An Introduction to Machine Learning

Connor Brown

Contents

1 Machine Learning **3**
- 1.1 Introduction . . . 3
- 1.2 Linear Regression . . . 4
  - 1.2.1 Gradient descent . . . 4
  - 1.2.2 Polynomial Regression . . . 6
- 1.3 Logistic Regression . . . 7
  - 1.3.1 Decision boundary . . . 7
  - 1.3.2 Cost function . . . 8
  - 1.3.3 Multi-class classification . . . 9
  - 1.3.4 Regularisation . . . 9
- 1.4 Neural Networks . . . 11
  - 1.4.1 Forward-propagation . . . 12
  - 1.4.2 Cost function & Back-propagation . . . 13
  - 1.4.3 Putting it all together . . . 14
- 1.5 Advice on applying machine learning algorithms . . . 14
  - 1.5.1 Evaluating a learning algorithm . . . 14
  - 1.5.2 Bias vs. Variance . . . 15
  - 1.5.3 Machine learning system design . . . 17
  - 1.5.4 Receiver Operating Characteristic (ROC) . . . 18
- 1.6 Support Vector Machines . . . 18
  - 1.6.1 Large Margin Classifiers . . . 18
  - 1.6.2 Kernels . . . 19
- 1.7 Unsupervised Learning . . . 21
  - 1.7.1 Clustering: K-means . . . 21
  - 1.7.2 Dimensionality Reduction: PCA . . . 24
- 1.8 Anomaly Detection . . . 27
  - 1.8.1 Multivariate gaussian distributions . . . 29
  - 1.8.2 Recommender Systems . . . 29
- 1.9 Large Scale Machine Learning . . . 32
  - 1.9.1 Gradient Descent . . . 32
  - 1.9.2 Map-reduce and data parallelism . . . 33
- 1.10 Ensemble Learning . . . 34
  - 1.10.1 Boosting . . . 35
  - 1.10.2 Bagging . . . 36
- 1.11 Decision Trees . . . 36
  - 1.11.1 Random Forests . . . 38

2 Deep Learning **39**
- 2.1 Neural Networks and Deep Learning . . . 39
  - 2.1.1 Logistic Regression as a Neural Network . . . 40
  - 2.1.2 Shallow Neural Networks . . . 41
  - 2.1.3 Activation Functions . . . 41
  - 2.1.4 Gradient descent for neural networks . . . 43
  - 2.1.5 Deep Neural Networks . . . 43

2.2 Improving Deep Neural Networks: Hyperparameter tuning, Regularisation and Optimisation . . . 44
2.2.1 Practical aspects of Deep Learning . . . 44
2.2.2 Bias & Variance . . . 44
2.2.3 Regularisation . . . 45
2.2.4 Setting up your optimisation problem . . . 47
2.2.5 Optimisation algorithms . . . 48
2.2.6 Hyperparameter tuning . . . 51
2.2.7 Batch normalisation . . . 52
2.3 Structuring Machine Learning Projects . . . 52
2.3.1 Error analysis . . . 53
2.4 Convolutional Neural Networks . . . 55
2.4.1 Introduction to CNNs . . . 55
2.4.2 Case Studies . . . 58
2.4.3 Network in Network and 1×1 convolutions . . . 60
2.4.4 Inception Network . . . 61
2.4.5 Practical Advice on ConvNets . . . 62
2.4.6 Object Detection . . . 63
2.4.7 YOLO (You Only Look Once) Algorithm . . . 64
2.4.8 Facial Recognition . . . 67
2.4.9 Neural Style Transfer . . . 69
2.5 Sequence Models . . . 72
2.5.1 Recurrent Neural Networks . . . 72
2.5.2 Natural Language Processing and Word Embeddings . . . 79
2.5.3 Sequence Models and Attention Mechanism . . . 79

3 Reinforcement Learning **80**

Chapter 1

Machine Learning

1.1 Introduction

This chapter will investigate the following areas of machine learning:

- supervised learning:
  - linear regression;
  - logistic regression;
  - neural networks;
  - SVMs;
  - decision trees.
- unsupervised learning:
  - K-means clustering;
  - PCA;
  - Anomaly detection.
- applications of machine learning:
  - recommender systems;
  - large scale machine learning.
- advice on building machine learning systems:
  - bias/variance;
  - regularisation;
  - deciding what should be improved in your model: evaluation of learning algorithms, learning curves, error analysis, ceiling analysis.

