Machine Learning in Science

(MLiS)

Part II

Dr Maggie Lieu[1], Dr Adam Moss[2], Dr Dominic Rose[3]

[1]maggie.lieu@nottingham.ac.uk
[2]adam.moss@nottingham.ac.uk
[3]dominic.rose1@nottingham.ac.uk

These lecture notes cover Machine Learning in Science (MLiS) Part 2. They begin with an introduction to deep learning, showing how one can build and train neural networks from first principles. We next introduce convolutional and recurrent neural networks, classes of network most commonly applied to image and sequential data, giving examples of the latest developments in the field. We then cover more advanced topics, such as how to deal with unlabelled data using unsupervised learning, how to transfer knowledge learnt in one domain to another using transfer learning, and how neural networks can be trained to take actions in an environment using reinforcement learning. Code examples complementary to the lecture notes are highlighted in framed boxes, and are hosted on Google Colab (`https://colab.research.google.com/`).

"Success in creating AI would be the biggest event in human history. Unfortunately, it might also be the last, unless we learn how to avoid the risks."

Stephen Hawking

Useful resources

- Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, (Available online at `https://www.deeplearningbook.org//`) This will be the main book we follow. Well written covering all the main topics you will need.
- Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson, Charles K. Fisher, David J. Schwab, *A high-bias, low-variance introduction to Machine Learning for physicists*, (Available at `https://arxiv.org/abs/1803.08823`) An excellent review paper written by physicists.
- Michael Elad, *Sparse and redundant representations: from theory to applications in signal and image processing*,
- Christoph Molnar, *Interpretable Machine Learning*, (Available online at `https://christophm.github.io/interpretable-ml-book/`).

Contents

1 Introduction to Deep Learning

In the last few years we have witnessed a revolution in the use of machine learning (ML) and deep learning (DL) to solve problems previously deemed intractable. Deep learning is a class of machine learning which aims to 'teach' a computer an abstract **representation** of data. While much of the conceptual and methodological basis for ML was developed in the 80s and 90s, their extensive practical application became feasible only recently due to a combination of increased computational power and of the ready availability of large data sets for training and analysis.

In general, the performance of many ML algorithms is heavily dependant on the representation of data, or **features** they learn from. For many tasks, it is difficult to know exactly what features should be provided. As an example, suppose we'd like to design an algorithm to predict what mark you will receive in this course. The primary **factors of variation** that affect your mark might include (1) the effort you put in and (2) your ability. However, it is difficult to observe these factors directly. Instead, we may observe other features such as the amount of time spent studying and your previous marks or degree class. Some factors of variation cannot easily be separated, and depend on multiple observables. We could then arrange these features into a hand-crafted graphical model, as shown in the left of Fig. 1.0.1, and attempt to learn the rela-

tionship between nodes of the graph.

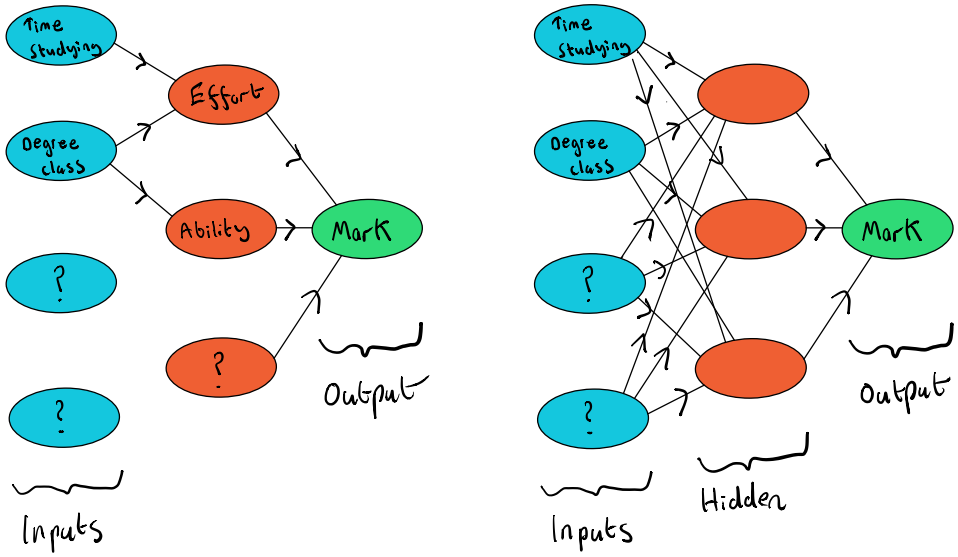


Figure 1.0.1: (Left) Graphical model predicting your mark in MLiS2. (Right) Deep learning approach using a neural network.

Deep learning aims to solve this problem by learning a representation of data that depends on other, simpler representations, without any manual 'feature engineering'. This representation is encoded by the parameters of a neural network (NN). The archetypal example of a neural network is the multilayer perceptron (MLP), which consists of at least 3 layers of nodes: an input layer, one or more **hidden** layers (called hidden as they are not observed) and an output layer. This is illustrated for our example problem in the right of Fig. 1.0.1. Each input feature is connected to a hidden node, and the model must determine which relations are most important to explain the observed data. Learning proceeds by updating the parameters of the model to best fit a set of labelled training data, which (hopefully!) generalises well to unseen test data. This is

called **supervised learning**, as each example in the input data has an associated **label** or **target**. In the following sections we will focus on supervised learning, and return to the case of unsupervised learning in section **??**.

1.1 Neural Network Basics

1.1.1 Notation

We follow a similar notation as introduced in the first semester. A single data point, labelled by i, is given by $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$, where the input vector $\mathbf{x}^{(i)} \in \mathbb{R}^P$ and the target vector $\mathbf{y}^{(i)} \in \mathbb{R}^K$. The components of $\mathbf{x}^{(i)}$ are therefore denoted as $x_P^{(i)}$, so that $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_P^{(i)})^T$. A neural network outputs a prediction $\hat{\mathbf{y}}^{(i)}(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the parameters of the network. If the target and output of the network is a scalar, such that $K = 1$, we indicate these by $\hat{y}^{(i)}$ and $\hat{y}^{(i)}$ respectively.

In total, we have N examples in our dataset (in the case of a **mini-batch** of data, the batch dimension takes the place of N), from which we can construct the input or **design** matrix $\mathbf{X} \in \mathbb{R}^{N \times P}$, so that each *row* corresponds to a different feature and each *column* contains a different example. Note that there are two conventions in use in the machine learning community, **feature by column**, and **feature by row**. We use the former, as this is closer to frameworks such as PyTorch and Tensorflow where the first dimension is the batch dimension. Please note that the choice of convention will impact on the matrix dimensions of intermediate layers and weights in the network.

The output matrix can similarly defined $\mathbf{Y} \in \mathbb{R}^{N \times K}$, along with the prediction matrix $\hat{\mathbf{Y}} \in \mathbb{R}^{N \times K}$. This dataset can be split

into training, test and validation sets, which we label as e.g. $\mathbf{X}^{(\text{train})}$ and $\mathbf{Y}^{(\text{train})}$. It can sometimes be helpful to have a first 'artificial' column of $\mathbf{X}$ of all ones, which we label by $p = 0$, so that $\mathbf{X} \in \mathbb{R}^{N \times (P+1)}$.

1.1.2 Loss Function

The goal of supervised learning is usually to minimise the prediction error. This can be achieved by minimising a **loss** function (also called the **objective** or **cost** function), given by

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i}^{N} \ell(\hat{\mathbf{y}}^{(i)}(\boldsymbol{\theta}), \mathbf{y}^{(i)}) \,, \tag{1.1.1}$$

where the per-example loss, $\ell(\hat{\mathbf{y}}^{(i)}), \mathbf{y}^{(i)})$, is dependent on the particular problem or task. Common loss functions are given in Table. 1.1.2.

For regression tasks on continuous target data, the Mean Square Error (MSE) and Mean Absolute Error (MAE) are typically used. For binary classification, where the targets $y \in \{0, 1\}$, the Binary Cross Entropy (BCE) is more suitable. More generally, for categorical data in M classes and targets $y \in \{0, 1, 2, \ldots M-1\}$, one can generalise the BCE to the Categorical Cross Entropy (CCE) by creating a 'one-hot' target vector such that

$$y_m^{(i)} = \begin{cases} 1, & \text{if } y^{(i)} = m \\ 0, & \text{otherwise} \end{cases} \tag{1.1.2}$$

with the prediction vector in the same 'one-hot' format. The loss function is often supplemented by additional terms that implement regularisation, as discussed in section 1.7.

Having defined a loss function, the optimal set of parameters $\boldsymbol{\theta}^\star$ are given by

$$\boldsymbol{\theta}^\star = \text{argmin}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})\,. \tag{1.1.3}$$

Most modern deep learning frameworks perform optimisation by variants of **gradient descent**, as discussed in section 1.5.

Loss	Applications	$\ell(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$
Mean Square Error (MSE)	Regression	$\left(\hat{y}^{(i)} - y^{(i)}\right)^2$
Mean Absolute Error (MAE)	Regression	$\lvert\hat{y}^{(i)} - y^{(i)}\rvert$
Binary Cross Entropy (BCE)	Binary classification	$-\left(y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\right)$
Categorical Cross Entropy (CCE)	Multi-class classification	$-\sum_{m=0}^{M-1} y_m^{(i)} \log \hat{y}_m^{(i)}$

Table 1.1: Common loss functions $\ell(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$ and their typical applications.

1.1.3 Neurons

The basic building block of a neural network is an artificial **neuron**, illustrated in the left of Fig. 1.1.1. A neuron takes an input vector $\boldsymbol{x}$ and outputs the scalar $a(\boldsymbol{x})$, given by

$$z = \boldsymbol{x}^T \boldsymbol{w} + b = \mathbf{x}^T \mathbf{w}\,, \quad a = \Phi(z)\,, \tag{1.1.4}$$

where $\mathbf{x} = (1, \boldsymbol{x})^T$, $\mathbf{w} = (b, \boldsymbol{w})^T$ and Φ is called the **activation** function, where the output a is called the **activation**. With this definition, the bias of the neuron b is absorbed into the term containing the weights $\mathbf{w}$. A **multi-layer perceptron** consists of many such neurons stacked into layers, with the output of one layer serving as the input for the next (see Fig. 1.1.1). Each

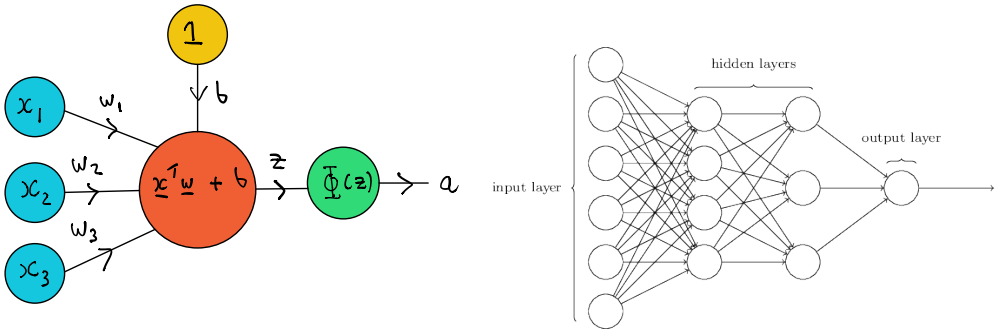


Figure 1.1.1: **Basic architecture of a multi-layer perceptron.** (Left) The basic component is a neuron, consisting of a linear transformation that weights the importance of various inputs, followed by an activation function. (Right) Neurons are arranged into layers with the output of one layer serving as the input to the next layer.

neuron has a different set of weights and bias. The first layer is called the input or visible layer, the middle layers the hidden layers and the final layer the output layer.

1.1.4 Activation Functions

The activation function Φ is usually non-linear, and common choices include the sigmoid function and hyperbolic tangent, shown in Fig. 1.1.2 and Table 1.1.4. More recently, the rectified linear units (ReLUs), leaky rectified linear units (leaky ReLUs), and exponential linear units (ELUs) have become popular. The reason for this is that neural networks are usually trained by **back-propagation** (see section 1.3), which requires taking the derivative of the activation function, and it is trivial to calculate gradients of these functions. Another reason is that if the input z of the activation function becomes large, the derivative of the sigmoid function and hyperbolic tangent becomes small. In this case, the activation function becomes saturated and such

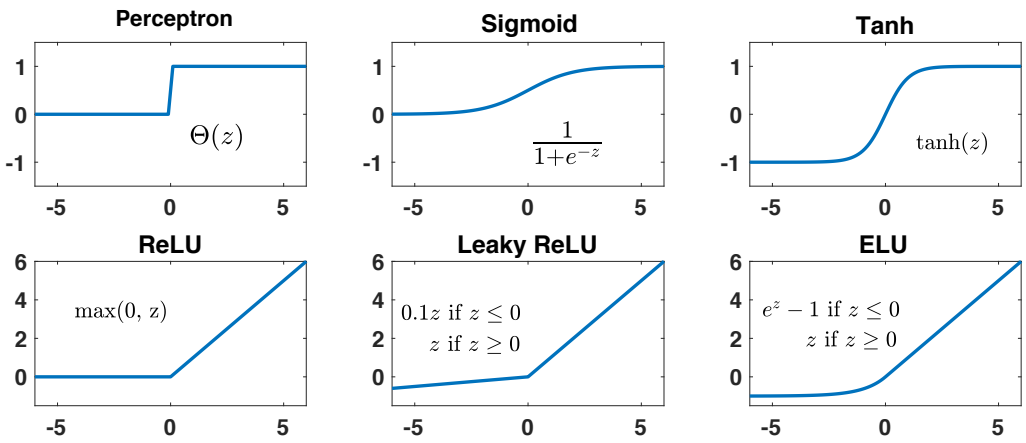


Figure 1.1.2: Common activation functions $\Phi(z)$ for neurons.

'vanishing gradients' makes it harder to train the network by back-propagation.

1.2 Basic Types of Network

1.2.1 Linear Model

`https://github.com/adammoss/MLiS2/blob/master/examples/intro/linear.ipynb`

The simplest type of neural network is the linear model, which maps an input vector $\boldsymbol{x}$ to a scalar output $\hat{y}$ by

$$\hat{y} = \mathbf{x}^T \mathbf{w}\,, \tag{1.2.1}$$

where $\mathbf{x} = (1, \boldsymbol{x})^T$ and $\mathbf{w} = (b, \boldsymbol{w})^T$. A linear model has no hidden layers, and, as the name suggests, a linear activation

Activation	$\Phi(z)$	$\frac{\partial \Phi}{\partial z}$
Linear	z	1
Heaviside	$\begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{otherwise} \end{cases}$	$\delta(z)$
Sigmoid	$\frac{1}{1+e^{-z}}$	$\frac{e^{-z}}{(1+e^{-z})^2} = \sigma(z)\,(1-\sigma(z))$
Hyperbolic tangent	$\tanh z$	$\operatorname{sech}^2 z = \frac{4}{(e^{-x}+e^{x})^2}$
ReLU	$\begin{cases} 0, & \text{if } z < 0 \\ z, & \text{otherwise} \end{cases}$	$\begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{otherwise} \end{cases}$
Leaky ReLU	$\begin{cases} \alpha z, & \text{if } z < 0 \\ z, & \text{otherwise} \end{cases}$	$\begin{cases} \alpha, & \text{if } z < 0 \\ 1, & \text{otherwise} \end{cases}$
ELU	$\begin{cases} e^z - 1, & \text{if } z < 0 \\ z, & \text{otherwise} \end{cases}$	$\begin{cases} e^z, & \text{if } z < 0 \\ 1, & \text{otherwise} \end{cases}$

Table 1.2: Common activation functions $\Phi(z)$ for neurons and their derivatives $\frac{\partial \Phi}{\partial z}$.

function. Computationally it is more efficient to pass through an entire **batch** of inputs, in which case

$$\hat{\mathbf{Y}} = \mathbf{X}\mathbf{w}\,. \tag{1.2.2}$$

Let us see how the linear model performs on two simple problems: learning the AND and XOR (exclusive OR) functions. The AND function is an operation on two binary values, x_1 and x_2, which is equal to 1 when $x_1 = x_2 = 1$ and 0 otherwise. The inputs and targets are then (with the first column of $\mathbf{X}$ all ones)

$$\mathbf{X} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}. \tag{1.2.3}$$

For the XOR problem, when exactly one of x_1 or x_2 is equal to 1 it returns 1, otherwise it returns 0. The inputs and targets

are then

$$\mathbf{X} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{1.2.4}$$

We choose to model this as a regression problem and so use the MSE loss. In reality, when dealing with binary data, the BCE is usually more appropriate, but using the MSE simplifies the maths. The loss is then (ignoring the factor of $1/N$)

$$J(\mathbf{w}) = (\mathbf{X}\mathbf{w} - \mathbf{Y})^T (\mathbf{X}\mathbf{w} - \mathbf{Y}) . \tag{1.2.5}$$

The optimal weights can be found by calculating the gradient of the loss and solving the **normal equations**

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = 2\mathbf{X}^T\mathbf{X}\mathbf{w} - 2\mathbf{X}^T\mathbf{Y} = \mathbf{0}, \tag{1.2.6}$$

which gives the solution $\mathbf{w}^\star = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{Y}$. For the AND problem, the optimal weights and prediction vector are

$$\mathbf{w} = \begin{bmatrix} -1/4 \\ 1/2 \\ 1/2 \end{bmatrix}, \quad \hat{\mathbf{Y}} = \begin{bmatrix} -1/4 \\ 1/4 \\ 1/4 \\ 3/4 \end{bmatrix}. \tag{1.2.7}$$

For the XOR problem, the optimal weights and prediction vector are

$$\mathbf{w} = \begin{bmatrix} 1/2 \\ 0 \\ 0 \end{bmatrix}, \quad \hat{\mathbf{Y}} = \begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{bmatrix}. \tag{1.2.8}$$

The linear model completely fails to represent the XOR function, outputting $1/2$ for all examples.

1.2.2 Perceptron

```
https://github.com/adammoss/MLiS2/blob/master/
examples/intro/perceptron.ipynb
```

The perceptron model is given by

$$\hat{y} = \Theta\left(\mathbf{x}^T\mathbf{w}\right) , \tag{1.2.9}$$

where Θ is the Heaviside step function. It is an example of a **binary classifier**, and is capable of learning whether an example belongs to a specific binary class (labelled by 0 and 1). Because of the non-linearity of the step function we cannot solve for the weights using the normal equations.

