JONATHON HARE

DIFFERENTIABLE PROGRAMMING AND DEEP LEARNING

BUILDING STRUCTURED LEARNING MACHINES WITH INNATE PRIORS AND INDUCTIVE BIASES

Differentiable Programming and Deep Learning

Building Structured Learning Machines with Innate Priors and Inductive Biases

Jonathon Hare

Differentiable Programming and Deep Learning
by Jonathon Hare

http://github.com/jonhare/differentiable-programming

This printing, July 2023

To my learners and deep-learners.

OK, Deep Learning has outlived its usefulness as a buzz-phrase. Deep Learning est mort. Vive Differentiable Programming!

Yeah, Differentiable Programming is little more than a rebranding of the modern collection of Deep Learning techniques, the same way Deep Learning was a rebranding of the modern incarnations of neural nets with more than two layers. But the important point is that people are now building a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization.

An increasingly large number of people are defining the networks procedurally in a data-dependent way (with loops and conditionals), allowing them to change dynamically as a function of the input data fed to them. It's really very much like a regular program, except it's parameterized, automatically differentiated, and trainable/optimizable. Dynamic networks have become increasingly popular (particularly for NLP), thanks to deep learning frameworks that can handle them...

Important note: this won't be sufficient to take us to "true" AI. Other concepts will be needed for that, such as what I used to call predictive learning and now decided to call Imputative Learning. More on this later...

Yann LeCun, January 2018

Contents

Preface

"Why another book about deep learning?" my colleague asked. It's a good question... Current textbooks focus on the *what* and *how* of deep learning rather than the *why*. There are numerous books that will tell you about how you can use a convolutional network to solve an image classification problem, and how you can add residual blocks to improve the performance, but few go into the true rationale for the use of convolutions or skip connections. At the same time deep learning research is moving fast and models are moving away from straight forward use of simple stacks of linear, convolutional and recurrent layers towards far more dynamic differentiable programs. These new programs necessarily incorporate many structural & innate priors and inductive biases to help the model learn and leverage the data in particular ways (to for example achieve disentanglement).

There is room for debate about how much a model should learn versus to what extent a model should be engineered, however, at the same time all of the building blocks we currently use have some form of inherent bias or innate prior within them. These biases and priors are prevalent throughout the learning machinery, from biases in the data, to biases in the model to inherent biases in the learning algorithm. My motivation when writing these words was to exactly try and address these issues by describing not only how particular models work, but when and why you might make particular design choices, and what the implication of those choices might be.

My target audience was advanced undergraduates and postgraduates, who have already studied basic machine learning, and understand the basic ideas of classification and regression. Whilst the early parts of the book provide something of a refresher in this area the scope is necessarily limited, and my aim was not to compete with the classic machine learning textbooks by Chris Bishop or David Mackay for example. I have strived to highlight right from the beginning how it is possible to construct learning machines that perform tasks that do not necessarily immediately fall into those covered by the realm of classical machine learning and statistical learning, as well as how our new machines need not be static functional mappings, but can incorporate their own dynamics.

The underlying mathematics of deep learning models and more general differentiable programs, and the approaches to optimising them, is a common theme of this book. The reader is advised that there is a lot of mathematics involved, however I have strived throughout to try to give the intuitive explanations of what that mathematics means or tells us. Calculus, particularly differentiation, clearly plays a big role throughout the text, and I have endeavored

to give this an in-depth coverage, including looking at the inner details of how the three main approaches to computing derivatives and gradients (numeric, symbolic and automatic), as well as stochastic gradient estimators, play a role in the wider world of differentiable programming.

The overall contents of this book came about from my lectures and notes from the Differentiable Programming and Deep Learning module I teach to Masters students. To a great extent, the chapters follow along with the order in which I lecture, however they of course go into far greater depth than is possible in a forty-five minute talk. In terms of structure, I've broken the topics into three parts; the first is all about the foundational building blocks of differentiable programming, and focuses on the underlying mathematics, algorithmic and computational techniques. The second part looks at contemporary deep learning architectures with a view to understanding why they are structured as they are. The third part looks at more advanced topics where structural biases play a key role in the building blocks of models, and looks to the future where such features will play a critical role in the next generation of learning AIs towards neuro-symbolic agents.

Acknowledgements

Nomenclature

Numbers and Arrays (scalars, vectors, matrices and tensors)

a	A scalar (real or integer).
$\boldsymbol{a}$	A vector.
$\boldsymbol{A}$	A matrix.
$\mathbf{A}$	A tensor.
$\boldsymbol{I}$	Identity matrix.
$\boldsymbol{I}_n$	$n \times n$ identity matrix.
$\boldsymbol{e}_i$	Standard basis vector $[0, \dots, 1, \dots, 0]$ with the 1 at the i-th position and zeros everywhere else.
a	Scalar random variable.
$\mathbf{a}$	Vector-valued random variable.
$\mathbf{A}$	Matrix-valued random variable.
$\mathbf{0}$	Vector, matrix or tensor of zeros (depending on context).
$\mathbf{1}$	Vector, matrix or tensor of ones (depending on context).
$a + bi$	Complex number made up of the tuple (a, b) with real part a and imaginary part b following the rule $(a, b) \cdot (c, d) = (ac - bd, ad + bc)$. The imaginary unit i is defined as $i^2 = -1$.
$a + b\epsilon$	Dual number made up of the tuple (a, b) and following the rule $(a, b) \cdot (c, d) = (ac, ad + bc)$. The symbol ϵ is taken to be an infinitesimally small, but non-zero, value that satisfies $\epsilon^2 = 0$.

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1.

$A_{i,j}$	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 $\mathbf{A}$.
$\mathbf{A}_{:,:,i}$	2-D slice of a 3-D tensor.
a_i	Element i of the random vector $\mathbf{a}$.

Sets, ranges, intervals and tuples

$\mathbb{A}$	A set.
$\mathbb{R}$	The set of all real numbers.
$\mathbb{Z}$	The set of all integer numbers.
$\mathbb{C}$	The set of all complex numbers.
$\mathbb{D}$	The set of all dual numbers.
$\{0, 1\}$	The set containing 0 and 1.
$\{0, \dots, n\}$	The set of all integers between 0 and n inclusive.
$[a, b]$	The *closed* real interval between a and b inclusive.
$[a, b)$	The *half-open* real interval between a inclusive and b exclusive.
$(a_1, \dots, a_n)$	The tuple or ordered sequence of elements a_i to a_n. Equivalent to the column vector $\begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix}^\top$

Functions

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$. f maps an input taken from the set $\mathbb{A}$ to a element from the set $\mathbb{B}$.
$f(\dots) \stackrel{\text{def}}{=} \dots$	Function $f(\dots)$ is defined to be equal to ...
$f(\boldsymbol{x}; \boldsymbol{\theta})$ or $f_{\boldsymbol{\theta}}(\boldsymbol{x})$	A function of $\boldsymbol{x}$ parameterised by $\boldsymbol{\theta}$. Sometimes we will write $f(\boldsymbol{x})$, omitting $\boldsymbol{\theta}$ to lighten notation.
$f \circ g$	Binary function composition of f and g. Equivalent to writing $f(g(\dots))$.
$\log(x)$	Natural logarithm of x.
$\mathrm{ReLU}(x)$	Rectified Linear Unit activation, $\max(x, 0)$.

$\text{logistic}(x)$	The logistic function, $\frac{1}{1+\exp(-x)}$. Sometimes referred to as the sigmoid function.
$\text{softplus}(x)$	Softplus, $\log(1+\exp(x))$.
$\|\boldsymbol{x}\|_p$	ℓ^p norm of $\boldsymbol{x}$.
$\|\boldsymbol{x}\|$ or $\|\boldsymbol{x}\|_2$	ℓ^2 norm of $\boldsymbol{x}$.
$[\boldsymbol{x};\boldsymbol{y}]$	Concatenation of two vectors into a larger vector; can be extended to more than two vectors, e.g. $[\boldsymbol{x};\boldsymbol{y};\boldsymbol{z}]$.

Linear Algebraic Operations

$\boldsymbol{A}^\top$	Transpose of $\boldsymbol{A}$.
$\boldsymbol{A}^*$	Conjugate (or Hermitian) transpose of $\boldsymbol{A}$. If $\boldsymbol{A}$ is real, then $\boldsymbol{A}^* = \boldsymbol{A}^\top$.
$\boldsymbol{A}^{-1}$	Inverse of $\boldsymbol{A}$.
$\boldsymbol{A}^+$	Moore-Penrose inverse (pseudoinverse) of $\boldsymbol{A}$.
$\boldsymbol{A} \odot \boldsymbol{B}$	Hadamard product. Element-wise product of $\boldsymbol{A}$ and $\boldsymbol{B}$.
$\det(\boldsymbol{A})$	Determinant of $\boldsymbol{A}$.

Calculus

$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$	Leibniz's notation for the first derivative of $f(x)$ with respect to x.
$\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2}$	Leibniz's notation for the second derivative of $f(x)$ with respect to x.
f'	Langrange's notation for the first derivative of f. If f is a function of a single variable, then f' is the derivative with respect to that variable and f'' represents the second derivative.
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x.
$\nabla_x y$	Gradient (vector) of y with respect to $\boldsymbol{x}$.
$\nabla_{\boldsymbol{X}} y$	Matrix containing derivatives of y with respect to $\boldsymbol{X}$.
$\nabla_{\mathbf{X}} y$	Tensor containing derivatives of y with respect to $\mathbf{X}$.
$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m\times n}$ of $f : \mathbb{R}^n \to \mathbb{R}^m$.
$\nabla_{\boldsymbol{x}}^2 f(\boldsymbol{x})$	The Hessian matrix of f at input point $\boldsymbol{x}$.

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 $\boldsymbol{X}_{i,:}$

Standard Formulae

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

The Taylor series of f at input point x. $f^{(n)}$ denotes the n-th derivative with $f^{(0)}$ defined as f, and both 0! and $(x-a)^0$ are defined as being 1.

1. Introduction

In recent years the fields of machine learning, computer vision, natural language processing, and others, have been completely changed by the deep learning revolution. This revolution was kick-started in 2012 when a Convolutional Neural Network[1] known as AlexNet demonstrated a massive improvement in performance over contemporary techniques in a problem of image classification. Since AlexNet, all sorts of problems have been tackled with deep learning and representation learning. So why then did Yann LeCun pronounce the death of deep learning in 2018, and propose that we should instead be talking about *Differentiable Programming* instead?

[1] A convolutional neural network is a neural network built of *convolutional layers* which have a number of interesting properties as we will see in chapter 11.

The point Yann was trying to make is that we can now do more than just build traditional neural networks and learning machines... We can build complex software systems from differentiable functional blocks (which may or may not have learnable parameters) and assemble these into complex software systems that can be "trained" using gradient-based optimisation. As with traditional procedural programming, constructs such as loops and conditionals can be used inside differentiable programs to enable them to behave in a data dependent way, and change dynamically as a function of their input.

So, we now have differentiable programming instead of deep learning, and it allows us to *do more*, or perhaps *be more flexible* with the models/software that we create, but this then poses another interesting question: how do we compose the building blocks of our software to achieve the goal we have in mind? This is where we need to start thinking about the structure we want to impose in our software or learning machines through innate priors and inductive biases. These are not new concepts; Yann's original convolutional neural network design used the innate prior of the convolution operation (or more correctly cross-correlation in most modern implementations) to directly impose

translational equivariance[2] onto part of a model and a *pooling* operation to achieve a degree of translational invariance. This network was designed to recognise hand written characters in images, so it was a desirable property that the network would be *invariant* to the exact position of the character in the image.

This introduction aims to show how all the building blocks of differentiable programming fit together without going into too much detail. Much of the content will be a refresher of pre-University topics such as basic calculus. You will probably encounter some ideas that you have not seen or thought of before too however. Almost all of the things we highlight in this introduction are covered in much greater detail at later points in the book, and the author has endeavored to provide cross-references to help you find the corresponding detailed discussion.

[2] Equivariance and invariance are properties of certain mathematical functions. A function that is invariant to a certain class of transformation produces the same output when given an input as it does to a transformed version of that same input. Equivariance is a property of functions that means that if the input is transformed, then the output is also transformed in the same way. We will define these properties more formally in **??**.

1.1. Differentiation

You probably originally studied calculus before University. How much can you remember beyond sets of formulas[3] like $\frac{\mathrm{d}}{\mathrm{d}x}x^n = nx^{n-1}$ or $\frac{\mathrm{d}}{\mathrm{d}x}\sin(x) = \cos(x)$? We will start by refreshing your memory of what a derivative of a function is, how it can be computed and how it can actually be used to solve a practical problem.

[3] The $\frac{\mathrm{d}}{\mathrm{d}x}$ notation is called Leibniz's notation; there are several other notations including Lagrange's or 'prime', Newton's and Euler's.

1.1.1. Recap: what is the derivative of a univariate[4] function?

[4] A function of one variable.

Firstly, recall that the slope or gradient of a straight line is $\frac{\mathrm{d}y}{\mathrm{d}x}$; that is it is the ratio of the change in the y-direction to the change in the x-direction. This is illustrated in fig. 1.1. Because the line is straight, you can pick *any point that is not on the line* and measure the horizontal and vertical distances $\mathrm{d}x$ and $\mathrm{d}y$ to the line, and the computed gradient will be the same.

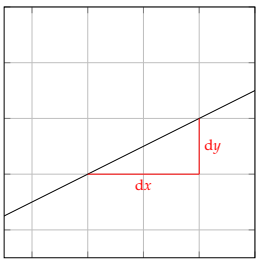


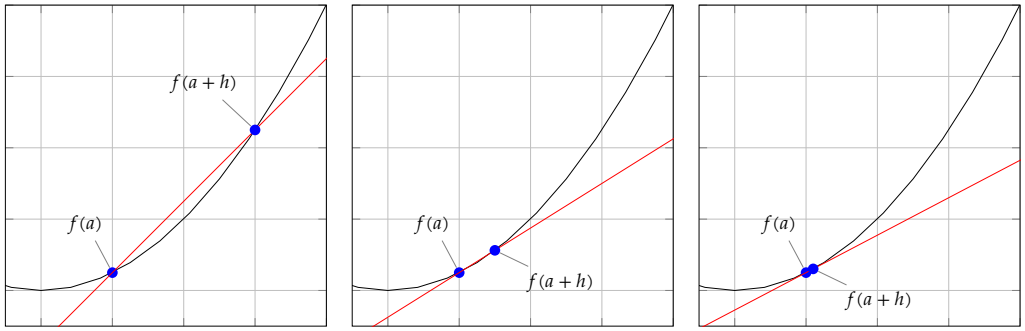
Figure 1.1: Computing the gradient of a straight line.

For an arbitrary real-valued univariate function, $f(a)$, we can approximate the derivative[5] $f'(a)$ using the gradient of the *secant line* defined by a point $(a, f(a))$ and a point $(a+h, f(a+h))$ a small distance h away from a, as illustrated in fig. 1.3. The gradient of this secant line is

$$f'(a) \approx \frac{f(a+h) - f(a)}{h} \,. \tag{1.1}$$

This expression is sometimes known as *Fermat's Difference Quotient* or *Newton's Quotient*. As h becomes smaller, the approxi-

[5] This is Langrange's notation; $f'(x) \equiv \frac{\mathrm{d}}{\mathrm{d}x}f(x)$.



(a) $y = x$ (b) $y = x$ (c) $y = x$

Figure 1.2: Approximating the gradient of a curve f at a using the secant line between $(a, f(a))$ and $(a + h, f(a + h))$. As h decreases in the limit towards zero (figures (a)-(c)) the approximation becomes more accurate and the secant line approaches the instantaneous tangent to the curve at a.

mated derivative becomes more accurate. If we take the limit as h *tends to* 0 (written as $h \to 0$), then we have an exact expression for the derivative,

$$\frac{\mathrm{d}f}{\mathrm{d}a} \equiv f'(a) = \lim_{h\to 0} \frac{f(a+h) - f(a)}{h} \,. \tag{1.2}$$

This derivative represents the slope of the instantaneous tangent to the curve f at a.

1.1.2. The derivative of simple expressions from first principles

The expression for the derivative in eq. (1.2) gives us the basis for computing the derivative of any arbitrary expression. For example, we can compute the derivative of $y = x^2$ as follows,

$$\begin{aligned}
\frac{\mathrm{d}y}{\mathrm{d}x} &= \lim_{h\to 0} \frac{(x+h)^2 - x^2}{h} \\
&= \lim_{h\to 0} \frac{x^2 + h^2 + 2hx - x^2}{h} \\
&= \lim_{h\to 0} \frac{h^2 + 2hx}{h} \\
&= \lim_{h\to 0} (h + 2x) \\
&= 2x \,.
\end{aligned}$$

Going back to those formulas that you might remember from when you first encountered differentiation, we can prove that

$\frac{\mathrm{d}}{\mathrm{d}x}x^n = nx^{n-1}$ for positive integer n by applying the same reasoning to $y = x^n$ and exploiting the binomial theorem[6]:

[6] With a few extra tricks involving *implicit differentiation* and the *derivatives of logarithms* one can prove the same holds for any real n.

.

$$\begin{aligned}
\frac{\mathrm{d}y}{\mathrm{d}x} &= \lim_{h\to 0}\frac{(x+h)^n - x^n}{h} \\
&= \lim_{h\to 0}\frac{x^n + nx^{n-1}h + \cdots + h^n - x^n}{h} \\
&= \lim_{h\to 0}\frac{nx^{n-1}h + O(h^2)}{h} \\
&= \lim_{h\to 0} nx^{n-1} + O(h) \\
&= nx^{n-1}
\end{aligned}$$

□

Similarly, we can prove that $\frac{\mathrm{d}}{\mathrm{d}x}\sin(x) = \cos(x)$ by utilising the trigonometric identity

$$\sin A - \sin B = 2\cos\frac{(A+B)}{2}\sin\frac{(A-B)}{2}$$

and noting[7] that $\lim_{x\to 0}\frac{\sin x}{x} = 1$. Taking $y = \sin x$ and starting as before with eq. (1.2),

[7] See exercise 1.1 to prove this for yourself.

.

$$\begin{aligned}
\frac{\mathrm{d}y}{\mathrm{d}x} &= \lim_{h\to 0}\frac{\sin(x+h) - \sin x}{h} \\
&= \lim_{h\to 0}\frac{2}{h}\cos\left(\frac{x+h+x}{2}\right)\sin\left(\frac{x-h+x}{2}\right) \\
&= \lim_{h\to 0}\frac{2}{h}\cos\left(\frac{2x+h}{2}\right)\sin\left(\frac{h}{2}\right) \\
&= \lim_{h\to 0}\cos\left(\frac{2x+h}{2}\right)\cdot\lim_{h\to 0}\frac{2}{h}\sin\left(\frac{h}{2}\right) \\
&= \lim_{h\to 0}\cos\left(\frac{2x+h}{2}\right)\cdot\lim_{\frac{h}{2}\to 0}\sin\left(\frac{h}{2}\right)\Big/\frac{h}{2} \\
&= \lim_{h\to 0}\cos\left(\frac{2x+h}{2}\right) \\
&= \cos x
\end{aligned}$$

□

1.1.3. Intuition: What does the derivative $\mathrm{d}y/\mathrm{d}x$ tell us

If your math teachers were like mine, then you were probably taught to think about the derivative of a function as being a

rate of change of that function. This is an excellent intuition for thinking about how for example how to relate position, velocity (the *rate of change of position*) and acceleration (the *rate of change of velocity*). This could well have been one of the first motivations for you using differentiation when your were first learning calculus. Formalising, let $\boldsymbol{r}(t)$ be some function that describes the position of an object at time some time t. Then, the instantaneous velocity $\boldsymbol{v}(t)$ of that object at time t is given by

$$\boldsymbol{v}(t) = \frac{\mathrm{d}\boldsymbol{r}(t)}{\mathrm{d}t} .$$

The acceleration, $\boldsymbol{a}(t)$, is given by the derivative of velocity with respect to time, or the *second derivative* of position with respect to time,

$$\boldsymbol{a}(t) = \frac{\mathrm{d}\boldsymbol{v}(t)}{\mathrm{d}t} = \frac{\mathrm{d}^2\boldsymbol{r}(t)}{\mathrm{d}t^2} .$$

Higher order derivatives are of course possible; for example jerk $\boldsymbol{j}(t)$ is given by the third derivative of position

$$\boldsymbol{j}(t) = \frac{\mathrm{d}\boldsymbol{a}(t)}{\mathrm{d}t} = \frac{\mathrm{d}^2\boldsymbol{v}(t)}{\mathrm{d}t^2} = \frac{\mathrm{d}^3\boldsymbol{r}(t)}{\mathrm{d}t^3} .$$

The reverse viewpoint. This rate of change perspective of what a derivative tells us is eminently useful for many problems. However, there is another way of thinking about what the derivative tells us which fits better with how we will use derivatives in differentiable programs and learning machines. In this alternative viewpoint we can think of the derivative $\mathrm{d}y/\,\mathrm{d}x$ as telling us by *how much* the y changes if we make a small change to the x.

To put this more into context, lets consider a supervised learning problem where we have a dataset $\mathbb{X}$ of N data pairs $\mathbb{X} = \{\boldsymbol{x}, \boldsymbol{y}\}_{n=1}^{N}$, and our objective is to learn some parameterised function f that makes a prediction $\hat{\boldsymbol{y}}$ given $\boldsymbol{x}$. We can write this as

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \theta) ,$$

where θ represents the parameter that controls how the predictions are made. Our learning problem is to find the value of θ that would *minimise* some error function $\mathcal{E}(\mathbb{X}, f)$ which is computed between all the $\boldsymbol{y}$'s in the dataset and the predictions $\hat{\boldsymbol{y}}$ for the corresponding $\boldsymbol{x}$'s. This error function could take many different forms[8] later, however at this stage it should be obvious that $\mathcal{E}$ depends on f and is thus dependent on the value of θ. Going

[8] We will encounter many of these throughout this book, but you might already know some of the broad ideas of empirical risk and notions of objective 'loss' functions like the mean squared error loss.

back to this reverse viewpoint of differentiation, we can think of the derivative $\mathrm{d}\mathcal{E}/\,\mathrm{d}\theta$ as telling us how much (and in which direction) the error will change if we make a small change in the value of θ. Not only can this tell us which direction we should adjust θ to reduce the error, but it also tells us about the sensitivity of the error to changes in θ.

1.1.4. Solving a simple problem with differentiation

Now we have an intuition for what the derivative means, lets turn our attention to solving a real-world problem by using differentiation. We will first solve this problem analytically because it is simple enough to have a closed form solution (indeed you might actually remember the answer from having attempted a similar question in the past). We will then work through the steps required to derive an algorithm that can numerically compute the solution, and implement that algorithm as a computer program.

We're going to ask *at what angle should a javelin be thrown to maximise the distance travelled?* We will assume that the initial speed is $u\,\mathrm{m\,s^{-1}}$ and that gravity acts with an acceleration $g\,\mathrm{m\,s^{-2}}$ towards the Earth. To simplify our model we will choose to ignore the launch height as it will be negligible compared to distance travelled, and also ignore the effect of any air resistance. With these assumptions in place, the kinematics equations that define the motion[9] of the javelin are

$$\begin{aligned} x &= ut\cos(\theta) \\ y &= ut\sin(\theta) - 0.5gt^2\ , \end{aligned}$$

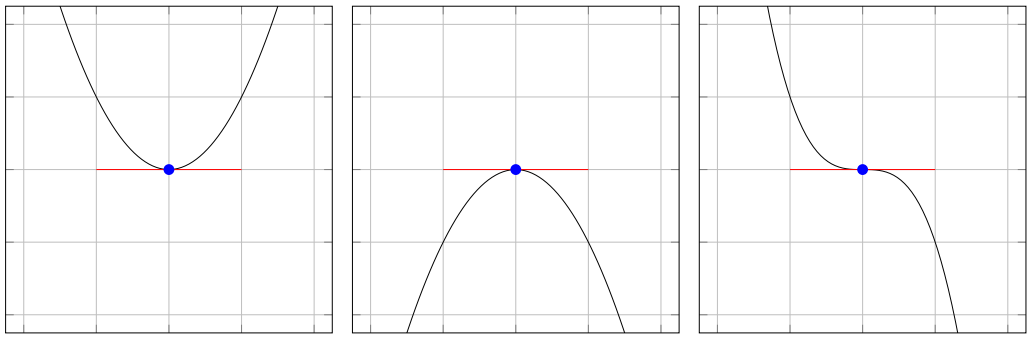
where the position of the javelin at time t is given by (x, y). Now, clearly the javelin hits ground when $y = 0$ and we only care about $t > 0$, so,

$$\begin{aligned} 0 &= ut\sin(\theta) - \frac{g}{2}t^2 \\ \implies t &= \frac{2u}{g}\sin(\theta)\ . \end{aligned}$$

Substituting this into the horizontal component and noting that $2\sin(\theta)\cos(\theta) = \sin(2\theta)$ then gives

$$\begin{aligned} x &= u\frac{2u}{g}\sin(\theta)\cos(\theta) \\ &= \frac{u^2}{g}\sin(2\theta)\ . \end{aligned}$$

[9] If these equations don't look familiar don't worry. The first just says that the javelin moves at a constant velocity horizontally in the absence of any air resistance (Newton's first law), and the second one says there is a continual vertical downward acceleration acting on the javelin due to the force of gravity (Newton's second law).



(a) minima (b) maxima (c) saddle point

Figure 1.3: Illustrations of local minima, maxima and saddle points in one dimension. At all these points the derivative of the sketched function is zero.

Now, we're in a position to solve an optimisation problem which we can write formally as

$$\begin{aligned} \arg\max_{\theta} \quad & \frac{u^2}{g}\sin(2\theta) \\ \text{s.t.} \quad & 0 \leq \theta \leq \frac{\pi}{2}\,. \end{aligned}$$

The arg max here tells us that we're looking for the value of θ that maximises the distance, rather than the distance itself (for which we would have written max). The constraint (s.t., 'such that') on the above serves to limit the range of possible directions in which the javelin is thrown to being from perfectly horizontal ($0\,\text{rad}$ or $0°$) to vertical ($\pi/2\,\text{rad}$ or $90°$). Now recall that the *stationary points* (the *minima*, *maxima* and *saddle points*) of a function are points where the derivative of that function are zero. In general a function might have any number of stationary points, but differentiation can always be used to find them by computing the derivative and setting the result to zero. Our function for x above actually has an infinite number of minima and maxima (its fundamentally just a sine-wave oscillating up and down), but no saddle points, however in the domain $0 \leq \theta \leq \frac{\pi}{2}$ there is just one maxima (which we can verify by looking at the sign of the second derivative). Going back to the javelin problem, we thus need to

compute the derivative with respect to θ and set it to zero:

$$\begin{aligned} 0 &= \frac{\mathrm{d}}{\mathrm{d}\theta}\left(\frac{u^2}{g}\sin(2\theta)\right) \\ &= \frac{2u^2}{g}\cos(2\theta) \\ &= \cos(2\theta) \end{aligned}$$

Clearly $\theta = \frac{\pi}{4}(2n-1)$ where $n \in \mathbb{Z}$, however for θ between 0 rad and $\frac{\pi}{2}$ rad, $\theta = \frac{\pi}{4}$ rad. This in turn implies that *irrespective of the initial velocity of the javelin, maximum distance is achieved when it is launched at* 45°.