Supervised learning

Supervised learning is where we give the algorithm *labelled data* to learn from. Supervised learning is generally used for *prediction* (e.g. given labelled training data, predict the price of a house with 3 bedrooms) or *classification* (e.g. given labelled training data, classify a picture as containing a cat or a dog).

Unsupervised learning

Unsupervised learning is where *unlabelled data* is given to the algorithm and we attempt to find *structure* in the data. For example, can we find different groups in our customer base that might be beneficial to target with a new product? The essence of unsupervised learning is *here is some data - find some structure in it.*

1.2 Linear Regression

Here, we seek to use linear regression on multiple features to help us predict some output. For example, we may try to predict the price of a house given its *size*, *number of bedrooms* etc., training our algorithm on the information in Table 1.1.

Size (feet^2)	No. bedrooms	No. floors	Age of home (years)	Price ($\$1,000s$)
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

Table 1.1: Predicting housing prices

Let

- n = number of features;
- m = number of training examples;
- $x^{(i)}$ = input features of i^{th} training example;
- $x_j^{(i)}$ = value of feature j in i^{th} training example;
- θ = parameter vector;
- $h_\theta(x)$ = predicted output, our *hypothesis function*.

We use the convention that $x_0 = 1$. Our hypothesis in linear regression is that

$$h_\theta(x) = \theta^T X \tag{1.1}$$

and we use the least-squares (or **mean squared error**) cost function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{1.2}$$

which we attempt to minimise over θ.

1.2.1 Gradient descent

Gradient descent is a minimisation method. Here, we can use it to minimise our cost function J across our parameter vector θ. Take Figure 1.1, for example. We estimate θ initially at the point marked with the *highest* cross.

If we want to minimise J, then we need to head *downhill*. Gradient descent finds *which direction will take you downhill as quickly as possible*. Of course, depending on your initialisation of θ, gradient descent may take you to a *local minimum* rather than a *global minimum*. Imagine if θ were initialised with a slightly smaller value for θ_1. This would result in a different minimum value for θ. This exemplifies the importance of a good initial guess for θ.

Gradient descent algorithm

The gradient descent algorithm is as follows: *Repeat the following until convergence:*

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \tag{1.3}$$

where θ_j is updated *simultaneously* $\forall j$. We call α the *learning rate* and this is a variable that the user decides on. Intuitively, the larger the value of α, the larger each of our steps downhill is. To gain understanding on why this equation makes sense, imagine a J is quadratic and you are at some point away from the minimum. How will θ change as you run the algorithm? How will the size of α effect the minimisation? If α is too

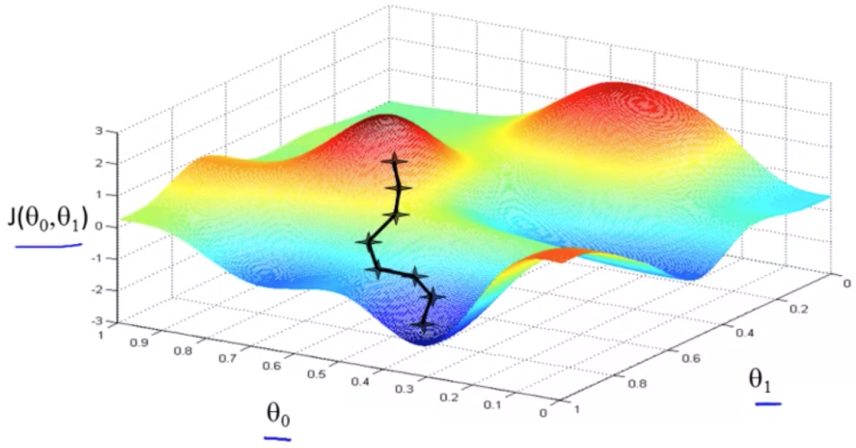

Figure 1.1: Finding the minimum for J across $\theta \in \mathbb{R}^2$

- small then your algorithm may take a long time to run;
- large then your step-size may be large enough to jump over the local minimum.

Note also that the step size automatically decreases as you approach a minimum. There is no need to adjust α across iterations of the algorithm.

We are now set to optimise our parameters θ. All we need to do now is calculate the gradient term in Equation 1.3 for J in Equation 1.2. Using calculus, we calculate the gradient as

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \tag{1.4}$$

Equation 1.4 is referred to as *batch gradient descent*, as it sums over the entire training set for each iteration of the algorithm. In Section 1.9.1 we will investigate other types of gradient descent that don't use the entire training set in every iteration. Our gradient descent algorithm in this case iterates over

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \tag{1.5}$$

where we have left out the $\frac{1}{m}$ term as this will not effect the minimisation anyway.