Perceptrons can be trained using the perceptron learning algorithm, which iteratively updates the weights depending on the size of the prediction error:

1. Initialise the weights to either 0 or a small random value.
2. Iteratively update the weights by

$$\mathbf{w} \rightarrow \mathbf{w} - \alpha\mathbf{X}^T\left(\hat{\mathbf{Y}} - \mathbf{Y}\right) , \tag{1.2.10}$$

   where α is the **learning rate**. The choice of learning rate is not actually important for the perceptron learning algorithm, but we shall use this form to compare with gradient descent methods.
3. Repeat step 2 until a stopping condition ia met. This could be, for example, a maximum number of iterations, or a condition that the loss has not improved significantly from the previous iteration.

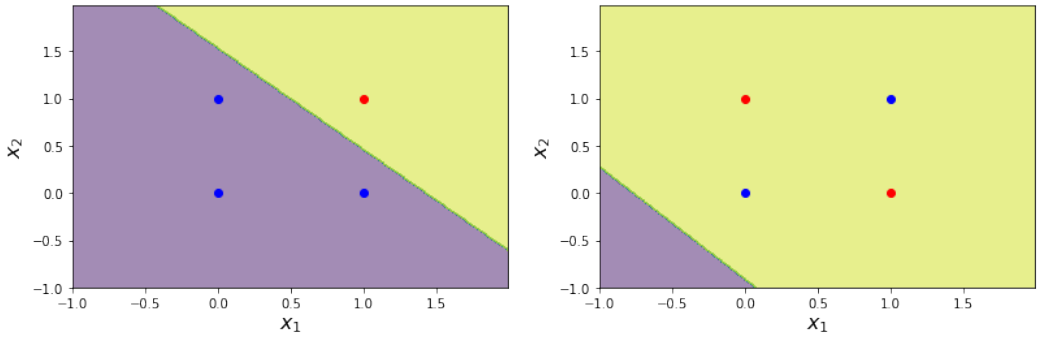


Figure 1.2.1: A perceptron learning the (left) AND function and (right) XOR function, for binary variables x_1 and x_2. Target variables equal to 0 and 1 are shown in blue and red respectively, along with the decision boundary of the prediction.

Let us again attempt to learn the AND and XOR functions. We set the initial weights to a small random value, the learning rate to $\alpha = 0.1$ and use a fixed number of iterations. The resulting **decision boundary** of the prediction, using the learnt weights, is shown in Fig. 1.2.1. The perceptron manages to classify the AND function correctly, but fails on the XOR function, giving all examples the same value or class. This is because the perceptron is a **linear classifier**, and will never be able to classify all examples correctly if the training data is not **linearly separable**, i.e. if all examples of one class cannot be separated from the other by a hyperplane.

The perceptron algorithm is guaranteed to converge on *some* solution in the case of a linearly separable training set, but it may still pick any solution, depending on the initial weights and training procedure. In the case of the AND function, for example, there are infinitely many solutions separating the two classes.

1.2.3 Logistic Regression

```
https://github.com/adammoss/MLiS2/blob/master/
examples/intro/logistic.ipynb
```

The logistic model is given by

$$\hat{y} = \sigma\left(\mathbf{x}^T\mathbf{w}\right) , \tag{1.2.11}$$

where σ is the sigmoid function. Logistic regression is commonly used to estimate the probability that an example belong to a specific binary class, i.e.

$$\hat{y} = P(y = 1|\mathbf{x}) . \tag{1.2.12}$$

There is no general closed form solution for the weights, so we will use a similar iterative procedure to the perceptron algorithm. Since the activation function is differentiable we can use **gradient descent**, and if the loss function is **convex** it is guaranteed to find the optimal solution. Gradient descent proposes a new set of weights

$$\mathbf{w} \to \mathbf{w} - \alpha\nabla_{\mathbf{w}} J(\mathbf{w}) , \tag{1.2.13}$$

where α is the learning rate, a positive scalar that determines the size of the update. There are several ways to choose α, which we will discuss in section 1.5. For now we will set it to a small constant. This process is then repeated for a number of **epochs**, defined as a complete pass over the training data, or until some other stopping condition is met, such as the loss not significantly improving from the previous epoch.

We will again attempt to learn the AND and XOR functions, but now use the BCE loss, given in Table. 1.1.2. In the logistic

model the gradient of the BCE loss with respect to the weights can be shown to be (see section 1.3)

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{N} \sum_{i}^{N} \left(\hat{y}^{(i)} - y^{(i)} \right) \mathbf{x}^{(i)} = \frac{1}{N} \mathbf{X}^T \left(\hat{\mathbf{Y}} - \mathbf{Y} \right) . \quad (1.2.14)$$

Compared to (1.2.10), gradient descent for logistic regression strongly resembles the perceptron learning algorithm.

We set the initial weights to a small random value, the learning rate to $\alpha = 0.1$ and use a fixed number of epochs. The resulting probability $P(y = 1|\mathbf{x})$, using the learnt weights, is shown in Fig. 1.2.2. Logistic regression manages to correctly predict the AND function, but fails on the XOR function, giving all examples an equal probability of 0.5. Similar to the perceptron, it is not able to classify all examples correctly if the training data is not linearly separable.

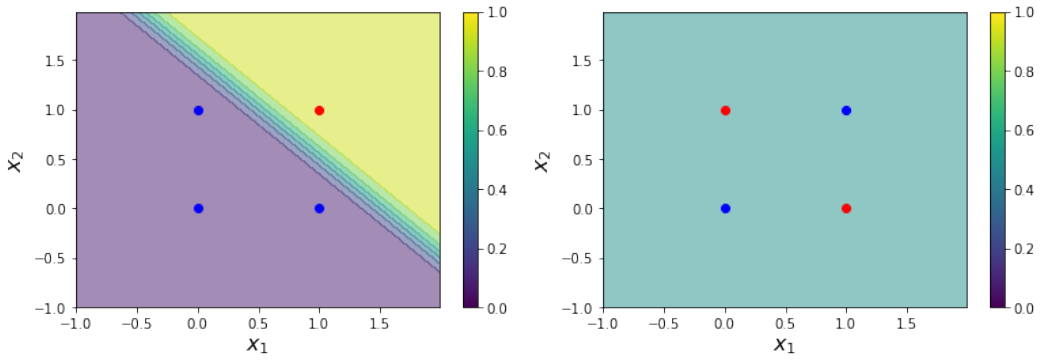


Figure 1.2.2: Logistic regression for the (left) AND function and (right) XOR function, for binary variables x_1 and x_2. Target variables equal to 0 and 1 are shown in blue and red respectively, along with the class probability $P(y = 1|\mathbf{x})$.

1.2.4 Multi-Layer Perceptron

```
https://github.com/adammoss/MLiS2/blob/master/
examples/intro/mlp.ipynb
```

So far all our models have failed to learn the XOR function. We now introduce a simple multi-layer perceptron (MLP) consisting of an input layer with two units, one hidden layer containing two hidden units and a single output unit, shown in the left of Fig. 1.2.3. This model is given by

$$\hat{y} = \boldsymbol{w}^{(T)}\boldsymbol{h} + b\,, \quad \boldsymbol{h} = \text{ReLU}\left(\boldsymbol{W}^{(T)}\boldsymbol{x} + \boldsymbol{c}\right)\,, \tag{1.2.15}$$

where the weights $\boldsymbol{W}$ connecting the input layer and hidden layer is now a matrix, with the number of rows equal to the input dimension and the number of columns equal to the number of hidden units (the hidden dimension), and the bias $\boldsymbol{c}$ is a vector. We apply a ReLU activation function to the hidden layer, as without it the whole network would remain a linear function of its input.

As before, we can absorb the bias into the weights and pass through an entire batch of inputs. The model is then

$$\hat{\mathbf{Y}} = \left(\text{ReLU}\left(\mathbf{X}\mathbf{W}\right)\right)\mathbf{w}\,, \tag{1.2.16}$$

illustrated in the right of Fig. 1.2.3. Here, each node of the graph represents an entire layer, with weight matrices connecting nodes and the bias absorbed into the weights. This style of network is much more compact, and we will use it from now onwards.

Note that for a single training example in (1.2.15), $\boldsymbol{x}$ and $\boldsymbol{h}$ are **column vectors**, since this is the default convention for

vectors. In the case of N training examples in (1.2.16) these are **matrices**, and since we use **feature-by-column** convention, with the number of rows equal to N, the transformation is now $\boldsymbol{XW}$.

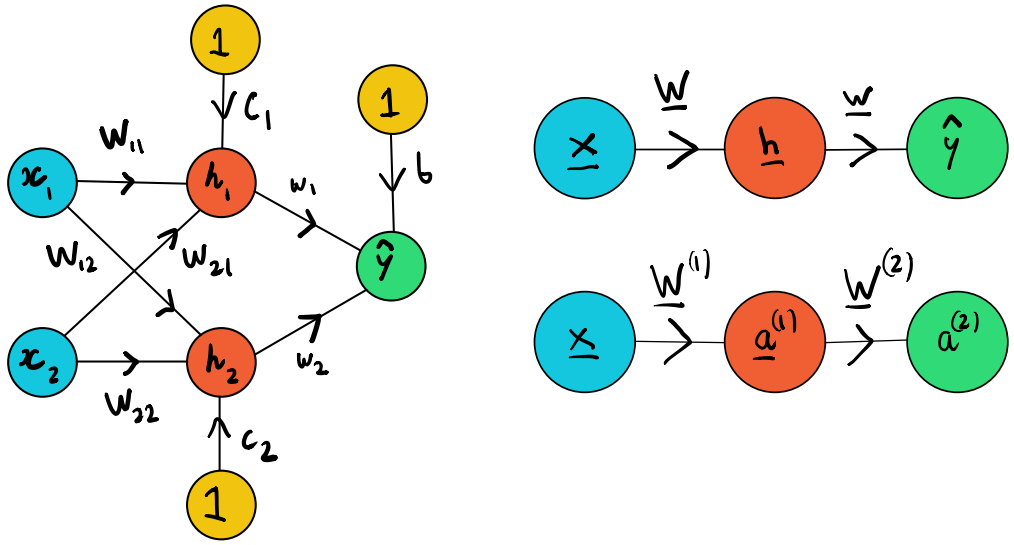


Figure 1.2.3: Multi-layer perceptron model used to learn the XOR function, with one hidden layer containing two hidden units, drawn in three different styles. (Left) Each unit is explicitly drawn with the connecting weights and biases. (Top-right) Each node of the graph represents an entire layer. (Bottom-right) Each node of the graph is also represented an entire layer, but we index the weights and activation by layer following the notation in 1.2.4.

When starting out with deep learning frameworks, one of the most common mistakes is not getting **tensor** (generalisations of vectors and matrices to arbitrary rank) dimensions correct when performing operations on them. Fig. 1.2.4 illustrates how the initial input matrix **X** for the AND and XOR datasets is propagated though the network of (1.2.16). Notice how the number of examples (the batch dimension) propa-

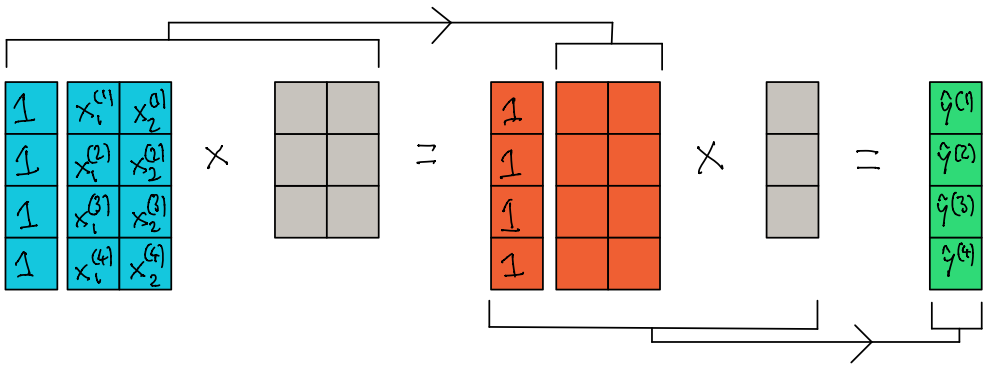


Figure 1.2.4: Illustration of how how the initial input matrix is propagated through the network of (1.2.16). The input matrix is shown in blue, the hidden layer in orange, the output matrix in green and the weight matrices in grey.

gates though the network, but the dimension of the weight matrix only depends on the dimensions of the preceding and following layers.

For the AND and XOR datasets, a gradient based solution can be used to find the best-fit parameters, but now the network is **deep**, it requires propagating gradients efficiently back through the network, which we discuss in section 1.3. For now we will simply specify the solution and show the network can reproduce the XOR function. The optimal weights are given by

$$\mathbf{W} = \begin{bmatrix} 0 & -1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}, \tag{1.2.17}$$

which reproduce exactly (1.2.4).

It is instructive to see how the MLP is able to learn the XOR function by plotting the representation of data it learns in the

$\boldsymbol{h}$ space, shown in Fig. 1.2.5. The original data is not linearly separable, but in the representation space it has mapped the two points with output 1 into a single point, and this space *is* now separable.

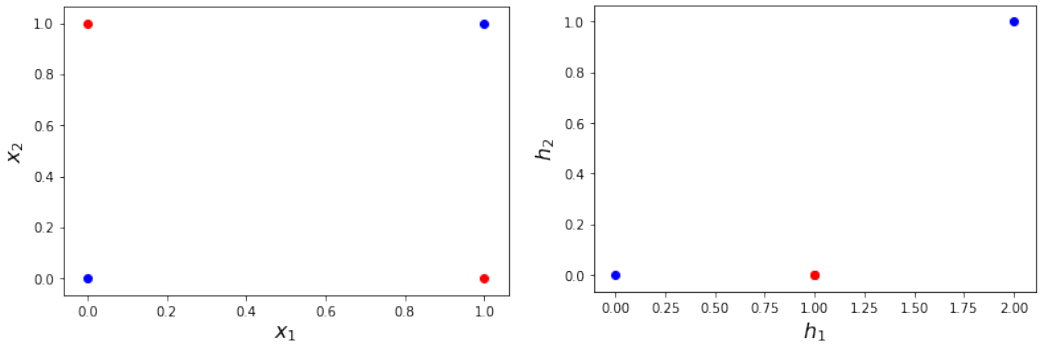


Figure 1.2.5: Data of the XOR function in the (left) original space (right) learnt representation space. Target variables equal to 0 and 1 are shown in blue and red respectively

Architecture

A key design consideration for neural networks is the **architecture**. For the multi-layer perceptron, this is determined by the *number of hidden layers* and *number of hidden units per layer*. We will assume that, excluding the input layer, there are L layers in our network with $l = 1, \ldots, L$ indexing the layer. In this structure, for a single training example $\boldsymbol{x}$, the first hidden layer and its activation is given by

$$\boldsymbol{z}^{(1)} = \boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)}, \quad \boldsymbol{a}^{(1)} = \Phi^{(1)}\left(\boldsymbol{z}^{(1)}\right), \tag{1.2.18}$$

the second hidden layer is given by

$$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(2)T}\boldsymbol{a}^{(1)} + \boldsymbol{b}^{(2)}, \quad \boldsymbol{a}^{(2)} = \Phi^{(2)}\left(\boldsymbol{z}^{(2)}\right), \tag{1.2.19}$$

until one reaches the output layer, given by

$$\boldsymbol{z}^{(L)} = \boldsymbol{W}^{(L)T}\boldsymbol{a}^{(L-1)} + \boldsymbol{b}^{(L)}, \quad \boldsymbol{a}^{(L)} = \Phi^{(L)}\left(\boldsymbol{z}^{(L)}\right). \tag{1.2.20}$$

The weighted input to the lth layer is therefore $\boldsymbol{z}^{(l)}$ and its activation is $\boldsymbol{a}^{(l)}$.

The weight $W_{ij}^{(l)}$ is the weight for the connection from the i-th neuron in layer $l-1$ to the j-th neuron in layer l. The loss J depends on the activation of the final layer, $\boldsymbol{a}^{(L)}$, which of course *indirectly* depends on all the activities of neurons in lower layers. This is shown for our example network in Fig. 1.2.3.

When we discuss convolutional neural networks in section 2, these will have different design considerations such as the number and size of convolutional filters. As we will see, deeper networks with fewer units per layer often generalise better to unseen test data, but they can be harder to train and optimise.

1.3 Back Propagation

Back-propagation can be summarised as an efficient way to apply the chain rule for partial differentiation, in order to calculate the gradient of the loss function with respect to the parameters of the network. For the MLP, it can be expressed by four basic equations. We will first write out each component explicitly to make the derivation clearer, then write these in vectorised form. Note that in this section we treat the weights and biases separately, and **do not** absorb the biases into the weights.

First, let us define the error of neuron j in layer l, $\Delta_j^{(l)}$, as the change in the loss function with respect to its weighted input

$z_j^{(l)}$:

$$\Delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} = \frac{\partial J}{\partial a_j^{(l)}} \Phi'^{(l)}(z_j^{(l)}), \quad \Phi'^{(l)}(z_j^{(l)}) = \frac{\partial \Phi^{(l)}(z_j^{(l)})}{\partial z_j^{(l)}} \tag{1.3.1}$$

This is the first of the four back-propagation equations. If the derivative of the activation function, $\Phi'^{(l)}(z_j^{(l)})$, becomes small it will be harder to train the network as the error will also decrease. The error can also be expressed in terms of the bias by

$$\Delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} = \frac{\partial J}{\partial b_j^{(l)}} \frac{\partial b_j^{(l)}}{\partial z_j^{(l)}} = \frac{\partial J}{\partial b_j^{(l)}}, \tag{1.3.2}$$

since $\partial b_j^{(l)} / \partial z_j^{(l)} = 1$.

In a feedforward network such as a multi-layer perceptron, the error depends on neurons in layer l only through the activation of neurons in the subsequent layer $l+1$, so we can use the chain rule to write

$$\begin{aligned} \Delta_j^{(l)} &= \frac{\partial J}{\partial z_j^{(l)}} = \sum_k \frac{\partial J}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \\ &= \sum_k \Delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \\ &= \left(\sum_k W_{jk}^{(l+1)} \Delta_k^{(l+1)} \right) \Phi'^{(l)}(z_j^{(l)}) . \end{aligned} \tag{1.3.3}$$

In vector form, these equations are

$$\boldsymbol{\Delta}^{(l)} = \frac{\partial J}{\partial \mathbf{a}^{(l)}} \Phi'^{(l)}(\mathbf{z}^{(l)}), \tag{1.3.4}$$

$$\boldsymbol{\Delta}^{(l)} = \frac{\partial J}{\partial \mathbf{b}^{(l)}}, \tag{1.3.5}$$

$$\boldsymbol{\Delta}^{(l)} = \left(\mathbf{W}^{(l+1)}\boldsymbol{\Delta}^{(l+1)}\right) \odot \Phi'^{(l)}(\mathbf{z}^{(l)}), \tag{1.3.6}$$

where $\odot$ is the element wise product. The final equation can be derived by differentiating the loss function with respect to the weight $W_{kj}^{(l)}$ as

$$\frac{\partial J}{\partial W_{kj}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial W_{kj}^{(l)}} = \Delta_j^{(l)} a_k^{(l-1)}. \tag{1.3.7}$$

Together, Eqs. (1.3.4), (1.3.5), (1.3.6), and (1.3.7) define the four back-propagation equations. The algorithm is then:

1. **Forward pass:** Starting with the input layer, perform a **forward pass** to compute $\mathbf{z}^{(l)}$ and $\mathbf{a}^{(l)}$ for each subsequent layer.