As an aside, this is actually quiet an interesting result that applies to any object being thrown or fired: the maximal distance *at the same level* is always achieved when the angle is 45°. Now in the physical world some of the modelling assumptions, particularly surrounding air resistance or drag[10] does have some effect and would reduce this angle. Our assumption that the launch height did not matter does in fact also have a bigger effect as you can see for yourself in exercise 1.2. Of course there are also additional factors related to the physiology of the athlete and environmental factors such as the wind that also have an effect and mean that the typical angle is somewhere around 32 ° to 36 °.

[10] For reasonable speeds below the speed of sound drag is equal to the square of the speed multiplied by the frontal-area of the projectile multiplied by the coefficient of drag. A javelin has a relatively tiny frontal area and low coefficient of drag, so the effect is very small.

1.1.5. Abstraction: Solving problems by minimising an objective

Before we look at constructing an algorithm and software implementation for solving our javelin problem, we will first take a step back and look at what we might abstract from how we solved this problem. To compute the *parameter* (the angle θ) for the javelin example we *maximised a function* representing the equation for distance travelled. We can solve all kinds of problems using a similar approach if we can (1) **formulate** an objective function (often referred to as a *loss* or *cost* function), and (2) **minimise**[11] the objective function with respect to the parameter(s).

[11] In learning problems its usually the convention that we look to minimise a loss function. In the javelin example we maximised a function, however maximisation and minimisation are trivially swapped by negating the function. In other words, maximising a distance is the same as minimising a negative distance.

This notion of minimising a objective function to find solutions is rather general and can be applied to many different types of problem. However, there are several potential issues that we need to consider:

1. The objective function might not be of a single parameter, but rather have many[12] parameters that need to be adjusted to

[12] Deep networks often have many millions of parameters that need to be optimised.

minimise the function. In such a case we can use the partial derivatives of the objective with respect to each parameter. To find a minima (there might be more than one) we could hypothetically solve the set of simultaneous equations that results from setting all of these partial derivatives to zero, although this might not be practical.

2. Clearly the objective function must be differentiable, or rather, you must be able to either compute or estimate the partial derivatives with respect to each of its parameters somehow. This restricts the flexibility of our functions[13].

3. It would in general be unreasonable to assume that in general our objective functions would have only a single minima. In reality they might have many local minima, often with the same minima repeated under permutations of the function's parameters. For example it is easy to conceive of a function such as $f(\theta_1, \theta_2) = \sin(\theta_1)\sin(\theta_2)$ that has a local minima of the same value under permutations of the ordering of the parameters — that is $\theta_1 = -\frac{\pi}{2}, \theta_2 = \frac{\pi}{2}$ and $\theta_1 = \frac{\pi}{2}, \theta_2 = -\frac{\pi}{2}$ are both local minima with $f(-\frac{\pi}{2}, \frac{\pi}{2}) = f(\frac{\pi}{2}, -\frac{\pi}{2}) = -1$).

4. In a function with many local minima it might be computationally intractable to find a globally optimal one (if one of more such minima even exist). Further, this intractability might mean that we might end up finding a saddle point rather than a (local) minima.

5. The objective function itself could be arbitrarily complex (that is up to you as creator of the function). In many cases it might be difficult to analytically compute the derivatives, or analytically computing the partial derivatives might be intractable.

[13] We look more formally at these restrictions in **??** where we consider the need for functions to be *continuous* and *differentiable almost everywhere* to behave well within our computational machinery, and look at the notions of *relaxations* that can allow approximations to functions that don't have these properties to be made.

1.1.6. Gradient Decent: A simple algorithm for minimising an objective function

The list of potential issues above highlights that in practice finding a global minima might be hard, and that for complex objective functions even writing down the analytic derivatives might be practically impossible. If we are however happy to live with the possibility that we might not find the best solution (or even an actual local minima, or all the possible solutions) then we could seek a numerical solution that find a set of parameters that have (close to) zero derivatives. The advantages of this will be that we

do not need to derive a closed form solution for the derivatives of the objective function[14], or solve a set of simultaneous equations.

[14] We will still need to be able to compute or estimate the local derivatives of the function about a particular set of parameters however.

Formulating a simple algorithm to numerically find the parameters that locally minimise a objective turns out to be rather intuitive, and has a simple physical world analogy. Imagine that you are out walking and stood on the hillside of a smooth *quadratic* valley as illustrated in **??**. What is the fastest way to the bottom? You simply walk down the hill in the most direct way possible; after every step you take you can choose the next step by evaluating the direction that will enable you to lose the most height. Eventually you will either collapse of exhaustion, or reach a point where any direction you can move in results in you gaining height.

We can formalise this idea as a simple iterative optimisation algorithm known as *gradient descent*. Given an scalar objective function $\ell(\theta)$ and an initial guess θ_0, at every step we update the parameter(s) θ by an amount directly proportional to the negative derivative[15],

$$\theta_{i+1} = \theta_i - \alpha \frac{\mathrm{d}\ell}{\mathrm{d}\theta} \tag{1.3}$$

[15] We're just considering a single scalar parameter here so we can just talk about the derivative with respect to that parameter. When we deal with many parameters (see **??**), we will define the algorithm in terms of the vector of partial derivatives of each parameter $\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{\theta})$, which is known as the *gradient* (hence why the algorithm is called "gradient descent").

where α is the *learning rate*. With a suitable value of α and with a *well-behaved* function ℓ, eq. (1.3) will converge to either a local minima or saddle point as illustrated in fig. 1.4. We can choose to terminate the algorithm in a number of ways such as when the gradient *is close* to zero (or equivalently there is little difference between $\ell(\theta_{i+1})$ and $\ell(\theta_i)$, e.g., $|\ell(\theta_{i+1} - \ell(\theta_i| < \epsilon)$. In machine learning problems we often set a maximum number of iterations (sometimes in combination with the gradient test on the training loss, or analysis of the validation generalisation error which is the magnitude of the difference between the training and validation loss).

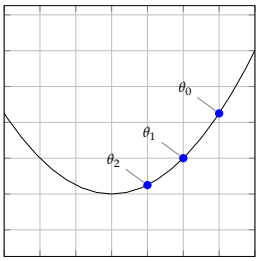


Figure 1.4: Gradient descent iteratively finds the parameter θ that minimises a function.

1.1.7. Javelin throwing again, but with Python code

Now we have a technique that will iteratively allow us to numerically estimate the parameter(s) that minimise a function, we construct a software implementation of the algorithm. We will return back to our original javelin problem where we have

already derived an expression for the derivative,

$$\frac{\mathrm{d}}{\mathrm{d}\theta}\left(\frac{u^2}{g}\sin(2\theta)\right) = \frac{2u^2}{g}\cos(2\theta)\,,$$

and construct a `python` implementation as follows:

```
from math import sin, cos

theta = 0.1    # our initial guess, theta_0
gamma = 0.001  # learning rate
u = 28 # 28 m/s initial velocity
g = 9.8 # constant 9.8 m/s^2 acceleration towards the
    ground

distance = u**2 * sin(2 * theta) / g
print(theta, distance)

for i in range(100):
  dtheta = 2 * u**2 * cos(2 * theta) / g
  theta = theta + gamma * dtheta

  distance = u**2 * sin(2 * theta) / g
  print(theta, distance)
```

If we run this code we will get the following output (truncated here), which shows that the algorithm converges to an angle of 0.785 rad (which is $\pi/4$ rad or 45°), and a maximal distance of 80 m:

```
0.1 15.893546463604896
0.25681065245459866 39.30676104058793
0.39616600450823336 56.95941704064276
0.5085159009993319 68.04406534735227
0.5926587597814116 74.12948048096732
0.6528192160171113 77.20409078000374
0.694749079686479 78.68883627904157
0.7235981382534541 79.389698659657
0.7433238318259351 79.71692719529094
0.756771734047573 79.86892021812099
...
0.7853981633974482 80.0
0.7853981633974482 80.0
```

Clearly in this case the solution we have found using the gradient descent algorithm matches the one we computed analytically. So what was the advantage of using gradient descent? In this particular case we still had to compute the derivative of the expression analytically, but we did skip the part where we had to solve for θ when that derivative was set to zero. Clearly for this problem that was not really much of a saving because the javelin problem is actually rather simple; if we construct problems with more

complex functions with many parameters, then this technique of using gradient descent really begins to show its power[16].

[16] It is not just deep learning problems that can benefit from using gradient descent to find the solutions. We will introduce some high-level examples at the end of this chapter. In **??** we review some other types of problem, which are otherwise computationally infeasible, that can be solved efficiently with gradients.

1.1.8. Derivatives of more general functions

Function composition is the process of taking the output of one function and applying another function to it. You will have seen examples of this formalised in different ways in the past; for example, if we had some input x, which we passed though a function g and passed the intermediate output though a function f to get an output y, we could write this as

$$y = f(g(x)) \ .$$

This is a function composition of f applied to g. You might also have seen this written as $f \circ g$, which you would read as "f of g". Of course more than two functions could be nested (for example $f \circ g \circ h$), and indeed the *depth* of nesting could in principle be in infinite. When we come to talk about deep learning, we will see one of the reasons it is called "deep" is in reference to the depth (or number of layers) of a neural network, which is exactly equivalent to the depth of the function composition.

Almost all complex functions can be broken into simpler parts either by construction (where a complex function is defined as a composition of simple primitive functions), or by approximation (for example through the low-order coefficients of a series[17] or through interpolation). The latter case of approximation can itself often be viewed as a composition of functions. In both approximation and construction it is often the case that the individual functions being composed have simple derivatives.

[17] For example the Taylor series and Fourier series can provide approximations of functions.

It is worth noting at this point that even elementary binary arithmetic operations are functions and can be composed[18]. For example, take the addition operator +; if we refer to the operands as a and b respectively (i.e., in $a + b$), then we can write

$$\text{add}(a, b) \stackrel{\text{def}}{=} a + b \ .$$

[18] Formally binary operations are functions of two variables, or *operands*, where the two domains and the codomain are the same set (so for example the addition operation could be defined for both inputs and the output being from the set of Real numbers).

Now, if we consider the expression $a + b + c$, we can actually see this is a function composition made by applying the add function twice,

$$a + b + c \equiv \text{add}(\text{add}(a, b), c) \ .$$

Although this is somewhat verbose, it does make the operation order explicit and also it maps to how operations will be computed by a machine. For example, in the right hand side of the

expression

$$a + b \,/\, c \equiv \text{add}(a, \text{divide}(b, c)) \,,$$

where $\text{divide}(a, b) \stackrel{\text{def}}{=} a \,/\, b$, there is no doubt that the division occurs before the addition. Returning to differentiation, recall that derivatives of function compositions are given by applying the chain rule,

$$h' = (f \circ g)' = (f' \circ g) \cdot g' \,,$$

which you might also remember being expressed as

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}u} \cdot \frac{\mathrm{d}u}{\mathrm{d}x} \,.$$

Derivatives of the elementary binary arithmetic operations are given by the sum rule,

$$(f + g)' = f' + g'$$

and the product rule,

$$(fg)' = f'g + fg' \,.$$

There are also the subtraction and quotient rules which follow on from these.

In **??** we will see that these elementary rules of differentiation can be extended to vector, matrix or tensor[19] -valued multivariate functions. Even with these extensions the simplicity of the rules remains, and it will become evident that for broad classes of function if you break it down into its constituent parts then computing the derivatives becomes very easy.

[19] The word tensor is somewhat overloaded. We'll most generally consider a tensor to be a multidimensional array of numbers. This is discussed more in **??**.

As a fun example of how these rules and the notion of decomposing a function into its constituent parts let us prove the derivative of $\sin x$ again. This time we will start by stating Euler's formulas which relate trigonometric functions to complex exponentials[20],

$$\begin{aligned} e^{ix} &= \cos x + i \sin x \,, \\ e^{-ix} &= \cos(-x) + i \sin(-x) = \cos x - i \sin x \,. \end{aligned}$$

[20] Don't worry if you have never studied complex numbers before; all you need to know is that i is a scalar constant which can be treated just like any other constant value when we perform algebraic manipulation or differentiation here. The value of i^2 is -1, but that is irrelevant for this proof.

Now, by adding these formulas together we can solve for $\cos x$,

$$\begin{aligned} e^{ix} + e^{-ix} &= \cos x + i \sin x + \cos x - i \sin x \\ &= 2 \cos x \\ \implies \cos x &= \frac{e^{ix} + e^{-ix}}{2} \,, \end{aligned}$$

and similarly by subtracting we can solve for $\sin x$,

$$\begin{aligned} e^{ix} - e^{-ix} &= \cos x + i \sin x - (\cos x - i \sin x) \\ &= 2i \sin x \\ \implies \sin x &= \frac{e^{ix} - e^{-ix}}{2i} , \end{aligned}$$

So, the implication here is that $\sin x$ can be broken down into the weighted difference of two exponential functions. Recall that by definition

$$\frac{\mathrm{d}e^x}{\mathrm{d}x} = e^x$$

and

$$\frac{\mathrm{d}e^{ax}}{\mathrm{d}x} = ae^{ax} .$$

Note that the latter is a trivial result of the chain rule,

$$\frac{\mathrm{d}e^u}{\mathrm{d}x} = \frac{\mathrm{d}e^u}{\mathrm{d}u} \cdot \frac{\mathrm{d}u}{\mathrm{d}x} = e^u \cdot \frac{\mathrm{d}u}{\mathrm{d}x}.$$

Now applying these exponential derivative formulations and utilising the complex exponential form of $\cos x$ we can prove the derivative of $\sin x$ is $\cos x$

.

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}x} \sin x &= \frac{\mathrm{d}}{\mathrm{d}x} \left(\frac{e^{ix} - e^{-ix}}{2i} \right) \\ &= \frac{1}{2i} \left(\frac{\mathrm{d}}{\mathrm{d}x} e^{ix} - \frac{\mathrm{d}}{\mathrm{d}x} e^{-ix} \right) \\ &= \frac{1}{2i} \left(ie^{ix} - (-i)e^{-ix} \right) \\ &= \frac{e^{ix} + e^{-ix}}{2} \\ &= \cos x \end{aligned}$$

□

1.2. Back to programming

Modern computer programs are wonderfully complex things, but computer programs are really just made up of function compositions and control statements. At the end of the day computer programs are just compositions of really simple functions that the microprocessor hardware can compute: arithmetic operations (add, multiply, divide, ...), logical operations (and, or, not, comparisons...), operations that move data, etc.

Many of these primitive operations performed by microprocessors have *well defined* gradients with respect to their operands. We've already seen that the chain rule tells us how to compute gradients of composite functions, and it thus follows that it should be possible to use this to compute gradients of certain types of computer program. In principle this implies we can find the locally optimal *parameters* of a computer program designed to solve a specific task by utilising the gradients of those parameters with respect to some objective, together with an optimisation scheme like the simple gradient descent approach we described earlier.

1.2.1. Differentiating through control logic and loops

It might not be immediately apparent how you might go about computing the derivatives of a piece of code that contains control statements or loops. Fortunately this actually turns out to be relatively simple when you consider the duality between a piece of code and its equivalent mathematical formulation. Take for example the following snippet of `python` code on the left which sets the value of a variable `b` as a function of another variable `a` and the mathematical equivalent (and its derivative) on the right:

Code	**Math**

`if a > 0.5:`
`    b = 1`
`else:`
`    b = 2 * a`
	$b(a) = \begin{cases} 0 & \text{if } a > 0.5 \\ 2a & \text{if } a \leq 0.5 \end{cases}$
	$\frac{\mathrm{d}b}{\mathrm{d}a} = \begin{cases} 1 & \text{if } a > 0.5 \\ 2 & \text{if } a \leq 0.5 \end{cases}$

In this example the `python` code snippet was implicitly considered to be a function. The following code with an explicit function definition would clearly have the same mathematical equivalent however.

```
function b(a):
    if a > 0.5:
        return 1
    else:
        return 2 * a
```

The mathematical form of this function (or rather the *if-else* statement) is called a *piecewise function*, as it is made up of multiple parts. The derivatives are computed by differentiating each part separately[21].

[21] There are caveats around the use of piecewise functions in a differentiable program optimised with gradient descent. This includes the need for the overall function to be continuous as previously mentioned. This example is continuous as it transitions without any discontinuity at $a = 0.5$.

Similar to the if-else example above, we can also consider what happens if a piece of code has a loop within it as in the following:

Code

```
1 b = 1
2 for i in range(3):
3     b = b + b * a
```

Math

$$
\begin{aligned}
b_0 &= 1 \\
b_1 &= b_0 + b_0 a \\
&= 1 + a \\
b_2 &= b_1 + b_1 a \\
&= 1 + 2a + a^2 \\
b_3 &= b_2 + b_2 a \\
&= 1 + 3a + 3a^2 + a^3 \\
\frac{\mathrm{d}b}{\mathrm{d}a} &= 3 + 6a + 3a^2
\end{aligned}
$$

As you can see in the math column, what we have done is to explicitly *unroll* the for-loop over each iteration, and we maintain the value of b at each *time-step* by indexing b with a subscript.

1.2.2. Can all programs be differentiable?

The examples in the previous section highlight how one can go about differentiating bits of numeric software. Indeed, we can differentiate through lots of types of programs and algorithms[22], but there are some significant limitations as not every operation or function has *useful* gradients. In particular, many functions have discontinuities or large areas of zero-gradient which makes them unsuited for use in a gradient-based optimisation setting. **??** looks at how some of these problems can be circumvented with different mathematical tricks.

[22] Even the Gradient Decent algorithm is itself differentiable! We look at an application of this in **??**.

What if we want to write programs that are not inherently numerical — for example programs that consume or generate textual data? In such cases we will often look to ways in which that data type can be *encoded* as numbers (most commonly as a vector of numbers, that we might refer to as a *distributed representation*). In some cases we might even use a (potentially differentiable) program, often trained with *self-supervised learning*, to learn distributed representations called embeddings[23].

[23] Distributed representations and embeddings are discussed in detail in **??**.

Even without these tricks, it is possible to build differentiable programs that solve a wide range real problems, such as implementations of neural networks that can be trained to recognise

images or systems that attempt to understand or generate natural language. All it takes for these examples is the careful construction of a program from differentiable building blocks, gradient descent to optimise the parameters, and a differentiable objective function coupled with large amounts of training data.

1.2.3. Do I really have to do the differentiation manually?

At this point you might be wondering about the practicality of a differentiable program. We've alluded to modern examples of differentiable programs having many variables and being made up of complex function compositions. Whilst we have demonstrated that the chain rule lies at the heart of any approach to differentiating complex compositions, at the end of the day all the differentiation we have done so far has involved analytic and algebraic manipulations performed by hand.

Thus far we have focused on analytically finding the gradient by taking a limit of the difference quotient. It turns out that there are actually other ways of computing gradients which will be useful in practical differentiable programs. Firstly the job of computing gradients of expressions analytically or symbolically is something that can be automated using a computer with a *Computer Algebra System* (CAS), which is a piece of software designed to apply analytic algebraic operations, including computing derivatives. A CAS works by systematically applying the same rules that we used by hand to reduce, simplify and solve algebraic expressions in symbolic form.

Secondly, if we return to the original difference quotient expression we can see that it should be possible to create numerical estimates of a gradient of a function by evaluating the difference of the function at two points and dividing by the distance between the points[24]. This numerical estimation of gradient is useful for checking for errors that could occur when we hand-compute and implement derivatives in code, and also has practical application in problems where it can be hard to compute derivatives exactly.

Thirdly, there is an approach to numerically computing exact (within the limits of the chosen floating point number representation) gradients of programs, called *automatic differentiation*. Automatic Differentiation is key to differentiating programs with millions of parameters[25]. As the 'programmer' *you* still need to have a really good intuition and understanding of what the im-

[24] There are many challenges with this approach, not least because computers use fixed bit-width floating point arithmetic which can allow errors to accumulate. We explore this more in **??**.

[25] Automatic differentiation was a key ingredient to enabling the deep learning revolution. It should be noted that the other approaches to differentiation still have their uses in deep learning and broader differentiable programming. For example in **??** we will see how symbolic differentiation can be combined to increase performance of particular differentiable programs.

plications of composing many functions will be on the gradients of the parameters with respect to the objective for optimisation to work successfully however.

1.2.4. What kinds of functional building blocks are common?

Most modern differentiable programs are build around the idea of performing operations on and with vectors, matrices and tensors. The basic building blocks of differentiable programs that are commonly in use today consist of binary operations which act on an input with a set of weights:

- *Vector addition*: the input vector $\boldsymbol{x}$ is added to a vector of weights $\boldsymbol{w}$: $\boldsymbol{x} + \boldsymbol{w}$. The vectors $\boldsymbol{x}$ and $\boldsymbol{w}$ must be from the same space and thus have the same dimensionality.
- *Element-wise multiplication*: the input vector $\boldsymbol{x}$ is multiplied element-wise with the vector of weights $\boldsymbol{w}$. This operation is known as the the Hadamard Product, denoted by $\boldsymbol{x} \odot \boldsymbol{w}$. The vectors $\boldsymbol{x}$ and $\boldsymbol{w}$ must be from the same space and thus have the same dimensionality.
- *Matrix-vector multiplication*: the input vector to the function is multiplied with a matrix of weights. Given a matrix of weights $\boldsymbol{W}$ and vector $\boldsymbol{x}$ this would be denoted $\boldsymbol{W}\boldsymbol{x}$.
- *Convolution*: the input vector $\boldsymbol{x}$ is 'convolved' with a set of weights $\boldsymbol{w}$. Convolution is denoted as $\boldsymbol{x} * \boldsymbol{w}$. In practice most implementations use a closely related operation called cross-correlation, denoted $\boldsymbol{x} \star \boldsymbol{w}$. The vector $\boldsymbol{w}$ is often referred to as the *kernel* and is usually has much lower dimensionality than the input $\boldsymbol{x}$. **??** covers convolution in detail.

The above operations are often generalised to work on inputs that are matrices or tensors. All of these operations are *linear mappings* (even when extended to matrices or tensors). Assuming $\mathbb{A}$ and $\mathbb{B}$ are vector spaces over some field $\mathbb{F}$ (such as the field of reals $\mathbb{R}$), recall that a function $f : \mathbb{A} \to \mathbb{B}$ is said to be a linear mapping if two conditions are satisfied: *additivity*,

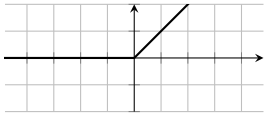
$$f(\boldsymbol{a} + \boldsymbol{b}) = f(\boldsymbol{a}) + f(\boldsymbol{b})$$

and *homogeneity*,

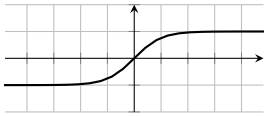
$$f(c\boldsymbol{a}) = cf(\boldsymbol{a}) \ ,$$

for any vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{A}$ and scalar $c \in \mathbb{F}$. Compositions of linear functions are also linear functions[26]. This means for example a series of matrix-vector multiplies could always be written as a single matrix-vector multiply[27]. As such, it is common to make the overall function be non-linear by following each linear operation with an *element-wise* nonlinearity[28]. Commonly used element-wise nonlinearities include:

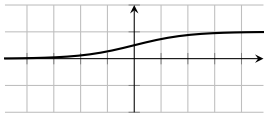
The *rectified linear unit* (ReLU). $\text{ReLU}(x) = \max(0, x)$.



The hyperbolic tangent, $\tanh(x)$.



A *sigmoid*[29] function such as the logistic $\frac{1}{1+e^{-x}}$.



Sometimes we also use *point-wise*[30] nonlinearities over an output, $\boldsymbol{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$ of a composition or operation, such as the softmax function,

$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K\,,$$

which has the effect[31] of making the largest element of the vector $\boldsymbol{z}$ tend towards 1, and the other elements tend towards 0.

1.3. Real Examples of Differentiable Programming

To wrap-up this introduction we give a number of examples of how differentiable programs could be formulated to solve a range of different tasks. We take a rather high-level *functional* perspective and consider the inputs, outputs and objective. We pay little attention to the inner formulation of the function, as that is addressed in later chapters.

[26] Prove this yourself in exercise 1.3.

[27] Interestingly the learning dynamics using gradient descent on a linear function is non linear, so learning weights of a single matrix versus learning weights of a multiplication of matrices could give very different results. See **??** for more on this topic.

[28] A nonlinearity applied to an element of a vector that is independent of other elements of the vector.

[29] A *sigmoid* function is an 'S-shaped' function that has commonly has the effect of squashing values into $[0, 1]$ allowing the output to be interpreted as a probability. If you see a reference to 'the sigmoid function', then this is actually referring to the logistic function. See **??**.

[30] A nonlinearity applied to an element of a vector that is a function of other elements of the vector.

[31] In many ways the softmax turns an arbitrary vector of values into a proper probability density function, with all values bounded in $[0, 1]$ and summing to 1. The input values to the softmax are often referred to as *logits* because they can be seen to represent log-probabilities. See **??** for more discussion.

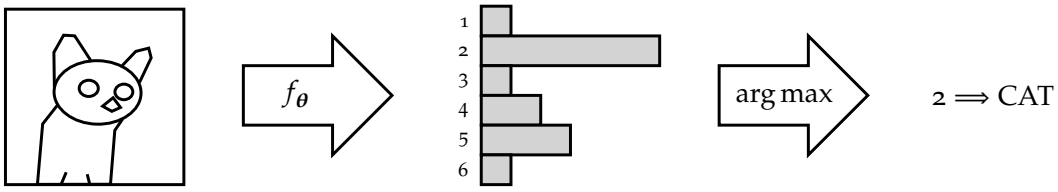


Figure 1.5: Image Classification. A function f with parameters $\boldsymbol{\theta}$ predicts a distribution over a set of labels given an input image. The highest scoring class label is selected as the prediction for the input image. During *training* the parameters $\boldsymbol{\theta}$ are optimised using gradients with respect to the known target score distribution for a large number of labelled training image examples belonging to each of the classes.

1.3.1. Image classification

We start with the application that kick-started the deep learning revolution: image classification. A typical image classification problem, illustrated in fig. 1.5, involves training a function f with parameters $\boldsymbol{\theta}$ to be able to predict a single label l given an input image $\mathbf{I}$,

$$l = f(\mathbf{I}; \boldsymbol{\theta}) \ .$$

The label l would be defined to be drawn from a set of n possible labels, $l \in 0, \ldots n$, where each integer label maps to a concept, such as $0 =$ 'dog', $1 =$ 'cat', $\ldots n =$ 'goldfish'. The image would typically be a 3D tensor with values representing the amount of red, green and blue colour at each spatial position or *pixel* of the image. We would have a large set of training data in the form of pairs of images and labels which would be used to learn an optimal set of parameters $\boldsymbol{\theta}$.

