_t$$

Fourth Finally, we decide what we're going to output. This **output gate** will be based on our cell state, but will be a filtered version. First, we run a `sigmoid` layer which decides what parts of the hidden state we're going to output. Then, we put the cell state through `tanh` (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate.

$$\begin{aligned}\boldsymbol{o}_t &= \sigma(\boldsymbol{\Theta}_o \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o) \\ \boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)\end{aligned}$$

10.6 Variants on LSTMs

Not all LSTMs are the same as the above. In fact, it seems like almost every paper involving LSTMs uses a slightly different version. The differences are minor, but it's worth mentioning some of them. Greff, et al. (2015) do a nice comparison of popular variants, finding that they're all about the same.

For a good alternate discussion of RNNs and LTSMs see `https://goo.gl/XodLUU`.

10.7 The Language Model

To support the Stanford course, Justin Johnson has written a Torch code called `LanguageModel` that he has made available via is `github` account.

This code can **learn** style from a text source and then produce more text in the same style. The code has implementations of both `VanillaRNN` and `LSTM` and one or the other can be chosen via a command line parameter.

The code comes with a sample text, `tiny-shakespeare.txt`, which is used as training data and after training can be sampled to produce Shakespearish like prose.

To test the code, we trained a similar model on some Latin poetry by the Roman poet, Gaius Valerius Catullus 54BC. Here is one of his poems, the ninth,

```
IX. ad Veranium

Verani, omnibus e meis amicis
antistans mihi milibus trecentis,
venistine domum ad tuos penates
fratresque unanimos anumque matrem?
venisti. o mihi nuntii beati!
visam te incolumem audiamque Hiberum
narrantem loca, facta nationes,
ut mos est tuus, applicansque collum
iucundum os oculosque suaviabor.
o quantum est hominum beatiorum,
quid me laetius est beatiusve?
```

After training for about two hours, samples of poetry such as the one below could be generated.

```
LXV. ad Remaeantem

olim aventem ad nihil solet patrium
Quod habet tuum sinire properippora?
ut reddedut gemina perpulacrimis in ventes
medio curo fladi, nec talicos domenia mente coetus
taliunt parditum fugiore aut tarem.
haec corocum subtegminas labore patere,
verum an faciat memores amore.
quos si sua die fuigore quereate monti,
non hoc iter issores malum languentem et Augasiati.
dantis irrumanti amore.
```

10.8 LSTM Notebook

in appendix A the reader will find a link to a Jupyter notebook that uses the Julia language together with the Flux package to build a character based LSTM that learns how to generate Latin poetry.

10.9 Next Chapter

In the next chapter, we will look at *Auto Encoders*.

Chapter 11

Auto-Encoders

11.1 Outline

This chapter introduces you to auto-encoding networks. This chapter is heavily influenced by a series of articles from *Irhum Shafkat* on **auto-encoders** published in [1].

The strategy behind an auto-encoder is to **choke** the network to a low dimensional space and then expand it again using inverse layers and finally use the input data to compute an MSE loss according to

$$\mathscr{L}_{\text{MSE}} = |\boldsymbol{X} - \hat{\boldsymbol{X}}| \tag{11.1}$$

and thus force the network to learn a low dimensional representation of its input. This low dimensional representation is often called the *latent space* of the network.

11.2 The Simple Autoencoder

An auto-encoder consists of 2 components: encoder and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code. See figure 11.1 below.