2. **Error at output layer:** calculate the error of the output layer using Eq. (1.3.4).

3. **Backward pass:** use Eq. (1.3.6) to propagate the error backwards and calculate $\boldsymbol{\Delta}^{(l)}$ for all layers.

4. **Calculate gradients:** use Eqs. (1.3.5) and (1.3.7) to calculate $\frac{\partial J}{\partial b_j^{(l)}}$ and $\frac{\partial J}{\partial W_{kj}^{(l)}}$.

These basic ideas underlie almost all modern automatic differentiation packages, with clever techniques used to minimise

the number of function evaluations. Once gradients have been calculated, parameters of the network can be updated via optimisation methods such as gradient descent, as discussed in section 1.5.

`https://github.com/adammoss/MLiS2/blob/master/examples/intro/backprop.ipynb`

Let us again attempt to learn the XOR function using the same network as in section 1.2.4, this time starting with random initial weights and using back-propagation with simple gradient descent to find the optimal parameters. The error functions for each neuron are

$$\Delta_1^{(2)} = \frac{\partial J}{\partial z_1^{(2)}} = \frac{\partial J}{\partial a_1^{(2)}} = \frac{\partial J}{\partial \hat{y}^{(i)}}\,, \tag{1.3.8}$$

$$\Delta_1^{(1)} = \Delta_1^{(2)} W_{11}^{(2)} \Theta(z_1^{(1)})\,, \tag{1.3.9}$$

$$\Delta_2^{(1)} = \Delta_1^{(2)} W_{21}^{(2)} \Theta(z_2^{(1)})\,, \tag{1.3.10}$$

where the first line follows since the output has a linear activation function, and the Heaviside step function Θ is the derivative of the ReLU activation function. We use the MSE loss, given in Table. 1.1.2, so that

$$\frac{\partial J}{\partial \hat{y}^{(i)}} = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)\,. \tag{1.3.11}$$

If we had applied a sigmoid activation to the final layer, instead of a linear function, it would have been more appropriate to use the BCE loss, since the output would have been normalised to $\{0, 1\}$. Writing out each component explicitly,

the gradients are

$$\begin{aligned}
\frac{\partial J}{\partial W_{11}^{(2)}} &= \Delta_1^{(2)} a_1^{(1)}\,, \\
\frac{\partial J}{\partial W_{21}^{(2)}} &= \Delta_1^{(2)} a_2^{(1)}\,, \\
\frac{\partial J}{\partial b_1^{(2)}} &= \Delta_1^{(2)}\,, \\
\frac{\partial J}{\partial W_{11}^{(1)}} &= \Delta_1^{(1)} x_1\,, \\
\frac{\partial J}{\partial W_{12}^{(1)}} &= \Delta_2^{(1)} x_1\,, \\
\frac{\partial J}{\partial W_{21}^{(1)}} &= \Delta_1^{(1)} x_2\,, \\
\frac{\partial J}{\partial W_{22}^{(1)}} &= \Delta_2^{(1)} x_2\,, \\
\frac{\partial J}{\partial c_1^{(1)}} &= \Delta_1^{(1)}\,, \\
\frac{\partial J}{\partial c_2^{(1)}} &= \Delta_2^{(1)}\,.
\end{aligned} \tag{1.3.12}$$

We then update the weights and biases using simple gradient descent, e.g.

$$W_{11}^{(2)} \to W_{11}^{(2)} - \alpha \frac{\partial J}{\partial W_{11}^{(2)}}\,. \tag{1.3.13}$$

In the example code we sum (or accumulate) gradients for the 4 training examples, and update the weights and biases only once per epoch. In practical examples with many thousands of training examples, gradients are accumulated and parameters

are updated for **mini-batches** of data. This is done for two reasons – firstly, the amount of memory (in particular GPU memory) restricts how many examples can be passed forwards and backwards through the network simultaneously. Secondly, updating parameters for batches of data rather than for the entire training set has actually been shown to improve convergence of the optimiser. This is called **stochastic gradient descent** and is discussed in section 1.5.

In our example we set the learning rate to be a small fixed constant of $\alpha = 0.1$. Starting with random initial weights and biases, drawn from a normal distribution with mean 0 and standard deviation 0.1, the history of the loss, weights and biases are shown as a function of epoch in Fig. 1.3.1. Although the loss decreases close to zero, it is noticeable that the optimal weights and biases are different to those in (1.2.17). The reason for this is that the loss, a function of 9 parameters, has degeneracy's between parameters and no unique global minima. The loss also has local minima where the optimiser can become stuck, and in our example we actually had to hand pick the seed for the random initial weights to ensure convergence to the global minima. Try using a different seed and different learning rates, and check the reliability of finding the global minima. This highlights the importance of the choice of optimiser and weight initialisation.

1.4 Deep Learning Frameworks

So far we have built and trained our neural networks from first principles. Fortunately, however, there are now powerful deep learning frameworks that do much of the heavy

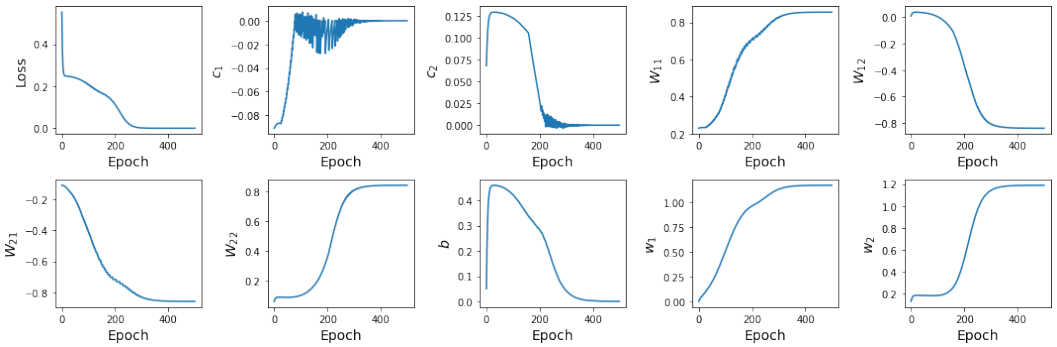


Figure 1.3.1: History of the loss, weights and biases as a function of epoch for the model in section 1.2.4 learning the XOR function.

lifting for you. They can leverage parallel processing from GPU(s) to train extremely deep networks with millions of parameters. Currently the two most popular deep learning libraries are **PyTorch** (`https://pytorch.org/`) and **TensorFlow** (`https://www.tensorflow.org/`). Each have their own advantages and disadvantages, but recently TensorFlow v2 (TF2) was released that addressed many of the shortcomings of TensorFlow v1 (TF1). In particular, to build a neural network in TF1 one needed to define a static computational graph, and to pass through data one needed to run a 'session'. This made debugging code difficult, but TF2 can be run much like any other Python code. Given it's excellent documentation, widespread use in industry and a wide ecosystem (e.g. TensorFlow Lite can be used on mobile devices), we will use TF2 as our default deep learning framework.

To illustrate its use, we will build the same model as in section 1.2.4 to learn the XOR function. TF2's high level Application Programming Interface (API) has roughly 3 levels of granularity:

1. A high level API called **keras**, which makes it particularly simply to build and train neural networks.
2. A lower level API that allows more control over building and training the network. Entire layers can still be defined, taking care of the weights and biases, and the forward pass is written imperatively. This API offers more control over the training procedure, defining a so called `GradientTape`, which is responsible for each step of the forward pass, loss function evaluation, backward pass and gradient descent step.
3. An extremely low level API that allows you to build the network from first principles, e.g. defining your own weights and biases.

`https://github.com/adammoss/MLiS2/blob/master/examples/intro/tensorflow_intro.ipynb`

In the example below we use keras to build the same **sequential** model (a neural network with a stack of layers) as in section 1.2.4. First we define a **dense** (fully connected) hidden layer with 2 units, specifying the input dimension (excluding the batch dimension, so in our case this is 2) if it is the first hidden layer following the input. We specify a ReLU activation function for this layer, then stack a dense layer on top with a single output unit. Since we did not specify an activation function, it is assumed to be linear.

```
from tensorflow import keras
from tensorflow.keras import layers
```

```
X = np.array([[0,0], [0,1], [1,0], [1,1]])
Y = np.array([[0], [1], [1], [0]])

def build_model():
  model = keras.Sequential([
    layers.Dense(2, activation='relu',
    input_shape=[X.shape[1]]),
    layers.Dense(1)
  ])

  optimizer = keras.optimizers.RMSprop(0.01)

  model.compile(loss='mse', optimizer=optimizer,
  metrics=['mae', 'mse'])

  return model

model = build_model()
model.summary()
model.fit(X, Y, epochs=300, verbose=1)
model.predict(X)
```

Once the model is defined, we can print out a summary of the network, checking that there are 9 trainable parameters. TF2 has a variety of different optimisers, discussed in section 1.5. In this example we use the **RMSprop** optimiser with a learning rate of 0.01. The model can then be trained by running for a specified number of epochs (other stopping conditions are possible), and metrics such as the MSE and MAE

can be outputted during training. The training procedure automatically calculates gradients using back-propagation, making it very easy to train complex networks. Finally, once the model is trained, we can make predictions using the optimal parameters of the model.

Note that, just as with our own back-propagation code, the optimiser can still get stuck in local minima, depending on the seed used to initialise the parameters of the network. This emphasises the need to carefully validate the result of trained models. One way, of doing this, for example, is to compare the loss between separate **training** and **test** datasets, and to perform multiple runs with different initial parameters.

1.5 Optimisation

Once gradients have been calculated, the next step is to update the weights and biases of the network, usually by some variant of gradient descent. So far, we have used a simple update rule,

$$\begin{aligned} \hat{\mathbf{g}} &= \frac{1}{N}\nabla_{\mathbf{w}}\sum_{i}^{N}\ell(\hat{\mathbf{y}}^{(i)}(\boldsymbol{\theta}),\mathbf{y}^{(i)})\,, \\ \mathbf{w} &\rightarrow \mathbf{w}-\alpha\hat{\mathbf{g}}\,, \end{aligned} \tag{1.5.1}$$

where α is the learning rate and $\hat{\mathbf{g}}$ is an estimate of the gradient from N training examples. This gives no insight on the optimal value of α, and if it is set incorrectly it can significantly impact model training and performance, as illustrated in Fig. 1.5.1. In particular, if it is too small, it will require many updates to reach a minima of the cost function, and if it is too large it can overshoot the minima or fail to converge.

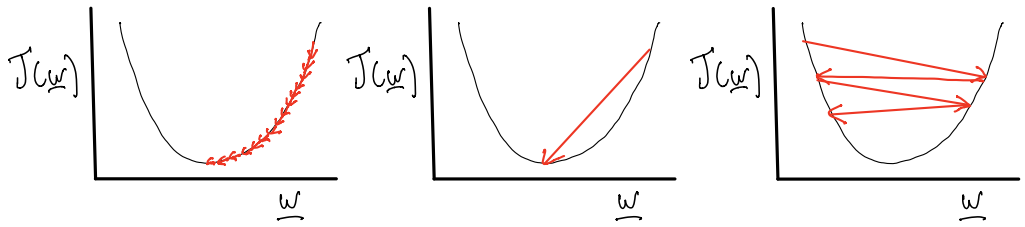


Figure 1.5.1: Illustration of a learning rate that is (left) too small; (middle) optimal; (right) too large.

The cost function for deep neural networks is almost guaranteed to have multiple local minima. One reason for this is weight-space symmetry, since there are multiple ways of arranging hidden units to give identical outputs. Furthermore, there are often degeneracy's between weights and biases, whereby one parameter can be rescaled relative to another with no change in cost. We encountered this issue in section 1.3, since if the incoming weight to a ReLU activation is scaled by β, this can be counteracted by scaling the outgoing weight by $1/\beta$. This means that every local minima lies on a degenerate hyper-surface in parameter space.

Although these issues mean that there can be a very large number of local minima, they have the same cost and therefore do not usually pose a problem. If the cost function has other local minima with higher cost, this can be more problematic as the optimiser may become trapped and fail converge to the global minima. There are several strategies to reduce the likelihood of this happening, such as the choice of optimiser, weight initialisation and regularisation. This is an active research area, and recent work has suggested that it may not be as important to find the true global minimum as opposed to

finding a low cost local minima as, for sufficiently large networks, most minima have similar cost.

In order to estimate the optimal value of α, one can Taylor expand the cost function around some point $\mathbf{w}_0$,

$$J(\mathbf{w}) = J(\mathbf{w}_0) + (\mathbf{w} - \mathbf{w}_0)^T \mathbf{g} + \frac{1}{2} (\mathbf{w} - \mathbf{w}_0)^T \mathbf{H} (\mathbf{w} - \mathbf{w}_0) \,, \tag{1.5.2}$$

where $\mathbf{g}$ is the true gradient, $\mathbf{g} = \nabla_{\mathbf{w}} J(\mathbf{w})$, and $\mathbf{H}$ is called the Hessian matrix, given by the second derivative of the cost function, $\mathbf{H} = \nabla^2_{\mathbf{w}} J(\mathbf{w})$. Starting from $\mathbf{w}_0$, a gradient update step of $\mathbf{w} = \mathbf{w}_0 - \alpha \hat{\mathbf{g}}$ gives

$$J(\mathbf{w}_0 - \alpha \hat{\mathbf{g}}) = J(\mathbf{w}_0) - \alpha \hat{\mathbf{g}}^T \mathbf{g} + \frac{1}{2} \alpha^2 \hat{\mathbf{g}}^T \mathbf{H} \hat{\mathbf{g}} \,. \tag{1.5.3}$$

We would like the loss to decrease as fast as possible for a gradient step, but the second term actually increases it. The optimal step size can be shown to be

$$\alpha^\star = \frac{\hat{\mathbf{g}}^T \mathbf{g}}{\hat{\mathbf{g}}^T \mathbf{H} \hat{\mathbf{g}}} \,, \tag{1.5.4}$$

i.e. depending on $\mathbf{H}^{-1}$. Inverting the Hessian is not feasible for large numbers of parameters, so some optimisers use methods to try to estimate it.

For large networks, a more severe problem than local minima are **saddle points**, points where the gradient is zero and the Hessian has both positive and negative eigenvalues. In standard gradient descent it can take a long time to escape from saddle points, as updates will be small due to the zero (or very small) gradient, and the cost will increase in some directions and decrease in others, resulting in a very small change in cost.

There may also be regions where the cost is highly sensitive to some directions in parameter space (i.e. large gradients)

and insensitive to others (i.e. small or flat gradients). In flat directions one would like to take large step sizes, but in steep directions one would like to take small step sizes.

1.5.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is one of the most used popular optimisation methods in machine learning. Rather than taking an estimate of the gradient over all training examples, a mini-batch of B examples is used, such that

$$\hat{\mathbf{g}} = \frac{1}{B} \nabla_{\mathbf{w}} \sum_{i}^{B} \ell(\hat{\mathbf{y}}^{(i)}(\boldsymbol{\theta}), \mathbf{y}^{(i)}) \,. \tag{1.5.5}$$

The mini-batch is usually drawn randomly from the training set, and this is repeated for subsequent gradient steps until the entire training set is used. Mini-batch sizes are constrained by the amount of memory required to process the entire batch at once, which is normally the case with neural networks. Although larger batch sizes provide a more accurate estimate of the gradient, it has been shown that smaller batch sizes can offer a regularising effect and generalise better, since they add some stochastic noise to the training process and reduce overfitting. A smaller mini-batch size also decreases the chance of becoming stuck in local minima.

1.5.2 Stochastic Gradient Descent with Momentum

The addition of momentum to SGD can accelerate learning, particularly for curving degeneracy's in the cost function with small gradients, and regions with noisy gradients. The update retains some 'memory' of the direction being moved in, such

that

$$\begin{aligned} \mathbf{v} &\rightarrow \gamma\mathbf{v} - \alpha\hat{\mathbf{g}}\,, \\ \mathbf{w} &\rightarrow \mathbf{w} + \mathbf{v}\,, \end{aligned} \tag{1.5.6}$$

where $\mathbf{v}$ can be interpreted as the momentum and γ is an inertia parameter. The default value in TF2 are $\alpha = 0.01$ and $\gamma = 0.0$ (i.e. no momentum).

1.5.3 RMSprop

One issue with the algorithms discussed so far is that the learning rate is the same for each parameter, whereas in practice the cost function is often more sensitive to particular directions. The RMSprop algorithm accumulates a weighted average of squared gradients for each parameter, and rescales the learning rate according to the inverse square of this. The net effect is a smaller step size in directions with large gradients, and a larger step size in directions with small gradients. The update rule is given by

$$\begin{aligned} \mathbf{r}_t &= \rho\mathbf{r}_{t-1} + (1-\rho)\hat{\mathbf{g}}_t \odot \hat{\mathbf{g}}_t\,, \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \alpha\frac{1}{\sqrt{\mathbf{r}_t + \epsilon}} \odot \hat{\mathbf{g}}_t\,, \end{aligned} \tag{1.5.7}$$

where ρ is the decay rate and ϵ is a small constant to stabilise division by very small numbers. We have also explicitly labelled the iteration step t. The default values in TF2 are $\alpha = 0.001$, $\rho = 0.9$ and $\epsilon = 10^{-7}$.

1.5.4 Adam

The Adam algorithm is another adaptive method, which improves on RMSprop by accumulating first and second order

moments of the gradient. The first moment is an estimate of the historical momentum, and the second moment is used to rescale the learning rate. The update rule is

$$\begin{aligned} \mathbf{s}_t &= \beta_1 \mathbf{s}_{t-1} + (1-\beta_1)\hat{\mathbf{g}}_t\,, && (1.5.8)\\ \mathbf{r}_t &= \beta_2 \mathbf{r}_{t-1} + (1-\beta_2)\hat{\mathbf{g}}_t \odot \hat{\mathbf{g}}_t\,, \\ \hat{\mathbf{s}}_t &= \frac{\mathbf{s}_t}{1-\beta_1^t}\,, && (1.5.9)\\ \hat{\mathbf{r}}_t &= \frac{\mathbf{r}_t}{1-\beta_2^t}\,, && (1.5.10)\\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \alpha \frac{1}{\sqrt{\hat{\mathbf{r}}_t} + \epsilon} \odot \hat{\mathbf{s}}_t\,, && (1.5.11) \end{aligned}$$

where β_1 and β_2 are decay rates and ϵ is a small constant to stabilise division by very small numbers. The default values in TF2 are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$.