The function defined above has a discrete integer output, which would cause problems with gradient methods[32]. As such, it's common to redefine the function to return a vector $\hat{\boldsymbol{y}} \in \mathbb{R}^n$,

$$\hat{\boldsymbol{y}} = f(\mathbf{I}; \boldsymbol{\theta}) \ ,$$

and define a objective function for training that tries to ensure that for a particular training example with label l that $\hat{y}_l$ is *larger* than all other elements of $\hat{\boldsymbol{y}}$. This small change allows for a model and loss with well defined gradients. Once the model has been trained it is possible to extract the actual class by computing $\arg\max \hat{\boldsymbol{y}}$,

$$l = \arg\max \hat{\boldsymbol{y}} = \arg\max f(\mathbf{I}; \boldsymbol{\theta}) \ .$$

This is illustrated in fig. 1.5. In practice we often conceptually model the vector $\hat{\boldsymbol{y}}$ as containing the *unnormalised log-probabilities* ("logits"), or as actual probabilities if the function f incorporates

[32] At some point you would need to discretise the output, but functions that *round* or take the *ceiling* or *floor* of a real number are not continuous and have useless derivatives.

[33] The categorical distribution is a discrete probability distribution representing the chance of a random variable taking on one of K possibilities. It is also known as the Generalised Bernoulli or multinoulli distribution.

a suitable normalisation, of a *categorical distribution*[33]. Logits are trivially transformed to normalised probabilities by applying the softmax. The true target label l can be converted to an idealised probability distribution, $\boldsymbol{y}$ through a one-hot encoding; this creates a vector with zeros at all elements except for the l-th element which takes a value of 1. Now that both targets and predictions can be modelled as probability distributions, an obvious learning objective would be to adjust the set of parameters $\boldsymbol{\theta}$ such that they maximise the likelihood that the model makes the correct prediction over all the training examples. Because it is is common to actually perform a minimisation we would minimise the negative likelihood, and for practical and numerical reasons it makes sense to work with the negative log-likelihood[34].

[34] This turns out to be exactly equivalent to minimising the Kullback–Leibler (KL) divergence between the predicted probability distribution and the distribution formed by the one-hot encoded target. As such this loss is often referred to as a *cross-entropy*. We recap and explore this in more detail in **??**

1.3.2. Object detection

Building on the idea of a function that performs image classification, could we go a step further and actually identify multiple objects in an image and simultaneously detect their bounds? This is the task of *object detection*. From a functional perspective this task involves training a function f with parameters $\boldsymbol{\theta}$ to be able to predict a (unordered) *set* of n_b tuples corresponding to the n_b objects in a given image

$$\mathbb{B} = \{(x_1, y_1, w_1, h_1, l_1), \dots, (x_{n_b}, y_{n_b}, w_{n_b}, h_{n_b}, l_{n_b})\} ,$$

where x, y, w, h are the (real valued) parameters of the object's bounding box[35], and as with image classification, $l \in 0, \dots n$, is an integer label mapping to a concept. Formally we can write this as

$$\mathbb{B} = f(\mathbf{I}; \boldsymbol{\theta}) .$$

[35] Here the top-left coordinate and width and height. Other parameterisations, such as the top-left and lower-right corners as illustrated in fig. 1.7, could be used however.

Now as with the image classification example we have to make some changes to make this differentiable and amenable to learning of the parameters. The key challenge to overcome is that the output of the function is a set of vectors rather than say a matrix of ordered vectors; whilst we can pretend that the output is unordered even if it actually is not, doing so causes major problems with the ability to learn the parameters[36]. There are many possible ways in which we can alter the object detection task to make it learnable. We briefly describe two different alternatives: *anchor-based* and *set-prediction*.

[36] Pretending an ordered output is actually unordered introduces discontinuities into the learning procedure which hampers or stops learning. See **??** for more on this topic and a detailed discussion of how to avoid it in general problems.

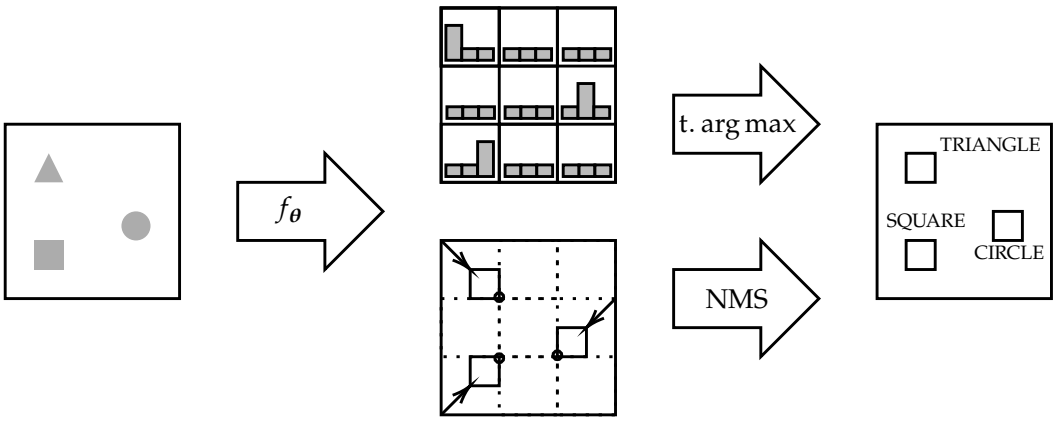


Figure 1.6: Object Detection with Anchors. A function f with parameters $\boldsymbol{\theta}$ predicts a label for a set of each of a list of *anchor* boxes, and simultaneously predicts changes in the offsets of the anchor box to make it more closely fit the underlying object in the image. In the figure, anchor boxes are denoted by the 3×3 grid covering the image space; in reality, multiple overlapping boxes of differing sizes and positions would be used. The highest scoring class label is selected as the prediction for each anchor if the probability exceeds a threshold (many anchor boxes will have low probability for all classes, and are thus ignored as belonging to the background). During *training* the parameters $\boldsymbol{\theta}$ are optimised using gradients with respect to a set of assignments of matching anchors (and their computed offsets and class assignment) computed from the overlap of the anchors and the true bounding boxes and class labels. After training, at inference time a non-differentiable algorithm called Non-Maximal Suppression (NMS) is used to filter out multiple anchors assigned to the same object.

Anchor-based approaches. Anchor-based approaches fundamentally change the prediction problem from being one of predicting variable length sets of tuples to being one of predicting fixed-length lists of tuples. As illustrated in fig. 1.6, a fixed set of predefined 'anchor boxes' is placed over the image and the job of the function is then to map each anchor box to an object by adjusting the position, and predicting the class of the object within the adjusted anchor. For training, the target output for a given image is created by mapping the true set of object bounds and classes to the best matching anchor boxes.

The downside of this approach is that often many anchor boxes will end up being assigned to the same object (both during training and during inference). A non-differentiable technique called non-maximal suppression can be used to select only the most relevant box for each object in the image. Unfortunately this inherently means that the performance of the object detection function is limited because it is being trained towards a proxy objective rather than the actual objective.

Set prediction approaches. An alternative approach involves using the original object detection formalisation and actually create a

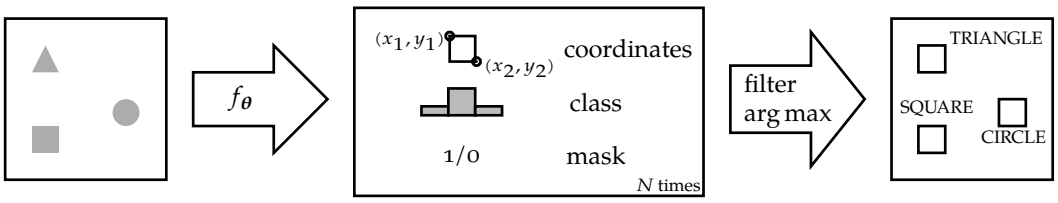


Figure 1.7: Object Detection with Set Prediction. A function f with parameters $\boldsymbol{\theta}$ directly predicts sets of N bounding boxes, labels and a binary mask. The function incorporates an internal optimisation procedure to fit predictions to the input image based on the learned parameters $\boldsymbol{\theta}$. During *training* the parameters $\boldsymbol{\theta}$ are optimised to directly match the ground truth boxes and labels (and the mask is used to remove unused/excess predictions). This form of training requires the use of a permutation invariant loss. At inference time, predictions with a zero-valued mask are removed and the label for each predicted box is computed by choosing the class with the highest probability.

function that truly returns unordered outputs, as shown in fig. 1.7. The advantage of this approach is that we will directly optimise the parameters of towards the task of detecting objects (so we could expect better performance). The downside is that defining a function that learns to produce sets of vectors is hard. One possible approach to generating sets is to actually use optimisation within the function $f_{\boldsymbol{\theta}}$ as follows:

$$\mathbb{B} = f_{\boldsymbol{\theta}}(\mathbf{I}) = \min_{\mathbb{B}} \quad \left\| h(\mathbf{I}; \boldsymbol{\theta}_h) - g(\mathbb{B}; \boldsymbol{\theta}_g) \right\|_2$$

where $\boldsymbol{\theta} = [\boldsymbol{\theta}_h; \boldsymbol{\theta}_g]$, h is a function that maps the image $\mathbf{I}$ to a compact vector representation[37], and g is a function that maps the set $\mathbb{B}$ to a latent representation.

Whilst this might look immensely complicated, the overall idea is quite simple: when f has been trained (i.e. $\boldsymbol{\theta}_h$ and $\boldsymbol{\theta}_g$ have been learned), then given an image $\mathbf{I}$, the function f will search for an optimal set of bounding boxes $\mathbb{B}$ by performing a minimisation of the distance between the latent representations of the image and set of bounding boxes. If the parameters of the function have been adequately trained, starting from a random set of vectors the minimisation procedure should quickly optimise the bounding boxes to fit the objects within the image. This minimisation itself can be implemented using the gradient descent algorithm we've seen previously. This is a an example of a much more powerful differentiable program, where we are not learning a fixed function, but rather learning the parameters of an algorithm that can solve a task.

Training the model requires the use of a *set loss*[38] that is invariant to the ordering of the predicted and ground-truth bounding boxes to ensure the correct mapping between each individual prediction and target. As is illustrated in fig. 1.7 we often con-

[37] Such representations are common in differentiable programs, and are often referred to as *latent representations* because they are said to capture the hidden characteristics of the object from which they are computed.

[38] A loss function that is *permutation invariant* with respect to the order within its two inputs. If the order of the vectors within one or both inputs changes, the output should remain the same. See **??**.

sider that f produces a fixed size set of N vectors (N being the maximum number of objects we expect to be able to find), and allow each predicted bounding box to choose whether it should be considered to be relevant by augmenting the bounding box vector with a binary mask. This allows us to find fewer than N objects in a given image by filtering out all the bounding boxes with a mask value close to zero.

1.3.3. Language Translation

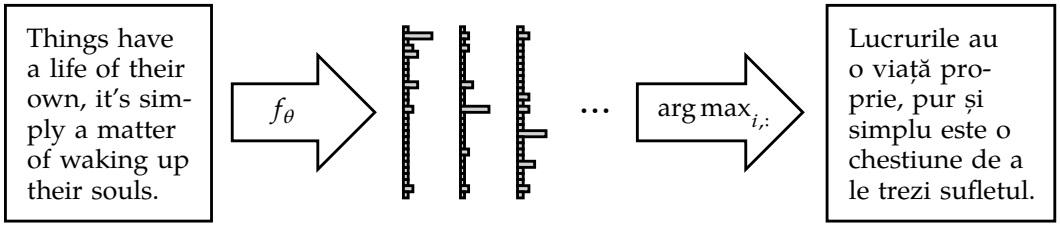


Figure 1.8: Language Translation. A function f with parameters $\boldsymbol{\theta}$ predicts a distribution over a sequence of a set of words from the target language given an input in the source language. In the simplest case, as illustrated, the highest scoring word is picked for each set in the sequence. One should note that f does not operate on a single input word at a time but rather considers sequences of consecutive words. Also, the output could have a different number of words to the input.

Turning attention away from images, we will next consider a problem involving natural language, and look at translating one language to another. This process involves a function that takes input text in the *source* language and provides as output text in the target language as shown in fig. 1.8. Now, a differentiable mathematical function f cannot naturally take a sequence of characters as input, so some process has to be used to *encode* or *embed* each word as a N_{in}-dimensional vector[39], and then often put the entire sequence into a matrix, $\boldsymbol{X} \in \mathbb{R}^{w_{in} \times N_{in}}$ by stacking the transposed vectors.

[39] We will explore encoding and embedding in **??**. For now, imagine a simple *one-hot encoding* of each word: consider that we might form a list of every possible unique word that could appear in the input which we call a *vocabulary*. If there are N words in the vocabulary, we can map the i-th word to an N-dimensional vector, $\boldsymbol{e}_i \in \mathbb{R}^N$ where the i-th element has a value of 1, and all other values are zero.

The output of the function is often also encoded as a sequence of logits over the entire output vocabulary. Formalising, if their were N_t possible words in the target language vocabulary, and w_t words are predicted, then,

$$\hat{\boldsymbol{Y}} = f(\boldsymbol{X}; \boldsymbol{\theta}) \ ,$$

where $\boldsymbol{Y}$ is an $w_t \times N_t$ matrix. By applying the arg max function across each row of $\boldsymbol{Y}$, it is possible to recover a sequence or vector of the indexes of the particular words in the target vocabulary, $\hat{\boldsymbol{y}}$, where

$$y_i = \arg\max \boldsymbol{Y}_{i,:} \ .$$

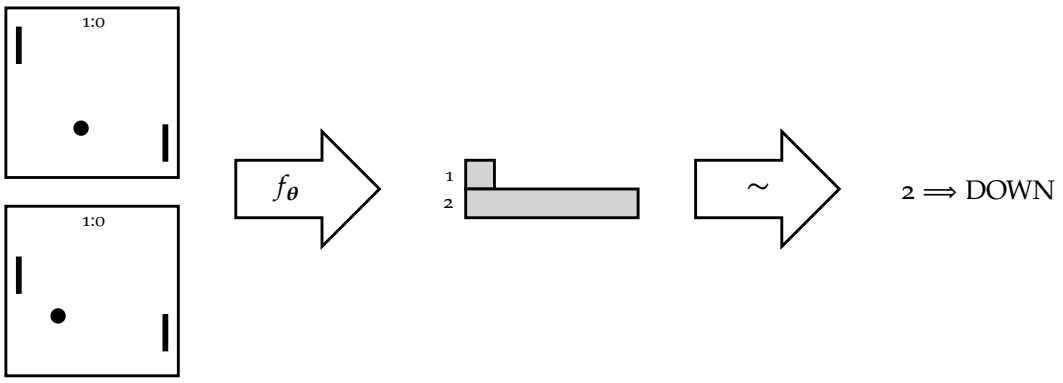


Figure 1.9: Playing games. We can train *agents* to solve sequential tasks such as playing games by defining a differentiable function (often referred to as a *policy*) with parameters $\boldsymbol{\theta}$. This policy predicts the probability distribution over possible actions (here illustrated to be discrete actions of moving the left paddle *up* or *down*, however modelling continuous actions is also possible), and the action that is taken is sampled from this. The input is a sequential pair of images showing the state of the game in the current and previous time steps. The agent is trained with the objective of maximising their score over the opponent. Clearly their is little information about what move will maximise the score given a single pair of images, however it can be shown that its possible to get a useful learning signal (the gradient with respect to $\boldsymbol{\theta}$) from the expectation over many games and the sequences of actions taken within them using what are known as *Policy Gradient methods*.

It should be noted that practical implementations of translation functions (and actually broader classes of functions that produce any sequences of discrete *tokens*) often employ more powerful *decoding strategies*[40] to turn each logit vector back into a word.

The function f is often constructed as two sub-functions which are composed: the *encoder* which turns the input text into a *latent representation*, and the *decoder* which turns that vector into the sequence of logit vectors. The decoder is commonly an *auto-regressive function* which produces the logits for the next output as a function of both the latent vector from the encoder, as well as the encodings of the previously predicted output words.

In terms of training the model and learning the parameters $\boldsymbol{\theta}$, essentially we can minimise the sum of the negative log-likelihoods between each predicted word and the target word for each position in the sequence. In practice we sometimes employ additional tricks such as *teacher forcing* (see **??**) during training which forcibly corrects errors during decoding.

[40] The decoding strategy using arg max is known as *greedy decoding*. Popular, more powerful choices include the *viterbi algorithm* and *beam search*, which are discussed in **??**.

1.3.4. Playing Games

You can use differentiable programming to write (and train) 'agents' that can play games. Take for example the classic com-

puter game pong, which simulated a game of table tennis between two players. In pong, the players have to move a paddle to deflect a ball, and a player accumulates points when the opposing player misses the ball. As before we can formalise our agent as a function f with parameters $\boldsymbol{\theta}$. Just like a human playing the game, we can use a visual input. Because this game involves motion we could choose to provide the current frame $\mathbf{I}_{curr}$ and the previous frame $\mathbf{I}_{prev}$ as inputs[41]. The output of the function is the *action* $\mathbb{A}$ that the agent chooses to perform. Formally we can write

$$\mathbb{A} = f(\mathbf{I}_{curr}, \mathbf{I}_{prev}; \boldsymbol{\theta}) \ .$$

In the case of a simple game of pong, the action $\mathbb{A}$ would be the discrete action of either moving the paddle up or moving the paddle down as illustrated in fig. 1.9.

As in the previous examples, selection of a discrete action is not amenable to gradient-based optimisation, so instead we could choose to model the probability of the actions; in this particular case with only two possible actions (*up* and *down*), we need only model the probability of moving up, and the probability of moving down is simply one minus this. As such, we can redefine f to have a single scalar result p_{up} in the range $[0, 1]$ (possibly achieved using a sigmoid function),

$$p_{up} = f(\mathbf{I}_{curr}, \mathbf{I}_{prev}; \boldsymbol{\theta}) \ .$$

When playing the game our agent will just need to *sample* a discrete action $a \sim p_{up}$ based on the probability.

In order to learn good values of $\boldsymbol{\theta}$ for our function we need an objective. Clearly maximising the agent's score relative to the opponent is an appropriate objective — the agent's score minus the opponents score will suffice as our *reward function*. This learning problem falls into the category of *reinforcement learning*.

It should be clear that between most pairs of consecutive frames from a game of pong the score will not change, so it will be impossible to get usable gradients that will allow us to adjust $\boldsymbol{\theta}$ at each step. A trick that will allow us to compute usable gradients comes from noting that whilst a single game or move might not have useful gradients, the expectation over many games and moves will; this is the topic of *policy gradient* methods[42]. If we denote the reward function as $r(a)$, which can simply take on a value of 1 if the action a was part of a sequence that led to a game being won and -1 if the game was lost, then we just need

[41] This is just one of many possible parameterisations of the function; we could for example provide a single image in the form of the difference between the current and previous frames, or provide multiple frames going back further in time, or even make the function recurrent with its own *memory* or *state*.

[42] Policy gradients are not the only way we can get useful gradients for problems involving discrete choices. Policy gradients and alternative gradient estimators for discrete choices are derived and discussed in **??**.

to maximise

$$\mathbb{E}_{a\sim p(\mathrm{a}|\boldsymbol{\theta})}[r(x)]\,,$$

where $p(\mathrm{a} \mid \boldsymbol{\theta})$ is modelled by our *policy* $f(\mathbf{I}_{curr}, \mathbf{I}_{prev}; \boldsymbol{\theta})$. With a little bit of simple algebraic manipulation it can be shown that the gradient of this expectation with respect to $\boldsymbol{\theta}$ is just a function of the expectation of the gradient of our policy f, so as long as f is differentiable then we can just follow the gradients to maximise our chance of winning many games.

Whilst pong is a relatively simple game, the broad idea of using policy gradients is rather general. Recently we have witnessed a lot of successes in the field of *deep reinforcement learning* with agents playing games such as Chess, Go and StarCraft, and winning against the best human players in the world at these games.

1.3.5. Drawing

- We could envisage a differentiable function that takes in a set of line coordinates and turns them into an image...
- With such a function we can optimise the line coordinates so they e.g. match a photograph, thus automatically creating a *sketch*.

1.4. Outlook

This book aims to expose the reader to modern differentiable programming in some depth. The first part of the book covers fundamentals from how differentiation can be used to solve complex problems, the building blocks of neural networks, and the increasingly important concepts of function reparameterisations and relaxations that allow us to build more complex models. The

second part looks at what is at the time of writing the contemporary set of building blocks and deep learning architectures. In this part we also dive into where some of the inspiration for these architectures has arisen from our understanding of biology and brains, and also look at how the architectural choices can result in the emergence of behaviours that mimic biological findings. Finally, in the third part we look at research topics where structures, priors and biases are being designed to induce certain behaviours to occur within models, or to apply constraints to parts of a model.

1.5. Exercises

Exercise 1.1

Construct a proof of the limit $\lim_{x \to 0} \frac{\sin x}{x} = 1$. Avoid using L'Hôpital's rule $\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f'(x)}{g'(x)}$, as that would result in a rather circular argument based on *knowing* $\sin'(x) = \cos(x)$ before we proved it in section 1.1.2.

Exercise 1.2

Compute the optimal angle for throwing a javelin. Assume that there is no air resistance as in section 1.1.4, but this time assume that the javelin flight starts from an initial height of h m.

What is the optimal angle when $h = 2$ m?

Exercise 1.3

Given $f : \mathbb{A} \to \mathbb{B}$ and $g : \mathbb{B} \to \mathbb{C}$ are two linear mappings, show that $g \circ f : \mathbb{A} \to \mathbb{C}$ is also a linear mapping.

Part I

Foundations

2. The Building Blocks of Learning Machines

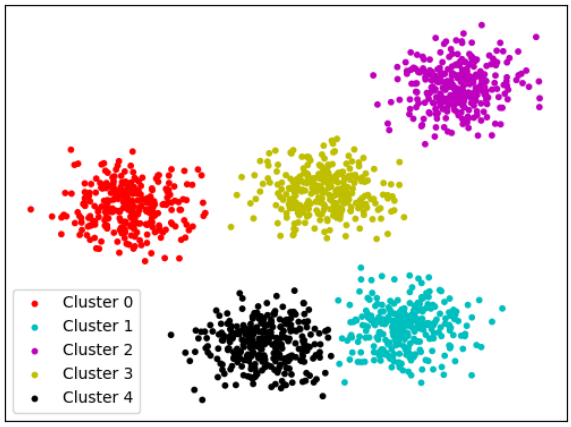
2.1. Learning Paradigms

Types of Learning

- Supervised Learning - learn to predict an output when given an input vector
- Unsupervised Learning - discover a good internal representation of the input
- Reinforcement Learning - learn to select an action to maximize the expectation of future rewards (payoff)
- Self-supervised Learning - learn with targets induced by a prior on the unlabelled training data
- Semi-supervised Learning - learn with few labelled examples and many unlabelled ones

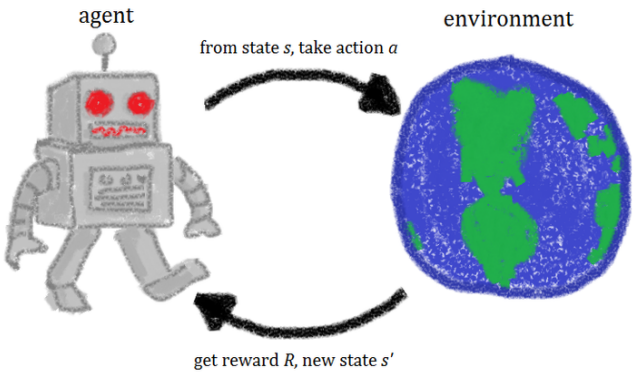
Supervised Learning

Unsupervised Learning

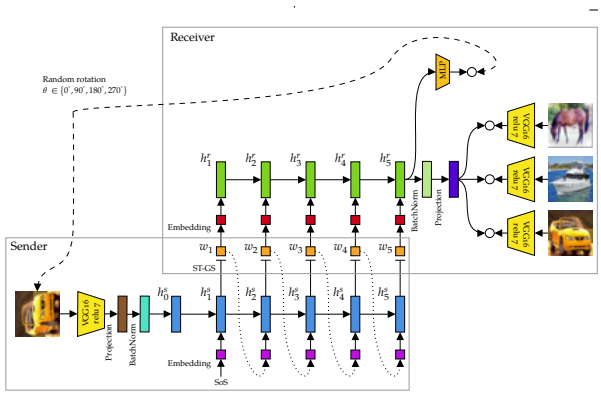


Reinforcement Learning

Newell, Alejandro, Kaiyu Yang, and Jia Deng. "Stacked hourglass networks for human pose estimation." ECCV'16. Springer, 2016.



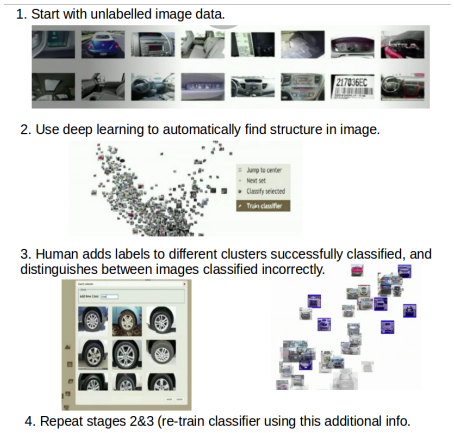
Self-supervised Learning



Semi-supervised Learning

Reference: Wikipedia `https://simple.wikipedia.org/wiki/Reinforcement_learning`

Daniela Mihai and Jonathon Hare. Avoiding hashing and encouraging visual semantics in referential emergent language games. EmeCom @ NeurIPS 2019. https://arxiv.org/abs/1911.05546



Generative Modelling

- Many unsupervised and self-supervised models can be classed as 'Generative Models'.
- Given unlabelled data X, a unsupervised generative model learns $P[X]$.
  - Could be direct modelling of the data (e.g. Gaussian Mixture Models)
  - Could be indirect modelling by learning to map the data to a parametric distribution in a lower dimensional space (e.g. a VAEs Encoder) or by learning a mapping from a parameterised distribution to the real data space (e.g. a VAE Decoder or GAN)
- These are characterised by an ability to 'sample' the model to 'create' new data

Generative vs. Discriminative Models (II)

Generative vs. discriminative approaches to classification use different statistical modelling.

- Discriminative models learn the boundary between classes. A discriminative models is a model of the conditional probability of the target Y given an observation X: $P[Y|X]$.