Feature Normalisation

Before applying gradient descent, it is important to perform *feature normalisation* if our features are on vastly different scales. If you do not perform feature normalisation in this case, your learning algorithm may take a long time to converge to the local optimum for θ.

It is common practice to try and scale the features into approximately the range $-1 \leq x_i \leq 1$. Letting μ be the mean value of our feature vector, one can perform feature normalisation using the following:

$$\frac{x_i - \mu}{\max(x_i) - \min(x_i)} \tag{1.6}$$

You can replace the denominator by σ, the standard deviation of x_i if you prefer.

A good way to choose the value of α is to plot J against *number of iterations* for your gradient descent algorithm for different values of α. You would want to see this curve decreasing monotonically and quickly for increasing number of iterations. It is common to try values of α changing by a factor of 10 e.g. $\alpha = \{..., 0.001, 0.01, 0.1, 1, ...\}$ and see which works best.

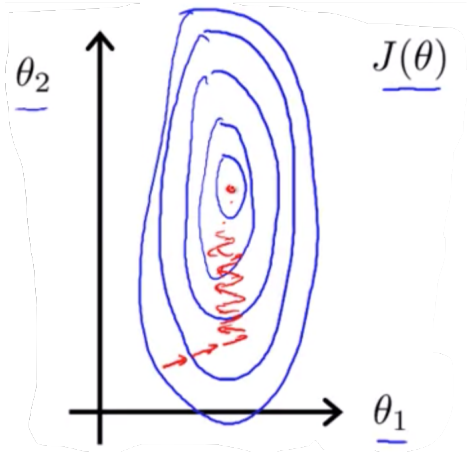

Figure 1.2: Feature normalisation may improve the speed of convergence here

One should note that there are more sophisticated optimisation algorithms that you could use, such as conjugate gradient, BFGS, and L-BFGS. With these algorithms you do not need to manually pick α and they are often faster than gradient descent.

1.2.2 Polynomial Regression

You may be able to improve the performance of your model by combining features you already have to create new ones. For example, say you have the model $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$. Then you could try including features $x_3 = x_1 x_2$ and $x_4 = x_1^2$ to see how this effects performance.

One shouldn't blindly apply this technique, however. You should investigate the data carefully before choosing which features to include. For example, the data seen in Figure 1.3 might be modelled well by $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$. Or perhaps it would be better modelled by $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$. Another possibility would be $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 \sqrt{x}$.

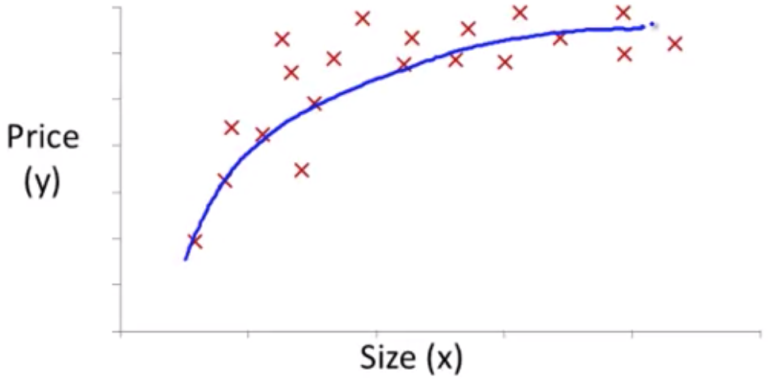

Figure 1.3: Polynomial Regression

Don't forget to apply feature normalisation if necessary here. It is often useful when including polynomial terms in your model.

The Normal Equation

In the case of least-squares regression, it is possible to solve for θ analytically using Equation 1.7.

$$\theta = (X^T X)^{-1} X^T y \tag{1.7}$$

This is useful in some cases, however gradient descent is much more generic. As long as you can estimate J and $\frac{\partial}{\partial\theta}J$. Feature scaling isn't necessary when using the normal equations (not proved here).

The problem with Equation 1.7 is that you need to calculate the term $(X^TX)^{-1}$ which can be very slow if the number of features n is large. This is an $O(n^3)$ operation. There is also the chance that this term is non-invertible. In this case you may have

- redundant features (linearly dependent) e.g. $x_1 = 3x_2 + 0.5$;
- too many features (e.g. $m \leq n$, more features than examples). This can be solved by deleting features or using regularisation.