1.5.5 Comparison

`https://github.com/adammoss/MLiS2/blob/master/examples/intro/optimiser.ipynb`

The Rosenbrock function is a non-convex function commonly used for bench-marking optimisation methods. It is given by

$$f(x,y) = (1-x)^2 + 100\left(y - x^2\right)^2\,, \qquad (1.5.12)$$

and has a global minima at $(x,y) = (1,1)$. It is especially challenging as the global minimum is inside a long, narrow and parabolic shaped flat valley. We benchmark several of the optimisers included with TensorFlow, including SGD, SGD + momentum, RMSprop and Adam, starting from $(x,y) = (0,1)$.

Learning rates were chosen to be (approximately) the largest values possible that converged to the true solution, and other hyper-parameters were set to their default values. Batch size is not important here, as know the true gradient rather than estimating it from training examples.

In Fig. 1.5.2 we show the trajectories of each update, along with the distance from the global minimum as a function of iteration. SGD takes the longest to reach the minimum, but the addition of momentum helps move along the flat region more quickly. The adaptive methods work best, and it is noticeable how initially, rather than moving almost vertically, they track towards the minima, as the result of the larger learning rate in the x direction. When training your neural networks, you are encouraged to try different optimisers and hyper-parameters.

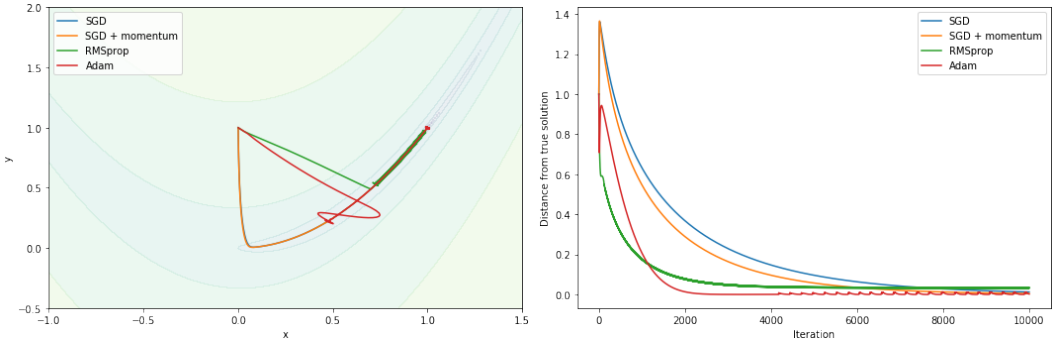


Figure 1.5.2: Benchmark of the SGD, SGD + momentum, RMSprop and Adam optimisers on the Rosenbrock function, starting from $(x, y) = (1, 0)$.

1.6 Initialisation

During back-propagation, it is important that gradients don't become either too small or too large as the algorithm pro-

gresses down to the lower layers. If the gradients become small, training will proceed at a much slower rate and may not converge. This is called the **vanishing gradients problem**. If the gradients become too large, the algorithm may diverge. This is called the **exploding gradients** problem, and is mostly encountered in recurrent neural networks, as discussed in section **??**.

```
https://github.com/adammoss/MLiS2/blob/master/
examples/intro/initialisation.ipynb
```

Weight initialisation To see why this can be a problem, consider a deep MLP with 100 hidden layers, each with 100 hidden units (setting the bias of each to zero for simplicity) and a sigmoid activation function. If we assume the input data (also with dimension 100) and the weights are normally distributed with zero mean and unit variance, i.e. $\mathcal{N}(0,1)$, the distribution of activations and their gradient after every 20th layer is shown in the top panel of Fig. 1.6.1. The activations immediately saturate and the gradients become small.

Ideally we would to preserve the signal variance throughout the network, such that it neither dies out nor saturates. The variance of the output of a neuron should therefore be equal to the variance of its inputs. To understand how this can be achieved, consider the input to the ith neuron in layer ℓ

$$z_i^{(\ell)} = \sum_{j=1}^{n_{\text{inputs}}} a_j^{(\ell-1)} W_{ji}^{(\ell)} \,. \tag{1.6.1}$$

The variance of this is

$$\mathrm{Var}[z_i^{(\ell)}] = \sum_{j=1}^{n_{\text{inputs}}} \mathrm{Var}[a_j^{(\ell-1)}] \, \mathrm{Var}[W_{ji}^{(\ell)}] \,, \tag{1.6.2}$$

which is equal to $n_{\text{inputs}}\mathrm{Var}[a_j^{(\ell-1)}] \, \mathrm{Var}[W_{ji}^{(\ell)}]$ if the variance of each input is the same. In order for the variance out to be equal to the variance in, the variance of the weights should therefore be rescaled by $1/n_{\text{inputs}}$. During back-propagation, we would also like to preserve the gradient variance, which can be achieved by rescaling the variance of the weights by $1/n_{\text{outputs}}$, where n_{outputs} is the number of outputs from a neuron. A simple heuristic that balances both requirements is a weight initialisation of either

$$W_{ij} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}, \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}\right) \,, \tag{1.6.3}$$

if the weights are initially drawn from a uniform distribution, and

$$W_{ij} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}\right) \,, \tag{1.6.4}$$

if the initial weights are drawn from a normal distribution. Both are commonly used and give similar results. This initialisation strategy is called **Xavier initialisation** or **Glorot initialisation**. Modern initialisation strategies are relatively simple and heuristic, and most are based on achieving desirable properties such as preserving variance. The only property we know with certainty is that the weights should not be the same, in order to break the weight-space symmetry between hidden units.

The results of using Xavier initialisation for our example are shown in the lower panel of Fig. 1.6.1. It can be seen that activations and their gradients retain similar variance throughout the the network. In TF2, the Xavier uniform distribution is used by default to initialise dense layers.

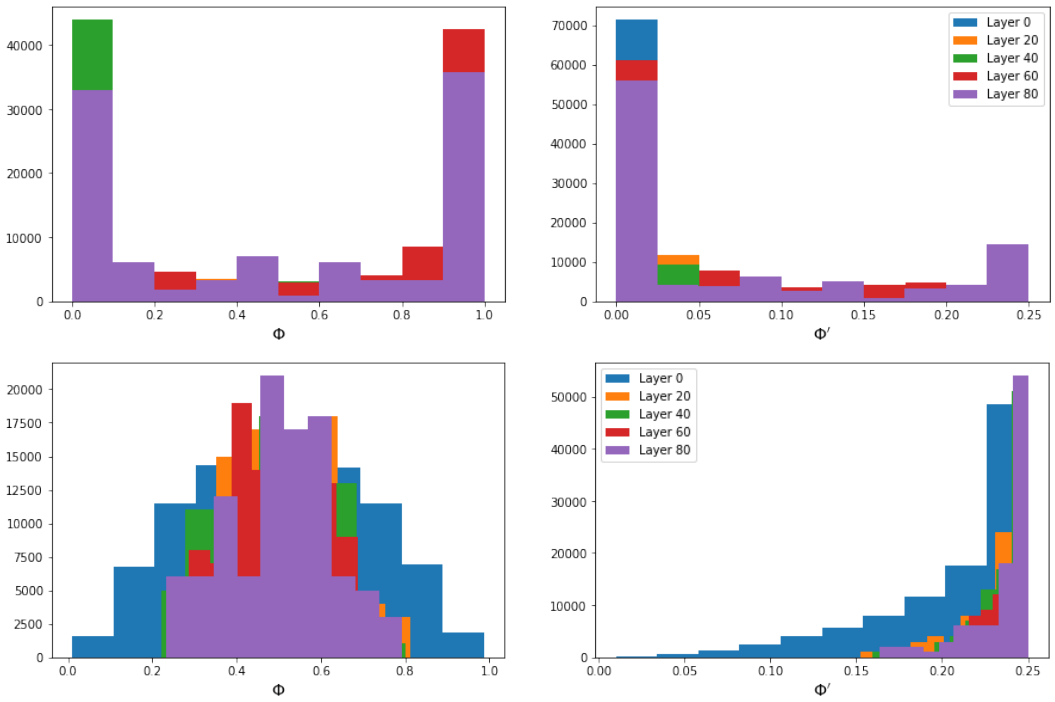


Figure 1.6.1: Histogram of activations $a = \Phi$ (left) and it's gradient Φ' (right) after every 20th layer for (top) zero mean and unit variance weight initialisation (bottom) Xavier initialisation.

Data initialisation During many machine learning tasks, it is good practice to apply **feature-scaling** to your initial data. In neural networks, extremely large or small inputs can cause the activations to either saturate or the signal to die out. This can be alleviated by using non-saturating activation functions

(e.g. leaky ReLU's), **batch normalisation** and **gradient clipping**, but feature-scaling is still an important step of initial data processing.

There are two common ways to normalise features: **min-max** and **standardisation**. Min-max normalisation rescales features to the range $[0, 1]$ by the transformation,

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}, \tag{1.6.5}$$

while standardisation rescales by the mean $\bar{x}$ and standard deviation σ,

$$x' = \frac{x - \bar{x}}{\sigma}. \tag{1.6.6}$$

Standardisation is much less affected by outliers, but does not bound inputs to a specific range, which may be a problem for some algorithms.

1.7 Regularisation

In the last few sections we have emphasised the importance of minimising the cost function on training data. However, the real goal of many machine learning tasks is not just to perform well on training data, but also to **generalise** well to unseen test or validation data. **Regularisation** can be defined as any strategy designed to reduce generalisation error, but not impact on the training error. There are many different types of regularisation, some of which we outline here. In deep learning an effective regulariser successfully balances the **bias-variance trade off**, that is it decreases the model variance responsible for **over-fitting**, while not overly increasing the bias.

1.7.1 Early Stopping

Early stopping is perhaps the simplest form of regularisation, and has been described by Geoffey Hinton as a “beautiful free lunch". When training large datasets, one should first separate it into training, test and holdout sets:

Training Training data is used to fit the parameters of the network.

Test Test data is used to to evaluate the performance of the trained model on unseen data. It can also be used to tune the various hyper-parameters of the network (e.g. number of layers, hidden units etc.).

Holdout Holdout data is used to assess the final performance of the tuned model.

During training, we often observe that the training loss continues to decrease, but the test loss eventually begins to rise. Early stopping terminates the training process after the test loss has not improved over the previously recorded best test loss for a specified number of epochs. Each time we obtain an improved test loss, the model parameters are stored, and when training terminates these are used as the best model. Early stopping has been shown to restrict the training procedure to a smaller volume of parameter space, which can reduce the chance of over-fitting.

1.7.2 L_1/L_2 regularisation

L_1 and L_2 regularisation are types of **parameter-norm penalties**, which can be implemented by adding a term to the cost

function

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \lambda\Omega(\mathbf{w})\,, \tag{1.7.1}$$

where λ is a hyper-parameter that determines the contribution of the penalty term $\Omega(\mathbf{w})$. In deep learning, the norm penalty is usually only a function of the weights at each layer, since biases require less data to fit accurately and do not contribute as much to the model variance. In TF2, norm penalties can be included on a per-layer basis by including a `kernel_regularizer` argument to the keras API.

The L_2 norm of a vector $\mathbf{x}$ is given by $||\mathbf{x}||_2 = \sqrt{\sum_i |x_i|^2}$, and adds a penalty term (also known as ridge-regression) of the form

$$\Omega(\mathbf{w}) = ||\mathbf{w}||_2^2\,. \tag{1.7.2}$$

For the specific case of the linear model in section 1.2.1, the solution of the normal equations is modified to

$$\mathbf{w}^\star = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{Y}\,, \tag{1.7.3}$$

where $\mathbf{I}$ is the identity matrix. The matrix $\mathbf{X}^T\mathbf{X}$ is proportional to the covariance matrix, so regularisation has the affect of increasing the variance of the input, which makes it down-weight the dependence on features which have small covariance with the targets.

In general, it can be shown that, near a local minima of the cost function, the affect of the regularisation is to rescale the best fit parameters $\mathbf{w}^\star$ along the axes defined by the eigenvectors of the Hessian $\mathbf{H}$. For directions with large eigenvalues that significantly affect the cost, the components of $\mathbf{w}^\star$ will be relatively unchanged. For directions with small eigenvalues that do not significantly affect the cost, components of $\mathbf{w}^\star$ will be decayed away.

The L_1 norm of a vector $\mathbf{x}$ is given by $||\mathbf{x}||_1 = \sum_i |x_i|$, and adds a penalty term (also known as LASSO regression) of the form

$$\Omega(\mathbf{w}) = ||\mathbf{w}||_1 \,. \tag{1.7.4}$$

In general, L_1 regularisation tends to give **sparse** solutions, meaning more components of $\mathbf{w}^\star$ will be zero. An intuitive justification for this can be obtained by plotting iso-contours of the unregularised cost function and the penalty term. In the case of L_1 regularisation, these will tend to intersect where components of $\mathbf{w}$ are zero.

1.7.3 Dropout

Another powerful regularization strategy that is computational simple to implement is **dropout**. During training, every neuron (including the input neurons but excluding the output neurons) has a probability p of having its input and output weights set to zero. This reduces the chance of introducing spurious correlations that do not significantly contribute to improving the loss. During testing no neurons are dropped out, so weights need to be rescaled by $\mathbf{w}_{\text{test}} = p\mathbf{w}_{\text{train}}$, otherwise each neuron will receive an average input signal larger than what it was trained on. Dropout can also be thought of as a type of **ensembling**, as each training step uses a slightly different (but obviously not completely independent) network, and these are combined during test time to produce a consensus prediction. The end result is a more robust network that generalises better.

1.7.4 Batch Normalisation

Batch normalisation is a method originally developed to reduce the vanishing/exploding gradients problem, but it has also been shown to have a regularising effect. It consists of adding an additional operation just before the activation function, that zero centers and normalises the inputs for each batch, then scales and shifts the results using two new learnable parameters. It can be added to individual layers in the network, each with separate learnable parameters, and is implemented in the TF2 keras API as a `BatchNormalization` layer.

1.7.5 Data Augmentation

The best way to reduce over-fitting is to use more data. Unfortunately, however, this may not be possible, but one can generate new training examples from existing ones by **augmenting** them. When using image data, for example, one could slightly shift, rotate, resize, crop, adjust the contrast, add small random noise etc, and this would force the model to be more tolerant to these properties and learn better representations of the data. It is preferable to generate these new examples during training, rather than saving them to disk, and TF2 offers several data augmentation methods. One should be careful not to apply transformations that would change the correct class, for example rotating by 180° would not be appropriate for tasks classifying the digits '6' and '9'.

2 Convolutional Neural Networks

As the name suggests, convolutional neural networks are conventional neural networks where the matrix multiplication has been replaced by convolution. In this chapter we introduce convolutional networks.

2.1 History

Before deep learning, image classification relied on feature extraction and data compression to make predictions. Often they would look for complex non-linear relationships between the features. Deep supervised neural networks, performed well but were too difficult to train, there are a huge number of parameters, and the sequential non-linear units creates a non-convex function for optimisation, which is difficult to converge to the global minima. In 1988 neural networks took a turn from applications on 1D arrays, to direct input of image pixels [LeCun et al., 1989b]. On dataset similar to what we now know as `MNIST`, a handwritten 10-digit recognition problem, it was shown that multi-layer networks applied in a hierarchical manner, which essentially reduces the number of free parameters and provided significant improvement in the generalisation performance of a network over single-layer networks. Over several years, this concept was refined [LeCun et al., 1989a], eventually converging to `LeNet-5` [LeCun et al., 1998], a con-

volutional neural network trained with gradient based back propagation (gradient descent). `LeNet-5` was a fundamental paradigm shift in machine learning, and deep learning.

2.2 Architecture

The architecture of a network is the sequence of layers that it contains. Each layer has different purposes, and not all layers have trainable parameters, for example convolutional and fully connected layers do, whereas pooling and activation layers don't.

2.2.1 Convolution

Convolution ($*$) is a mathematical operation on 2 signals f and g, that produces a new smoothed signal s at time t,

$$s(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.2.1}$$

$s(t)$ is typically referred to as the feature map, f is the input and g is the kernel filter that acts as an average weighting applied to f. Of course convolution does not have to be time specific, and in general we tend to have signals at discrete intervals rather than continuous ones, in which case what we have is discrete convolution,

$$s(t) = \sum f(\tau)g(t-\tau). \tag{2.2.2}$$

The convolutional layer makes up the principle layer type in CNNs. Whilst time series data is most commonly a 1D problem, image data is a 2D problem. In a 2D discrete convolution layer, the input ($\boldsymbol{I}$) is a feature map of size $I_x \times I_y \times I_z$. For

example an input of 50×50×3 may correspond to a 3 channel (red, green and blue) 50×50 pixel RGB image. The input feature map can be represented by a 3D matrix, where the pixel values of the image are represented as values within the matrix. A kernel ($\boldsymbol{K}$) of size $K_x \times K_y \times I_z \times N$ is applied, where N is the number of filters, and produces an output feature map ($\boldsymbol{O}$) of size $O_x \times O_y \times O_z$, which will later be defined in Equation 2.2.8.