Jeremy Howard. The wonderful and terrifying implications of computers that can learn. TEDxBrussels. `http://www.ted.com/talks/jeremy_howard_the_wonderful_and_terrifying_implications_of_computers_that_can_learn`

- Generative models of labelled data model the distribution of individual classes. Given an observable variable X and a target variable Y, a generative model is a statistical model that tries to model $P[X|Y]$ and $P[Y]$ in order to model the joint probability distribution $P[X, Y]$.[1]

2.2. A Supervised Learning Example

Two Types of Supervised Learning

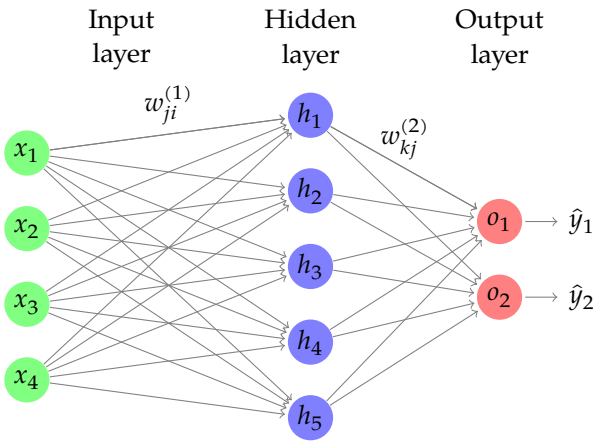
- Classification: The machine is asked to specify which of k categories some input belongs to.
  - Multiclass classification - target is one of the k classes
  - Multilabel classification - target is some number of the k classes
  - In both cases, the machine is a function $f : \mathbb{R}^n \to \{1, ..., k\}$ (although it is most common for the learning algorithm to actually learn $\hat{f} : \mathbb{R}^n \to \mathbb{R}^k$).
- Regression: The machine is asked predict k numerical values given some input. The machine is a function $f : \mathbb{R}^n \to \mathbb{R}^k$.
- Note that there are lots of exceptions in the form the inputs (and outputs) can take though! We'll see lots of variations in the coming weeks.

How Supervised Learning Typically Works

- Start by choosing a model-class: $\hat{y} = f(\mathbf{x}; \mathbf{W})$ where the model-class f is a way of using some numerical parameters, $\mathbf{W}$, to map each input vector $\mathbf{x}$ to a predicted output $\hat{y}$.
- Learning means adjusting the parameters to reduce the discrepancy between the true target output y on each training case and the output $\hat{y}$, predicted by the model.

[1] Some such models can be sampled conditionally based on a prior Y - e.g. a Conditional VAE: https://papers.nips.cc/paper/5775-learning-structured-output-representation-using-deep-conditional-generative-models

Let's look at an unbiased Multilayer Perceptron...



Without loss of generality, we can write the above as:

$$\hat{\mathbf{y}} = g(f(\mathbf{x}; \mathbf{W}^{(1)}); \mathbf{W}^{(2)}) = g(\mathbf{W}^{(2)} f(\mathbf{W}^{(1)} \mathbf{x}))$$

where f and g are activation functions.

2.3. Activation Functions

Common Activation Functions

- Identity
- Sigmoid (aka Logistic)
- Hyperbolic Tangent (tanh)
- Rectified Linear Unit (ReLU) (aka Threshold Linear)

Final layer activations

$$\hat{y} = g(\mathbf{W}^{(2)} f(\mathbf{W}^{(1)} \mathbf{x}))$$

- What form should the final layer function g take?
- It depends on the task (and on the chosen loss function)...
  - For regression it is typically linear (e.g. identity), but you might choose others if you say wanted to clamp the range of the network.

- For binary classification (MLP has a single output), one would choose Sigmoid
- For multilabel classification, typically one would choose Sigmoid
- For multiclass classification, typically you would use the Softmax function

Softmax

The softmax is an activation function used at the output layer of a neural network that forces the outputs to sum to 1 so that they can represent a probability distribution across a discrete mutually exclusive alternatives.

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad \forall i = 1, 2, \ldots, K$$

- Note that unlike the other activation functions you've seen, softmax makes reference to all the elements in the output.
- The output of a softmax layer is a set of positive numbers which sum up to 1 and can be thought of as a probability distribution.
- Note:

$$\begin{aligned}
\frac{\partial \,\text{softmax}(\mathbf{z})_i}{\partial z_i} &= \text{softmax}(z_i)(1 - \text{softmax}(z_i)) \\
\frac{\partial \,\text{softmax}(\mathbf{z})_i}{\partial z_j} &= \text{softmax}(z_i)(1(i = j) - \text{softmax}(z_j)) \\
&= \text{softmax}(z_i)(\delta_{ij} - \text{softmax}(z_j))
\end{aligned}$$

2.4. Objective Functions

Ok, so let's talk loss functions

- The choice of loss function depends on the task (e.g. classification/regression/something else)
- The choice also depends on the activation function of the last layer
  - For numerical reasons (see Log-Sum-Exp in a few slides) many times the activation is computed directly within the loss rather than being part of the model

  - Some classification losses require *raw outputs* (e.g. a linear layer) of the network as their input
    * These are often called *unnormalised log probabilities* or *logits*
    * An example would be hinge-loss used to create a Support Vector Machine that maximises the margin — e.g.: $\ell_{hinge}(\hat{y}, y) = \max(0, 1 - y \cdot \hat{y})$ with a true label, $y \in \{-1, 1\}$, for binary classification.

- There are many different loss functions we might encounter (MSE, Cross-Entropy, KL-Divergence, huber, L1 (MAE), CTC, Triplet, ...) for different tasks.

The Cost Function (measure of discrepancy)

Recall from Foundations of Machine Learning:

- Mean Squared Error (MSE) loss for a single data point (here assumed to be a vector, but equally applicable to a scalar) is given by

  $\ell_{MSE}(\hat{\mathbf{y}}, \mathbf{y}) = \sum_i (\hat{y}_i - y_i)^2 = (\hat{\mathbf{y}} - \mathbf{y})^\top (\hat{\mathbf{y}} - \mathbf{y})$

- We often multiply this by a constant factor of $\frac{1}{2}$ — can anyone guess/remember why?

- $\ell_{MSE}(\hat{\mathbf{y}}, \mathbf{y})$ is the predominant choice for regression problems with linear activation in the last layer

- For a classification problem with Softmax or Sigmoidal (or really anything non-linear) activations, MSE can cause slow learning, especially if the predictions are very far off the targets
  - Gradients of ℓ_{MSE} are proportional to the difference in target and predicted multiplied by the gradient of the activation function[2]
  - The Cross-Entropy loss function is generally a better choice in this case

Binary Cross-Entropy

For the binary classification case:

$$\ell_{BCE}(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

[2] http://neuralnetworksanddeeplearning.com/chap3.html

- The cross-entropy cost function is non-negative, $\ell_{BCE} > 0$
- $\ell_{BCE} \approx 0$ when the prediction and targets are equal (i.e. $y = 0$ and $\hat{y} = 0$ or when $y = 1$ and $\hat{y} = 1$)
- With Sigmoidal final layer, $\frac{\partial \ell_{BCE}}{\partial \mathbf{w}_i^{(2)}}$ is proportional to just the error in the output $(\hat{y} - y)$ and therefore, the larger the error, the faster the network will learn!
- Note that the BCE is the negative log likelihood of the Bernoulli Distribution

Binary Cross-Entropy — Intuition

- The cross-entropy can be thought of as a **measure of surprise**.
- Given some input x_i, we can think of $\hat{y}_i$ as the estimated probability that x_i belongs to class 1, and $1 - \hat{y}_i$ is the estimated probability that it belongs to class 0.
- Note the extreme case of infinite cross-entropy, if your model believes that a class has 0 probability of occurrence, and yet the class appears in the data, the 'surprise' of your model will be infinitely great.

Binary Cross-Entropy for multiple labels

In the case of multi-label classification with a network with multiple sigmoidal outputs you just sum the BCE over the outputs:

$$\ell_{BCE} = -\sum_{k=1}^{K} [y_k \log(\hat{y}_k) + (1 - y_k) \log(1 - \hat{y}_k)]$$

where K is the number of classes of the classification problem, $\hat{y} \in \mathbb{R}^K$.

Numerical Stability: The Log-Sum-Exp trick

$$\ell_{BCE}(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

- Consider what might happen early in training when the model might confidently predict a positive example as negative
  - $\hat{y} = \sigma(z) \approx 0 \implies z << 0$

  - if $\hat{y}$ is small enough, it will become 0 due to limited precision of floating-point representations
  - but then $\log(\hat{y}) = -\inf$, and everything will break!

- To tackle this problem implementations usually combine the sigmoid computation and BCE into a single loss function that you would apply to a network with linear outputs (e.g. `BCEWithLogitsLoss`).

- Internally, a trick called 'log-sum-exp' is used to *shift* the centre of an exponential sum so that only numerical underflow can potentially happen, rather than overflow[3].

  - Ultimately this means you'll always get a numerically reasonable result (and will avoid NaNs and Infs originating from this point).

Multiclass classification with Softmax Outputs

- Softmax can be thought of making the K outputs of the network mimic a probability distribution.

- The target label y could also be represented as a distribution with a single 1 and zeros everywhere else.

  - e.g. they are "one-hot encoded".

- In such a case, the obvious loss function is the *negative log likelihood* of the Categorical distribution (aka Multinoulli, Generalised Bernoulli, Multinomial with one sample)[4]: $\ell_{NNL} = -\sum_{k=1}^{K} y_k \log \hat{y}_k$

  - Note that in practice as y_k is zero for all but one class you don't actually do this summation, and if y is an integer class index you can write $\ell_{NNL} = -\log \hat{y}_y$.

- Analogously to what we saw for BCE, Log-Sum-Exp can be used for better numerical stability.

  - PyTorch combines LogSoftmax with NNL in one loss and calls this "Categorical Cross-Entropy" (so you would use this with a *linear output layer*)

[3] https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/

[4] Note: Keras calls this function 'Categorical Cross-Entropy'; you would need to have a Softmax output layer to use this

2.5. Gradient-based learning

Reminder: Gradient Descent

- Define total loss as $\mathcal{L} = -\sum_{(\mathbf{x},y)\in\mathbf{D}} \ell(g(\mathbf{x},\boldsymbol{\theta}),y)$ for some loss function ℓ, dataset $\mathbf{D}$ and model g with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the **total loss** $\mathcal{L}$ by the learning rate η multiplied by the gradient: [1em] `for each Epoch:` $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}$

Reminder: Stochastic Gradient Descent

- Define loss function ℓ, dataset $\mathbf{D}$ and model g with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Stochastic Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the loss of a **single item** ℓ by the learning rate η multiplied by the gradient: [1em] `for each Epoch:` `for each` $(\mathbf{x},y) \in \mathbf{D}$`:` $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\ell$

2.6. The programming perspective: Tensors and Vectorisation

Now we've recapped some of the basics of learning machines we turn our attention to programming, and translating these ideas and algorithms into code that can be executed. Most importantly however, we are going to focus on writing efficient code that makes full use of modern computing hardware. We will not go

into the gritty details of writing machine code, but rather focus on a higher-level mathematical and programming abstraction using objects called *tensors* that let us write and construct differentiable programs with ease.

Throughout this book we utilise a tensor programming framework called `PyTorch` which interfaces with the `python` programming language. `PyTorch` is just one of a number of popular tensor programming frameworks, and you might have heard of others such as `TensorFlow` or `Jax`. These frameworks take slightly different approaches but broadly offer the same feature set. We predominantly use `PyTorch` in this book as it is probably the most popular framework for differentiable programming and deep learning research at the time of writing and is what we use in our research lab. Most importantly though, `PyTorch` lets us easily write a wide range of differentiable programs that go beyond simple deep learning models as we'll see later on.

2.6.1. Tensors, operations and views

Broadly speaking a tensor is defined as a linear mapping between sets of algebraic objects[5].

A tensor T can be thought of as a generalization of scalars, vectors and matrices to a single algebraic object.

We can just think of this as a multidimensional array[6].

- A $0D$ tensor is a scalar
- A $1D$ tensor is a vector
- A $2D$ tensor is a matrix
- A $3D$ tensor can be thought of as a vector of identically sized matrices
- A $4D$ tensor can be thought of as a matrix of identically sized matrices or a sequence of $3D$ tensors
- ...

[5]This statement is always entirely true

[6]This statement will upset mathematicians and physicists because its not always true for them (but it is for us!).

Operations

Element-wise operations.

Batch-wise operations and broadcasting.

Tensor-wise operations.

Einstein summation.

Viewing and reshaping tensors

2.6.2. Working with Batches of Data

2.6.3. Superscalar Compute and Code Vectorisation

Example: vectorised logistic regression

2.6.4. Tensor Tricks

Operations on Tensors in PyTorch

- PyTorch lets you do all the standard matrix operations on 2D tensors
  - including important things you might not yet have seen like the hadamard product of two $N \times M$ matrices: $\mathbf{A} \odot \mathbf{B}$)
- You can do element-wise add/divide/subtract/multiply to ND-tensors
  - and even apply scalar functions element-wise (log, sin, exp, ...)
- PyTorch often lets you *broadcast* operations (just like in numpy)
  - if a PyTorch operation supports broadcast, then its Tensor arguments can be automatically expanded to be of equal sizes (without making copies of the data).[7]

[7] Important - read and understand this after the lab next week: https://pytorch.org/docs/stable/notes/broadcasting.html

Homework

PyTorch Tensor 101:
https://colab.research.google.com/gist/jonhare/d98813b2224dddbb234d2031510878e1/notebook.ipynb

s

3. The Power of Differentiation

Clearly differentiable programming involves the process of computing derivatives and estimating gradients. This chapter will start by recapping the fundamentals of gradient-based optimisation, differentiation and partial differentiation. We'll look at how differentiation can be extended to functions involving operations on matrices and tensors, and the notions of gradient vectors and matrices of derivatives (the Jacobian).

The chapter ends by walking through two practical applications in which these concepts can be applied to learn predictive models from actual data: the linear soft-margin support vector machine, and a neat approach to computing the Singular Value Decomposition and performing matrix completion that was the basis of the collaborative filtering based recommender system[43] that won the *Netflix Prize*.

[43] A recommender system is a piece of software that attempts to predict ratings or rankings of items. In collaborative filtering, the system makes predictions about the interests (the 'items') of a particular user based on the recorded preferences of other users of the system.

3.1. The big idea: optimisation by following gradients

- Fundamentally, we're interested in machines that we train by optimising parameters
  - How do we select those parameters?
- In deep learning/differentiable programming we typically define an objective function that we *minimise* (or *maximise*) with respect to those parameters
- This implies that we're looking for points at which the gradient of the objective function is zero w.r.t the parameters
- Gradient based optimisation is a *big* field!

  - First order methods, second order methods, subgradient methods...

- With deep learning we're primarily interested in first-order methods[1].
  - Primarily using variants of gradient descent: a function $F(\mathbf{x})$ has *a* minima[2] at a point $\mathbf{x} = \mathbf{a}$ where $\mathbf{a}$ is given by applying $\mathbf{a}_{n+1} = \mathbf{a}_n - \alpha\nabla F(\mathbf{a}_n)$ until convergence from some initial point $\mathbf{a}_0$.

3.2. Recap: what are gradients and how do we find them?

3.2.1. The derivative in 1D

- Recall that the gradient of a straight line is $\frac{\Delta y}{\Delta x}$.
- For an arbitrary real-valued function, $f(a)$, we can approximate the derivative, $f'(a)$ using the gradient of the *secant line* defined by $(a, f(a))$ and a point a small distance, h, away $(a+h, f(a+h))$: $f'(a) \approx \frac{f(a+h)-f(a)}{h}$.
  - This expression is 'Newton's Quotient'.
  - As h becomes smaller, the approximated derivative becomes more accurate.
  - If we take the limit as $h \to 0$, then we have an exact expression for the derivative: $\frac{df}{da} = f'(a) = \lim_{h\to 0} \frac{f(a+h)-f(a)}{h}$.

3.2.2. Numerical approximation of the derivative

Clearly one can estimate a derivative numerically by evaluating the difference of a function at two points that are close together. Rather than using the Newton's quotient, for numerical computation of derivatives it is better to use a *centralised* definition of the derivative, the *symmetric derivative*,

$$f'(a) = \lim_{h\to 0} \frac{f(a+h) - f(a-h)}{2h} .$$

[1] Second order gradient optimisers are potentially better, but for systems with many variables are currently impractical as they require computing the Hessian.
[2] not necessarily global or unique

The expression under this limit is known as the *symmetric difference quotient*. Functions that are differentiable (in the usual sense of Newton's quotient) are also symmetrically differentiable, however not all functions that are symmetrically differentiable are differentiable. A common example would be $f(x) = |x|$, which is symmetrically differentiable with a symmetric derivative of 0 at $x = 0$, but is not differentiable because the derivative at 0 is undefined[44].

[44] This is an example of a function that is *differentiable almost everywhere*. Such functions are common components in differentiable programming, usually also with the condition that they are *continuous* functions.

For small values of h the symmetric difference quotient has *less error* than the standard one-sided difference quotient. To understand this we need to think about two sources of error: the truncation error from the difference quotient expressions and the roundoff error from using floating-point number representations. The truncation error of the derivative can be analysed by considering the Taylor series expansion of the Newton's Quotient,

$$\begin{aligned}\frac{f(x+h)-f(x)}{h} &= \frac{f(x)+hf'(x)+\frac{h^2}{2}f''(x)+\cdots-f(x)}{h}\\ &= f'(x)+\frac{hf''(x)}{2}+O(h^2)\end{aligned}$$

Thus the truncation error is approximately $\frac{hf''(x)}{2}$. Now consider the expansion of the symmetric difference quotient,

$$\begin{aligned}\frac{f(x+h)-f(x-h)}{2h} &= \frac{f(x)+hf'(x)+\frac{h^2}{2}f''(x)+\cdots-f(x)+hf'(x)-\frac{h^2}{2}f''(x)+\ldots}{2h}\\ &= f'(x)+\frac{h^2f'''(x)}{3!}+O(h^3)\end{aligned}$$

This time the error term is approximately $\frac{h^2f'''(x)}{6}$. Because h is positive and $h \ll 1$, it follows that $h^2 \ll h$, so we expect there to be lower error with the symmetric difference quotient.

Obviously smaller values of h will give estimates that are closer to the true derivative of the function in question, however the second type of error, the roundoff error, comes into play: if h is chosen to be too small there will be a large rounding error. Assume that because of our finite precision floating point representations that there is a rounding error of ϵ in representing the value of $f(x)$. Consider the worst case scenario of the rounding error of $f(x)$ and error of $f(x+h)$ having the same magnitude but opposite sign; in this case the rounding error in representing $f'(x)$ would be $\frac{2\epsilon}{h}$. We can write the total error (the truncation error plus the rounding error) for approximations using Newton's

quotient as

$$E(h) = Kh + \frac{2\epsilon}{h} ,$$

where K is a constant made up of the second derivative and divisor. Clearly as $h \to 0$ the truncation error goes to zero, however at the same time the roundoff error goes to infinity. To find the optimal value of h that minimises the error we can differentiate, set to zero and rearrange for the optimal h, which we will call $h_0^{(newton)}$:

$$\frac{\mathrm{d}E(h)}{\mathrm{d}h} = K - \frac{2\epsilon}{h^2} = 0$$

$$\implies h_0^{(newton)} = \sqrt{\frac{2\epsilon}{K}} .$$

If we assume[45] that $\sqrt{\frac{2}{K}} \approx 1$ and that $\epsilon \approx 10^{-n}$, then $h_0^{(newton)} \approx 10^{-\frac{n}{2}}$. This implies the error would be,

$$E(h_0^{(newton)}) \approx K \cdot 10^{-\frac{n}{2}} + \frac{2 \cdot 10^{-n}}{10^{-\frac{n}{2}}} \approx (K+2)10^{-\frac{n}{2}} \approx 10^{-\frac{n}{2}} .$$

The take-away message from this is that if our computer hardware can give us number representations with n digits of precision, then using Newton's quotient we can expect to get about $\frac{n}{2}$ digits correct.

If we follow the same principle to derive an optimal $h_0^{(symmetric)}$ for the symmetric difference quotient, we arrive at

$$E(h) = \tilde{K}h^2 + \frac{\epsilon}{h} , \text{ and,}$$

$$h_0^{(symmetric)} = \sqrt[3]{\frac{\epsilon}{2\tilde{K}}} .$$

Assuming $\sqrt[3]{\frac{1}{2\tilde{K}}} \approx 1$, and that $\epsilon \approx 10^{-n}$ as before, then the error is,

$$E(h_0^{(symmetric)}) \approx \tilde{K} \cdot 10^{\frac{-2n}{3}} + \frac{10^{-n}}{10^{\frac{-n}{3}}} \approx (\tilde{K}+1) \cdot 10^{\frac{-2n}{3}} \approx 10^{\frac{-2n}{3}} .$$

This implies that for the same hardware precision as above we would expect to get about $\frac{2n}{3}$ digits correct with the symmetric difference quotient, and thus the symmetric quotient is a much better choice in terms of minimising the total error.

The above analysis looks at the magnitude of error in terms of correct digits. In digital computer hardware this isn't something we have exact control over because of the way floating point

[45] Note that the optimal h_0 is a function of the second derivative of the function (which is wrapped up in the K term). Clearly the implication of this is that if we want to actually use the optimal step size for computing the derivatives of a function we already need to know something about the function which in general is not possible, hence the need to make assumptions and approximations.

numbers are represented in binary. The most common representations of floating point numbers are 32-bit *single precision*, and 64-bit *double precision* defined by the IEEE 754 standard[46]. These floating point representations actually have a different *number of digits* precision for different numbers. Broadly speaking the double precision format (approx 15.95 correct digits) has slightly over twice the number of digits of accuracy than the single format (approx 7.22 correct digits). In terms of *good* values for the step size h when the function is unknown, for the symmetric difference formula a value of $h^* = \sqrt[3]{\epsilon}$, where ϵ is the machine epsilon[47] for the particular floating point representation and hardware combination being used. For Newton's quotient $f(x+h) - f(x)$, a value of $h^* = \sqrt{\epsilon}x$ is appropriate if x is not zero.

[46] There is also a *half precision* (usually 16 bits), which is sometimes used in differentiable programming (but not with numerical differentiation as the errors would be too high), and *long double precision* (typically 80 bits or 128 bits depending on the hardware architecture) which is often used for things like high-precision physics simulations.

[47]

Finally, a further, a problem particularly with the single precision format is that if even if x is a *representable floating point number* (that is the rounding error is zero), then $x + h$ almost certainly is not representable and will be rounded to the nearest representable number. The implication of this is that $(x + h) - x$ will not be h, so this needs to be accounted for in the difference quotient. Unfortunately compiler optimisations based on the axioms of arithmetic, where the compiler recognises that mathematically $(x + h) - x$ should *equal* h, can actually make this hard to implement correctly.

To put this discussion of the optimal step size and the trade-off in errors into context, we can actually simulate the computation of numerical gradients of a function for which we know the true derivative with code, and plot the total error. The following code example does just this for the function $y = x^3$ using both Newton's quotient and the symmetric difference with both single and double precision.

```
1 import numpy as np
2 import pandas as pd
3 import seaborn as sb
4
5
6 def f(x):
7   return x**3
8
9 def fprime(x):
10  return 3*x**2
11
12 def newton(fcn, x, h=0.0001):
13  return (fcn(x + h) - fcn(x)) / h
14
15 def symmetric(fcn, x, h=0.0001):
16  return (fcn(x + h) - fcn(x - h)) / (2 * h)
```

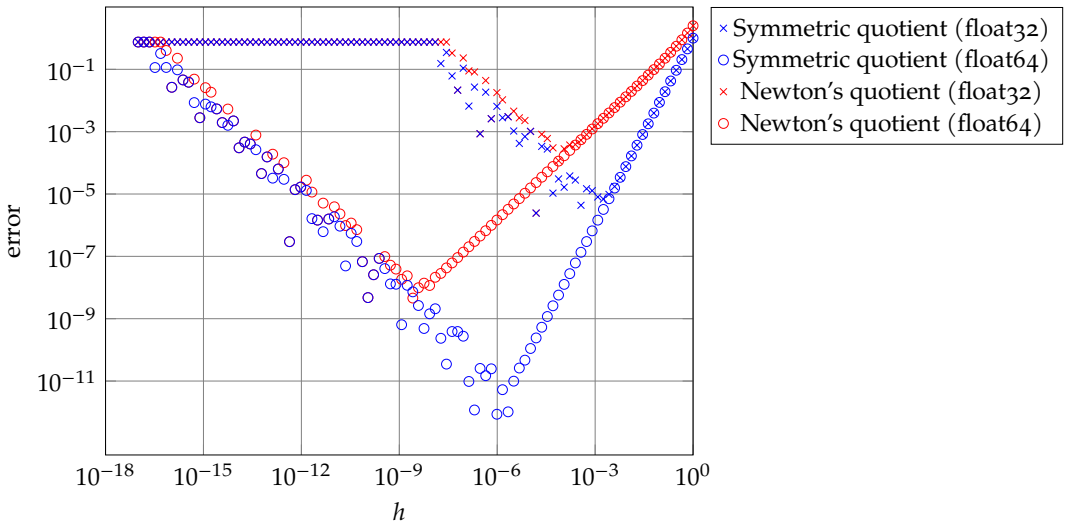


Figure 3.1: Error tradeoff in numerically estimating the derivative of $y = x^3$ as a function of the step h in the difference quotient. The competing effects of the truncation error which reduces as $h \to 0$ (the *tight* linear relationship on the right side) and the roundoff error which decreases as h increases (the *noisy* linear relationship on the left) can be clearly seen. Use of the symmetric difference quotient results in lower error given the optimal step size. Error with 32-bit floating point representations, even with optimal h, is likely to be too great to rely on for real applications.

```
17
18 data = {'h':[], 'dtype': [], 'algorithm':[], 'error':[]}
19
20 for dtype in ['float32', 'float64']:
21   for algorithm in [newton, symmetric]:
22     x = np.array([0.5], dtype=dtype)
23     h = np.logspace(-17, -0, 100, dtype=dtype)
24
25     estimate = algorithm(f, x, h)
26     data['h'].append(h)
27     data['error'].append(np.abs(estimate - fprime(x)))
28     data['algorithm'].append([algorithm.__name__] * len(
       estimate))
29     data['dtype'].append([dtype] * len(estimate))
30
31 df = pd.DataFrame({k: np.hstack(v) for k, v in data.items
       ()})
32
33 splot = sb.scatterplot(data=df, x='h', y='error',
34                        hue='dtype', style='algorithm')
35 splot.set(xscale='log', yscale='log')
```

Running the above code results in the log-log plot illustrated in fig. 3.1. One can clearly see the trade-off in the dominance of the two types of error, with the rounding errors on the left sides of the trend and truncation errors on the right. The significant effect of the choice of floating point precision and approximation

algorithm is also evident.