1.3 Logistic Regression

We can use logistic regression to help us in classification problems. For example,

- is an email spam or not spam?
- is this transaction fraudulent?

In the above examples, we are trying to predict a binary output $y \in \{0, 1\}$, where 0 is *negative* (e.g. not spam) and 1 is *positive* (e.g. a fraudulent transaction). This idea can be extended to *multi-class classification* problems too (e.g. $y \in \{0, 1, 2, 3, ..., 10\}$). We will focus on the binary case for now.

Given $y \in \{0, 1\}$, it makes sense that we want $0 \leq h_\theta(x) \leq 1$. We do this using the *sigmoid function*, seen in Equation 1.8.

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T X}} \tag{1.8}$$

We will denote $g(z) = \frac{1}{1+e^{-z}}$ for simplicity. The sigmoid function can be seen in Figure 1.4.

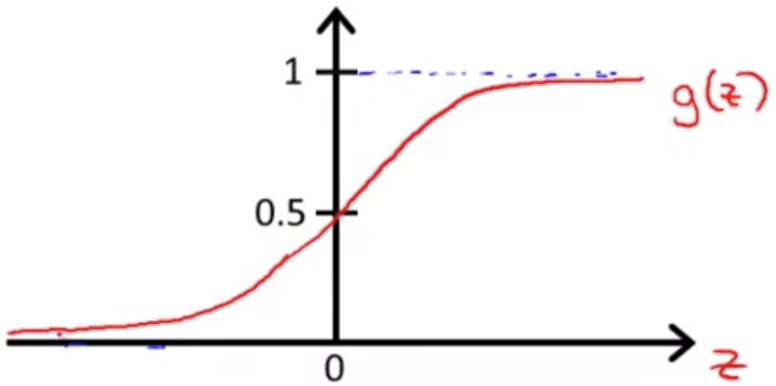

Figure 1.4: Sigmoid function

One can interpret the value of $h_\theta(x)$ as the *probability* that the output is *positive*. That is

$$h_\theta(x) = \mathbb{P}(y = 1 \mid x; \theta). \tag{1.9}$$

1.3.1 Decision boundary

A *decision boundary* is the value of $h_\theta(x)$ above which you will predict $y = 1$ (and below which you will predict $y = 0$). Looking at Figure 1.4, a reasonable suggestion for a decision boundary would be at $h_\theta(x) = 0.5$ (i.e. predict $y = 1$ if $\theta^T X \geq 0$).

However, there may be situations in which one wouldn't want to commit to saying the result is positive if it is *just* above the decision boundary (e.g. telling someone whether they have a disease or not when $h_\theta(x) = 0.51$). An alternative to this is having some kind of region in which you say

you are *unsure* and will not commit to predicting the result. For example, for $0.3 \leq h_\theta(x) \leq 0.7$ you could say that you are *unsure* of the result, for $h_\theta(x) > 0.7$ you are confident the output is *positive*, and $h_\theta(x) < 0.3$ you are confident the output is *negative*.

Non-linear decision boundaries

Sometimes we may need a non-linear hypothesis function h in order to capture the decision boundary. For example, the data seen in Figure 1.5 may require $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$.

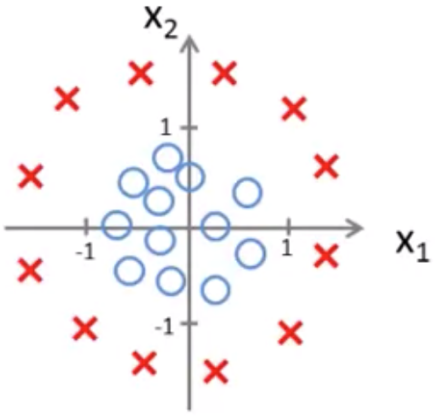

Figure 1.5: A situation where we need a non-linear decision boundary

Adding more polynomial terms to our hypothesis will result in more complicated decision boundaries. We will see later in Section 1.6 how Support Vector Machines can be used to build these boundaries also.

1.3.2 Cost function

We shall use the cost function seen in Equation 1.10 for our logistic regression model. This is commonly known as the **cross-entropy** cost function.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} \log(h_\theta(x^{(i)}))y^{(i)} + \log(1 - h_\theta(x^{(i)}))(1 - y^{(i)}) \tag{1.10}$$

We do not use Equation 1.2 as our cost function in the case of logistic regression as for the hypothesis function $h_\theta(x) = g(\theta^T X)$ this equation is *non-convex* (i.e. you cannot guarantee that we will be able to find the global minimum for J).