If you recall that the output of a fully connected neural network node is,

$$\hat{\boldsymbol{y}} = \Phi\left(\sum_i w_i x_i + b\right), \tag{2.2.3}$$

where x is the input, w are the multiplicative weights, b is the additive bias, and Φ is the activation function. Mathematically, the 2D discrete convolutional layer is given as,

$$\hat{\boldsymbol{Y}} = \Phi(\sum \boldsymbol{I} * \boldsymbol{K} + \boldsymbol{b}) \tag{2.2.4}$$

Here $\boldsymbol{b}$ is the bias, which is a vector of length O_z, i.e. a scalar value additive bias applied to every pixel for each output channel. For stride one and no padding (section 2.2.2, section 2.2.2), the convolutional part is,

$$\boldsymbol{O}_{i,j,k} = (\boldsymbol{I} * \boldsymbol{K})(i,j,k) = \sum_{l,m,n} \boldsymbol{K}_{l,m,k,n} \cdot \boldsymbol{I}_{i+l-1,j+m-1,n} \tag{2.2.5}$$

More intuitively, let's take an example arbitrary $4 \times 4(\times 1)$ image written as a matrix,

$$I = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 4 & 4 \\ 6 & 5 & 7 & 1 \\ 1 & 2 & 2 & 8 \end{bmatrix}.$$

Let's say we have a $3 \times 3(\times 1 \times 1)$ filter kernel, in matrix form,

$$\boldsymbol{K} = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 2 \\ 2 & 1 & 1 \end{bmatrix}.$$

The convolution of the filter is applied at each position of the image. Again, for stride one and no padding, each kernel element is **multiplied** by the overlapping image values and **summed**, resulting in an output size of $2 \times 2 \times 1$,

$$\begin{bmatrix} \mathbf{0}_{\times 1} & \mathbf{1}_{\times 2} & \mathbf{2}_{\times 3} & 3 \\ \mathbf{4}_{\times 3} & \mathbf{5}_{\times 4} & \mathbf{4}_{\times 2} & 4 \\ \mathbf{6}_{\times 2} & \mathbf{5}_{\times 1} & \mathbf{7}_{\times 1} & 1 \\ 1 & 2 & 2 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 0+2+6+12+20+8+12+5+7=72 & \end{bmatrix}$$

$$\begin{bmatrix} 0 & \mathbf{1}_{\times 1} & \mathbf{2}_{\times 2} & \mathbf{3}_{\times 3} \\ 4 & \mathbf{5}_{\times 3} & \mathbf{4}_{\times 4} & \mathbf{4}_{\times 2} \\ 6 & \mathbf{5}_{\times 2} & \mathbf{7}_{\times 1} & \mathbf{1}_{\times 1} \\ 1 & 2 & 2 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 72 & 1+4+9+15+16+8+10+7+1=71 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ \mathbf{4}_{\times 1} & \mathbf{5}_{\times 2} & \mathbf{4}_{\times 3} & 4 \\ \mathbf{6}_{\times 3} & \mathbf{5}_{\times 4} & \mathbf{7}_{\times 2} & 1 \\ \mathbf{1}_{\times 2} & \mathbf{2}_{\times 1} & \mathbf{2}_{\times 1} & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 72 & 71 \\ 4+10+12+18+20+14+2+2+2=84 & \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & \mathbf{5}_{\times 1} & \mathbf{4}_{\times 2} & \mathbf{4}_{\times 3} \\ 6 & \mathbf{5}_{\times 3} & \mathbf{7}_{\times 4} & \mathbf{1}_{\times 2} \\ 1 & \mathbf{2}_{\times 2} & \mathbf{2}_{\times 1} & \mathbf{8}_{\times 1} \end{bmatrix} \rightarrow \begin{bmatrix} 72 & 71 \\ 84 & 84 \end{bmatrix}.$$

More practically, the image matrix can be reshaped into a

column vector of size 16×1

$$I^* = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 4 \\ 4 \\ 6 \\ 5 \\ 7 \\ 1 \\ 1 \\ 2 \\ 2 \\ 8 \end{bmatrix},$$

and the filter can be unrolled into a convolutional matrix of size $(I_x - K_x + 1)(I_y - K_y + 1) \times (I_x I_y)$, in our case a 4×16 sparse matrix,

$$\boldsymbol{K}^* = \begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{3} & 0 & \mathbf{3} & \mathbf{4} & \mathbf{2} & 0 & \mathbf{2} & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & \mathbf{2} & \mathbf{3} & 0 & \mathbf{3} & \mathbf{4} & \mathbf{2} & 0 & \mathbf{2} & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & \mathbf{3} & 0 & \mathbf{3} & \mathbf{4} & \mathbf{2} & 0 & \mathbf{2} & \mathbf{1} & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{2} & \mathbf{3} & 0 & \mathbf{3} & \mathbf{4} & \mathbf{2} & 0 & \mathbf{2} & \mathbf{1} & \mathbf{1} \end{bmatrix},$$

where the 16 columns correspond to the 16 element in I, and each row corresponds to the vectorized filter at each of the 4 steps of the convolution, i.e. in the first convolutional step, K_{11} is applied at $I_{11} = I_1^*$, so the vectorized form of the filter begins at $K*_{11}$, whereas in the third convolutional step, K_{11} is

applied at $I_{21} = I_5^*$, so the vectorized filter begins at begins at $K*_{35}$. This way, the output of the convolutional layer is simply the matrix multiplication of the two, reshaped to the output shape,

$$\boldsymbol{K}^* \cdot I^* = \begin{bmatrix} 72 \\ 71 \\ 84 \\ 84 \end{bmatrix} \rightarrow \begin{bmatrix} 72 & 71 \\ 84 & 84 \end{bmatrix}. \tag{2.2.6}$$

The values in the filter kernel are unknown parameters and are determined during the training of the network.

The total number of parameters in a convolutional layer is,

$$n_{param} = (K_x \times K_y \times O_z + 1) \times I_z, \tag{2.2.7}$$

where the addition of 1 is due to the biases. Additionally, Convolutional layers have 4 hyperparameters which control the output volume of each layer that must be defined. These are the kernel size, number of filters, padding and stride.

2.2.2 Kernel size

The kernel size is also sometimes called the **receptive field**.

Number of filters and channels

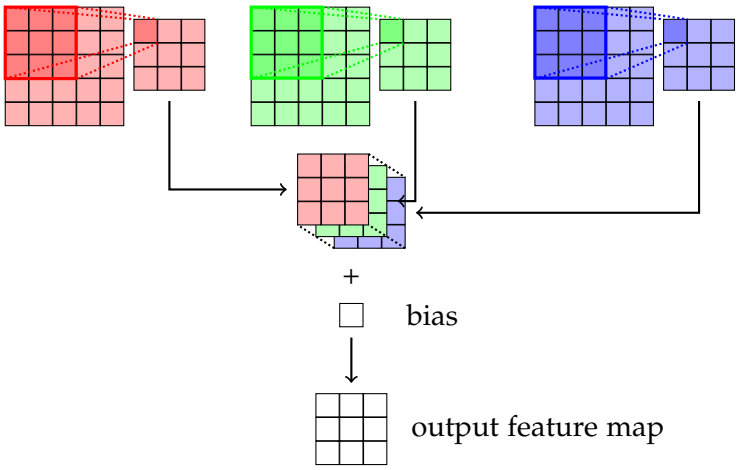


Figure 2.2.1: 2D convolution for multi-channel inputs. Here we show a 3 channel 5×5 input (RGB) image that is convolved with the kernel of the corresponding channel. The resulting feature maps are summed to give a single channel map and a bias is added to produce the single channel output feature map.

In the previous example the input was a 1 channel image, and the number of filters was 1 with depth 1. In practice, most images have 3 channels (red, green, blue) and the number of channels increases with the depth of the network. In the case of Inception modules (see section 2.5) where parallel feature maps are concatenated together, the number of channels can increase very quickly. For multi-channel inputs, a filter with the same number of channels is applied and the resulting feature maps are summed together before a before the bias is applied to produce the output feature map (Figure 2.2.1).

The number of filters determines the number of channels of the output (feature map), and the depth of the filter must always be the same as the number of input channels. Each

filter is looking for different features within the data. Each of the feature maps have their own distinct bias applied, and these are then concatenated together to output a feature map with the same number of channels as the number of filters.

Padding

So far, due to the way the convolution is applied, the size of the input image will very quickly shrink down in size after a series of convolutions and this equivalent to loss of information. Additionally, the pixels at the edges of each input feature are convolved with the kernel less frequently than the inner pixels. A solution to this is to use padding. Padding refers to how much margin is added around the input. Typically this is filled with zeroes, and therefore is also sometimes known as zero-padding.

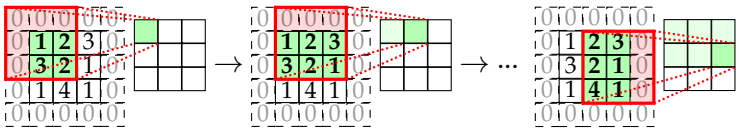


Figure 2.2.2: Convolution of a 3×3 input with a 3×3 kernel and half (same) padding to output a 3×3 output feature map. The red square shows where the convolution is applied to the cells, and the dark green shows where the operation maps to.

Half padding which is sometimes referred to as **same padding**, is when the padding value is $P = (K-1)/2$. This results in an output image with the same dimensions as the input image (Figure 2.2.2).

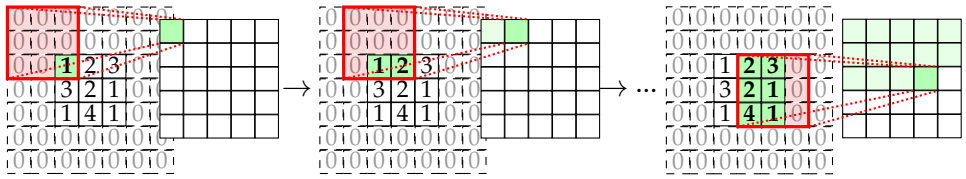


Figure 2.2.3: Convolution of a 3×3 input with a 3×3 kernel and full padding to output a 3×3 output feature map. The red square shows where the convolution is applied to the cells, and the dark green shows where the operation maps to.

Full padding occurs when the padding value is $P = K - 1$. This ensures that every possible convolution of the kernel with the input is taken into account equally (Figure 2.2.3).

Alternatively any other value of padding can be used and it does not have to be symmetric in any of the 4 axes (bottom, left, top, right). At some point you may come across the term *valid padding* which refers to no padding.

Stride

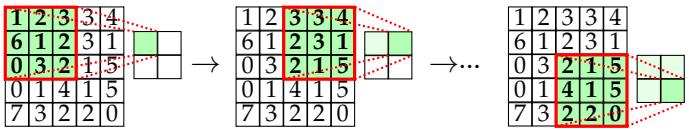


Figure 2.2.4: Convolution of a 5×5 input with a 3×3 kernel with stride 2 to produce a 2×2 output feature map.

The stride determines how many pixels each filter moves across before it is applied to the image.

The output size of a fully connected layer used in traditional neural networks is independent of the input size. This however is not the case for convolutional layers, where the output size is not only dependent on the input size, but also the stride,

padding and kernel size. The output of the convolutional layer is a matrix of size $W_2 \times H_2 \times D_2$ pixels, where,

$$\begin{aligned} O_x &= (I_x - K + 2P)/S + 1 \\ O_y &= (I_y - K + 2P)/S + 1 \\ O_z &= N, \end{aligned} \tag{2.2.8}$$

Where N is the number of filters, P is the padding and S is the stride, and K is the kernel size. These equations similarly apply for the pooling layers, where by default $P = 0$.

Fun fact: for S>1, if $I - K + 2P$ is a multiple of S, then the output size will always be the same regardless of the input size. The number of parameters in the network is therefore independent of the input image size, and only dependent on the size and number of filters.

Activation

After convolving the image with the filter(s), as we said before in Equation 2.2.4, a bias value is added and an activation function is applied elementwise for the aggregation of the information. The most common activation function used in CNNs is the rectified linear unit (ReLU). It is common because the derivative is simple and therefore fast to compute.

Activation functions add non-linearity to the network and is typically applied after each convolutional layer since convolutional layers are linear operations.

2.2.3 Pooling

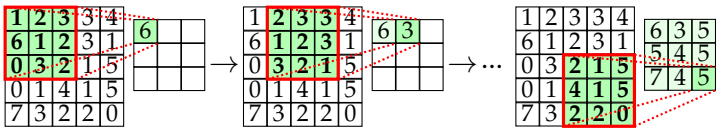


Figure 2.2.5: A max pool operation where a 5×5 input is convolved with a 3×3 kernel that returns the maximum value in regions of size 3×3 in the image.

The purpose of the pooling layer is down sampling, i.e. to reduce the dimensionality of the input. However additionally it helps with the invariance of the network to small shifts and transformations in the data. This layer has no trainable parameters but the kernel size, stride and padding must be defined. There are many types of pooling layers for example the max pool layer has a max pool kernel, which summarises areas on the input with the same size as the kernel to the maximum of those values (Figure 2.2.5). The average pool layer similarly returns the mean value in areas with the size as the kernel.

2.2.4 Fully connected

The convolutional and pooling layers can be thought of as feature extractors. They enhance the relevant features or absence of features. The fully connected (or dense) layers aggregate together this information across the final feature maps. In this layer every input element has its own weight to map it to the output layer. They are typically the last layer that provides the results of the **final classification**. The fully connected layer takes 2 hyperparameters - the number of nodes, which is typically the same as the number of classes (C), and the activation

function. If the layer is used as a classification then usually this will be a softmax activation,

$$S(x_i) = \frac{\exp(x_i)}{\sum_c^C \exp(x_c)} \tag{2.2.9}$$

to ensure all the sum of scores over all classes is one because softmax is a discrete probability distribution. The derivative of the softmax is then,

$$\frac{\delta S(x_i)}{\delta x_j} = \begin{cases} S(x_i)(1 - S(x_j)), & i = j \\ -S(x_i)S(x_j), & i \neq j \end{cases} \tag{2.2.10}$$

Note that softmax can be numerically unstable for large values of x, since the exponential of a large number can be astronomically large! Therefore it's important to rescale the data first, often the largest value of $\boldsymbol{x}$ is subtracted from each x such that the largest value of the data becomes 1.

2.3 Other layers

2.3.1 Batch normalisation

In convolutional neural networks, after each layer the values of the elements in the feature maps can grow very quickly. This can cause problems for the convergence of weights because the parameter search can be very large and thus slow. CNNs are therefore restricted to low learning rate values and can be sensitive to the initialisation of the weights. Batch normalisation [Ioffe and Szegedy, 2015] was introduced to improve the efficiency and performance of CNNs,

$$\hat{x_i} = \frac{x_i - \mu_x}{\sqrt{\sigma_x^2 + \epsilon}}, \tag{2.3.1}$$

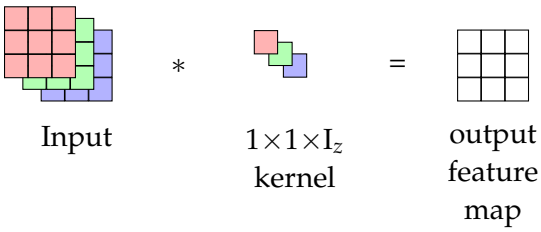


Figure 2.3.1: Pointwise convolution of a 3×3×3 input with a 1×1×3 kernel. The number of kernels determines the number of output channels.

where μ_x is the mean of x and σ_x^2 is its variance. ϵ is an arbitrary small constant included for numerical stability. The layer renormalises the input so that it has mean 0 and variance 1 when ϵ=0. This however can change the representation of the layer, so an additional scale (γ) and shift (β) parameters are introduced to restore the representation of the layer and these parameters are learnt during training,

$$y = \gamma \hat{x}_i + \beta. \tag{2.3.2}$$

In the cases of stochastic and batch gradient descent, batch norm can be applied on mini-batches. They are applied after the activation function. Aside from improving efficiency, batch norm also introduces noise into the network, and thus helps regularise the network. Batch norm should be used instead of dropout (subsection 2.4.2).

2.3.2 Pointwise convolution

Since the number of channels quickly increase with the depth of the network, this also means that the number of parameters is increasing and consequently the computational efficiency. It's usually a good idea to reduce the number of parameters in the layers however pooling only reduces the dimensionality

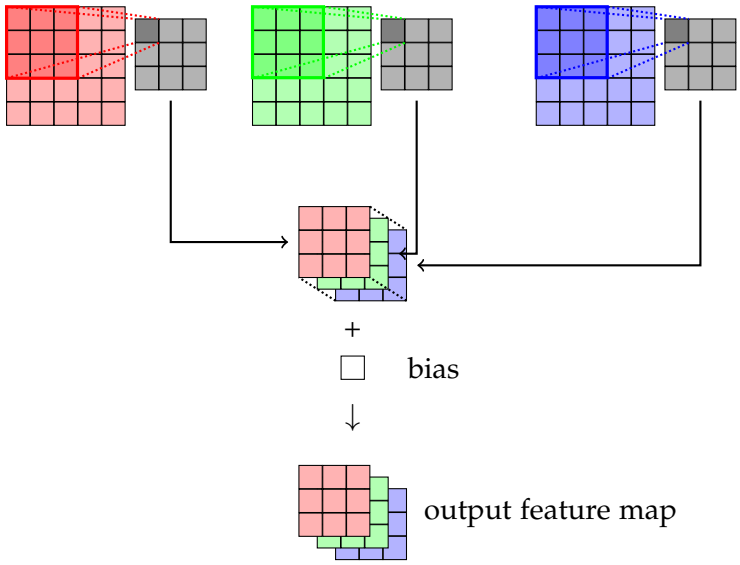


Figure 2.3.2: Depthwise convolution. A 5×5×3 input is convolved with a single 3×3 kernel.

in the width and height. Pointwise convolutions (Figure 2.3.1) are $1 \times 1 \times I_z$ kernel filters. Where I_z are the number of input channels. The number of pointwise filters applied will give the same number of output channels in this layer. Pointwise convolutions can be used to reduce or increase the depth of the input, and thus reduce or increase it's dimensionality. Alternatively pointwise convolutions can be used to increase the non-linearity of the layers. Pointwise convolution was first described in the Network-in-Network paper [Lin et al., 2013] as a cross channel pooling filter.

2.3.3 Depthwise convolution

In the depthwise convolution, each channel of the multi-channel input is convolved with the respective channel of the filter as per the standard multi-channel convolution, however in depth-

wise convolution the output features are not summed together but instead concatenated (Figure 2.3.2). Also this operation specifically uses only 1 filter. Later (section 2.5) we will talk about the significance of this operation for for dimensionality reduction.

2.3.4 Dilated convolution

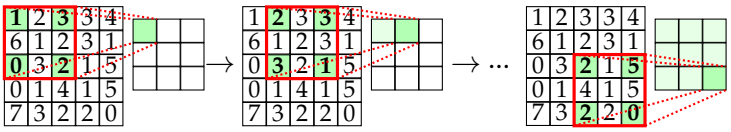


Figure 2.3.3: Convolution of a 5×5 input with a 2×2 diluted convolutional kernel with stride 1 to produce a 3×3 output feature map. Only the green shaded pixel values in the input are convolved with the kernel to produce the green shaded output pixel value in the output feature map.

Dilated convolution, is a convolution layer, where the kernel is applied with padding between kernel values. This layer is powerful when you want to integrate information from different spatial scales, balancing between high accuracies on pixel-level whilst also inferring information from the global context. For example, super resolution and denoising problems.