3.2.3. Vector functions

Now we have a good understanding of the derivatives of scalar-valued functions, lets next consider the gradients of functions that return *vectors*, such as $\boldsymbol{y}(t)$. Such functions can be split into their constituent *coordinate functions*: $\boldsymbol{y}(t) = (y_1(t), \dots, y_n(t))$.

- – This can be split into its constituent coordinate functions: $\mathbf{y}(t) = (y_1(t), \dots, y_n(t))$.
  - – Thus the derivative is a vector (the 'tangent vector'), $\mathbf{y}'(t) = (y_1'(t), \dots, y_n'(t))$, which consists of the derivatives of the coordinate functions.
  - – Equivalently, $\mathbf{y}'(t) = \lim_{h\to 0} \frac{\mathbf{y}(t+h)-\mathbf{y}(t)}{h}$ if the limit exists.

Recap: what are gradients and how do we find them?

Functions of multiple variables: partial differentiation

- What if the function we're trying to deal with has multiple variables[3] (e.g. $f(x, y) = x^2 + xy + y^2$)?
  - – This expression has a pair of *partial derivatives*, $\frac{\partial f}{\partial x} = 2x + y$ and $\frac{\partial f}{\partial y} = x + 2y$, computed by differentiating with respect to each variable x and y whilst holding the other(s) constant.
- In general, the partial derivative of a function $f(x_1, \dots, x_n)$ at a point $(a_1, \dots, a_n)$ is given by: $\frac{\partial f}{\partial x_i}(a_1, \dots, a_n) = \lim_{h\to 0} \frac{f(a_1\dots,a_i+h,\dots,a_n)-f(a_1\dots,a_i,\dots,a_n)}{h}$.
- The vector of partial derivatives of a scalar-value multivariate function, $f((x_1, \dots, x_n)$ at a point $(a_1, \dots, a_n)$, can be arranged into a vector: $\nabla f(a_1, \dots, a_n) = (\frac{\partial f}{\partial x_1}(a_1, \dots, a_n), \dots, \frac{\partial f}{\partial x_n}(a_1, \dots, a_n))$.
  - – This is the **gradient** of f at a.
- In the case of a vector-valued multivariate function, the partial derivatives form a matrix called the **Jacobian**.

[3] A multivariate function

Recap: what are gradients and how do we find them?
Functions of vectors and matrices: partial differentiation

- For the kinds of functions (and programs) that we'll look at *optimising* in this course have a number of typical properties:
  - They are scalar-valued
    - We'll look at programs with *multiple losses*, but ultimately we can just consider optimising with respect to the *sum* of the losses.
  - They involve multiple variables, which are often wrapped up in the form of vectors or matrices, and more generally tensors.
  - **How will we find the gradients of these?**

Recap: what are gradients and how do we find them?
The chain rule for vectors

Suppose that $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and f maps from $\mathbb{R}^n$ to $\mathbb{R}$.

If $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}.$$

Equivalently, in vector notation:

$$\nabla_{\mathbf{x}} z = (\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^\top \nabla_{\mathbf{y}} z$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian matrix of g.

Recap: what are gradients and how do we find them?
The chain rule for Tensors

- Conceptually, the simplest way to think about gradients of tensors is to imagine flattening them into vectors, computing the vector-valued gradient and then reshaping the gradient back into a tensor.
  - In this way we're still just multiplying Jacobians by gradients.
- More formally, consider the gradient of a scalar z with respect to a tensor $\mathbf{X}$ to be denoted as $\nabla_{\mathbf{X}} z$.

- Indices into $\mathbf{X}$ now have multiple coordinates, but we can generalise by using a single variable i to represent the complete tuple of indices.
  - For all index tuples i, $(\nabla_{\mathbf{X}} z)_i$ gives $\frac{\partial z}{\partial \mathbf{X}_i}$.
- Thus, if $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$ then $\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} \mathbf{Y}_j) \frac{\partial z}{\partial \mathbf{Y}_j}$.

Recap: what are gradients and how do we find them?

Example: $\nabla_{\mathbf{W}} f(\mathbf{XW})$

- Let $\mathbf{D} = \mathbf{XW}$ where the rows of $\mathbf{X} \in \mathbb{R}^{n \times m}$ contain some fixed *features*, and $\mathbf{W} \in \mathbb{R}^{m \times h}$ is a matrix of weights.
- Also let $\mathcal{L} = f(\mathbf{D})$ be some scalar function of $\mathbf{D}$ that we wish to minimise.
- What are the derivatives of $\mathcal{L}$ with respect to the weights $\mathbf{W}$?

Recap: what are gradients and how do we find them?

Example: $\nabla_{\mathbf{W}} f(\mathbf{XW})$

- Start by considering a specific weight, W_{uv}: $\frac{\partial \mathcal{L}}{\partial W_{uv}} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial D_{ij}} \frac{\partial D_{ij}}{\partial W_{uv}}$.
- We know that $\frac{\partial D_{ij}}{\partial W_{uv}} = 0$ if $j \neq v$ because D_{ij} is the dot product of row i of $\mathbf{X}$ and column j of $\mathbf{W}$.
- Therefore, we can simplify the summation to only consider cases where $j = v$: $\sum_{i,j} \frac{\partial \mathcal{L}}{\partial D_{ij}} \frac{\partial D_{ij}}{\partial W_{uv}} = \sum_i \frac{\partial \mathcal{L}}{\partial D_{iv}} \frac{\partial D_{iv}}{\partial W_{uv}}$.
- What is $\frac{\partial D_{iv}}{\partial W_{uv}}$?

$$
\begin{aligned}
D_{iv} &= \sum_{k=1}^{q} X_{ik} W_{kv} \\
\frac{\partial D_{iv}}{\partial W_{uv}} &= \frac{\partial}{\partial W_{uv}} \sum_{k=1}^{q} X_{ik} W_{kv} = \sum_{k=1}^{q} \frac{\partial}{\partial W_{uv}} X_{ik} W_{kv} \\
\therefore \frac{\partial D_{iv}}{\partial W_{uv}} &= X_{iu}
\end{aligned}
$$

Recap: what are gradients and how do we find them?

Example: $\nabla_{\mathbf{W}} f(\mathbf{XW})$

- Putting every together, we have: $\frac{\partial \mathcal{L}}{\partial W_{uv}} = \sum_i \frac{\partial \mathcal{L}}{\partial D_{iv}} X_{iu}$.
- As we're summing over multiplications of scalars, we can change the order: $\frac{\partial \mathcal{L}}{\partial W_{uv}} = \sum_i X_{iu} \frac{\partial \mathcal{L}}{\partial D_{iv}}$.
- and note that the sum over i is doing a dot product with row u and column v if we transpose X_{iu} to $X^\top_{ui}$: $\frac{\partial \mathcal{L}}{\partial W_{uv}} = \sum_i X^\top_{ui} \frac{\partial \mathcal{L}}{\partial D_{iv}}$.
- We can then see that if we want this for all values of $\mathbf{W}$ it simply generalises to: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{X}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{D}}$.

Recap: Singular Value Decomposition and its applications

Let's now change direction - we're going to look at an early success story resulting from using some differentiation and the Singular Value Decomposition (SVD). [1em] For complex $\mathbf{A}$:

$$\mathbf{A} = \mathbf{U\Sigma V}^*$$

where $\mathbf{V}^*$ is the *conjugate transpose* of $\mathbf{V}$.[1em] For real $\mathbf{A}$:

$$\mathbf{A} = \mathbf{U\Sigma V}^\top$$

Recap: Singular Value Decomposition and its applications

- SVD has many uses:
  - Computing the Eigendecomposition:
    - Eigenvectors of $\mathbf{AA}^\top$ are columns of $\mathbf{U}$,
    - Eigenvectors of $\mathbf{A}^\top\mathbf{A}$ are columns of $\mathbf{V}$,
    - and the non-zero values of $\mathbf{\Sigma}$ are the square roots of the non-zero eigenvalues of both $\mathbf{AA}^\top$ and $\mathbf{A}^\top\mathbf{A}$.
  - Dimensionality reduction
    - ...use to compute PCA
  - Computing the Moore-Penrose Pseudoinverse
    - for real $\mathbf{A}$: $\mathbf{A}^+ = \mathbf{V\Sigma}^+\mathbf{U}^\top$ where $\mathbf{\Sigma}^+$ is formed by taking the reciprocal of every non-zero diagonal element and transposing the result.

  - Low-rank approximation and matrix completion
    * if you take the ρ columns of $\mathbf{U}$, and the ρ rows of $\mathbf{V}^\top$ corresponding to the ρ largest singular values, you can form the matrix $\mathbf{A}_\rho = \mathbf{U}_\rho \mathbf{\Sigma}_\rho \mathbf{V}_\rho^\top$ which will be the *best* rank-ρ approximation of the original $\mathbf{A}$ in terms of the Frobenius norm.

Example: Computing SVD using gradients - The Netflix Challenge

- There are many standard ways of computing the SVD:
  - e.g. 'Power iteration', or 'Arnoldi iteration' or 'Lanczos algorithm' coupled with the 'Gram-Schmidt process' for orthonormalisation
- but, these don't necessarily scale up to really big problems
  - e.g. computing the SVD of a sparse matrix with 17770 rows, 480189 columns and 100480507 non-zero entries!
  - this corresponds to the data provided by Netflix when they launched the *Netflix Challenge* in 2006.
- OK, so what can you do?
  - The 'Simon Funk' solution: realise that there is a really simple (and quick) way to compute the SVD by following gradients...

Example: Computing SVD using gradients - The Netflix Challenge
Deriving a gradient-descent solution to SVD

- One of the definitions of rank-ρ SVD of a matrix $\mathbf{A}$ is that it minimises reconstruction error in terms of the Frobenius norm.
- Without loss of generality we can write SVD as a 2-matrix decomposition $\mathbf{A} = \hat{\mathbf{U}}\hat{\mathbf{V}}^T$ by rolling in the square roots of $\mathbf{\Sigma}$ to both $\hat{\mathbf{U}}$ and $\hat{\mathbf{V}}$: $\hat{\mathbf{U}} = \mathbf{U}\mathbf{\Sigma}^{0.5}$ and $\hat{\mathbf{V}}^\top = \mathbf{\Sigma}^{0.5}\mathbf{V}^\top$.
- Then we can define the decomposition as finding $\min_{\hat{\mathbf{U}},\hat{\mathbf{V}}}(\|\mathbf{A} - \hat{\mathbf{U}}\hat{\mathbf{V}}^\top\|_F^2)$

Example: Computing SVD using gradients - The Netflix Challenge
Deriving a gradient-descent solution to SVD

Start by expanding our optimisation problem:

$$\begin{aligned}\min_{\hat{\mathbf{U}},\hat{\mathbf{V}}}(\|\mathbf{A}-\hat{\mathbf{U}}\hat{\mathbf{V}}^\top\|_F^2) &= \min_{\hat{\mathbf{U}},\hat{\mathbf{V}}}(\sum_r\sum_c(A_{rc}-\hat{U}_r\hat{V}_c)^2)\\ &= \min_{\hat{\mathbf{U}},\hat{\mathbf{V}}}(\sum_r\sum_c(A_{rc}-\sum_{p=1}^{\rho}\hat{U}_{rp}\hat{V}_{cp})^2)\end{aligned}$$

Let $e_{rc} = A_{rc} - \sum_{p=0}^{\rho}\hat{U}_{rp}\hat{V}_{cp}$ denote the error. Then, our problem becomes:

$$\text{Minimise } J = \sum_r\sum_c e_{rc}^2$$

We can then differentiate with respect to specific variables $\hat{U}_{rq}$ and $\hat{V}_{cq}$

Example: Computing SVD using gradients - The Netflix Challenge
Deriving a gradient-descent solution to SVD

We can then differentiate with respect to specific variables $\hat{U}_{rq}$ and $\hat{V}_{cq}$:

$$\begin{aligned}\frac{\partial J}{\partial \hat{U}_{rq}} &= \sum_r\sum_c 2e_{rc}\frac{\partial e}{\partial \hat{U}_{rq}} = -2\sum_r\sum_c \hat{V}_{cq}e\\ \frac{\partial J}{\partial \hat{V}_{cq}} &= \sum_r\sum_c 2e_{rc}\frac{\partial e}{\partial \hat{V}_{cq}} = -2\sum_r\sum_c \hat{U}_{rq}e\end{aligned}$$

and use this as the basis for a gradient descent algorithm:

$$\begin{aligned}\hat{U}_{rq} &\Leftarrow \hat{U}_{rq} + \lambda\sum_r\sum_c \hat{V}_{cq}e_{rc}\\ \hat{V}_{cq} &\Leftarrow \hat{V}_{cq} + \lambda\sum_r\sum_c \hat{U}_{rq}e_{rc}\end{aligned}$$

Example: Computing SVD using gradients - The Netflix Challenge
Deriving a gradient-descent solution to SVD

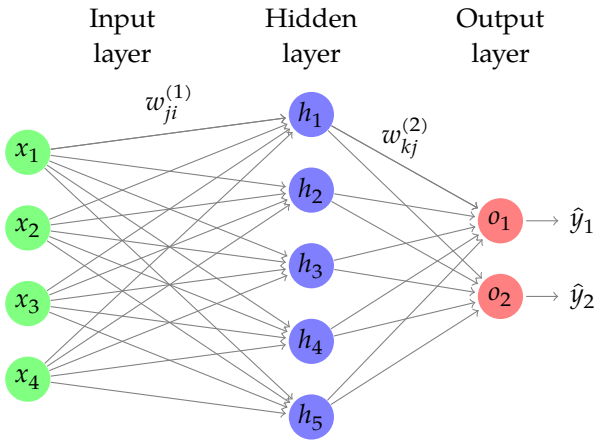
- A stochastic version of this algorithm (updates on one single item of $\mathbf{A}$ at a time) helped win the Netflix Challenge competition in 2009.
- It was both *fast* and *memory efficient*

4. Perceptrons, MLPs and Backpropagation

Topics

- A quick look at an MLP again
- The chain rule (again)
- Uninititive gradient effects
- A closer look at basic stochastic gradient descent algorithms

The unbiased Multilayer Perceptron (again)...



Without loss of generality, we can write the above as:

$$\hat{\mathbf{y}} = g(f(\mathbf{x}; \mathbf{W}^{(1)}); \mathbf{W}^{(2)}) = g(\mathbf{W}^{(2)} f(\mathbf{W}^{(1)} \mathbf{x}))$$

where f and g are activation functions.

Gradients of our simple unbiased MLP

- Let's assume MSE Loss

$$\ell_{MSE}(\hat{\mathbf{y}}, \mathbf{y}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

- What are the gradients?

$$\nabla_{\mathbf{W}^*} \ell_{MSE}(g(\mathbf{W}^{(2)} f(\mathbf{W}^{(1)} \mathbf{x})), \mathbf{y})$$

- Clearly we need to apply the chain rule (vector form) multiple times
- We could do this by hand
- (But we're not that crazy!)

Let's go back to a simpler expression

$$\begin{aligned} f(x, y, z) &= (x + y)z \\ &\equiv qz \text{ where } q = (x + y) \end{aligned}$$

Clearly the partial derivatives of the subexpressions are trivial:

$$\begin{aligned} \partial f / \partial z = q \qquad \partial f / \partial q = z \\ \partial q / \partial x = 1 \qquad \partial q / \partial y = 1 \end{aligned}$$

and the chain rule tells us how to combine these:

$$\begin{aligned} \partial f / \partial x = \partial f / \partial q \cdot \partial q / \partial x = z \\ \partial f / \partial y = \partial f / \partial q \cdot \partial q / \partial y = z \end{aligned}$$

so $\nabla_{[x,y,z]} f = [z, z, q]$

A computational graph perspective

$$f(x, y, z) = (x + y)z$$

An intuition of the chain rule

- Notice how every operation in the computational graph given its inputs can immediately compute two things:
  1. its output value
  2. the *local* gradient of its inputs with respect to its output value
- The chain rule tells us literally that each operation should take its local gradients and multiply them by the gradient that *flows* backwards into it

This is backpropagation

- The backprop algorithm is just the idea that you can perform the forward pass (computing and caching the local gradients as you go),
- and then perform a backward pass to compute the total gradient by applying the chain rule and re-utilising the cached local gradients
- Backprop is just another name for 'Reverse Mode Automatic Differentiation'...

Unintuitive effects I: Multiplication

- Consider the multiplication operation $f(a, b) = a * b$.
- The gradients are clearly $\partial f / \partial b = a$ and $\partial f / \partial a = b$.
  - (in a computational graph these would be the local gradients w.r.t the inputs)
- If a is large and b is tiny the gradient assigned to b will be large, and the gradient to a small.
- This has implications for e.g. linear classifiers ($\mathbf{w}^\top \mathbf{x}_i$) where you perform many multiplications
  - the magnitude of the gradient is directly proportional to the magnitude of the data
  - multiply $\mathbf{x}_i$ by 1000, and the gradients also increase by 1000

  - if you don't lower the learning rate to compensate your model might not learn
  - **Hence you need to always pay attention to data normalisation!**

Unintuitive effects II: vanishing gradients of the sigmoid

- It used to be popular to use sigmoids (or tanh) in the hidden layers...
- Gradient of $\sigma(x) = \sigma(x)(1 - \sigma(x))$
- Thus as part of a larger network where this is the local gradient, if x is large (+ve or -ve), then all gradients backwards from this point will be zero due to multiplication of the chain rule
  - Why might x be large?
- Maximum gradient is achieved when $x = 0 \quad (\sigma(x) = 0.5,\ dx = 0.25)$
  - This means that the maximum gradient that can flow out of a sigmoid will be a quarter of the input gradient
    * What's the implication of this in a deep network with sigmoid activations?

Unintuitive effects III: dying ReLUs

- Modern networks tend to use ReLUs
- Gradient is 1 for $x > 0$ and 0 otherwise
- Consider ReLU($\mathbf{w}^\top \mathbf{x}$)
  - What happens if $\mathbf{w}$ is initialised badly?
  - What happens if $\mathbf{w}$ receives an update that means that $\mathbf{w}^\top \mathbf{x} < 0 \ \forall \ \mathbf{x}$?
- These are dead ReLUs - ones that never fire for all training data
  - Sometimes you can find that you have a large fraction of these
  - if you get them from the beginning, check weight initialisation and data normalisation
  - if they're appearing during training, maybe η is too big?

Unintuitive effects IV: Exploding gradients in recurrent networks

- Recurrent networks apply a function recursively for some number of timesteps
- Often this recursion involves a multiplication at each timestep, the gradients of which are all multiplied together because of the chain rule...
- Consider $z = a \prod_n^\infty b$
  - $z \to 0$ *if* $|b| < 1$
  - $z \to \infty$ *if* $|b| > 1$
- Same thing happens in the backward pass of an RNN (although with matrices rather than scalars, so the reasoning applies to the largest eigenvalue)

5. Automatic Differentiation

What is Automatic Differentiation (AD)?

To solve optimisation problems using gradient methods we need to compute the gradients (derivatives) of the objective with respect to the parameters.

- In neural nets we're talking about the gradients of the loss function, $\mathcal{L}$ with respect to the parameters $\boldsymbol{\theta}$: $\nabla_{\boldsymbol{\theta}}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$
- AD is important - it's been suggested that "Differentiable programming" could be the term that ultimately replaces deep learning[1].

What is Automatic Differentiation (AD)?

Computing Derivatives

There are three ways to compute derivatives:

- Symbolically differentiate the function with respect to its parameters
  - by hand
  - using a CAS
- Make estimates using finite differences
- Use Automatic Differentiation

Problems

Static - can't "differentiate algorithms"

Problems

Numerical errors - will compound in deep nets

[1] http://forums.fast.ai/t/differentiable-programming-is-this-why-we-switched-to-pytorch/9589/5

What is Automatic Differentiation (AD)?

Automatic Differentiation is:

- a method to get exact derivatives efficiently, by storing information as you go forward that you can reuse as you go backwards.
  - Takes code that computes a function and uses that to compute the derivative of that function.
  - The goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

Lets think about differentiation and programming

[
Math]

$$x = ?$$
$$y = ?$$
$$a = x\,y$$
$$b = \sin(x)$$
$$z = a + b$$

[
Code]

```
x = ?
y = ?
a = x * y
b = sin(x)
z = a + b

```

The Chain Rule of Differentiation

Recall the chain rule for a variable/function z that depends on y which depends on x:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

In general, the chain rule can be expressed as:

$$\frac{\partial w}{\partial t} = \sum_{i}^{N} \frac{\partial w}{\partial u_i}\frac{\partial u_i}{\partial t} = \frac{\partial w}{\partial u_1}\frac{\partial u_1}{\partial t} + \frac{\partial w}{\partial u_2}\frac{\partial u_2}{\partial t} + \cdots + \frac{\partial w}{\partial u_N}\frac{\partial u_N}{\partial t}$$

where w is some output variable, and u_i denotes each input variable w depends on.

Applying the Chain Rule

Let's differentiate our previous expression with respect to some yet to be given variable t:

Expression

$$
\begin{aligned}
x &= ? \\
y &= ? \\
a &= x\,y \\
b &= \sin(x) \\
z &= a + b
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial x}{\partial t} &= ? \\
\frac{\partial y}{\partial t} &= ? \\
\frac{\partial a}{\partial t} &= x\frac{\partial y}{\partial t} + y\frac{\partial x}{\partial t} \\
\frac{\partial b}{\partial t} &= \cos(x)\frac{\partial x}{\partial t} \\
\frac{\partial z}{\partial t} &= \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}
\end{aligned}
$$

If we substitute $t = x$ in the above we'll have an algorithm for computing $\partial z/\partial x$. To get $\partial z/\partial y$ we'd just substitute $t = y$.

Translating to code I

We could translate the previous expressions back into a program involving *differential variables* `{dx, dy, ...}` which represent $\partial x/\partial t, \partial y/\partial t, \ldots$ respectively:

```
1 dx = ?
2 dy = ?
3 da = y * dx + x * dy
4 db = cos(x) * dx
5 dz = da + db
```

What happens to this program if we substitute $t = x$ into the math expression?

Translating to code II

```
dx = 1
dy = 0
da = y * dx + x * dy
db = cos(x) * dx
dz = da + db
```

The effect is remarkably simple: to compute $\partial z/\partial x$ we just seed the algorithm with `dx=1` and `dy=0`.

Translating to code III

```
dx = 0
dy = 1
da = y * dx + x * dy
db = cos(x) * dx
dz = da + db
```

To compute $\partial z/\partial y$ we just seed the algorithm with `dx=0` and `dy=1`.

Making Rules

- We've successfully computed the gradients for a specific function, but the process was far from automatic.
- We need to formalise a set of rules for translating a program that evaluates an expression into a program that evaluates its derivatives.
- We have actually already discovered 3 of these rules:

```
c = a + b   =>  dc = da + db
c = a * b   =>  dc = b * da + a * db
c = sin(a)  =>  dc = cos(a) * da
```

More rules

These initial rules:

```
c=a+b     =>  dc=da+db
c=a*b     =>  dc=b*da+a*db
c=sin(a)  =>  dc=cos(a)*da
```

can easily be extended further using multivariable calculus:

```
c=a-b     =>  dc=da-db
c=a/b     =>  dc=da/b-a*db/b**2
c=a**b    =>  dc=b*a**(b-1)*da+log(a)*a**b*db
c=cos(a)  =>  dc=-sin(a)*da
c=tan(a)  =>  dc=da/cos(a)**2
```

Forward Mode AD

- To translate using the rules we simply replace each primitive operation in the original program by its differential analogue.
- The order of computation remains unchanged: if a statement K is evaluated before another statement L, then the differential analogue of K is evaluated before the analogue statement of L.
- This is **Forward-mode Automatic Differentiation**.

Interleaving differential computation

A careful analysis of our original program and its differential analogue shows that its possible to interleave the differential calculations with the original ones:

```
1  x  = ?
2  dx = ?
3
4  y  = ?
5  dy = ?
6
7  a  = x * y
8  da = y * dx + x * dy
9
10 b  = sin(x)
11 db = cos(x) * dx
12
13 z  = a + b
14 dz = da + db
```

Dual Numbers

- This implies that we can keep track of the value and gradient at the same time.
- We can use a mathematical concept called a "Dual Number" to create a very simple direct implementation of AD.

Reverse Mode AD

- Whilst Forward-mode AD is easy to implement, it comes with a very big disadvantage...
- **For every variable we wish to compute the gradient with respect to, we have to run the *complete program* again.**

- This is obviously going to be a problem if we're talking about the gradients of a function with very many parameters (e.g. a deep network).

- A solution is **Reverse Mode Automatic Differentiation**.

Reversing the Chain Rule

The chain rule is symmetric — this means we can turn the derivatives upside-down:

$$\frac{\partial s}{\partial u} = \sum_i^N \frac{\partial w_i}{\partial u}\frac{\partial s}{\partial w_i} = \frac{\partial w_1}{\partial u}\frac{\partial s}{\partial w_1} + \frac{\partial w_2}{\partial u}\frac{\partial s}{\partial w_2} + \cdots + \frac{\partial w_N}{\partial u}\frac{\partial s}{\partial w_N}$$

In doing so, we have inverted the input-output role of the variables: u is some input variable, the w_i's are the output variables that depend on u. s is the yet-to-be-given variable. [1em] In this form, the chain rule can be applied repeatedly to every input variable u (akin to how in forward mode we repeatedly applied it to every w). Therefore, given some s we expect this form of the rule to give us a program to compute both $\partial s/\partial x$ and $\partial s/\partial y$ in one go...