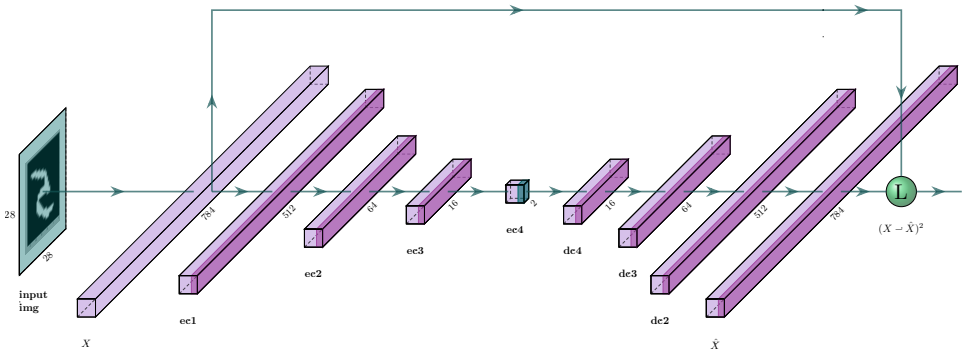


Figure 11.1: A simple auto-encoder for MINST data.

After training the auto-encoder we can feed in sample data into the encoder to be compressed into a two dimensional code and then we can observe what emerges from the decoder.

In fact it is now possible to sample from latent space directly and see how a position in latent space decodes to an output image. In figure 11.2 below we see 10 samples taken from the *latent space class centroids* of the MNIST training data.

Figure 11.2: MNIST decoding from latent space samples

As can be seen the representations of hand drawn, 2, 4 and 5 digits is poor. This is due to the *overlap* of classes in latent space.

The fundamental problem with this simple auto-encoder is that there is no control of the latent space positioning of the code. Sample classes are *smeared* across the 2D plane and do not allow easy classification.

For example, if the class centroids of the training data in latent space are used to classify new data according to nearest centroid in latent space then we do not achieve much better than a 50% accuracy level.

The reason is that classes overlap and do not bunch in latent space, see figure 11.3 below for the positions of the MNIST training samples in latent space.

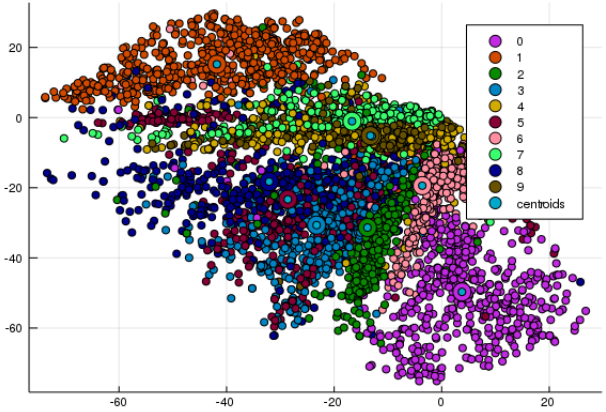


Figure 11.3: 2D latent space for the simple auto-encoder on 5000 MNIST training samples, training accuracy = 55%, testing accuracy = 53%.

In the next section the *Variational Auto-Encoder* is introduced in an attempt to control the positioning of the code in latent space.

11.3 The Variational Auto-Encoder

Variational auto-encoders attempt to control the position of the code in latent space and overcome the non-contiguous latent space problem by introducing an extra part to the code layer.

The VAE achieves this by making its encoder not a single vector of size n, but rather, two vectors of size n: a vector of means, $\boldsymbol{\mu}$, and another vector encoding a diagonal covariavce matrix, $\boldsymbol{\Sigma}$.

These two vectors form the parameters for a gaussian representation of the input from which we **sample** to obtain an encoding which we pass onward to the decoder. This stochastic generation means, that even for the same input, while the mean and standard deviations remain the same, the actual encoding will vary on every pass simply due to sampling.

To prevent the fragmentation of the model's latent space the loss function of the VAE is commonly augmented with an extra term that encourages samples to cluster near the origin. This is achieved using a measure for gaussian distributions known as Kullback-Leibler (KL) divergence.