2.3.5 Locally connected layer

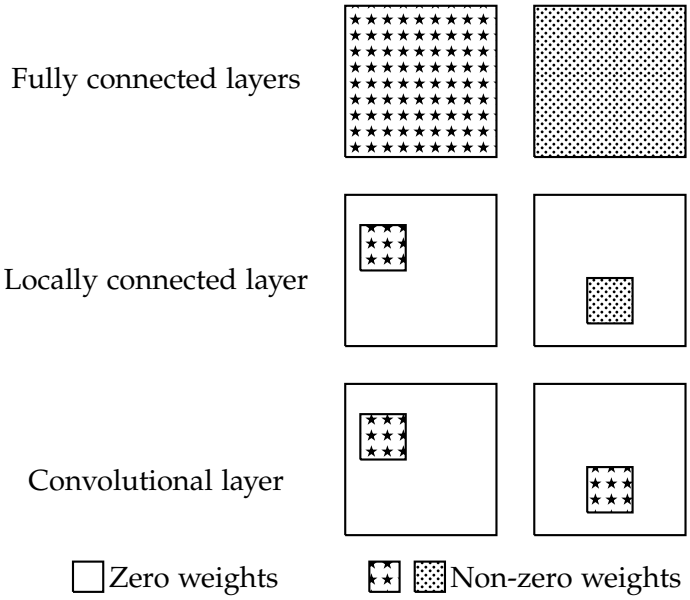


Figure 2.3.4: A visual representation comparison of fully connected layers, locally connected layers and convolutional layers. In fully connected layers all elements of the input have a weight and there is no weight sharing between elements. In convolutional layers, weights are shared across local regions of the size of the filter, and the filter is applied across the entire input. In locally connected layers the filter is only applied over a small region of the input and there is no weight sharing.

The locally connected layer can be seen as a cross between a standard convolutional layer a fully connected layer. In the fully connected layers each input pixel has its own weights and biases connecting it to each output pixel and therefore the number of parameters is very high. For a 5×5 input, the number of parameters would be $5\times5+1 = 26$ (note the addition of 1 to account for the bias). In convolutional layers, local regions share the weights of the filter kernel to significantly reduce the number of parameters. The same 5×5 input using a 3×3 kernel would only have $3\times3+1 = 10$ parameters. In

convolutional layers you are essentially scanning across the entire image for a specific feature. Locally connected layers have a median number of parameters, like the convolutional layer its looking for specific features and therefore uses a small filter kernel with weights, but these weights are not shared over all regions of the image. Instead they focus on specific regions (Figure 2.3.4). This type of layer is useful for example when classifying humans based on portrait images, the nose is always at the centre of the image, and therefore the kernels responsible for picking out these features will only be applied at the centre of the image.

2.3.6 Up sampling

Convolutional neural networks are not limited to classification problems. In some example cases which we will touch upon later, you may want to keep the dimensions of the network output the same or perhaps even larger than the input dimensions. For this up sampling layers will be required.

Transposed convolution

The transposed convolution is an up sampling layer. It takes an input and returns a spatially larger output. It is also sometimes known as a deconvolution layer, however this can be misleading since it implies a deconvolution operation - the inverse of a convolution operation, which is not what is happening. The transposed convolution is a more fitting name, since mathematically it is the transpose of the convolutional matrix

(Equation 2.2.1) that represents the kernel filter,

$$\boldsymbol{K}^{*\mathsf{T}} = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 \\ \mathbf{2} & \mathbf{1} & 0 & 0 \\ \mathbf{3} & \mathbf{2} & 0 & 0 \\ 0 & \mathbf{3} & 0 & 0 \\ \mathbf{3} & 0 & \mathbf{1} & 0 \\ \mathbf{4} & \mathbf{3} & \mathbf{2} & \mathbf{1} \\ \mathbf{2} & \mathbf{4} & \mathbf{3} & \mathbf{2} \\ 0 & \mathbf{2} & 0 & \mathbf{3} \\ \mathbf{2} & 0 & \mathbf{3} & 0 \\ \mathbf{1} & \mathbf{2} & \mathbf{4} & \mathbf{3} \\ \mathbf{1} & \mathbf{1} & \mathbf{2} & \mathbf{4} \\ 0 & \mathbf{1} & 0 & \mathbf{2} \\ 0 & 0 & \mathbf{2} & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{2} \\ 0 & 0 & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix} \tag{2.3.3}$$

that is multiplied by the unrolled input matrix,

$$D = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix}, \tag{2.3.4}$$

to produce an output matrix with higher dimensions than

the input.

$$\boldsymbol{K}^{*\intercal} \cdot D^* = \begin{bmatrix} 1 \\ 4 \\ 7 \\ 6 \\ 6 \\ 17 \\ 21 \\ 7 \\ 11 \\ 20 \\ 13 \\ 4 \\ 6 \\ 5 \\ 4 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 4 & 7 & 6 \\ 6 & 17 & 21 & 7 \\ 11 & 20 & 13 & 4 \\ 6 & 5 & 4 & 1 \end{bmatrix} \tag{2.3.5}$$

With same padding, the output size of the convolution layer is the $I \times S$, and with valid padding, the output size is $(I-1)S + K$. Images created using transposed convolution can often be seen to leave a checkerboard like artefact [Odena et al., 2016] caused by an uneven overlap of the mapping from the input to the output at each kernel operation. This **occurs when the kernel size is non-divisible by the stride**, and whilst trying to avoid such choices of hyperparameters in these layers can help, they generally cannot completely mitigate the problem because CNNs will typically use multiple layers of transpose convolutional layers further complicating the observed effect.

Resize-convolution

An alternative method for up sampling that avoids checkerboard artefacts is to use resize-convolution. This involves first upsampling the input feature map using an interpolation method such as **Bi-linear interpolation** or **nearest neighbour interpolation** and then applying a normal convolutional layer.

2.4 Regularisation

In deep learning there are many more parameters than a standard neural network, and consequently you more likely to over fit to your training data if your training data set is small. Recall that this means that whilst training the model may appear to be performing well but later does not perform well on test data. One way to improve the generalisability of your model is to increase the size of the training dataset.

2.4.1 Augmentation

Augmentation is the one of the simplest ways to increase the effective sample size. Taking samples of data from the training set there are many transformations you can apply to create additional training data. Examples of augmentation include cropping an image, changing the brightness or constrast, rescaling and translation (moving the image by a few pixels). At first thought, it might be tempting to use all the possible augmentation transforms to maximise the training data size, however, not all transformations have big gains on the performance of the network and may take a lot more time to train. Additionally care should be taken when deciding the type of

transformation, for example a horizontal flip on a image used for detecting car lanes would be a good idea, however the same transformation applied on an image where you are trying to distinguish the letter 'b' from the letter 'd' would only confuse the network.

2.4.2 Dropout

Another way to prevent over fitting on a small dataset is to train multiple different architectures on the dataset and take average. These methods are known as ensemble learning methods, however these methods not only consume computational power, but can also be draining on the memory to store the networks. An easier alternative is to use the dropout technique [Srivastava et al., 2014]. During dropout random nodes on in the model are randomly ignored according to a set probability during the training. This simulates having a large number of different architectures. Having dropout in a layer means the activation to the layer has a sparse activation and therefore this allows the network to learn a sparse representation. Dropout is a commonly used technique in deep networks, however dropout in a fully connected layer is not the same as dropout on a convolutional layer. After a convolutional layer, dropout can be seen as only adding noise to the network and therefore it's generally preferable to use batch norm instead.

Q. Show that the fully connected layer dropout is not equivalent to the convolutional dropout layer, and show why the latter is not effective for regularisation.

2.5 Architectures

So now you know the layers that make up a convolutional neural network, you must also realise there are many ways to build and tune your network. The architecture is what refers to the layout of different neurons, layers and activation functions. Typically the architecture is designed through trial and error, although there are some techniques that are proven to work. In this section we will talk about the design of a selection of well known architectures.

Whilst convolutional neural networks were powerful, by 2010 they were still largely unadopted by the machine learning community. Stanford University's Fei-Fei Li realised that even the best machine learning tools could not work well unless the data reflected the real world, and so in 2006 began on a project known as Imagenet. Imagenet is a compilation of over 14 million images of everyday objects. The images consisted of over 200,000 different object categories with hand classified labels and bounding boxes. Following in the footsteps of the PASCAL VOC challenge[1], the database came with an annual competition (ILSVRC) to classify these images, and this provided the motivation for the next generation of machine learning codes. Before imagenet, researchers would create bespoke architectures for a specific class of object - one for classifying cats and another for classifying dogs. Imagenet allowed for the development of multipurpose architectures. In the early years, the competition was dominated by feature engineering algorithms, and the **top-5 error percentage** (the percentage of images where the correct classification did not feature in the

[1] `http://host.robots.ox.ac.uk/pascal/VOC/`

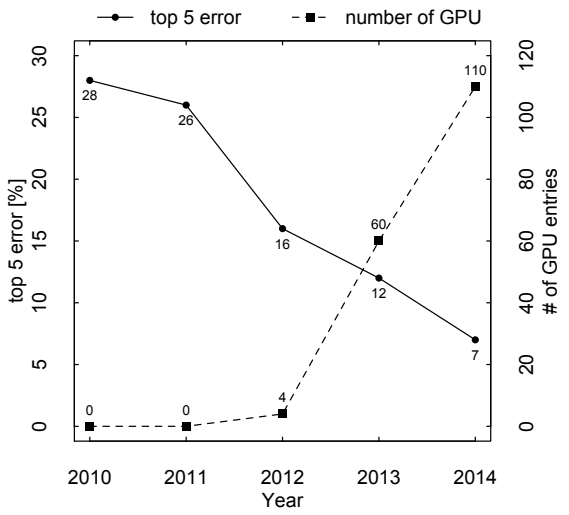


Figure 2.5.1: The top 5 error of the winning entry in the ILSVRC competitions and the corresponding number networks that used GPUs.

top 5 classes predicted by the network) hovered around 26-28%. This changed in 2012, with the winning entry `Alexnet` [Krizhevsky et al., 2012], a convolutional neural network.

Alexnet

LeNet-5 essentially had 5 layers; 3 convolutional and 2 fully connected layers, giving it around 60,000 parameters. Alexnet on the other hand has 5 convolutional layers and 3 fully connected layers giving it around 60 million parameters. For this reason, Alexnet is known to be the first real deep convolutional neural network. Previously, CNN's were limited depthwise due to the lack of computational power which in turn limited their performance, but as shown in Figure 2.5.1, 2012 was the year that GPU's became popular in the machine learning community and hence the performance and popularity of CNNs.

The record breaking performance of Alexnet has been contributed by training on the huge Imagenet dataset, the use of regularisation techniques like data augmentation and dropout and the use of the ReLU activation which significantly improved training efficiency. The major point however is that Alexnet was one of the 1st networks to use GPU in ILSVRC, the Alexnet architecture is designed for use with 2 GPUs, but the network still took almost a week to train.

MobileNet

The design of architectures is often a trade off between efficiency and accuracy. A big, complex network may provide higher accuracies, however this may come at the cost of requiring a high number of parameters and slow training of the network. MobileNet [Howard et al., 2017] is designed to be a small and low latency architecture such that training can even be performed on mobile devices. The main concept behind architecture is that the standard convolutional filter is computationally expensive. For layers where there is more than 1 filter used, it can be more favourable to replace the standard convolutional layer with 2 layers, a depthwise convolution followed by a pointwise convolution with the same depth as the number of filters required when using the standard convolutional layer, collectively this is called the depthwise separable filter. It not only reduces the number of parameters in the layer but also the number of multiplication operations and hence the speed. MobileNet uses a series of 3×3 depthwise separable filters for a factor of 8 increase over conventional convolutional layers.

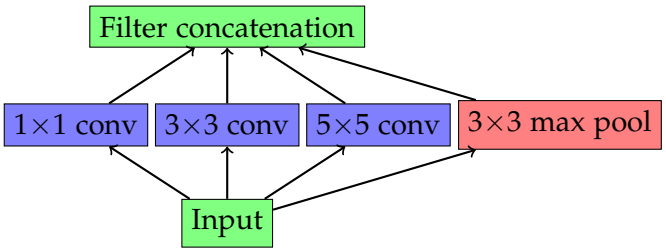


Figure 2.5.2: The naive Inception cell contains various sized filters in parallel. The full Inception cell also contains various 1×1 convolutions to obtain consistent dimensions

VGGNet

It's quite common that you will see 3×3 filters used in CNNs since another architecture, VGGNet [Simonyan and Zisserman, 2014] demonstrated that two 3×3 convolutional layers is equivalent to a 5×5 receptive field and thus able to capture larger features whilst maintaining a small number of parameters to train for. The VGGNet architecture only used 3×3 filters and is an important architecture in the history of CNNs because it demonstrated the power that could be achieved by building simple but deep architectures. Complex architectures not only drain memory and power, but are also more susceptible to over fitting.

Inception

Inception is another notable architecture and milestone for CNNs. Whilst previous architectures used linearly stacked convolutional layers, Inception employed what is known as an inception module. In the former, it would be quite difficult to tailor an architecture for image classification with multi-scale objects. Smaller size filters are more adapted for small scale,

local features and larger size filters more adapted to global features. The main concept behind the inception module is to apply different sized filters in parallel and then concatenate them together. In Inception v1, the inception module was essentially a 1×1, a 3×3 and a 5×5 filter (Figure 2.5.2). This then allows for learning on multiple scales.

NASNet

Designing a high performance architecture is difficult, there are many different layers and hyper-parameters for each of these layers. Transfer learning, which we will talk about in **??** can help but often you want an architecture tailored more specifically for your problem. This motivated the development of Neural Architecture Search (NAS), a technique to find the optimal neural architecture. AutoML is a recurrent neural network that Google created to perform NAS [Zoph and Le, 2017]. The network would be provided with building blocks of operations (e.g. 3×3 convolution, concatenation, max pool) from which it would propose a new neural network comprising of different combinations of these operations. NASNet is the result of AutoML and is one of the best performing CNNs. What's interesting about this network is that is comprises mainly of 2 cells, called the normal cell and the reduction cell. These cells make use of the parallel operations similar to that of Inception.

Resnet

The 2015 Imagenet competition was won by Microsoft with `Resnet` [He et al., 2015], an ultra deep CNN with almost $20\times$

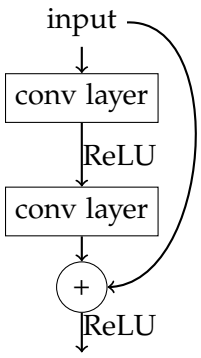


Figure 2.5.3: Residual cell used in `Resnet` consists of a convolution-ReLU-convolution before the output is summed with the input and another ReLU is applied.

the number of layers as `Alexnet` (152 layers). Given that the best human top 5 error on the Imagenet dataset is about 5%, `Resnet` impressed the community by surpassing humans with a 3.6% top 5 error[2]. Naively, it may appear that simply increasing the depth of network can improve testing accuracies, however as the authors of the paper show, this is not the case. The main feature of `Resnet` is the use of a residual cell (Figure 2.5.3). The residual net cell consists of the operations of a convolutional layer, followed by ReLU activation, followed by convolution. The output of this is then summed with the input before the another ReLU activation is applied. If a traditional Convolutional layer takes an input x and outputs $s(x)$, the residual cell takes input x and outputs $s(x) + x$. The cell is essentially computing the residual changes to the input image, which is easier to optimise than traditional convolutional layer weights.

The last ILSVRC competition ran in 2017 with the top 5 error of the winning entry 2.3%.

[2]Try yourself here: `https://cs.stanford.edu/people/karpathy/ilsvrc/`

Region based-CNNs

Region-based CNNs [Girshick et al., 2013] or R-CNNs are CNNs designed specifically for object detection. Here the problem of interest is no longer simply classification, but also localisation of objects. The input to an R-CNN would be an image, and the output would be a vector containing the class of object, classification score and coordinates for a bounding box around the object of interest. RCNNs do this by splitting the problem into two networks. The first part extracts region proposals from the image, and whilst in principle any region proposal would work, RCNN uses a selective search to generate thousands of regions in the image where there is a high probability of an positive detection. These proposals are then resized to a common size for input to the second part of R-CNNs, a CNN that extracts the features in the proposed regions. Here the output vector enters a linear SVM and bounding box regressor to train for the classification and bounding box coordinates. Variants of R-CNN have since improved efficiency of object detection over the traditional R-CNN [Ren et al., 2015].

2.6 Applications

When we talk about CNNs, we are typically referring to 2D convolution which acts on 2D data such as the images we have demonstrated. However, the applications of convolutional neural networks are not exclusive to image classification. In fact CNNs can be applied to 1D data such as audio or other time series data, and they can be applied on 3D data such as videos. For 1D data however, CNNs are searching for

fixed-length features irrespective of the location. In this section we discuss some of the applications of CNNs on various types of data and examples of what has been done.

2.6.1 Image recognition

Image recognition is the general term to describe computer vision tasks for identifying objects in images. It can be divided into 4 main categories,

1. Classification

Image classification is the most obvious application of CNNs. Here the task is to predict the label (class) that features in a image. For example an image may contain a cat, and the task could be to predict if its a Persian breed or a Siamese breed. Most of the CNN architectures in section 2.5 that we have spoke about have been developed for image classification.

2. Localisation

In image classification, you typically have only one object in the image, and although you may have more, the object types will be the same. In some scenarios, you might not be interesting in the object itself but instead where they are located in an image. Image localisation using CNNs can work in exactly the same way as classification. But here instead of returning a score for a given class, the CNN would return 8 numbers to describe the location of the box, this could be the coordinates of the 4 corners of the box ($\mathbf{C}_{i,j}$), or more typically the width, height and centre coordinate of the box. The loss function could then for example be the L2 loss, equivalent to the

Euclidean distance between the true coordinates and the predicted $\hat{\mathbf{C}}_{i,j}$,

$$l = \sum_{i=1}^{2} \sum_{j=1}^{2} ||\mathbf{C}_{i,j} - \hat{\mathbf{C}}_{i,j}||_2, \tag{2.6.1}$$

where $||x||_2$ is the L2 norm.

3. Detection

For images wherein exists multiple objects of different classes, image detection can be used to **both localise objects and classify** them. Object detection is a combination of image classification and localisation. In this scenario, the output of the network would be the bounding box parameters for the detected objects in the image and the classification scores within each bounding box. One way to do this would to simply use sliding boxes of various scales to apply a classical cnn for image classification on each patch of the input image. Not only would this be very inefficient, but also the choice of box scale and aspect-ration needs to be determined.

Architectures like R-CNN (section 2.5) were developed to obtain a more principled selection of boxes to apply the standard CNN, using selective search to propose regions of interest, a CNN to classify and then regression to optimise. Similarly the variations of R-CNN such as fast RCNN and faster RCNN, split the tasks of classification and localisation. The methods do not process the entire image at once, but instead look at high resolution patches. Never the less the selective search algorithm can be very slow - thousands of forward passes through the CNN are required for each proposal region of each image. So whilst **R-CNN** architectures demonstrate good

performance for detection, they are not ideal for real time processing.

Single shot detectors[SSDs, Liu et al., 2015] [including YOLO, Redmon et al., 2015, and variants] use a single CNN architecture to obtain both bounding box parameters and object class. In an SSD, convolutional layers are first applied to reduce the dimensionality of the input. The input is split into a grid of cells and at each cell, anchors (predefined initial guesses of the box) are used to then make the detections. Since these types of models use low resolution images to make the detection, they are very fast but on the trade off for accuracy. For this reason they are more adapted for applications where real time processing is required. Additionally the performance on low resolution images is scale dependent, can affect the detection of small scale objects.