Reversing the chain rule: Example

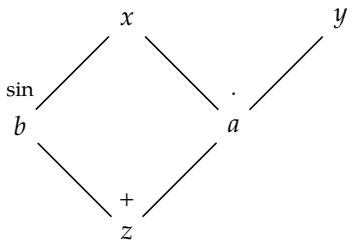
$$\frac{\partial s}{\partial u} = \sum_i^N \frac{\partial w_i}{\partial u}\frac{\partial s}{\partial w_i}$$

$$\begin{aligned} x &= ? \\ y &= ? \\ a &= x\,y \\ b &= \sin(x) \\ z &= a + b \end{aligned}$$

$$
\begin{aligned}
\frac{\partial s}{\partial z} &= ? \\
\frac{\partial s}{\partial b} &= \frac{\partial z}{\partial b}\frac{\partial s}{\partial z} = \frac{\partial s}{\partial z} \\
\frac{\partial s}{\partial a} &= \frac{\partial z}{\partial a}\frac{\partial s}{\partial z} = \frac{\partial s}{\partial z} \\
\frac{\partial s}{\partial y} &= \frac{\partial a}{\partial y}\frac{\partial s}{\partial a} = x\,\frac{\partial s}{\partial a} \\
\frac{\partial s}{\partial x} &= \frac{\partial a}{\partial x}\frac{\partial s}{\partial a} + \frac{\partial b}{\partial x}\frac{\partial s}{\partial b} \\
&= y\,\frac{\partial s}{\partial a} + \cos(x)\,\frac{\partial s}{\partial b} \\
&= (y + \cos(x))\,\frac{\partial s}{\partial z}
\end{aligned}
$$

Visualising dependencies

Differentiating in reverse can be quite mind-bending: instead of asking what input variables an output depends on, we have to ask what output variables a given input variable can affect. [1em] We can see this visually by drawing a dependency graph of the expression: [1em]



Translating to code

Let's now translate our derivatives into code. As before we replace the derivatives ($\partial s/\partial z, \partial s/\partial b, \ldots$) with variables (`gz, gb,` ...) which we call *adjoint variables*:

```
1 gz = ?
2 gb = gz
3 ga = gz
```

```
4 gy = x * ga
5 gx = y * ga + cos(x) * gb
```

If we go back to the equations and substitute $s = z$ we would obtain the gradient in the last two equations. In the above program, this is equivalent to setting `gz = 1`. [1em] **This means to get the both gradients $\partial z/\partial x$ and $\partial z/\partial y$ we only need to run the program once!**

Limitations of Reverse Mode AD

- If we have multiple output variables, we'd have to run the program for each one (with different seeds on the output variables)[2]. For example:

$$\begin{cases} z = 2x + \sin x \\ v = 4x + \cos x \end{cases}$$

- We can't just interleave the derivative calculations (since they all appear to be in reverse)...How can we make this automatic?

Implementing Reverse Mode AD

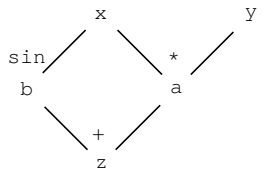
There are two ways to implement Reverse AD:

- We can parse the original program and generate the *adjoint* program that calculates the derivatives.
  - Potentially hard to do.
  - Static, so can only be used to differentiate algorithms that have parameters predefined.
  - But, efficient (lots of opportunities for optimisation)
- We can make a *dynamic* implementation by constructing a graph that represents the original expression as the program runs.

Constructing an expression graph

The goal is to get something akin to the graph we saw earlier:

[2] there are ways to avoid this limitation...



The "roots" of the graph are the independent variables `x` and `y`. Constructing these nodes is as simple as creating an object:

```
class Var:
  def __init__(self, value):
    self.value = value
    self.children = []
    ...
  ...

x = Var(0.5)
y = Var(4.2)
```

Each `Var` node can have *children* which are the nodes that depend directly on that node. The children allow nodes to link together in a **Directed Acyclic Graph**.

Building expressions

By default, nodes do not have any children. As expressions are created each expression u registers itself as a child of each of its dependencies w_i together with its weight $\partial w_i / \partial u$ which will be used to compute gradients:

```
class Var:
  ...
  def __mul__(self, other):
    z = Var(self.value * other.value)

    # weight = dz/dself = other.value
    self.children.append((other.value, z))

    # weight = dz/dother = self.value
    other.children.append((self.value, z))
    return z
  ...
...
# "a" is a new Var that is a child of both x and y
a = x * y
```

Computing gradients

Finally, to get the gradients we need to propagate the derivatives. To avoid unnecessarily traversing the tree multiple times we will *cache* the derivative of a node in an attribute `grad_value`:

```
class Var:
  def __init__(self):
    ...
    self.grad_value = None

  def grad(self):
    if self.grad_value is None:
      # calculate derivative using chain rule
      self.grad_value = sum(weight * var.grad() for weight
    , var in self.children)
    return self.grad_value
  ...
...
a.grad_value = 1.0
print("da/dx = {}".format(x.grad()))
```

Aside: Optimising Reverse Mode AD

- The Reverse AD approach we've outlined is not very space efficient. One way to get around this is to avoid storing the children directly and instead store indices in an auxiliary data structure called a *Wengert list* or *tape*.
- Another interesting approach to memory reduction is trade-off computation for memory of the caches. The Count-Trailing-Zeros (CTZ) approach does just this[3].
- **But,** in reality memory is relatively cheap if managed well...

AD in the PyTorch autograd package

- PyTorch's AD is remarkably similar to the one we've just built:
  - it eschews the use of a tape
  - it builds the computation graph as it runs (recording explicit `Function` objects as the children of `Tensors` rather than grouping everything into `Var` objects)
  - it caches the gradients in the same way we do (in the `grad` attribute) - hence the need to call `zero_grad()` when recomputing the gradients of the same graph after a round of backprop.

[3]Andreas Griewank (1992) Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation, Optimization Methods and Software, 1:1, 35-54, DOI: 10.1080/10556789208805505

- PyTorch does some clever memory management to work well in a reference-counted regime and aggressively frees values that are no longer needed.
- The backend is actually mostly written in C++, so its fast, and can be multi-threaded (avoids problems of the GIL).
- It allows easy "turning off" of gradient computations through `requires_grad`.
- In-place operations which invalidate data needed to compute derivatives will cause runtime errors, as will variable aliasing...

6. Optimisation and some tricks for convergence

Gradient descent and SGD (again), and mini-batch SGD

We'll start up by looking again at gradient descent algorithms and their behaviours...

Reminder: Gradient Descent

- Define total loss as $\mathcal{L} = -\sum_{(\mathbf{x},y)\in\mathbf{D}} \ell(g(\mathbf{x},\boldsymbol{\theta}),y)$ for some loss function ℓ, dataset $\mathbf{D}$ and model g with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the **total loss** $\mathcal{L}$ by the learning rate η multiplied by the gradient: [1em]
`for each Epoch:` $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}$

Gradient Descent

- Gradient Descent has good statistical properties (very low variance)
- But is very data inefficient (particularly when data has many similarities)
- Doesn't scale to effectively infinite data (e.g. with augmentation)

Reminder: Stochastic Gradient Descent

- Define loss function ℓ, dataset $\mathbf{D}$ and model g with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Stochastic Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the loss of a **single item** ℓ by the learning rate η multiplied by the gradient: [1em] `for each Epoch:` `for each` $(\mathbf{x}, y) \in \mathbf{D}$**:** $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \ell$

Stochastic Gradient Descent

- Stochastic Gradient Descent has poor statistical properties (very high variance)
- But is computationally inefficient (poor utilisation of resources - particularly with respect to vectorisation)

Mini-batch Stochastic Gradient Descent

- Define a batch size b
- Define batch loss as $\mathcal{L}_b = -\sum_{(\mathbf{x},y)\in\mathbf{D}_b} \ell(g(\mathbf{x}, \boldsymbol{\theta}), y)$ for some loss function $\ell$and model g with learnable parameters $\boldsymbol{\theta}$. $\mathbf{D}_b$ is a subset of dataset $\mathbf{D}$ of cardinality b.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Mini-batch Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the loss of a **mini-batch** $\mathbf{D}_b$, $\mathcal{L}_b$ by the learning rate η multiplied by the gradient: [1em] partition the dataset $\mathbf{D}$ into an array of subsets of size b `for each Epoch:` `for each` $\mathbf{D}_b \in$ *partitioned*$(\mathbf{D})$**:** $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_b$

Mini-batch Stochastic Gradient Descent

- Mini-batch Stochastic Gradient Descent has reasonable statistical properties (much lower variance than SGD)
- Allows for computationally efficiency (good utilisation of resources)
- Ultimately we would normally want to make our batches as big as possible for lower variance gradient estimates, but:
  - Must still fit in RAM (e.g. on the GPU)
  - Must be able to maintain throughput (e.g. pre-processing on the CPU; data transfer time)

So, what about the learning rate?

- Choice of learning rate is extremely important
- But we have to reason about the 'loss landscape'
  - Most convergence analysis of optimisation algorithms assumes a convex loss landscape
    - Easy to reason about
    - Can be shown that (S)GD will converge to the optimal solution for a variety of learning rates
    - Can give insights into potential problems in the non-convex case
  - Deep Learning is highly non-convex
    - Many local minima
    - Plateaus
    - Saddle points
    - Symmetries (permutation, etc)
    - Certainly no single global minima

**GD in the convex case: failure modes*

Accelerated Gradient Methods

- Accelerated gradient methods use a *leaky* average of the gradient, rather than the instantaneous gradient estimate at each time step
- A physical analogy would be one of the momentum a ball picks up rolling down a hill...
- As you'll see, this helps address the *GD failure modes, but also helps avoid getting stuck in local minima

pause

Momentum I

It's common for the 'leaky' average (the 'velocity', v_t) to be a weighted average of the instantaneous gradient g_t and the past velocity[1]:

$$v_t = \beta v_{t-1} + g_t$$

where $\beta \in [0, 1]$ is the 'momentum'.

[1]There are quite a few variants of this; here we're following the PyTorch variant

Momentum II

- The momentum method allows to accumulate velocity in directions of low curvature that persist across multiple iterations
- This leads to accelerated progress in low curvature directions compared to gradient descent

MB-SGD with Momentum

Learning with momentum on iteration t (batch at t denoted by $b(t)$) is given by:

$$\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} + \nabla_{\boldsymbol{\theta}} \mathcal{L}_{b(t)}$$
$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta \mathbf{v}_t$$

Note $\beta = 0.9$ is a good choice for the momentum parameter.

SGD with Momentum - potentially better convex convergence

Learning rate schedules

- In practice you want to decay your learning rate over time
- Smaller steps will help you get closer to the minima
- But don't do it to early, else you might get stuck

- Something of an art form!
  - 'Grad Student Descent' or GDGS ('Gradient Descent by Grad Student')

Reduce LR on plateau

- Common Heuristic approach:
  - if the loss hasn't improved (within some tolerance) for k epochs
  - then drop the lr by a factor of 10
- Remarkably powerful!

Cyclic learning rates

- Worried about getting stuck in a non-optimal local minima?
- Cycle the learning rate up and down (possibly annealed), with a different lr on each batch
- See `https://arxiv.org/abs/1506.01186`

More advanced optimisers

- Adagrad
  - Decrease learning rate dynamically per weight.
  - Squared magnitude of the gradient (2nd moment) used to adjust how quickly progress is made - weights with large gradients are compensated with a smaller learning rate.
  - Particularly effective for sparse features.
- RMSProp
  - Modifies Adagrad to decouple learning rate from gradient magnitude scaling
  - Incorporates leaky averaging of squared gradient magnitudes
  - LR would typically follow a predefined schedule
- Adam

  - Essentially takes all the best ideas from RMSProp and SDG+Momentum
  - Bias corrected momentum and second moment estimation
  - Shown that it might still diverge (or be non optimal, even in convex settings)...
  - LR is still a hyperparameter (you might still schedule)

Take-away messages

- The loss landscape of a deep network is complex to understand (and is far from convex)
- If you're in a hurry to get results use Adam
- If you have time (or a Grad Student at hand), then use SGD (with momentum) and work on tuning the learning rate
- If you're implementing something from a paper, then follow what they did!

/sectionThe innate biases of optimisation with SGD

7. Deeper Networks: Universal approximation, overfitting and regularisation

No Free Lunch

- Statistical learning theory claims that a machine can generalise well from a finite training set.
- This contradicts basic inductive reasoning which says to derive a rule describing every member of a set one must have information about every member.
- Machine learning avoids this problem by learning probabilistic[1] rules which are *probably* correct about *most* members of the set they concern.
- But, **no free lunch theorem** states that every possible classification machine has the *same error* when averaged over *all possible* data-generating distributions.
  - **No machine learning algorithm is universally better than any other!**
  - Fortunately, in the real world, data is generated by a small subset of generating distributions...

pause

[1] or perhaps more generally rules which are not certain

The Universal Approximation Theorem

Let $\psi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0, 1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, such that we may define:

$F(x) = \sum_{i=1}^{N} v_i \psi(w_i^T x + b_i)$ as an approximate realization of the function f; that is,

$|F(x) - f(x)| < \varepsilon \ \forall \ x \in I_m$. [1em] $\implies$ simple neural networks can represent a wide variety of interesting functions when given appropriate parameters.

So a single hidden layer network can approximate most functions?

- Yes!
- But, ...
  - to get the precision you require (small ε), you might need a really large number of hidden units (very large N).
  - worse-case analysis shows it might be exponential (possibly one hidden unit for *every* input configuration)
  - We've not said anything about learnability...
    - The optimiser might not find a good solution[2].
    - The training algorithm might just choose the wrong solution as a result of overfitting.
    - *There is no known universal procedure for examining a set of examples and choosing a function that will generalise to points out of the training set.*

Then Why Go Deep?

- There are functions you can compute with a deep neural network that shallow networks require exponentially more hidden units to compute.

[2]note that it has been shown that the gradients of the function are approximated by the network to an arbitrary precision

  - The following function is more efficient to implement using a deep neural network: $y = x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n$

- We should care about the data generating distribution (c.f. NFL).
  - Real-world data has significant structure; often believed to be generated by (relatively) simple low-dimensional latent processes.
  - Implies a prior belief that the underlying factors of variation in data can be explained by a hierarchical composition of increasingly simple latent factors

- Alternatively, one could just consider that a deep architecture just expresses that the function we wish to learn is a program made of multiple steps where each step makes use of the previous steps outputs.

- **Empirically, deeper networks just seem to generalise better!**

What are the problems?

- Learnability is still hard
  - Problems of gradient flow
  - Horrible symmetries in the loss landscape
  - Overfitting

pause

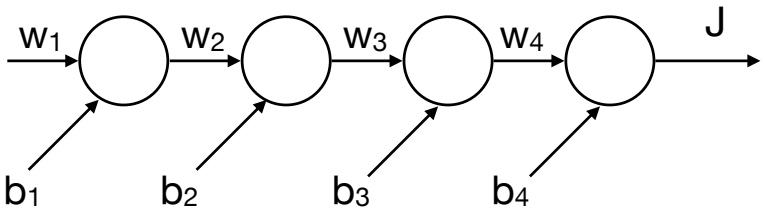
Vanishing and Exploding Gradients

- The vanishing and exploding gradient problem is a difficulty found in training NN with gradient-based learning methods and backpropagation.
- In training, the gradient may become vanishingly small (or large), effectively preventing the weight from changing its value (or exploding in value).
- This leads to the neural network not being able to train.
- This issue affects many-layered networks (feed-forward), as well as recurrent networks.

- In principle, optimisers that rescale the gradients of each weight should be able to deal with this issue (as long as numeric precision doesn't become problematic).

pause

Issues with Going Deep



pause

Residual Connections

- One of the most effective ways to resolve diminishing gradients is with residual neural networks (ResNets)[3].
- ResNets are artificial neural networks that use *skip connections* to jump over layers.
- The vanishing gradient problem is mitigated in ResNets by reusing activations from a previous layer.
- Is this the full story though? Skip connections also break symmetries, which could be much more important...

pause

[3]K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

Residual Connections

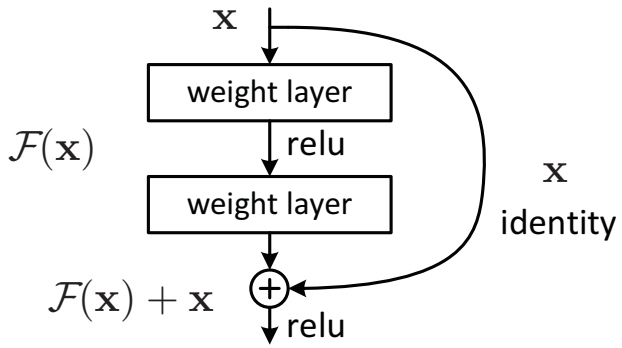


Figure 2. Residual learning: a building block.

.

pause

Residual Connections

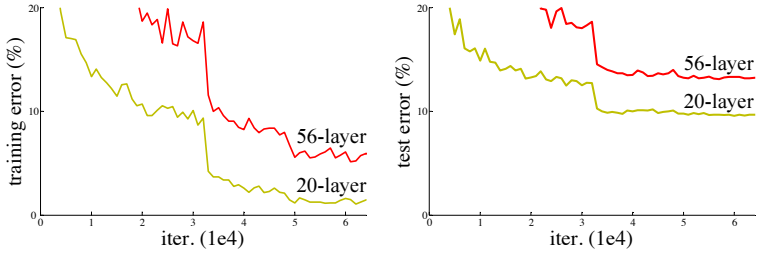


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

pause

Residual Connections

K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

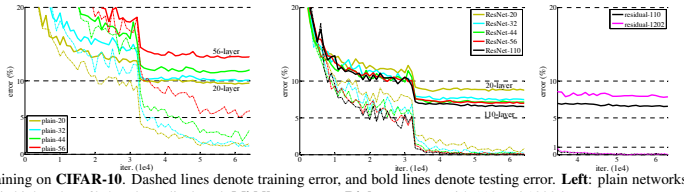


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left**: plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle**: ResNets. **Right**: ResNets with 110 and 1202 layers.

Overfitting

- Neural networks with a large number of parameters (and hidden layers) are powerful, however, overfitting is a serious problem in such systems.
- Just as you've seen in simple machines (e.g. Ridge Regression and LASSO), regularisation can help mitigate overfitting
- In deep networks, we might:
  - Use the architecture to regularise (e.g. ConvNets)
  - Use weight regularisers (L1, L2 [weight decay], etc, ...)
  - Use a stochastic weight regulariser (like dropout)
  - Regularise by smoothing the optimisation landscape (e.g. Batch Normalisation)

pause

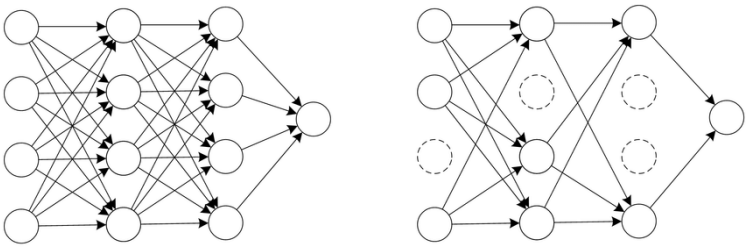
Dropout

- Dropout is a form of regularisation
- The key idea in dropout is to randomly drop neurons, including all of the connections, from the neural network during training.
- Motivation: the best way to regularise a fixed size model is to average predictions over all possible parameter settings, weighting each setting by the posterior probability given the training data.
  - Clearly this isn't actually tractable - dropout is an approximation of this idea.

K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

– The idea of averaging predictions to resolve the bias-variance dilemma is called ensembling.

Dropout



(a) Standard Neural Network

(b) Network after Dropout

pause

How Does Dropout Work?

- In the learning phase, we set a dropout probability for each layer in the network.
- For each batch we then randomly decide whether or not a given neuron in a given layer is removed.

pause

How is Dropout implemented?

- We define a random binary mask $m^{(l)}$ which is used to remove neurons, and note, $m^{(l)}$ changes for each iteration of the back-propagation algorithm.
- For layers, $l = 1$ to $L - 1$, for the forward pass of backpropagation, we then compute

$$a^{(l)} = \sigma(w^{(l)}a^{(l-1)} + b^{(l)}) \odot m^{(l)} \tag{7.1}$$

Image from: https://www.researchgate.net/figure/Dropout-neural-network-model-a-is-a-standard-neural-network-b-is-the-same-network_fig3_309206911

- For layer L,

$$a^{(L)} = \sigma(w^{(L)}a^{(L-1)} + b^{(l)}) \tag{7.2}$$

- For the backward pass of the backpropagation algorithm,

$$\delta^L = \Delta_a J \odot \sigma'(z^L) \tag{7.3}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \odot m^{(l)} \tag{7.4}$$

pause

Why Does Dropout Work?

- Neurons cannot co-adapt to other units (they cannot assume that all of the other units will be present)
- By breaking co-adaptation, each unit will ultimately find more general features

8. Embeddings and distributed representations

9. Function reparameterisations and relaxations

What are differentiable relaxations and reparameterisations?

- We've seen that we can build arbitrary computational graphs from a variety of building blocks
- But, those blocks need to be differentiable to work in our optimisation framework
  - More specifically they need to be continuous and *differentiable almost everywhere*.
- That limits what we can do... Can we work around that?
  - Relaxations — make continuous (and potentially differentiable everywhere) approximations.
  - Reparameterisations — rewrite functions to factor out stochastic variables from the parameters.

Aside: continuity and differentiable almost everywhere

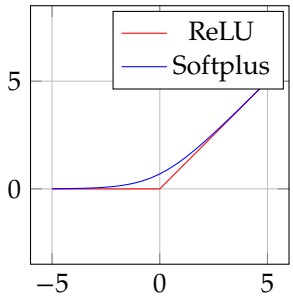
- Consider the ReLU function $f(x) = max(0, x)$
  - ReLU is *continuous*
    - it does not have any abrupt changes in value
    - small changes in x result in small changes to $f(x)$ everywhere in the domain of x
  - ReLU is *differentiable almost everywhere*
    - No gradient at $x = 0$; only *left* and *right* gradients at that point

    - There are *subgradients* at $x = 0$; implementations usually just arbitrarily pick $f'(0) = 0$

- Functions that are differentiable almost everywhere or have subgradients tend to be compatible with gradient descent methods
  - We expect that the loss landscape is different for each batch & that we'll never actually reach a minima, and we only need to *mostly* take steps in the right direction.

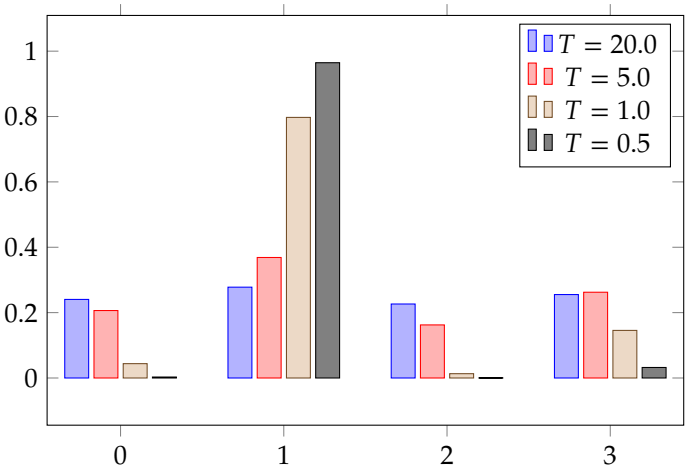
Relaxing ReLU

- Softplus ($\text{softplus}(x) = \ln(1 + e^x)$) is a relaxation of ReLU that is *differentiable everywhere*.
- Its derivative is the Sigmoid function
- Not widely used; counter-intuitively, even though it neither saturates completely and is differentiable everywhere, empirically it has been shown that ReLU works better.



Interpretations of softmax

- Up until now we've really considered softmax as a generalisation of sigmoid (which represents a probability distribution over a binary variable) to many output categories.
  - softmax transforms a vector of logits into a probability distribution over categories.
- As you might guess from the name, softmax is a relaxation...
  - but not of the max function like the name would suggest!



- softmax can be viewed as a continuous and differentiable relaxation of the arg max function with one-hot output encoding.
- The arg max function is not continuous or differentiable; softmax provides an approximation:

$\mathbf{x} =$	[	1.1	4.0	-0.1	2.3]
$\arg\max(\mathbf{x}) =$	[	0	1	0	0]
$\text{softmax}(\mathbf{x}) =$	[	0.044	0.797	0.013	0.146]

The Softmax function with temperature

Consider what happens if you were to divide the input logits to a softmax by a scalar temperature parameter T.

$$\text{softmax}(\mathbf{x}/T)_i = \frac{e^{x_i/T}}{\sum_{j=1}^{K} e^{x_j/T}} \qquad \forall i = 1, 2, \ldots, K$$

arg max — softmax *with temperature*

$\mathbf{x} =$	[	1.1	4.0	-0.1	2.3]
$\text{softmax}(\mathbf{x}/1.0) =$	[	0.044	0.797	0.013	0.146]
$\text{softmax}(\mathbf{x}/0.8) =$	[	0.023	0.868	0.005	0.104]
$\text{softmax}(\mathbf{x}/0.6) =$	[	0.008	0.937	0.001	0.055]
$\text{softmax}(\mathbf{x}/0.4) =$	[	6.997e-04	9.852e-01	3.484e-05	1.405e-02]
$\text{softmax}(\mathbf{x}/0.2) =$	[	5.042e-07	9.998e-01	1.250e-09	2.034e-04]

arg max — *scalar approximation*

- What if you want to get a scalar approximation to the index of the arg max rather than a probability distribution approximating the one-hot form?
  - Caveat: we are not actually going get a guaranteed integer representation as that would be non-differentiable; we'll have to live with a float that is an approximation[1].
- First, consider how to convert a one-hot vector to index representation in a differentiable manner: $[0, 0, 1, 0] \to 2$
  - Just dot product with a vector of indices: $[0, 1, 2, 3]$
- The same process can be applied to the softmax distribution
  - As temperature $T \to 0$, $\text{softmax}(\mathbf{x}/T) \cdot [0, 1, \ldots, N] \to \arg\max(\mathbf{x})$ for $\mathbf{x} \in \mathbb{R}^N$.

arg max — *scalar approximation*

$$\mathbf{x} = [\ 1.1 \quad 4.0 \quad -0.1 \quad 2.3\ ]^\top$$
$$\mathbf{i} = [\ 0.0 \quad 1.0 \quad 2.0 \quad 3.0\ ]^\top$$
$$\text{softmax}(\mathbf{x}/1.0)^\top \mathbf{i} = 1.2606$$
$$\text{softmax}(\mathbf{x}/0.8)^\top \mathbf{i} = 1.1894$$
$$\text{softmax}(\mathbf{x}/0.6)^\top \mathbf{i} = 1.1037$$
$$\text{softmax}(\mathbf{x}/0.4)^\top \mathbf{i} = 1.0274$$
$$\text{softmax}(\mathbf{x}/0.2)^\top \mathbf{i} = 1.0004$$

max

- A similar trick applies to finding the maximum value of a vector:
  - Use $\text{softmax}(\mathbf{x})$ as an approximate one-hot arg max, and dot product with the vector $\mathbf{x}$.
  - As temperature $T \to 0$, $\text{softmax}(\mathbf{x}/T)^\top \mathbf{x} \to \max(\mathbf{x})$.