Given two multivariate Gaussian distributions, $p(x) = \mathcal{N}(\boldsymbol{\mu}_p, \boldsymbol{\Sigma}_p)$ and $q(x) = \mathcal{N}(\boldsymbol{\mu}_q, \boldsymbol{\Sigma}_q)$ the KL divergence between them can be derived as

$$\begin{aligned} \mathrm{KL}(p,q) &= -\int p(x)\log\left[\frac{p(x)}{q(x)}\right] \\ &= \frac{1}{2}\left\{\log\frac{\det(\boldsymbol{\Sigma}_q)}{\det(\boldsymbol{\Sigma}_p)} - d + \mathrm{Tr}(\boldsymbol{\Sigma}_q^{-1}\boldsymbol{\Sigma}_p) + (\boldsymbol{\mu}_p - \boldsymbol{\mu}_q)^T\boldsymbol{\Sigma}_q^{-1}(\boldsymbol{\mu}_p - \boldsymbol{\mu}_q)\right\} \end{aligned} \tag{11.2}$$

A derivation of this result is often set as an exercise to the reader, see [4] for a solution.

To encourage clustering around the origin $q(x)$ is fixed at $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and $p(x)$ is the encoding of the sample as $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Then it turns out that when the encoding is of dimension $d = 2$ with $\boldsymbol{\mu} = [\mu_1, \mu_2]$ and $\boldsymbol{\Sigma} = \begin{bmatrix} \Sigma_{11} & 0 \\ 0 & \Sigma_{22} \end{bmatrix}$ then the contribution to the loss of a KL term for each sample reduces to

$$\mathcal{L}_{\mathrm{KL}} = \frac{1}{2}\left\{-\log(\Sigma_{11}\Sigma_{22}) - 2 + \Sigma_{11} + \Sigma_{22} + \mu_1^2 + \mu_2^2\right\} \tag{11.3}$$

The total loss over a batch is then computed as the sum of the KL loss and the MSE loss. The improved design for a variational auto-encoder is shown in figure 11.4 below.

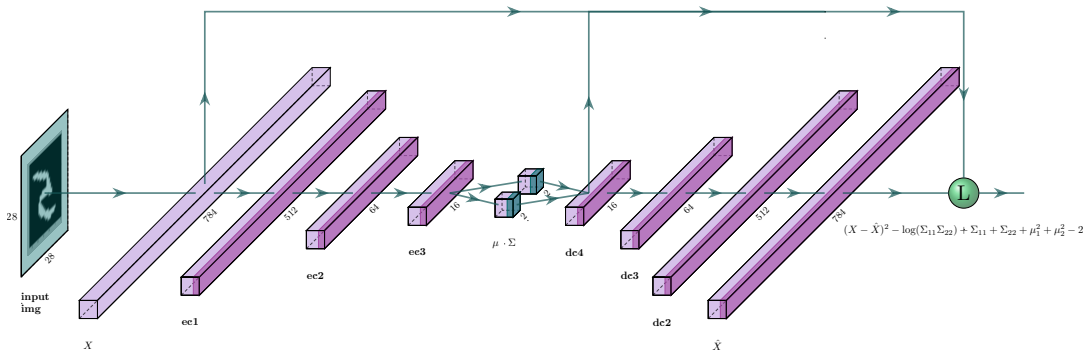


Figure 11.4: A variational auto-encoder for MINST data.

Input samples are now grouped near the origin in latent space, see figure 11.5 below

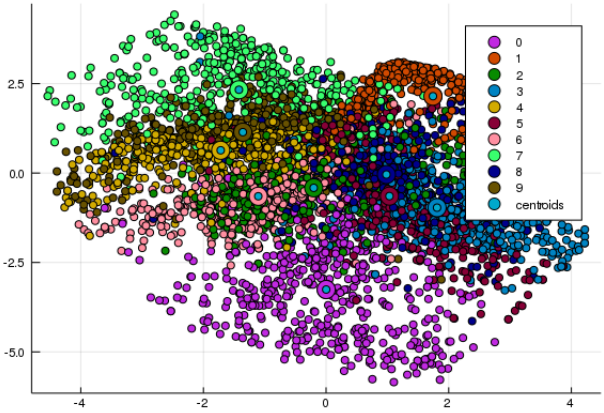


Figure 11.5: The 2D latent space of a variational auto-encoder for 5000 MINST training samples, training accuracy = 56%, testing accuracy = 55%..