4. Segmentation

Bounding boxes are a good way to localise objects in an image, however taking it a step further, localisation could be done with segmentation. In segmentation, objects are classified on pixel-by-pixel basis. This is can be a desired result if for example you have multiple overlapping objects or perhaps are interested in the outlines of objects.

Image segmentation can itself be split into two different categories,

- **Semantic segmentation**, classifies each pixel of an input to identify the object class it belongs to. Again using a sliding box CNN approach to classify every pixel would be possible but very inefficient. Fully convolutional layers

are commonly used for image segmentation. They use a combination of downsampling (e.g. convolution) and upsampling (e.g. transpose convolution) layers to produce an output with the same size as the input image [see e.g. Ronneberger et al., 2015].

- **Instance segmentation**, classifies each pixel of an input to identify the object instance it belongs to. For example a litter of puppies segmented using the semantic would produce a single segment for all the puppies, but using the instance segmentation it would provide segments for each individual puppy. Therefore instance segmentation can be seen as a combination of semantic segmentation and object detection. `Mask-RCNN` is a popular architecture for instance segmentation, it is based upon `Faster R-CNN`, but in addition to the object class, score and bounding box, the output to the CNN will also produce an image mask to represent the segmentation. The image mask is produced taking into account each region of interest separately.

Applications of image segmentation include facial recognition, medical imaging and autonomous driving.

2.6.2 Image enhancement

In addition, fully convolutional networks are also a popular choice for image enhancement and restoration problems including denoising, deblurring, artefact removal and super resolution. The latter of which involves taking a low resolution image and transforming it into a high resolution output.

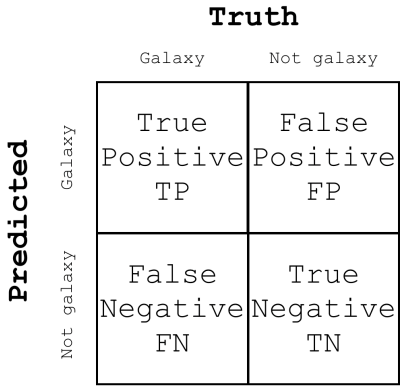


Figure 2.7.1: Confusion matrix for a model that does binary classification of galaxies. The matrix values should be filled with the numbers from the model prediction.

2.7 Model Evaluation

During training of the model, the loss is often the metric that is used to monitor how the model is performing as a function of training time. Simultaneously monitoring both the loss of the training data set and the validation data, can tell you if the model is performing as it should (i.e. dropping loss), if it's underfitting (i.e dropping validation loss) or it's overfitting (i.e. increasing validation loss). However, the loss cannot tell you how well your model is performing. The value of the loss is relative, and can vary depending on the loss function you decide to use. Instead other metrics are required to describe the performance of the network and to enable comparison with other models.

2.7.1 Confusion matrix

In the context of CNN's, classification is the most common task. For binary classification, if for example we were interested in knowing if an image is a galaxy or not a galaxy, we can log the classifications of a network on a test data set in a confusion matrix (Figure 2.7.1). Within the confusion matrix, the true positives (TP) are the number of images correctly classified as galaxies, true negatives (TN) are the number of images correctly classified as non-galaxies, false positves (FP) are the number of non-galaxies incorrectly classified as galaxies and false negatives (FN) are the number of galaxies incorrectly classified as non-galaxies. The sum of the columns tell you the total number of galaxies (P) and non-galaxies (N) in the test sample and this matrix can be extended for multi-class classification by extending the number of rows and columns to represent each class.

2.7.2 Accuracy

The confusion matrix is a great way to spot biases in the model, however it would be more ideal to have a single metric to evaluate the model. The **accuracy** of a model is given by,

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{P} + \mathrm{N}}. \tag{2.7.1}$$

In general the accuracy is good metric to use for classification since it tells us with a single number how well the network performs on a scale of 0-1. However accuracy is not a good metric if the data classes are unbalanced. Accuracy favours the larger class, for example if there were 9 non-galaxies and 1 galaxy, if

the model predicts all objects as non-galaxies, it would still get a 90% accuracy, even though it is a terrible galaxy classifier!

2.7.3 Precision, Recall and F1 score

Instead we can look to other metrics that are better representations for unbalanced data. **Precision** tells us the purity of the predictions,

$$\mathrm{PRE} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \tag{2.7.2}$$

and **recall** tells us the completeness of the predictions,

$$\mathrm{REC} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}. \tag{2.7.3}$$

The former is more important where you need a predictions free of contaminants - such as if you a sample of supernova lightcurves, you wouldn't want a variable star to contaminate the signal. The latter is more important where you need to know even if there is a slight possibility, e.g. cancer detections. There is also the **F1 score** which is a mid point, between being pure and complete,

$$\mathrm{F1} = 2\frac{\mathrm{PRE} \times \mathrm{REC}}{\mathrm{PRE} + \mathrm{REC}} \tag{2.7.4}$$

2.7.4 Multiclass metrics

For multiple classes, the above metrics can also be calculated as a **micro-average** to give equal weights to each individual classification, e.g.,

$$\mathrm{PRE}_{\mathrm{mic}} = \frac{\mathrm{TP}_1 + \mathrm{TP}_2 + ...}{\mathrm{TP}_1 + \mathrm{TP}_2 + ... + \mathrm{FP}_1 + \mathrm{FP}_2 + ...}. \tag{2.7.5}$$

This metric is biased towards large classes, and therefore it might be preferable to instead use the **macro-average** and give equal weight to each class, e.g.,

$$\text{PRE}_{\text{mac}} = \frac{\text{PRE}_1 + \text{PRE}_2 + ... + \text{PRE}_K}{K}, \tag{2.7.6}$$

where k are the number of classes.

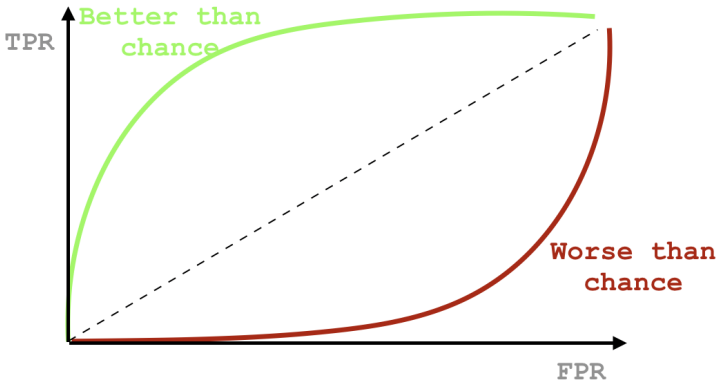


Figure 2.7.2: Receiver operating characteristic curve. The dashed line shows where the curve should lie if the the model performance is equal to chance. The green line shows the curve for a network that performs better than chance, and the red line shows a network that performs worse than chance.

2.7.5 ROC curve and AUC

The receiver operating characteristic (ROC) curve (Figure 2.7.2) is a common visual used to show the performance of a classification network, it plots the true positive rate (TPR) which is the recall, against the false positive rate (FPR),

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}, \tag{2.7.7}$$

for different threshold values of positive prediction. The area under the ROC curve (AUC) summaries the ROC into a single

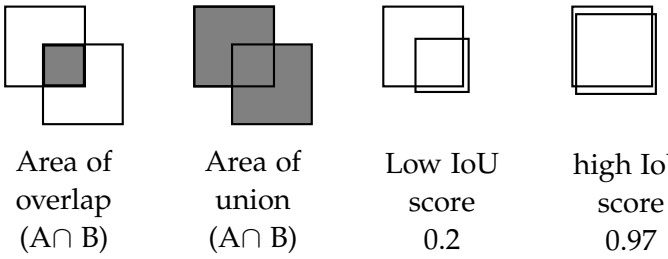


Figure 2.7.3: IoU scores are a popular metric used to quantify accuracy in image localization tasks. The metric takes into account both the area of overlap and the area of union.

value between 0 and 1,

$$\text{AUC} = \int_{x=0}^{1} \text{TPR}(\text{FPR}^{-1}(x))dx. \tag{2.7.8}$$

2.7.6 IoU and mAP

A common metric used in interpreting the performance of a localisation model is the intersection over union (IoU) score, which measures the area of overlap between the true bounding box A and the predicted box B (Figure 2.7.3),

$$\text{IOU}(A, B) = \frac{|A \cap B|}{|A \cup B|}. \tag{2.7.9}$$

The IoU score scales from 0-1 and higher values are better. Additionally, the average precision, given as the area under the precision-recall curve for various threshold values of IoU scores is another common metric, and for multiple classes the mean average precision (mAP) can be used - the mean average precision over each class.

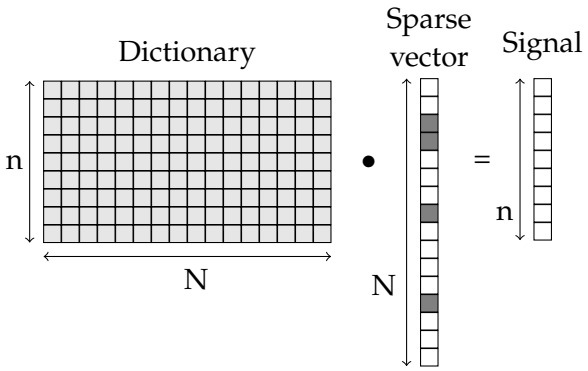


Figure 2.7.4: Any image can be approximated by a linear combination of basis images, and therefore can be represented as the dot product of a dictionary of bases and a sparse matrix. Here there are N atoms and each atom (column) in the dictionary represents a small image patch of size n, the matrix multiplication with the size N sparse vector produces the signal with the same size as each atom.

```
https://github.com/MaggieLieu/MLiS_examples/blob/
master/CNN_examples.ipynb
```

2.8 Sparse coding

In deep learning, we are working in the domain of big data, and while the data may be very different, the data contains structure. The purpose of machine learning is to exploit the structure in data. Whilst there are a range of models that can be applied to a data set to do the same thing, in general, reliable models tend to be simple ones. Sparse representation is already popular concept for priors in the signal processing community. Sparse coding is an unsupervised learning method to learn a set of bases that can more efficiently represent data. It reduces the complexity of neural networks with

minimal decrease in performance.

In image processing, each small patch in an image ($\boldsymbol{X}$) has a sparse representation ($\boldsymbol{\alpha}$), with respect to a common dictionary ($\boldsymbol{D}$)(Figure 2.7.4), such that,

$$\boldsymbol{X} = \boldsymbol{D}\boldsymbol{\alpha}. \tag{2.8.1}$$

$\boldsymbol{D}$ and $\boldsymbol{\alpha}$ are redundant matrices, since typically their combined dimensions are larger than the input. It may seem strange that we would want to replace a small input matrix with 2 larger matrices, but later you will see that the nature of the sparse matrix $\boldsymbol{\alpha}$ can lead to increased efficiencies in convolutional networks.

When $\boldsymbol{X}$ and $\boldsymbol{D}$ are known, $\boldsymbol{\alpha}$ can still take on many different solutions. We require $\boldsymbol{\alpha}$ to take on the **sparsest** solution - that containing the least non-zero values.

$$\min||\boldsymbol{\alpha}||_0 \quad \text{s.t.} \quad \boldsymbol{X} = \boldsymbol{D}\boldsymbol{\alpha}, \tag{2.8.2}$$

Where $||\alpha||_0$ is the number of non-zero values in α. More generally, if the input $\hat{\boldsymbol{X}}$ is a noisy observation of $\boldsymbol{X}$ then,

$$\min||\boldsymbol{\alpha}||_0 \quad \text{s.t.} \quad ||\hat{\boldsymbol{X}} - \boldsymbol{D}\boldsymbol{\alpha}||_2 \leq \epsilon, \tag{2.8.3}$$

where $||\alpha||_2$ is the L2 norm of α and ϵ is the noise. Both Equation 2.8.2 and Equation 2.8.3 are NP-hard problems - their computational cost to solve increases exponentially with the number of atoms in the dictionary.

This problem of atom decomposition can be solved using approximation algorithms, of which there are two main approaches, greedy methods such as orthogonal matching pursuit (OMP) gradually introduces non-zeroes to the sparse representation to solve Equation 2.8.3, and stops when the error

falls below ϵ, and relaxation methods such as basis pursuit (BP) relaxes the L0-norm in Equation 2.8.3 into a L1-norm which is easier to solve for.

2.8.1 Mulit-layered convolutional sparse coding

Larger images ($\boldsymbol{X}$) can be written as the sum of N filters ($\boldsymbol{K}$) convolved with their sparse representations, where $\boldsymbol{\Gamma}_i$ is a feature map that holds the sparse representation of the ith filter.

$$\boldsymbol{X} = \sum_{i=1}^{N} \boldsymbol{K}_i * \boldsymbol{\Gamma}_i \tag{2.8.4}$$

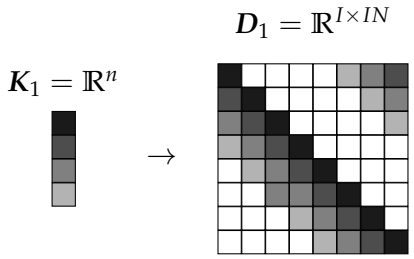


Figure 2.8.1: A kernel K, can be unrolled into an $I \times IN$ matrix where I is the size of the flattened input and N is the number of filters.

Like we did with the convolution and transpose convolution layers, the filters in convolutional sparse coding can benefit from being unrolled (Figure 2.8.1) such that a 1x4 filter can be written as the dot product with the unrolled sparse matrix,

$$\boldsymbol{X} = \boldsymbol{D}\boldsymbol{\Gamma}. \tag{2.8.5}$$

For the first layer in a convolutional neural network, we can write the input signal $\boldsymbol{X}$ of size I in terms of the convolutional dictionary $\boldsymbol{D}_1$ with dimensions $I \times IN_1$ and the locally sparse

representation $\mathbf{\Gamma}_1$ of size IN_1, $\boldsymbol{X} = \boldsymbol{D_1}\mathbf{\Gamma}_1$ where locally sparse refers to the constrain that

$$||\mathbf{\Gamma}_1||_0 \leq N_1(2n_0 - 1), \tag{2.8.6}$$

with N_1 - the number of filters in the first layer, and n_0 - the length of the filters in layer 1. In the second convolutional layer, the input is the sparse representation of layer 1, and this too can represented in a similar form $\mathbf{\Gamma}_1 = \boldsymbol{D_2}\mathbf{\Gamma}_2$. In this layer $\boldsymbol{D_2}$ has dimensions, $IN_1 \times IN_2$ and $\mathbf{\Gamma}_2$ has dimensions IN_2. $\mathbf{\Gamma_2}$ is subject to different sparsity constraints to the first layer,

$$||\mathbf{\Gamma}_2||_0 \leq N_2(2n_1N_1 - 1), \tag{2.8.7}$$

where N_2 are the number of filters in the second layer and n_1 are their lengths. This structure is then cascaded through the layers of the CNN. It can therefore be shown that, a M-layered sparse convolutional neural network can be written as, $\boldsymbol{X} = \boldsymbol{D_1}\boldsymbol{D_2}...\boldsymbol{D_M}\mathbf{\Gamma}_M$ or more succinctly $\boldsymbol{X} = \boldsymbol{D}_{eff}\mathbf{\Gamma}_M$, where the effective dictionary $\boldsymbol{D}_{eff}$ is a chain of each layer's dictionary.

2.8.2 Comparison to classic CNN

The sparse vector in layer 1 has a solution,

$$\min ||\mathbf{\Gamma_1}||_0 \quad \text{s.t.} \quad \hat{\boldsymbol{X}} = \boldsymbol{D}_1\mathbf{\Gamma}_1 + \boldsymbol{E}, \tag{2.8.8}$$

where $\boldsymbol{E}$ is the noise. This can be described as solving,

$$\hat{\mathbf{\Gamma}} = \min_{\Gamma} \frac{1}{2}||\hat{\boldsymbol{X}} - \boldsymbol{D}_1^{\intercal}\mathbf{\Gamma}_1||_2^2 + \beta||\mathbf{\Gamma}_1||_0 \tag{2.8.9}$$

for some scalar value of β. One simple way to solve this is to use a threshold algorithm, where the inner product of the atoms in the dictionary and the signal is computed and then

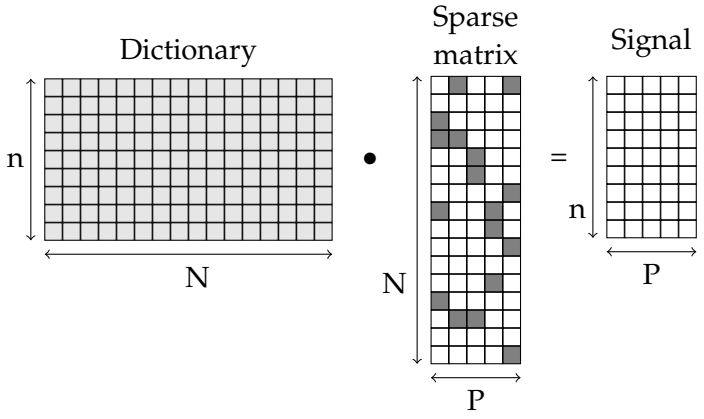


Figure 2.8.2: Multiple (P) signals can be represented by a single dictionary multiplied by a sparse matrix, where each column corresponds to the sparse vector representation of the each column in the signal matrix.

the lowest responses are removed via some threshold function,

$$\mathcal{T}_1(\boldsymbol{D}_1^\intercal \hat{\boldsymbol{X}}) \tag{2.8.10}$$

The thresholding is applied sequentially for each layer such that the 2nd layer would take the form,

$$\mathcal{T}_2(\boldsymbol{D}_2^\intercal \mathcal{T}_1(\boldsymbol{D}_1^\intercal \hat{\boldsymbol{X}})). \tag{2.8.11}$$

On the basis that the sparse vector is non-negative, one possible threshold function is the soft non-negative function,

$$\mathcal{T}(x) = \begin{cases} 0, & x \leq \beta \\ x, & x > \beta \end{cases} \tag{2.8.12}$$

Therefore Equation 2.8.11 can be written as,

$$\mathrm{ReLU}_2(\boldsymbol{D}_2^\intercal \mathrm{ReLU}_1(\boldsymbol{D}_1^\intercal \hat{\boldsymbol{X}} + \beta_1) + \beta_2). \tag{2.8.13}$$

This is equivalent to a 2 layer convolutional neural network.