Equation 1.10 makes sense for our problem. If $y = 1$, then only the first term in the summation is used which has low cost for large $h_\theta(x)$ (i.e. we have predicted there is a high probability that $y = 1$). Similarly for $y = 0$ we will penalise highly for large $h_\theta(x)$. Equation 1.10 can be derived using the principal of maximum likelihood estimators. That is,

$$\begin{aligned}
\mathbb{P}(y^{(i)} \mid x^{(i)}; \theta) &= h_\theta(x^{(i)})^{y^{(i)}}(1 - h_\theta(x^{(i)}))^{1-y^{(i)}} \\
\mathcal{L}(\theta) &= \prod_{i=1}^{m} h_\theta(x^{(i)})^{y^{(i)}}(1 - h_\theta(x^{(i)}))^{1-y^{(i)}} \\
\ell(\theta) &= \log(\mathcal{L}(\theta)) \\
&= \sum_{i=1}^{m} \log(h_\theta(x^{(i)}))y^{(i)} + \log(1 - h_\theta(x^{(i)}))(1 - y^{(i)})
\end{aligned}$$

adding the minus sign to make the cost positive (i.e. so that we *minimise* the cost in our algorithm). We have added the $\frac{1}{m}$ term to simplify Equation 1.11.

Again, we can use gradient descent to minimise the cost function: *Iterate over*

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}. \tag{1.11}$$

Feature normalisation can also be used in logistic regression algorithms to help gradient descent converge faster.

1.3.3 Multi-class classification

Here we will look at the *one-vs-all* algorithm where we aim to build a classification model with $n \geq 3$ classes. In the one-vs-all algorithm, we take each class in turn and create a decision boundary of that class against the rest of the data points, like in Figure 1.6.

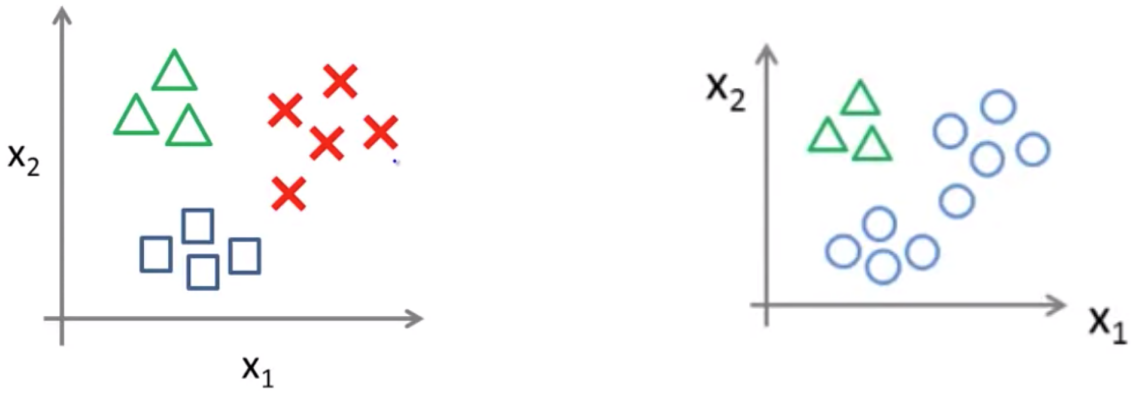

Figure 1.6: Multi-class classification (left) and one-vs-all algorithm (right)

Running this for each class, we will have a classifier $h_\theta^{(i)}(x)$ for each class i to predict $\mathbb{P}(y = i \mid x; \theta)$. For a given example, we will pick the class i that maximises $h_\theta^{(i)}(x)$ $\forall i$.

1.3.4 Regularisation

Overfitting is a common problem in machine learning algorithms. We shall look at using regularisation to reduce the amount of overfitting in our model. If we have too many features in our model then we may fit the training set very well, but fail to generalise to new examples.

Bias and Variance

Some useful terminology that we will see going forward:

- *Bias*: the extent to which we *underfit* the data.
  - *High bias* $\iff$ we have *underfit* the data;
  - *Low bias* $\iff$ we have *overfit* the data.
- *Variance*: the extent to which we *overfit* the data.
  - *High variance* $\iff$ we have *overfit* the data;
  - *Low variance* $\iff$ we have *underfit* the data.

More specifically, *bias* is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (hence underfitting). Conversely, *variance* is an error from sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (hence overfitting).