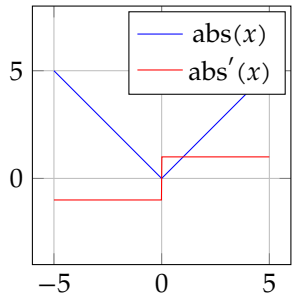
[1] for now — we'll address this in a few slides time!

$$\mathbf{x} = [\ 1.1 \quad 4.0 \quad -0.1 \quad 2.3\ ]^\top$$

$$\text{softmax}(\mathbf{x}/1.0)^\top \mathbf{x} = 3.571$$
$$\text{softmax}(\mathbf{x}/0.8)^\top \mathbf{x} = 3.736$$
$$\text{softmax}(\mathbf{x}/0.6)^\top \mathbf{x} = 3.881$$
$$\text{softmax}(\mathbf{x}/0.4)^\top \mathbf{x} = 3.974$$
$$\text{softmax}(\mathbf{x}/0.2)^\top \mathbf{x} = 3.999$$

L1 norm

- L1 norm is the sum of absolute values of a vector
- We've seen that an L1 norm regulariser can induce sparsity in a model
- abs is continuous and differentiable almost everywhere, but...
- unlike ReLU, the gradients left and right of the discontinuity point in equal and opposite directions
  - This can cause oscillations that prevent or hamper learning

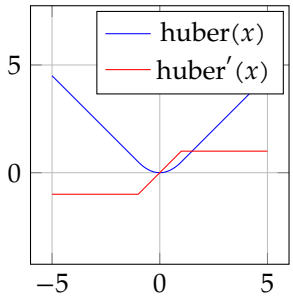


Relaxing the L1 norm

- Huber loss (aka Smooth L1 loss) relaxes L1 by mixing it with L2 near the origin:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

- In both cases gradients reduce in magnitude and switch direction smoothly which can lead to much less oscillation.



Backpropagation through random operations

- Up until now all the models we've considered have performed deterministic transformations of input variables **x**.
- What if we want to build a model that performs a stochastic transformation of **x**?
- A simple way to do this is to augment the input **x** with a random vector **z** sampled from some distribution
  - The network would learn a function $f(\mathbf{x}, \mathbf{z})$ that is internally deterministic, but appears stochastic to an observer that does not have access to **z**.
  - provided that f is continuous and differentiable (almost everywhere) we can perform gradient based optimisation as usual.

Differentiable Sampling

Consider

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

How can we take derivatives of y with respect to μ and σ^2?

Differentiable Sampling

If we rewrite

$$y = \mu + \sigma z \text{ where } z = \mathcal{N}(0, 1)$$

Then it is clear that y is a function of a deterministic operation with variables μ and σ with an (extra) input z.

- Crucially the extra input is an r.v. whose distribution is not a function of any variables whose derivatives we wish to calculate.
- The derivatives $dy/d\mu$ and $dy/d\sigma$ tell us how an infinitesimal change in μ or σ would change y if we could repeat the sampling operation with the *same* value of z

The reparameterisation trick

- The 'trick' of factoring out the source of randomness into an extra input z is often called the **reparameterisation trick**.
- It doesn't just apply to the Gaussian distribution!
  - More generally we can express any probability distribution $p(\mathrm{y}; \boldsymbol{\theta})$ or $p(\mathrm{y}|\mathbf{x}; \boldsymbol{\theta})$ as $p(\mathrm{y}; \boldsymbol{\omega})$ where $\boldsymbol{\omega}$ contains the parameters $\boldsymbol{\theta}$ and if applicable inputs $\mathbf{x}$.
  - A sample $\mathbf{y} \sim p(\mathrm{y}; \boldsymbol{\omega})$ can be rewritten as $\mathbf{y} = f(\mathbf{z}, \boldsymbol{\omega})$ where $\mathbf{z}$ is a source of randomness.
  - We can thus compute derivatives $\partial \mathbf{y}/\partial \boldsymbol{\omega}$ and use gradient based optimisation as long as
    - f is continuous and differentiable almost everywhere
    - $\boldsymbol{\omega}$ is not a function of $\mathbf{z}$
    - and $\mathbf{z}$ is not a function of $\boldsymbol{\omega}$

Backpropagation through discrete stochastic operations

- Consider a stochastic model $\mathbf{y} = f(\mathbf{z}, \boldsymbol{\omega})$ where the outputs are **discrete**.
  - This implies f must be a step function.
  - Derivatives of a step function at the step are undefined.
  - Derivatives are zero almost everywhere.
  - If we have a loss $\mathcal{L}(\mathbf{y})$ the gradients don't give us any information on how to update the parameters $\boldsymbol{\theta}$ to minimise the loss
- Potential solutions:
  - REINFORCE
  - A relaxation and another 'trick': Gumbel Softmax and the Straight-through operator

REINFORCE: REward Increment = nonnegative Factor × Offset Reinforcement × Characteristic Eligibility

- $\mathcal{L}(f(\mathbf{z}, \boldsymbol{\omega}))$ has useless derivatives
- But the expected loss $\mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\mathcal{L}(f(\mathbf{z}, \boldsymbol{\omega}))$is often smooth and continuous.
  - This is not tractable with high dimensional $\mathbf{y}$.
  - But, it can be estimated without bias using an Monte Carlo average.
- REINFORCE is a family of algorithms that utilise this idea.

REINFORCE: REward Increment = nonnegative Factor × Offset Reinforcement × Characteristic Eligibility

The simplest form of REINFORCE is easy to derive by differentiating the expected loss:

$$\mathbb{E}_{\mathbf{z}}[\mathcal{L}(\mathbf{y})] = \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) p(\mathbf{y}) \tag{9.1}$$

$$\frac{\partial \mathbb{E}[\mathcal{L}(\mathbf{y})]}{\partial \boldsymbol{\omega}} = \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) \frac{\partial p(\mathbf{y})}{\partial \boldsymbol{\omega}} \tag{9.2}$$

$$= \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y}) p(\mathbf{y}) \frac{\partial \log p(\mathbf{y})}{\partial \boldsymbol{\omega}} \tag{9.3}$$

$$\approx \frac{1}{m} \sum_{\mathbf{y}^{(i)}\sim p(\mathbf{y}), i=1}^{m} \mathcal{L}(\mathbf{y}^{(i)}) \frac{\partial \log p(\mathbf{y}^{(i)})}{\partial \boldsymbol{\omega}} \tag{9.4}$$

- This gives us an unbiased MC estimator of the gradient.
- Unfortunately this is a very high variance estimator, so it would require many samples of y to be drawn to obtain a good estimate
  - or equivalently, if only one sample were drawn, SGD would converge very slowly and **require** a small learning rate.

Sampling from a categorical distribution: Gumbel Softmax

The generation of a discrete token, t, from a vocabulary of K tokens is achieved by sampling a categorical distribution

$$t \sim \text{Cat}(p_1, \dots, p_K)\,;\, \sum_i p_i = 1.$$

Generating the probabilities $p_1, \ldots, p_K$ directly from a neural network has potential numerical problems; it's much easier to generate logits, $x_1, \ldots, x_K$. [0.5em] The gumbel-softmax reparameterisation allows us to sample directly using the logits:

$$t = \underset{i \in \{1,\cdots,K\}}{\text{argmax}} x_i + z_i$$

where $z_1, \ldots z_K$ are i.i.d Gumbel(0,1) variates which can be computed from Uniform variates through $-\log(-\log(-ll(0,1)))$.

Differentiable Sampling: Straight-Through Gumbel Softmax

Ok, but how does that help? argmax isn't differentiable! [0.5em] ...but we've already seen that we can relax arg max using

$$\text{softargmax}(\mathbf{y}) = \sum_i \frac{e^{y_i/T}}{\sum_j e^{y_j/T}} i$$

where T is the temperature parameter.

Differentiable Sampling: Straight-Through Gumbel Softmax

But... this clearly gives us a result that will be non-integer; we cannot round or clip because it would be non-differentiable. [0.5em] The Straight-Through operator allows us to take the result of a true argmax that has the gradient of the softargmax:

$$\text{STargmax}(\mathbf{y}) = \text{softargmax}(\mathbf{y}) + \text{stopgradient}(\text{argmax}(\mathbf{y}) - \text{softargmax}(\mathbf{y}))$$

where stopgradient is defined such that $\text{stopgradient}(\mathbf{a}) = \mathbf{a}$ and $\nabla \,\text{stopgradient}(\mathbf{a}) = 0$.

Straight-Through Gumbel Softmax

Combine the gumbel softmax trick with the STargmax to give you discrete samples, with a usable gradient[2].

Summary

- Differentiable programming works with functions that are continuous and differentiable almost everywhere.

[2]The ST operator is biased but low variance; in practice it works very well and is better than the high-variance unbiased estimates you could get through REINFORCE.

- Some non-continuous functions can be relaxed to make them more amenable to gradient based optimisation by making continuous approximations.
- Some continuous functions with discontinuous gradients can be relaxed to make optimisation more stable.
- Reparameterisations can allow us to differentiate through random operations such as sampling
- We can even make networks output/utilise discrete variables by combining relaxations and reparameterisations.

Part II

Basic Deep Learning Architectures

10. A Biological Perspective

11. Convolutional Networks

12. Recurrent Neural Networks

pause

Recurrent Neural Networks - Motivation

x:	Jon	and	Ethan	gave	deep	learning	lectures
y:	1	0	1	0	0	0	0

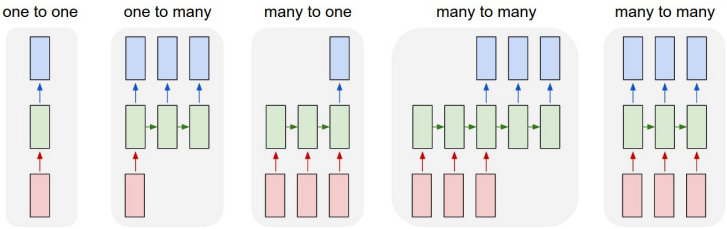
pause

Recurrent Neural Networks - Motivation

x:	$x^{(1)}$	...	$x^{(t)}$	...	$x^{(T_x)}$
x:	Jon	...	Ethan	...	lectures
y:	$y^{(1)}$	...	$y^{(t)}$	...	$y^{(T_y)}$
y:	1	...	1	...	0

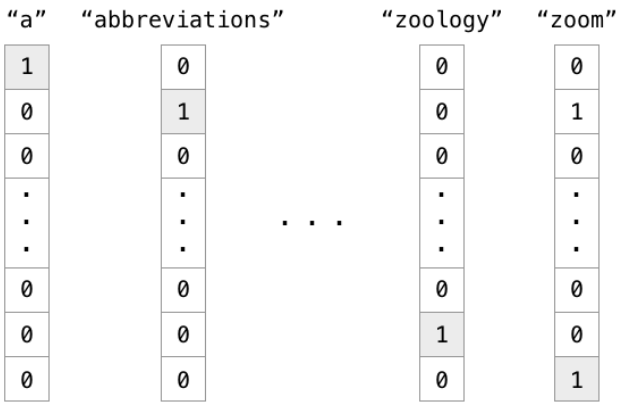
In this example, $T_x = T_y = 7$ but T_x and T_y can be different.

Recurrent Neural Networks



Aside: One Hot Encoding

How can we represent individual words (or other discrete tokens)?



Why Not a Standard Feed Forward Network?

- For a task such as "Named Entity Recognition" a MLP would have several disadvantages
  - The inputs and outputs may have varying lengths
  - The features wouldn't be shared across different temporal positions in the network
    - Note that 1-D convolutions can be (and are) used to address this, in addition to RNNs - more on this in a later lecture

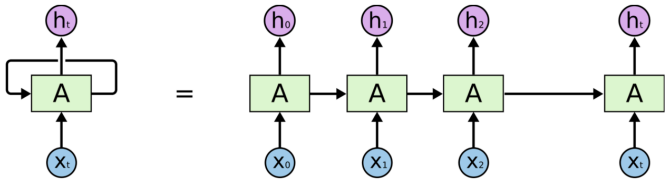
Image from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Image from https://ayearofai.com

- To interpret a sentence, or to predict tomorrows weather it is necessary to remember what happened in the past
- To facilitate this we would like to add a feedback loop delayed in time

pause

Recurrent Neural Networks



[1]

- RNNs are a family of ANNs for processing sequential data
- RNNs have directed cycles in their computational graphs

pause

Recurrent Neural Networks

RNNs combine two properties which make them very powerful.

- Distributed hidden state that allows them to store a lot of information about the past efficiently. This is because several different units can be active at once, allowing them to remember several things at once.
- Non-linear dynamics that allows them to update their hidden state in complicated ways[2].

[1] Image taken from `https://towardsdatascience.com`

[2] Often said to be difficult to train, but this is not necessarily true - dropout can help with overfitting for example

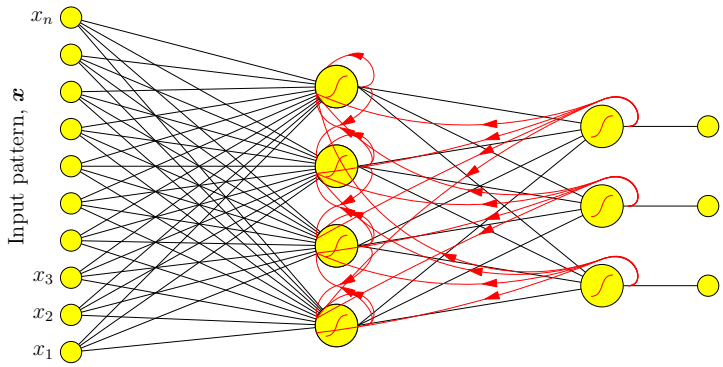
pause

Recurrent Neural Networks

RNNs are Turing complete in the sense they can simulate arbitrary programs[3].

If training vanilla neural nets is optimisation over functions, training recurrent nets is optimisation over programs.

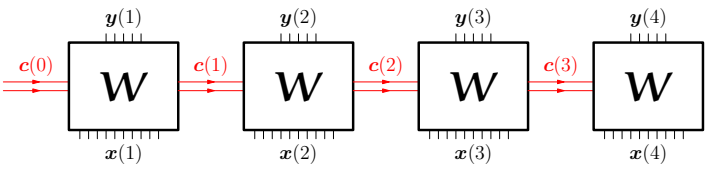
Recurrent Network



pause

Training Recurrent Networks

- Given a set of inputs $\mathcal{D} = ((\mathbf{x}(t), \mathbf{y}(t))|t = 1, 2, \ldots, T)$



[3]Don't read too much into this - like universal approximation theory, just because they can doesn't mean its necessarily learnable!

- Minimise an error (here MSE, but your choice):

$$E(\mathbf{W}) = \sum_{t=1}^{T} \|\mathbf{y}(t) - \mathbf{f}(\mathbf{x}(t), \mathbf{c}(t-1)|\mathbf{W})\|^2$$

- This is known as *back-propagation through time*

An RNN is just a recursive function invocation

- $\mathbf{y}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{c}(t-1)|\mathbf{W})$
- and the state $\mathbf{c}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{c}(t-1)|\mathbf{W})$
- If the output $\mathbf{y}(t)$ depends on the input $\mathbf{x}(t-2)$, then prediction will be

$$\mathbf{f}(\mathbf{x}(t), \mathbf{g}(\mathbf{x}(t-1), \mathbf{g}(\mathbf{x}(t-2), \mathbf{g}(\mathbf{x}(t-3)|\mathbf{W})|\mathbf{W})|\mathbf{W})|\mathbf{W})$$

- it should be clear that the gradients of this with respect to the weights can be found with the chain rule

What is the state update $g()$?

- It depends on the variant of the RNN!
  - Elman
  - Jordan
  - LSTM
  - GRU

Elman Networks ("Vanilla RNNs")

$$\mathbf{h}_t = \sigma_h(\mathbf{W}_{ih}\mathbf{x}_t + \mathbf{b}_{ih} + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_{hh})$$
$$\mathbf{y}_t = \sigma_y(\mathbf{W}_y\mathbf{h}_t + \mathbf{b}_y)$$

- σ_h is usually tanh
- σ_y is usually identity (linear) – the y's could be regressed values or logits
- the state $\mathbf{h}_t$ is referred to as the "hidden state"
- the output at time t is a projection of the hidden state at that time
- the hidden state at time t is a summation of a projection of the input and a projection of the previous hidden state

Going deep: Stacking RNNs

- RNNs can be trivially stacked into deeper networks
- It's just function composition: $[1em]\ \mathbf{y}(t) = \mathbf{f}_2(\mathbf{f}_1(\mathbf{x}(t), \mathbf{c}_2(t-1)|\mathbf{W}_1), \mathbf{c}_2(t-1)|\mathbf{W}_2)\ [1em]$
- The output of the inner RNN at time t is fed into the input of the outer RNN which produces the prediction y
- Also note: RNNs are most often not used in isolation - it's quite common to process the inputs and outputs with MLPs (or even convolutions)

Example: Character-level language modelling

- We'll end with an example: an RNN that learns to 'generate' English text by learning to predict the next character in a sequence
- This is "Character-level Language Modelling"

Example: Character-level language modelling

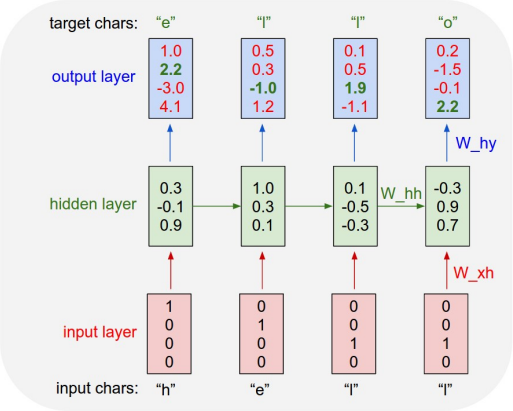


Image from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Training a Char-RNN

- The training data is just text data (e.g. sequences of characters)
- The task is unsupervised (or rather self-supervised): given the previous characters predict the next one
  - All you need to do is train on a reasonable sized corpus of text
  - Overfitting could be a problem: dropout is very useful here

Sampling the Language Model

- Once the model is trained what can you do with it?
- if you feed it an initial character it will output the logits of the next character
- you can use the logits to select the next character and feed that in as the input character for the next timestep
- how do you 'sample' a character from the logits?
  - you could pick the most likely (maximum-likelihood solution), but this might lead to generated text with very low variance (it might be boring and repetitive)
  - you could treat the softmax probabilities defined by the logits as a categorical distribution and sample from them
    * you might increase the 'temperature', T, of the softmax to make the distribution more diverse (less 'peaky'): $q_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$

A lot of the ideas in this lectu on the input c(t − 1), g (x i (x(t − 2), g (t − 1)|W) Iged snllhomitpon" ares Mnt Net) th

pl Onaafed a tre the sidisicters of to prediction couponet on the logits its venvows usts sevouvd be this in as useuled at on in the pan Lerate'atectsrray to paet inputs D = Pxxpraition the rople, the next

vog the state atite

- Sampled from a single layer RNN[4].

[4]LSTM, 128 dim hidden size, with linear input projection to 8-dimensions and output to the number of characters (84). Trained on the text of these slides for 50 epochs.

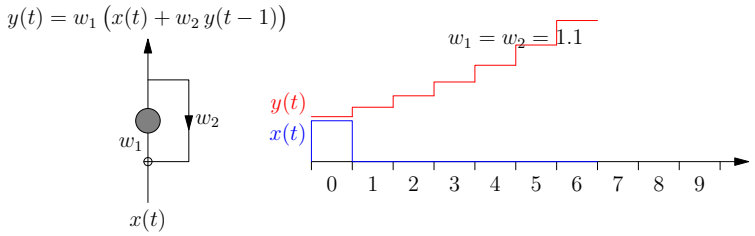
Recap: An RNN is just a recursive function invocation

- $\mathbf{y}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{c}(t)|\mathbf{W})$
- and the state $\mathbf{c}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{c}(t-1)|\mathbf{W})$
- If the output $\mathbf{y}(t)$ depends on the input $\mathbf{x}(t-2)$, then prediction will be

$$\mathbf{f}(\mathbf{x}(t), \mathbf{g}(\mathbf{x}(t), \mathbf{g}(\mathbf{x}(t-1), \mathbf{g}(\mathbf{x}(t-2), c(t-2)|\mathbf{W})|\mathbf{W})|\mathbf{W})|\mathbf{W})$$

- it should be clear that the gradients of this with respect to the weights can be found with the chain rule
- The back-propagated error will involve applying $\mathbf{f}$ multiple times
- Each time the error will get multiplied by some factor a
- If $\mathbf{y}(t)$ depends on the input $\mathbf{x}(t-\tau)$ then the back-propagated signal will be proportional to $a^{\tau-1}$
- This either vanishes or explodes when τ becomes large

Vanishing and Exploding Gradients



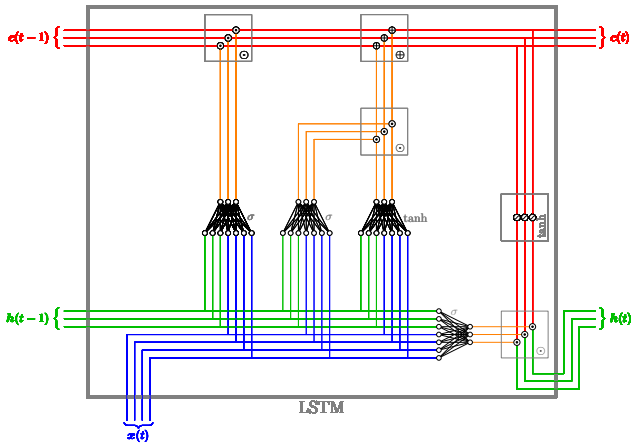
LSTM Architecture

- LSTMs (long-short term memory) was designed to solve this problem
- Key ideas: to retain a 'long-term memory' requires

$$\mathbf{c}(t) = \mathbf{c}(t-1)$$

- Sometimes we have to forget and sometimes we have to change a memory
- To do this we should use 'gates' that saturate at 0 and 1
- Sigmoid functions naturally saturate at 0 and 1

LSTM Architecture



Update Equations

Initially, for $t = 0$, $\mathbf{h}(0) = \mathbf{0}$

- Inputs $\mathbf{z}(t) = \Big(\mathbf{x}(t), \mathbf{h}(t-1)\Big)$
- Network updates ($\mathbf{W}_*$ and $\mathbf{b}_*$ are the learnable parameters)

$$\mathbf{f}(t) = \sigma\Big(\mathbf{W}_f\,\mathbf{z}(t) + \mathbf{b}_f\Big) \qquad \mathbf{i}(t) = \sigma(\mathbf{W}_i\,\mathbf{z}(t) + \mathbf{b}_i)$$
$$\mathbf{g}(t) = \mathbf{tanh}\Big(\mathbf{W}_g\,\mathbf{z}(t) + \mathbf{b}_g\Big) \qquad \mathbf{o}(t) = \sigma(\mathbf{W}_o\,\mathbf{z}(t) + \mathbf{b}_o)$$

- Long-term memory update

$$\mathbf{c}(t) = \mathbf{f}(t) \odot \mathbf{c}(t-1) + \mathbf{g}(t) \odot \mathbf{i}(t)$$

- Output $\mathbf{h}(t) = \mathbf{o}(t) \odot \mathbf{tanh}(\mathbf{c}(t))$

Training LSTMs

- We can train an LSTM by unwrapping it in time.
- Note that it involves four dense layers with sigmoidal (or tanh) outputs.
- This means that typically it is very slow to train.
- There are a few variants of LSTMs, but all are very similar. The most popular is probably the Gated Recurrent Unit (GRU).

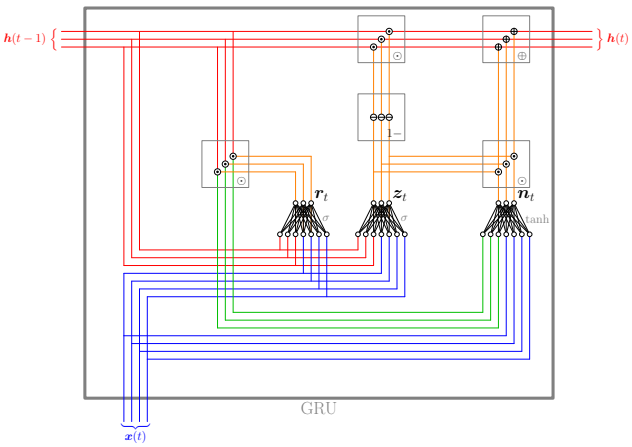
pause

LSTM Success Stories

- LSTMs have been used to win many competitions in speech and handwriting recognition.
- Major technology companies including Google, Apple, and Microsoft are using LSTMs as fundamental components in products.
- Google used LSTM for speech recognition on the smartphone, for Google Translate.
- Apple uses LSTM for the "Quicktype" function on the iPhone and for Siri.
- Amazon uses LSTM for Amazon Alexa.
- In 2017, Facebook performed some 4.5 billion automatic translations every day using long short-term memory networks[5].

pause

Gated Recurrent Unit (GRU)



[5] https://en.wikipedia.org/wiki/Long_short-term_memory

pause

Gated Recurrent Unit (GRU)

- $\mathbf{x}(t)$: input vector
- $\mathbf{h}(t)$: output vector (and 'hidden state')
- $\mathbf{r}(t)$: reset gate vector
- $\mathbf{z}(t)$: update gate vector
- $\mathbf{n}(t)$: new state vector (before update is applied)
- $\mathbf{W}$ and $\mathbf{b}$: parameter matrices and biases

pause

Gated Recurrent Unit (GRU)

Initially, for $t = 0$, $\mathbf{h}(0) = \mathbf{0}$

$$\begin{aligned}
\mathbf{z}(t) &= \sigma(\mathbf{W}_z(\mathbf{x}(t), \mathbf{h}(t-1)) + \mathbf{b}_z) \\
\mathbf{r}(t) &= \sigma(\mathbf{W}_r(\mathbf{x}(t), \mathbf{h}(t-1)) + \mathbf{b}_r) \\
\mathbf{n}(t) &= \tanh(\mathbf{W}_n(\mathbf{x}(t), r(t) \odot h(t-1)) + \mathbf{b}_h) \\
\mathbf{h}(t) &= (1 - \mathbf{z}(t)) \odot \mathbf{h}(t-1) + \mathbf{z}(t) \odot \mathbf{n}(t)
\end{aligned}$$

pause

GRU or LSTM?

- GRUs have two gates (reset and update) whereas LSTM has three gates (input/output/forget)
- GRU performance on par with LSTM but computationally more efficient (less operations & weights).

Most implementations follow the original paper and swap $(1 - \mathbf{z}(t))$ and $(\mathbf{z}(t))$ in the $\mathbf{h}(t)$ update; this doesn't change the operation of the network, but does change the interpretation of the update gate, as the gate would have to produce a 0 when an update was to occur, and a 1 when no update is to happen (which is somewhat counter-intuitive)!

- In general, if you have a very large dataset then LSTMs will likely perform slightly better.
- GRUs are a good choice for smaller datasets.

13. Auto-encoders

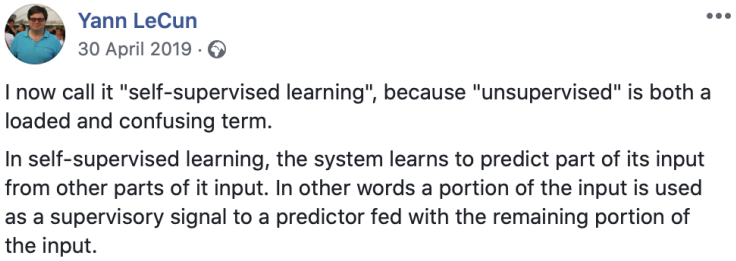
Low Dimensional Representations

- One of the common features of many of the deep learning models we have looked at to this point is that they often try to reduce the dimensionality of the input data in order to capture some kind of underlying information.
- In the last lecture this was particularly evident when when we looked at embedding models like word2vec which explictly try to capture relationships in the data in a low dimensional 'latent' space.