The Variational Auto-Encoder still suffers from class overlap in latent space. In the next section we introduce *semi-supervision* to overcome the overlap problem.

11.4 A Semi-Supervised Variational Auto-Encoder

If class labels for samples are available then we can *supervise* the auto-encoder by encouraging samples to cluster near class means in latent space. To do this we first run a variational auto-encoder on training data without considering the training labels and then we compute class means in latent space and then continue running the auto-encoder but this time encourage samples of the same class to cluster near the current class means.

To do this we now choose $\boldsymbol{\mu}_q = \boldsymbol{c}$, the current class centroid for the sample and our KL contribution to the loss now becomes

$$\mathcal{L}_{\text{KL}} = \frac{1}{2}\left\{-\log(\Sigma_{11}\Sigma_{22}) - 2 + \Sigma_{11} + \Sigma_{22} + (\mu_1 - c_1)^2 + (\mu_2 - c_2)^2\right\} \quad (11.4)$$

After each epoch we recompute the class centroids. We then centre them about the origin and we also include a drift term to force some separation between them. In figure 11.6 below we show the resulting latent space for a semi-supervised variational auto-encoder on a training set comprising half the MNIST data set.

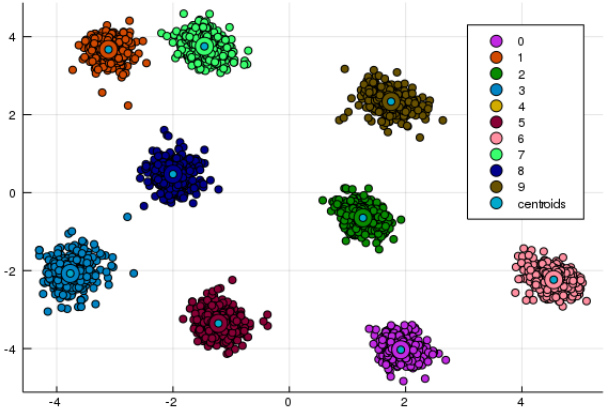


Figure 11.6: The 2D latent space of a semi-supervised variational auto-encoder for 5000 MINST training samples, training accuracy = 99.9%.

The reader can view an animation of the clustering in latent space as learning progresses by visiting [11].

Whenever training occurs in machine learning we must also check the performance of the model on holdout test data. In figure 11.7 below we show clustering of MNIST holdout test data near the class centroids evolved from the training data.

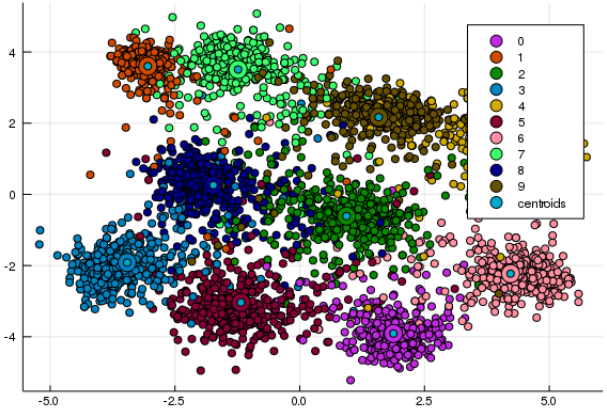


Figure 11.7: The 2D latent space of a semi-supervised variational auto-encoder for 5000 MINST test samples, testing accuracy = 90%.

This semi-supervised model now provides the practitioner with a means of *sampling* hand drawn digits of any class. Given a class to sample we choose a point *near*

the class centroid in latent space and then pass that point through the decoder to produce a sample image. Samples for each of the ten classes are shown in figure 11.8 below.