2.8.3 Learning the dictionary

Traditionally, sparse coding uses pre-defined dictionaries, or analytically derived dictionaries from known transforms (e.g. wavelets, discrete cosine transform etc.). An alternative approach is to learn the dictionary from the data. This can be done if there are multiple signals, and hence multiple sparse representations of the data with respect to a common dictionary (Figure 2.8.2). To obtain a sparse representation for all the signals we need to solve,

$$\min_{D,\Gamma} \sum_{j=1}^{P} ||\boldsymbol{X}_j - \boldsymbol{D}\boldsymbol{\Gamma}_j||_2^2 + \beta||\boldsymbol{\Gamma}_j||_0 \tag{2.8.14}$$

where P is the number of signals. The dictionary can be learnt in either a supervised or unsupervised manner, with the general steps as follows,

1. Initiate the atoms in the dictionary by randomly sampling them from the signal.

2. Using the dictionary and signal, compute the sparse representation by cycling through each column of the sparse matrix and using a pursuit algorithm, i.e. solving

$$\boldsymbol{\Gamma}^*(\boldsymbol{X}_j, \boldsymbol{D}) = \min_{\Gamma} ||\boldsymbol{\Gamma}_j||_0 \quad \text{s.t.} \quad \boldsymbol{X}_j = \boldsymbol{D}\boldsymbol{\Gamma}_j \tag{2.8.15}$$

3. These sparse representation can be fed to a classifier with parameters $\boldsymbol{U}$ and the dictionary is updated by solving,

$$\min_{D,U} \sum_{j} \ell(h(\boldsymbol{X}_j), \boldsymbol{U}, \boldsymbol{\Gamma}^*(\boldsymbol{X}_j, \boldsymbol{D})) \tag{2.8.16}$$

   where ℓ is the loss with respect to the true signal $(h(\boldsymbol{X})_j)$. This can be solved with gradient descent in a supervised

manner, or if $\boldsymbol{U}$ is of no importance, it reduces to the unsupervised form,

$$\min_{D} ||\boldsymbol{X} - \boldsymbol{D}\boldsymbol{\Gamma}^*||_2^2 \tag{2.8.17}$$

using K-SVD, this is solved atom wise, by taking all signals that use the given atom, subtracting the contribution from other atoms, and using a method such as singular value decomposition (SVD) to update the atom and reduce the residuals. Although other methods also exist.

4. Steps 2 and 3 are repeated over many iterations.

2.9 Geometric deep learning

So far, we have only touched upon CNNs in the euclidean domain i.e. grid based, structured data and most research up until recently have focussed on CNNs in this domain.

Nevertheless, there are many fields such as biology and physics, where the data of interest is non-euclidean in nature. Non-euclidean data include things like molecules, trees, networks and manifolds. This kind of data is generally not structured.

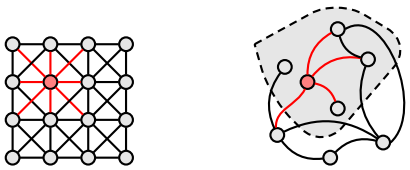
2.9.1 Graph theory

Graphs $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consist of a set of vertices $\mathbf{V} = \{v_1, v_2 ... v_n\}$ known as the vertex set and a set of edges $\mathbf{E} = \{e_{11}, e_{12} ... e_{i,j}\}$ that link pairs of vertices together. The number of neighbours of a node in a graph is known as the degree, defined as,

$$d(v_i) = \sum e_{i,j} \tag{2.9.1}$$

where the sum is made over neighbouring nodes.

2.9.2 Graph convolution



2D convolution Graph convolution

Figure 2.9.1: On the left we have 2D convolution represented in graph form. The circles represent nodes and the lines the connections. The data is structured and all neighbouring nodes, determined by the filter size are connected. Convolution aggregates information together of neighbouring nodes. On the right is a graph convolution, the nodes are not structured and can vary in size, and therefore neighbours are less well defined.

Graph neural networks [Gori et al., 2005, Scarselli et al., 2008] were inspired by recurrent neural networks which will be covered later in this course. They take graph data as inputs and sequentially processes it through hidden layers that are equivalent to intermediate feature representations in neural networks in order to classify either the vertices (nodes) or the edges (connections) according to a class label. A popular example of graph data would be social networks where individuals could be represented by nodes, edges could be used to represent the interactions between people and the classes could for example be their age, additionally you may have feature vectors for each of the nodes, such as their favourite music genre and favourite food. Alternatively if the edges represent transactions between people, the classes could be a fraud transaction and a valid transaction. Standard neural networks would not work well on this data, since nodes are not fixed spatially, they can move around, and nodes that are close spa-

tially do not necessarily connect via an edge. Also, given how large graph data can grow, it may make sense to use a method analagous to convolutional neural networks - graph convolutional networks [GCNs, Kipf and Welling, 2016].

In 2D convolution, the data is structured (Figure 2.9.1). Each pixel can be represented as a node, and the filter size determines the which nodes are neighbours. The filters extract features in the local neighbourhood and aggregate neighbouring features in subsequent layers. Here the nodes are ordered and they have the same size (all pixels are equivalent). In graph convolution, the data is not structured. Like in 2D convolution, in order to get the representation of a node, one way is to take the average of its neighbouring nodes, however graph data is not structured. Nodes can vary in size and the filter size cannot be used to determine which nodes are neighbours. The definition of convolution, the method of sequentially aggregating node features, on graph data must therefore be redefined.

In order to generalise convolution to graph data, we need define the parameterised filters used in the convolutional layers. There are 2 main methods to do so, spatial methods (e.g. GraphSage, Hamilton et al. [2017] and Neural FPS Duvenaud et al. [2015]) are problem specific architectures that aggregate features in neighbouring nodes, and spectral methods (e.g. GCNs, Kipf and Welling [2016] and Chebnet, Defferrard et al. [2016]) involve eigenvalue decomposition like in Principle Component Analysis, however here, it is the smaller eigenvalues that describe the structure.

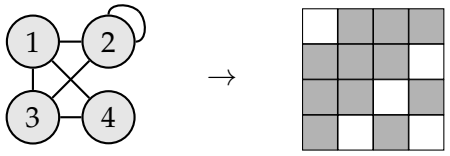


Figure 2.9.2: On the left you have the visualisation of some graph data with nodes and connections that are labelled arbitrarily. The right shows the corresponding adjacent matrix, where black elements are corresponding to connections and white elements correspond to lack of connections. At coordinates (1,2) and (2,1) the elements of the adjacent matrix are black showing the connection between nodes 1 and 2. At (1,1) the element is white to show there is no self-connection at node 1.

Graph data can be represented visually as shown in (Figure 2.9.1), but in order to use them in GCN we need to project them onto an adjacency matrix $\boldsymbol{A}$ which encodes the information about the nodes and the connections and has size N^2 where N is the number of nodes in the input (Figure 2.9.2). Note that the adjacency matrix has to be symmetric and so one limitation is that edges are treated as undirected. This restriction means that representational hierarchy can only be built by graph convolutional operation followed by pooling or expansion and this is what was common for early classic CNN methods, which has since been replaced by strided and transpose convolutions and skip connections. There are however ways around this by for example using Bipartite graph convolutions [Nassar, 2018] or weighted directed graphs where each edges is represented by a 3-tuple rather than a 2-tuple [Clemente and Grassi, 2017]. Furthermore adjacent matrices unlike classic CNNs are not invariant to small shifts or transformations (Figure 2.9.3). However regardless of how the graphs are sampled to produce the adjacent matrix, the targets will always be the same.

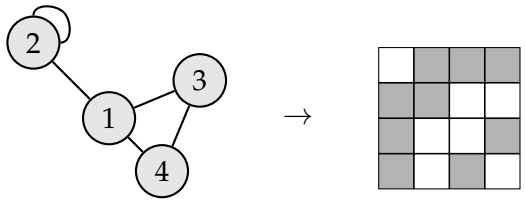


Figure 2.9.3: Here we have changed just one pixel in the adjacent matrix from Figure 2.9.2, but the structure of the graph data can be drastically changed.

The input to the graph convolution layer is a feature matrix $\boldsymbol{X}$ which consists of a value for each of the F features, for each of the N nodes and therefore has size $F \times N$. Each neural layer is a non-linear function of the adjacent and feature matrix,

$$\boldsymbol{H}^{(l)} = f(\boldsymbol{H}^{(l-1)}, \boldsymbol{A}) \tag{2.9.2}$$

Where l is the lth layer in the network. In the first layer $\boldsymbol{H}^{(0)} = \boldsymbol{X}$, but since the features may not be known a priori, they can either be randomly initialised or $\boldsymbol{X}$ can be taken as the identity matrix. The difference between spectral and spatial GCNs are how $f(x, y)$ is defined and parameterised.

Following in the context of CNNs, we could naively use,

$$f(\boldsymbol{H}^{(l)}, \boldsymbol{A}) = \Phi(\boldsymbol{A}\boldsymbol{H}^{(l)}\boldsymbol{K}^{l}), \tag{2.9.3}$$

where $\boldsymbol{K}^{l}$ is a matrix of weights of the the lth layer and $\Phi(x)$ is a non-linear activation function like ReLU. The problem, is that $\boldsymbol{A}\boldsymbol{H}^{(l)}$ sums up the feature vectors of all neighbouring nodes but not the feature vector of the node itself unless a self connection is imposed, therefore we replace $\boldsymbol{A}$ with $\hat{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$ where $\boldsymbol{I}$ is the identity matrix. Additionally for large graph data, the number of neighbours of nodes can be very large, and consequently the number of aggregated features will become

increasingly large at each layer. For this reason $\boldsymbol{A}$ is normalised by the degree matrix $\boldsymbol{D}$, a diagonal matrix where $D_{ii} = \sum_j A_{ij}$. Since $\boldsymbol{A}$ is a symmetric matrix, to ensure the normalised matrix is also symmetric, this is applied as $\boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$. Thus Equation 2.9.3 becomes,

$$f(\boldsymbol{H}^{(l)}, \boldsymbol{A}) = \Phi(\hat{\boldsymbol{D}}^{-1/2}\hat{\boldsymbol{A}}\hat{\boldsymbol{D}}^{-1/2}\boldsymbol{H}^{(l)}\boldsymbol{K}^{l}). \tag{2.9.4}$$

Just like CNNs, a simple 2-layer semi supervised GCN could look something like,

$$\hat{\boldsymbol{Y}} = \text{softmax}(\hat{\boldsymbol{A}}\,\text{ReLU}(\hat{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}^{(0)})\boldsymbol{W}^{(1)}) \tag{2.9.5}$$

In semi supervised classification, some nodes are labeled and others are not. Any loss can be used, e.g. cross entropy, however, when calculating the loss, only the labelled nodes are taken into account. This assumes that locally connected nodes are likely to share the same label.

Graph downsampling

The hierarchical structure in CNNs is largely down to the downsampling (pooling) layers. To use these layers however graph networks require first clustering the graph before pooling layers can be applied. This is also known as graph coarsening, and can be done using various algorithms specific to the graph data. For point cloud data there is VoxelGrid and Self organising networks, for general weighted graphs there is Graculus. The max or average pooling occurs across collapsed vertices, and this coarsening structure can be represented as a binary tree.

GCN applications

GNNs can be thought of not only looking for patterns in the direct connections between nodes, but also the indirect connections. The features of nodes become increasingly more abstract at each consecutive layer. The applications are not limited to classification, but also generation of new drug candidates, detecting fraudulent bank transactions, etc. Graph convolutional networks are still an area of active research and one limitation is that graph data is difficult to fit into GPU memory because the adjacency matrix needs to be stored. Nevertheless it has been shown that for large graphs, CPU can be used without too much loss to the efficiency.

2.9.3 Riemannian manifolds

The graph networks mentioned so far are in a fixed domain - at any instance the connections between nodes are fixed. There are cases however where we would want to have varying domains, for example on 3D moving objects, you would want the same network to work at all instances.

A 3D shape can be described by a topological surface called a **Manifold** χ. At any point x on the surface, the local area has a Euclidean representation of χ, known as the **tangent plane** $T_x\chi$. The **Riemannian metric** is defined as the family of inner products of all tangent planes about x,

$$g_x\langle .,. \rangle_{T_x\chi} : T_x\chi \times T_x\chi \to \mathbb{R}, \tag{2.9.6}$$

and this describes the smooth local intrinsic structure at x, and allows us to meausre lengths and angles. The **Scalar fields** f describe a scalar quantity at all points over χ, $f : \chi \to \mathbb{R}$,

and the **vector fields** F describe a tangent vector quantity at all points over χ, $F : \chi \to T\chi$.

The **Laplacian** is an important operator for learning on non-euclidean domains, because its eigenfunctions are used to generalise the classical fourier bases in spectral analyses. The Laplacian is defined as the divergence ($\nabla \cdot F(x)$) of the gradient ($\nabla f(x)$) of a function on a Euclidean space,

$$\Delta f(x) = -\nabla \cdot \nabla f(x). \tag{2.9.7}$$

In non-Euclidean space, the Laplacian is similar but uses the **intrinsic gradient** - the direction of greatest change in slope as represented by a tangent vector, $\nabla f : L^2(\chi) \to L^2(T\chi)$, and the **intrinsic divergence** - the density of outward flux of F from an infinitesimal sphere about tangent vector fields, $\nabla \cdot F : L^2(T\chi) \to L^2(\chi)$. In otherwords, the Laplacian is the difference between the average function on an infinitesimal sphere around a point and the value of the function at the point itself. On a manifold, the eigendecomposition of the laplacian is,

$$\Delta_\chi \phi_k = \lambda_k \phi_k \tag{2.9.8}$$

where λ_k are the eigenvalues and ϕ_k are the eigenvectors. Additionally the non-Euclidean Laplacian has the properties that it is **intrinsic** so can be expressed solely in terms of the Riemannian metric, it is **isometry invariant**, so it doesn't change with the deformation of the manifold and it's **positive semidefinite**.

A function f on χ can be written as a linear combination of

the eigenvectors,

$$\begin{aligned} f(x) &= \sum_{i \geq 0} \langle f, \phi_i \rangle_{L^2(\chi)} \phi_i(x) \\ &= \sum_{i \geq 0} \hat{f}_i \phi_i(x) \end{aligned} \tag{2.9.9}$$

with $\hat{f}$ serving as the coordinates of the eigenbasis. Here $\hat{f}$ is the forward fourier transform of f. In Euclidean space, the convolution of two functions in the spectral domain is the element-wise product of their Fourier transforms,

$$\widehat{(f * g)}(\omega) = \hat{f}(\omega) \cdot \hat{g}(\omega). \tag{2.9.10}$$

In non-euclidean space, the operation $x - x'$ cannot be defined on the manifold, so an equivalent transformation cannot be made, however if Equation 2.9.10 is taken as a definition, then we can write,

$$(f * g)(x) = \sum_{k \geq 0} \langle f, \phi_k \rangle_{L^2(\chi)} \langle g, \phi_k \rangle_{L^2(\chi)} \phi_k(x). \tag{2.9.11}$$

It's clear that the spectral convolution in non-Euclidean space is basis dependent. This means a filter applied in this domain is basis dependent and will not generalise across domains.

A basis independent, non-euclidean convolution can be obtained by construction of a local system of geodesic polar coordinates on a manifold,

$$\boldsymbol{u}(x, x') = (p(x, x'), \theta(x, x')) \tag{2.9.12}$$

where p is the radial component that describes the geodesic distance from x to x' and θ is the angular component that describes the direction from x to x'. Additionally, their locally defined weights can be written,

$$w_l(\boldsymbol{u}(x, x')) = (v_p(x, x'), v_\theta(x, x')) \tag{2.9.13}$$

Domain	Euclidean	Non-Euclidean
Spatial	$(f * g)(x) = \int f(x')g(x - x')dx'$	$(f * g)(x) = \int (D(x)f)(\boldsymbol{u})g(\boldsymbol{u})d\boldsymbol{u}$
Spectral	$\widehat{(f * g)}(\omega) = \hat{f}(\omega) \cdot \hat{g}(\omega)$	$\widehat{(f * g)}(\omega)_k = \langle f, \phi_k \rangle_{L^2(\chi)} \langle g, \phi_k \rangle_{L^2(\chi)}$

Table 2.1: Spatial and spectral convolutions for Euclidean and non-Euclidean domains

For a function f, the **patch operator** produces a local representation of f around point x,

$$(D(x)f)(\boldsymbol{u}) = \frac{\int_\chi v_p(x, x')v_\theta(x, x')f(x')dx'}{\int_\chi v_p(x, x')v_\theta(x, x')dx'}. \tag{2.9.14}$$

The spatial convolution can then be defined as a filter g applied on local patches,

$$(f * g)(x) = \sum_{p,\theta} (D(x)f)(\boldsymbol{u})g(\boldsymbol{u}). \tag{2.9.15}$$

Table 2.1 summarises the different convolution operations across different spaces and domains.

`https://github.com/MaggieLieu/MLiS_examples/blob/master/Graph_examples.ipynb`

References

Gian Paolo Clemente and Rosanna Grassi. Directed clustering in weighted networks: A new perspective. Chaos Solitons & Fractals, 107, 12 2017. doi: 10.1016/j.chaos.2017.12.007.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. CoRR, abs/1606.09375, 2016. URL http://arxiv.org/abs/1606.09375.

David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, Advances in Neural Information Processing Systems 28, pages 2224–2232. Curran Associates, Inc., 2015. URL http://papers.nips.cc/paper/5954-convolutional-networks-on-graphs-for-learning-molecular-f pdf.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. arXiv e-prints, art. arXiv:1311.2524, Nov 2013.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., volume 2, pages 729–734. IEEE, 2005.

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive

Representation Learning on Large Graphs. arXiv e-prints, art. arXiv:1706.02216, Jun 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. arXiv e-prints, art. arXiv:1512.03385, Dec 2015.

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv e-prints, art. arXiv:1704.04861, Apr 2017.

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv e-prints, art. arXiv:1502.03167, Feb 2015.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907, 2016.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. Neural computation, 1(4):541–551, 1989a.

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, 1998.

Yann LeCun et al. Generalization and network design strategies. In Connectionism in perspective, volume 19. Citeseer, 1989b.

Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network. arXiv e-prints, art. arXiv:1312.4400, Dec 2013.

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. arXiv e-prints, art. arXiv:1512.02325, Dec 2015.

Marcel Nassar. Hierarchical Bipartite Graph Convolution Networks. arXiv e-prints, art. arXiv:1812.03813, Nov 2018.

Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. Distill, 2016. doi: 10.23915/distill.00003. URL `http://distill.pub/2016/deconv-checkerboard`.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. arXiv e-prints, art. arXiv:1506.02640, Jun 2015.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. arXiv e-prints, art. arXiv:1506.01497, Jun 2015.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. arXiv e-prints, art. arXiv:1505.04597, May 2015.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural net-

work model. IEEE Transactions on Neural Networks, 20(1): 61–80, 2008.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15:1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017. URL https://arxiv.org/abs/1611.01578.