Self-supervised Learning

Self-supervised Learning

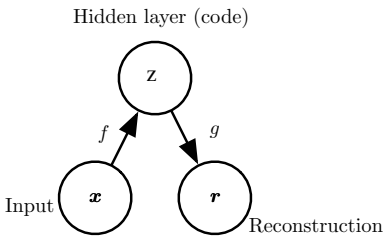
- The word2vec models are examples of *self-supervised learning*
  - CBOW learns to predict the focus word from the context words
  - Skip-gram learns to predict the context words from the focus word



- Let's now consider a different type self-supervised of task where we want to learn a model that learns to copy its input to its output.

Autoencoders

- An **autoencoder** is a network that is trained to copy its input to its output
  - Internally there is some hidden vector $\mathbf{h}$ that describes a **code** that represents the input.
  - Conceptually the autoencoder consists of two parts:
    - The encoder $\mathbf{h} = f(\mathbf{x})$
    - The decoder $\mathbf{r} = g(\mathbf{h})$
  - and has loss that tries to minimise the reconstruction error (typically SSE/MSE: $\|\mathbf{x} - \mathbf{r}\|_2^2$)



Autoencoder constraints

- Clearly a linear autoencoder with a sufficient number of weights (e.g. if the dimension of $\mathbf{h}$ was greater than or equal to that of x) could learn set $g(f(\mathbf{x})) = \mathbf{x}$ everywhere, but this obviously wouldn't be useful!
- In practice we apply *restrictions*[1] to stop this happening.
- The objective is to use these restrictions to force the autoencoder to learn useful properties of the data.

[1] these are 'inductive biases'

Undercomplete Autoencoders

- Undercomplete autoencoders have $\dim(\mathbf{h}) << \dim(\mathbf{x})$.
- This forces the encoder to learn a *compressed representation* of the input.
- The representation will capture the most *salient* features of the input data.

pause

Undercomplete Autoencoders — Linear

Consider the single-hidden layer linear autoencoder network given by:

$$h = \mathrm{W}_e\mathbf{x} + \mathbf{b}_e$$
$$r = \mathrm{W}_d\mathbf{h} + \mathbf{b}_d$$

where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{h} \in \mathbb{R}^m$ and $m < n$.[1em] With the MSE loss, this autoencoder will learn to span the same subspace as PCA for a given set of training data. [1em] Note that the autoencoder weights are not however constrained to be orthogonal (like they would be in PCA)

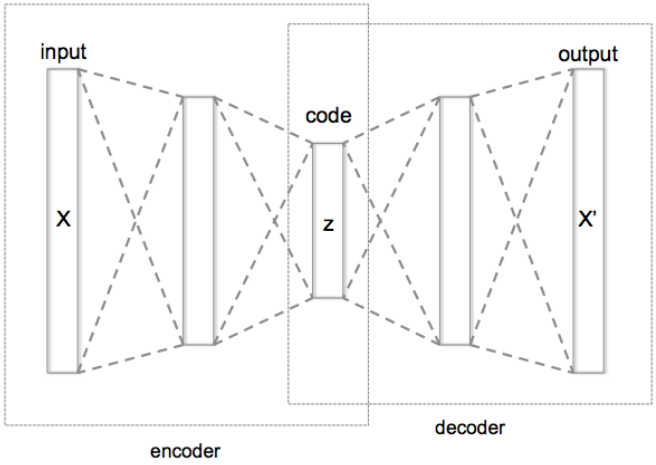
Undercomplete Autoencoders — deeper and nonlinear

- A linear autoencoder with a single hidden layer learns to map into the same subspace as PCA.
- Clearly, a deeper, linear autoencoder would also do the same thing.
- What happens if you introduce non-linearity?
  - Interestingly, a single hidden layer network with non-linear activations on the encoder (keeping the decoder linear) and MSE loss also just learns to span the PCA subspace[2]!

[2]Bourlard, H., Kamp, Y. Auto-association by multilayer perceptrons and singular value decomposition. Biol. Cybern. 59, 291–294 (1988). https://doi.org/10.1007/BF00332918

– But, if you add more hidden layers with non-linear activations (to either the encoder, decoder or both) you can effectively perform a powerful non-linear generalisation of PCA

Deep Autoencoders



Deep Autoencoders - caveat

- There is a slight catch: if you give the deep autoencoder network too much capacity (too many weights) it will learn to perform the copying task without extracting anything useful about the data.
- Of course this means that will likely not generalise to unseen data.
- Extreme example:
  - Consider a powerful encoder that maps $\mathbf{x}$ to $\mathbf{h} \in \mathbb{R}^1$
  - Each training example $\mathbf{x}^{(i)}$ could e.g. be mapped to i.
  - The decoder just needs to memorise the training examples so that it can map back from i.

Image taken from wikipedia

Undercomplete Autoencoders — Convolutional

- Thus far, we only considered autoencoders with vector inputs/outputs and fully-connected layers.
- There is nothing stopping us using any other kinds of layers though...
- If we're working with image data, where we know that much of the structure is 'local', then using convolutions in both the decoder makes sense

Convolutional Autoencoder

Regularised Autoencoders

- Rather than (necessarily) forcing the hidden vector to have a lower dimensionality than the input, we could instead utilise some form of regularisation to force the network to learn interesting representations...
- Many ways to do this; let's look at two of them:
  - Denoising Autoencoders
  - Sparse Autoencoders

Denoising Autoencoders

- Denoising autoencoders take a partially corrupted input and train to recover the original undistorted input.
- To train an autoencoder to denoise data, it is necessary to perform a preliminary stochastic mapping to corrupt the data ($x \to \tilde{x}$).
  - E.g. by adding Gaussian noise.
- The loss is computed between the reconstruction (computed from the noisy input) against the original noise-free data.

Sparse Autoencoders

- In a sparse autoencoder, there can be more hidden units than inputs, but only a small number of the hidden units are allowed to be active at the same time.
- This is simply achieved with a regularised loss function: $\ell = \ell_{mse} + \Omega(\mathbf{h})$
- A popular choice that you've seen before would be to use an l1 penalty $\Omega(\mathbf{h}) = \lambda \sum_i |h_i|$
  - this of course does have a slight problem... what is the derivative of $y = |x|$ with respect to x at $x = 0$?

Autoencoder Applications

- Any basic AE (or its variant) can be used to learn a compact representation of data.
  - You can learn useful features from data without the need for labelled data.
  - Denoising can help generalise over the test set since the data is distorted by adding noise.
- Pretraining networks
- Anomoly Detection
- Machine translation
- Semantic segmentation

Beyond Deterministic Autoencoders: Stochastic Encoders and Decoders

- When we trained supervised classification networks we usually assume that the network produces an output distribution $p(\mathbf{y}|\mathbf{x})$ and try to minimise the negative log-likelihood $-\log(p(\mathbf{y}|\mathbf{x}))$.
- In a decoder of an autoencoder we could do the same thing and have the decoder learn $p_{decoder}(\mathbf{x}|\mathbf{h})$ by minimising $-\log(p(\mathbf{x}|\mathbf{h})$.
  - A linear output layer could parameterise the mean of a Gaussian distribution for real-valued $\mathbf{x}$; in this case the negative log likelihood yields the MSE criterion.

  - Binary $\mathbf{x}$ would correspond to a Bernoulli distribution parameterised by sigmoid outputs
  - Discrete (or categorical) $\mathbf{x}$ would correspond to a softmax distribution.

- What about the encoder - could we make that output $p(\mathbf{h}|\mathbf{x})$?

14. Generative models

Introduction

- What is generative modelling and why do we do it?
- Differentiable Generator Networks
- Variational Autoencoders
- Generative Adversarial Networks

14.1. Generative Modelling and Differentiable Generator Networks

Recap: Generative Models

- Learn models of the data: $p(\mathrm{x})$
- Learn *conditional* models of the data: $p(\mathrm{x}|\mathrm{y} = y)$
- Some generative models allow the probability distributions to be evaluated explicitly
  - i.e. compute the probability of a piece of data x: $p(\mathrm{x} = x)$
- Some generative models allow the probability distributions to be sampled
  - i.e. draw a sample x based on the distribution: $x \sim p(\mathrm{x})$
- Some generative models can do both of the above
  - e.g. a Gaussian Mixture Model is an explicit model of the data using k Gaussians
    - The likelihood of data x is the weighted sum of the likelihood from each of the k Gaussians

    * Sampling can be achieved by sampling the categorical distribution of k weights followed by sampling a data point from the corresponding Gaussian

Why do generative modelling?

- Try to understand the processes through which the data was itself generated
  - Probabilistic latent variable models like VAEs or topic models (PLSA, LDA, ...) for text
  - Models that try to disentangle latent factors like β-VAE
- Understand how likely a new or previously unseen piece of data is
  - outlier prediction, anomaly detection, ...
- Make 'new' data
  - Make 'fake' data to use to train large supervised models?
  - 'Imagine' new, but plausible, things?

Differentiable Generator Networks

- Generative Modelling is not new; we've known how to make arbitrarily complex probabilistic graphical models for many years.
  - ...But difficult to train and scale to real data, relying on MCMC.
- The past few years has seen major progress along four loose strands:
  - Invertible density estimation - A way to specify complex generative models by transforming a simple latent distribution with a series of invertible functions.
  - Autoregressive models - Another way to model $p(x)$ is to break the model into a series of conditional distributions: $p(x) = p(x_1)p(x_2|x_1)p(x_3|x_2, x_1)$ ...
  - Variational autoencoders - Latent-variable models that use a neural network to do approximate inference.

  - Generative adversarial networks - A way to train generative models by optimizing them to fool a classifier

- **Common thread in recent advances is that the loss functions are end-to-end differentiable.**

Differentiable Generator Networks: key idea

- We're interested in models that transform samples of latent variables $\mathbf{z}$ to
  - samples x, or,
  - distributions over samples x
- The model is a (differentiable) function $g(\mathbf{z}, \boldsymbol{\theta})$
  - typically g is a neural network.

Example: drawing samples from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$

- Consider a simple generator network with a single affine layer that maps samples $\mathcal{N}(\mathbf{0}, \mathbf{I})$ to $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$: [1em]

$$\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \longrightarrow \boxed{g_{\boldsymbol{\theta}}(\mathbf{z})} \longrightarrow \mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

- Note: Exact solution is $\mathbf{x} = g_{\boldsymbol{\theta}}(\mathbf{z}) = \boldsymbol{\mu} + \mathbf{L}\mathbf{z}$ where $\mathbf{L}$ is the Cholesky decomposition of $\boldsymbol{\Sigma}$: $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^\top$, lower triangular $\mathbf{L}$.

Generating samples

More generally, we can think of g as providing a nonlinear change of variables that transforms a distribution over $\mathbf{z}$ into the desired distribution over $\mathbf{x}$:

$$p_z(\mathbf{z}) \longrightarrow \boxed{g(\mathbf{z})} \longrightarrow p_x(\mathbf{x})$$

For any *invertible, differentiable, continuous* g:

$$p_z(\mathbf{z}) = p_x(g(\mathbf{z})) \left| \det \left(\frac{\partial g}{\partial \mathbf{z}} \right) \right|$$

Which implicitly imposes a probability distribution over $\mathbf{x}$:

$$p_x(\mathbf{x}) = \frac{p_z(g^{-1}(\mathbf{x}))}{\left|\det\left(\frac{\partial g}{\partial \mathbf{z}}\right)\right|}$$

Note: usually use an indirect means of learning g rather than minimise $-\log(p(\mathbf{x}))$ directly

Generating distributions

- Rather than use g to provide a sample of $\mathbf{x}$ directly, we could instead use g to define a conditional distribution over x, $p(\mathbf{x}|\mathbf{z})$
  - For example, g might produce the parameters of a particular distribution - e.g.:
    - means of Bernoulli
    - mean and variance of a Gaussian
- The distribution over $\mathbf{x}$ is imposed by marginalising $\mathbf{z}$:$p(\mathbf{x}) = \mathbb{E}_{\mathbf{z}} p(\mathbf{x}|\mathbf{z})$

Distributions vs Samples

- In both cases (g generates samples and g generates distributions) we can use the reparameterisation tricks we saw last lecture to train models.
- Generating distributions:
  - \+ works for both continuous and discrete data
  - \- need to specify the form of the output distribution
- Generating samples:
  - \+ works for continuous data
    - \+ discrete data is recently possible - we need the STargmax
  - \+ don't need to specify the distribution in explicit form

Complexity of Generative Modelling

- In classification both input and output are given
  - Optimisation only needs to learn the mapping

- Generative modelling is more complex than classification because
  - learning requires optimizing intractable criteria
  - data does not specify both input $\mathbf{z}$ and output $\mathbf{x}$ of the generator network
  - learning procedure needs to determine how to arrange $\mathbf{z}$ space in a useful way and how to map $\mathbf{z}$ to $\mathbf{x}$

14.2. Variational Autoencoders

Variational Autoencoders (VAEs)

The Variational Autoencoder uses the following generative process to draw samples:

$$\mathbf{z} \sim p_{\text{model}}(\mathbf{z}) \longrightarrow \boxed{p_{\text{model}}(\mathbf{x}|\mathbf{z};\boldsymbol{\theta}) = p_{\text{model}}(\mathbf{x}; g_{\boldsymbol{\theta}}(\mathbf{z}))} \longrightarrow \mathbf{x} \sim p_{\text{model}}(\mathbf{x}|\mathbf{z};\boldsymbol{\theta})$$

- The learning problem is to find $\boldsymbol{\theta}$ that maximises the probability of each $\mathbf{x}$ in the training set under $p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z};\boldsymbol{\theta})p(\mathbf{z})d\mathbf{z}$
- $p_{\text{model}}(\mathbf{z})$ is most often chosen to be $\mathcal{N}(\mathbf{0}, \mathbf{I})$
- $p_{\text{model}}(\mathbf{x}|\mathbf{z})$ is chosen according to the data; typically Gaussian for real-valued data (most often just predicting the means, with a fixed diagonal covariance) or Bernoulli for binary data.
  - Intuition: we don't exactly want to exactly create the training examples; we want to create things *like* the training examples

Variational Autoencoders (VAEs)

- Conceptually we can compute $p(\mathbf{x}) \approx \frac{1}{n}\sum_i^n p(\mathbf{x}|\mathbf{z}_i;\boldsymbol{\theta})$ for n samples of $\mathbf{z}$, $\{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$ and just use gradient ascent to do the optimisation
  - This isn't tractable in practice; n would need to be *extremely* big!
- For most $\mathbf{z}$, $p(\mathbf{x}|\mathbf{z})$ will be nearly zero, and hence contribute almost nothing to our estimate of $p(\mathbf{x})$

- The key idea behind the VAE is to learn to sample values of $\mathbf{z}$ that are likely to have produced $\mathbf{x}$, and compute $p(\mathbf{x})$ just from those
  - Introduce a new function $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ which can take a value of $\mathbf{x}$ and produce the distribution over $\mathbf{z}$ values that are likely to produce $\mathbf{x}$.
  - The space of $\mathbf{z}$ values that are likely under q should be much smaller than the space of than under prior $p(\mathbf{z})$.
  - We can now compute $\mathbb{E}_{\mathbf{z}\sim q_{\boldsymbol{\phi}}} p(\mathbf{x}|\mathbf{z};\boldsymbol{\theta})$ easily
    - if the PDF $q(\mathbf{z})$, is not $\mathcal{N}(\mathbf{0},\mathbf{I})$, then how does that help us optimize $p(\mathbf{x})$?
    - and how does this expectation relate to $p(\mathbf{x})$?

Variational Inference

$$
\begin{aligned}
\text{Log-probability} \quad & \log p(x) = \log \int p(x|z)p(z)dz \\
\text{Proposal} \quad & \log p(x) = \log \int p(x|z)p(z)\frac{q(z|x)}{q(z|x)}dz \\
\text{Importance weight} \quad & \log p(x) = \log \int p(x|z)\frac{p(z)}{q(z|x)}q(z|x)dz \\
\text{Jensen's inequality} \quad & \log p(x) \geq \int q(z|x)\log\left(p(x|z)\frac{p(z)}{q(z|x)}\right)dz \\
\text{Rearrange} \quad & \log p(x) \geq \int q(z|x)\log p(x|z)dz - \int q(z|x)\log\frac{q(z|x)}{p(z)}dz \\
\text{ELBO} \quad & \log p(x) \geq \mathbb{E}_{z\sim q(z|x)}\log p(x|z) - D_{\mathrm{KL}}(q(z|x)||p(z))
\end{aligned}
$$

Jensen's inequality: $\log \int p(x)g(x)dx \geq \int p(x)\log g(x)dx$ Log product rule: $\log(a\cdot b) = \log a + \log b$ Log quotient rule: $\log(a/b) = \log a - \log b$

The Evidence LOwer Bound (ELBO) / variational lower bound

The ELBO expression we just derived is a cornerstone of variational inference:

$$
\begin{aligned}
\mathcal{L}(q) &= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\log p_{\text{model}}(\mathbf{x}|\mathbf{z}) - D_{\mathrm{KL}}(q(\mathbf{z}|\mathbf{x})||p_{\text{model}}(\mathbf{z})) \\
&\leq \log p_{\text{model}}(\mathbf{x})
\end{aligned}
$$

- The expectation term looks just like a reconstruction log-likelihood found in normal autoencoders

  - If $p_{\text{model}}(\mathbf{x}|\mathbf{z})$ is Gaussian, then this is MSE between the true training $\mathbf{x}$ and a generated sample computed from $\mathbf{z}$, averaged across many $\mathbf{z}$'s (each a function of $\mathbf{x}$)

- The KL term is forcing the approximate posterior $q(\mathbf{z}|\mathbf{x})$ towards the prior $p_{\text{model}}(\mathbf{z})$.

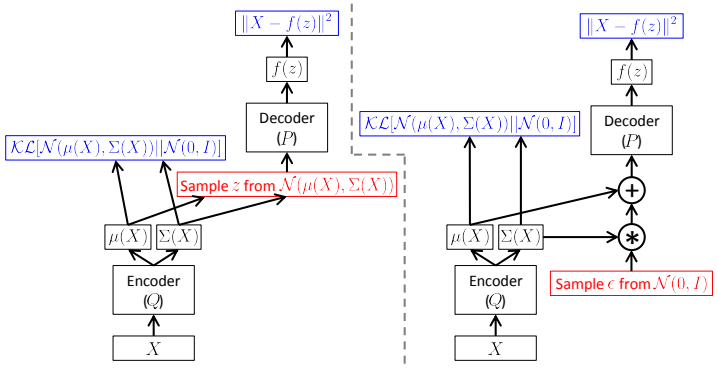
Why is it called an autoencoder?

- $q(\mathbf{z}|\mathbf{x})$ is referred to as an encoder; it's used to take $\mathbf{x}$ and turn it into a $\mathbf{z}$

- $p_{\text{model}}(\mathbf{x}; g_{\boldsymbol{\theta}}(\mathbf{z}))$ is referred to as a decoder network; it takes a $\mathbf{z}$ and decodes it into a target $\mathbf{x}$

- From a practical standpoint, a VAE is a normal autoencoder with two key differences:
  - the encoder generates a distribution that must be sampled
    - the network produces the sufficient statistics of the distribution (e.g. means and diagonal co-variances for a typical VAE with Gaussian $q(\mathbf{z}|\mathbf{x})$)
  - the decoder generates a distribution, which, during training the NLL of the true data $\mathbf{x}$ is compared against

VAE: Diagram

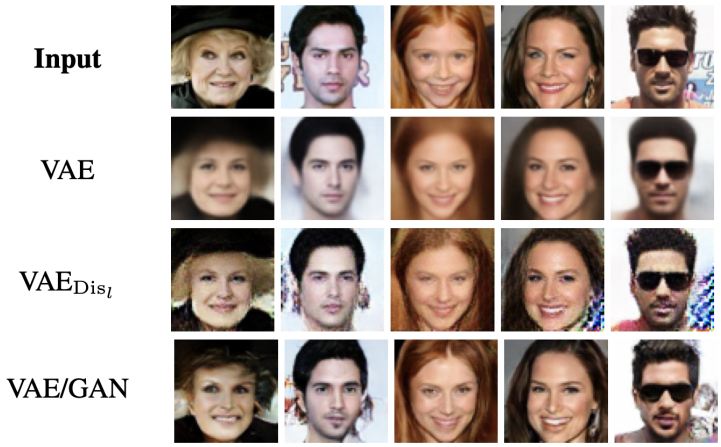


From Carl Doersch's Tutorial on VAEs - https://arxiv.org/pdf/1606.05908.pdf

VAE Models and Performance

- VAEs can be used with any kind of data
  - the distributions and network architecture just needs to be set accordingly
  - e.g. it's common to use convolutions in the encoder and transpose convolutions in (Gaussian) decoder for image data
- VAEs have nice learning dynamics; they tend to be easy to optimise with stable convergence
- VAEs have a reputation for producing blurry reconstructions of images
  - Not fully understood why, but most likely related to a side effect of maximum-likelihood training
- VAEs tend to only utilise a small subset of the dimensions of $\mathbf{z}$
  - Pro: automatic latent variable selection
  - Con: better reconstructions should be possible given the available code-space

Reconstructions Example



Sampling Example

14.3. Generative Adversarial Networks

Generative Adversarial Networks (GANs)

- New (old?![1]) method of training deep generative models
- Idea: pitch a generator and a discriminator against each other
  - Generator tries to draw samples from $p(x)$
  - Discriminator tries to tell if sample came from the generator (fake) or the real world
- Both discriminator and generator are deep networks (differentiable functions)
- LeCun quote 'GANs, the most interesting idea in the last ten years in machine learning'

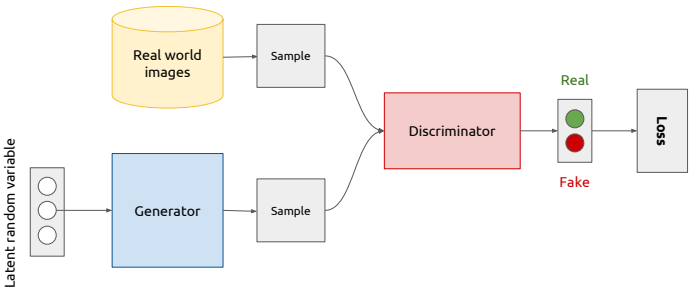
Aside: Adversarial Learning vs. Adversarial Examples

The approach of GANs is called adversarial since the two networks have *antagonistic* objectives.

This is not to be confused with *adversarial examples* in machine learning.

[1] c.f. Schmidhuber

Generative adversarial networks (conceptual)



See these two papers for more details: `https://arxiv.org/pdf/1412.6572.pdf` `https://arxiv.org/pdf/1312.6199.pdf`

pause

More Formally

- The **generator**

$$\mathbf{x} = g(\mathbf{z})$$

  is trained so that it gets a random input $\mathbf{z} \in \mathbb{R}^n$ from a distribution (typically $\mathcal{N}(\mathbf{0}, \mathbf{I})$ or $\mathcal{U}(\mathbf{0}, \mathbf{I})$) and produces a sample $\mathbf{x} \in \mathbb{R}^d$ following the data distribution as output (ideally). Usually $n << d$.

- The **discriminator**

$$y = d(\mathbf{x})$$

  gets a sample $\mathbf{x}$ as input and predicts a probability $y \in [0, 1]$ (or real-valued logit of a Bernoulli distribution) determining if it is real or fake.

More Practically

- Training a standard GAN is difficult and often results in two undesirable behaviours

  - Oscillations without convergence. No guarantee that the loss will actually decrease...
    * It has been shown that a GAN has saddle-point solution, rather than a local minima.
  - The **mode collapse** problem, when the generator models very well a small sub-population, concentrating on a few modes.
- Additionally, performance is hard to assess and often boils down to heuristic observations.

Deep Convolutional Generative Adversarial Networks (DCGANs)

- Motivates the use of GANS to learn reusable feature representations from large unlabelled datasets.
- GANs known to be unstable to train, often resulting in generators that produce "nonsensical outputs".
- Model exploration to identify architectures that result in **stable** training across datasets with higher resolution and deeper models.

Architecture Guidelines for Stable DCGAN

- Replace pooling layers with strided convolutions in the discriminator and fractional-strided (transpose) convolutions in the generator.
  - This will allow the network to learn its own spatial downsampling.
- Use batchnorm in both the generator and the discriminator.
  - This helps deal with training problems due to poor initialisation and helps the gradient flow.
- Eliminate fully connected hidden layers for deeper architectures.
- Use ReLU activation in the generator for all layers except for the output, which uses tanh.
- Use LeakyReLU activation in the discriminator for all layers.

Summary

- Generative modelling is a massive field with a long history
- Differentiable generators have had a profound impact in making models that work with real data at scale
- VAEs and GANs are currently the most popular approaches to training generators for spatial data
- We've only scratched the surface of generative modelling
  - Auto-regressive approaches are popular for sequences (e.g. language modelling).
    - But also for images (e.g. PixelRNN, PixelCNN)
  - typically RNN-based
  - but not necessarily - e.g. WaveNet is a convolutional autoregressive generative model

15. Attention

Part III

Structure, Innate Priors and Inductive Biases

16. Learning Representations of Sets

17. Differentiable relaxations of drawing

18. Learning to communicate

19. Learning with graphs and geometric data

20. Self-supervised and multi-objective learning

21. Neural arithmetic and logic units

22. Searching for architectures

Solutions

Answer of exercise 1.1

geometric proof + squeezing

Answer of exercise 1.2

...

Answer of exercise 1.3

Linear mappings satisfy $f(\boldsymbol{a}+\boldsymbol{b}) = f(\boldsymbol{a}) + f(\boldsymbol{b})$ and $f(c\boldsymbol{a}) = cf(\boldsymbol{a})$. Letting $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{A}$, then,

$$\begin{aligned}(g \circ f)(\boldsymbol{a}+\boldsymbol{b}) &= g(f(\boldsymbol{a}) + f(\boldsymbol{b})) \\ &= g(f(\boldsymbol{a})) + g(f(\boldsymbol{b})) \\ &= (g \circ f)(\boldsymbol{a}) + (g \circ f)(\boldsymbol{b})\end{aligned}$$

and,

$$\begin{aligned}(g \circ f)(c\boldsymbol{a}) &= g(f(c\boldsymbol{a})) \\ &= g(cf(\boldsymbol{a})) \\ &= cg(f(\boldsymbol{a})) \\ &= c(g \circ f)(\boldsymbol{a})\end{aligned}$$

Thus $g \circ f$ must also be a linear mapping.

Bibliography