Figure 11.8: Sample digits drawn from near class centroids in latent space.

We can also generate new MNIST examples by sampling anywhere in latent space and passing the sample to the decoder. Results are shown in the grid in figure 11.9

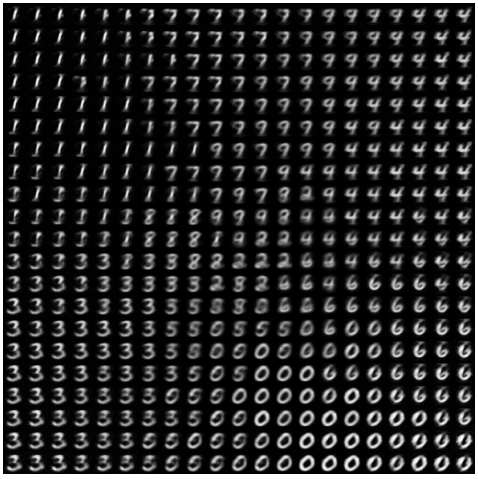


Figure 11.9: Encoding of samples from across the SS-VAE's latent space.

11.5 Autoencoder Notebooks

in appendix A the reader will find a link to a Jupyter notebook that uses the Julia language together with the Flux package to implement the auto-encoders described in this chapter.

Bibliography

[1] Towards Data Science. https:towardsdatascience.com.

[2] Randal J. Barnes. Matrix Calculus. https://atmos.washington.edu/~dennis/MatrixCalculus.pdf, 2006.

[3] Nando de Freitas. Deep Learning Video Lectures. https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/, 2015.

[4] John Duchi. Derivations for linear algebra and optimization. https://web.stanford.edu/~jduchi/projects/general_notes.pdf, 2007.

[5] Alan Edelman. The Julia Academy. https://academy.juliabox.com/, 2018.

[6] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[7] Justin Johnson. Recurrent Neural Networks, lecture 10, 2017. http://cs231n.stanford.edu/2018/.

[8] Jeremy Jordan. Jeremy Jordan Blog. https://www.jeremyjordan.me.

[9] Robert Lucianil. Efficient Neural Network Loss Landscape Generation. https://nextjournal.com/r3tex/loss-landscape, 2019.

[10] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[11] Hugh Murrell. Animation of semi-supervised VAE clustering. https://github.com/HughMurrell/DeepLearningNotes/blob/master/notebooks/julia/animation/vae_ss_ani.gif, 2019.

[12] Hugh Murrell and Nando de Freitas. Deep learning notes using julia with flux. https://HughMurrell.github.io/DeepLearningNotes, 2019.

[13] Christopher Olah. Understanding lstms. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[14] Sebastian Ruder. Optimising Gradient Descent. http://ruder.io/optimizing-gradient-descent/, 2017.

[15] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[16] Pavithra Solai. Back Propagation for CNNs. https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c, 2018.

Appendix A

Julia with Flux

- **Introduction to the Julia programming language** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_01_julia_introduction.ipynb
- **Linear Algebra with Julia** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_02_linear_algebra.ipynb
- **Linear Regression** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_03_linear_regression.ipynb
- **Gaussians and Entropy** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_04_gaussian_entropy.ipynb
- **Basis Function Regression** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_05_basis_function_regression.ipynb
- **Stochastic Gradient Descent** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_06_stochastic_gradient_descent.ipynb
- **Softmax Formulation** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_07_softmax_formulation.ipynb
- **Layered Networks** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_08_layered_network.ipynb
- **Convolutional Networks** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_09_convolutional_network.ipynb
- **Long Short Term Memory** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_10_LSTM_network.ipynb
- **Auto Encoders** https://HughMurrell.github.io/DeepLearningNotes/notebooks/julia/nb_11_SS_VAE_network.ipynb