Overfitting

To address overfitting, we can

1. reduce the number of features (see here for an interesting python implementation of feature selection);
    - select which features to keep e.g. model selection algorithms.
2. use *regularisation*.
    - keep all of the features but reduce some of the features' magnitude.

Regularisation means our hypothesis is simpler and easier to interpret.

Regularised Linear Regression

In linear regression, we implement regularisation in our cost function by

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{i=1}^{n}\theta_j^2\right], \tag{1.12}$$

where λ is our *regularisation parameter* which controls the trade-off between the goal of fitting the training set well (i.e. the first summation term) and the goal of regularising our parameters (i.e. the second summation term). Note that in Equation 1.12, we do not regularise θ_0. This is by convention and in practice makes little difference to performance of the algorithm.

Extending this, we can then easily implement gradient descent as before

$$\theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)} + \frac{\lambda}{m}\theta_j\mathbb{I}\{j > 0\}\right] \tag{1.13}$$

Regularised Logistic Regression

We now look at applying regularisation to logistic regression. Our new cost function is

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\log(h_\theta(x^{(i)}))y^{(i)} + \log(1 - h_\theta(x^{(i)}))(1 - y^{(i)}) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{1.14}$$

Extending this, we can then easily implement gradient descent (which is the same as in Equation 1.13 but with a different definition for $h_\theta(x)$)

$$\theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)} + \frac{\lambda}{m}\theta_j\mathbb{I}\{j > 0\}\right] \tag{1.15}$$

LASSO Regression

LASSO regression extends on the idea of regularisation in linear regression. LASSO regression has cost function

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\|\theta\|_1\right], \tag{1.16}$$

i.e. we use the $\mathcal{L}_1$-norm here ($\|\theta\|_1 = |\theta_0| + ... + |\theta_n|$). Technically, the regularisation seen in Equation 1.12 is called *Ridge regression* (which uses the $\mathcal{L}_2$-norm squared). Lasso regression *suddenly* shrinks parameter values to zero with increasing λ whereas ridge regression *gradually* shrinks parameters *towards* zero with increasing λ, as seen in Figure 1.7.

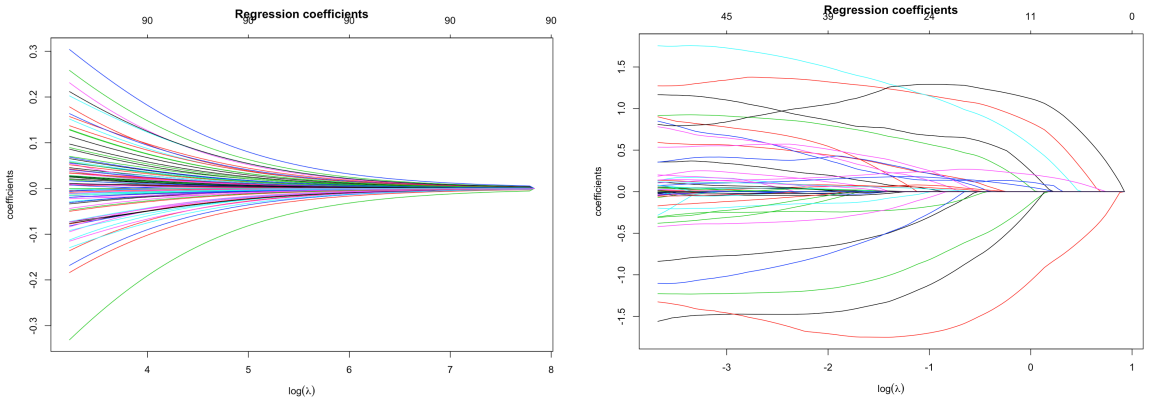

Figure 1.7: Ridge regression (left) and LASSO regression (right)

1.4 Neural Networks

Neural networks are a great way to learn complex hypotheses even when you have a large number of features. Here we will focus mainly on applying neural networks to the problem of classification.

Before diving into the theory, let's first discuss some terminology that will be useful to us going forward. There are many similarities between the logistic/linear regression model and neural networks.

- **weights**: the parameter values of our network θ;
- **input values**: alias for our x terms;
- **bias unit**: alias for our intercept term $x_0 = 1$;
- **output values**: alias for the output of $h_\theta(x)$ given *input values* and the network's *weights*;
- **neuron**: points in our network i.e. the circles seen in Figure 1.8;
- **activation function**: the function g applied to the weights and input values from a given layer to give the output values of that layer.

The above terminology will become clearer as you continue reading.

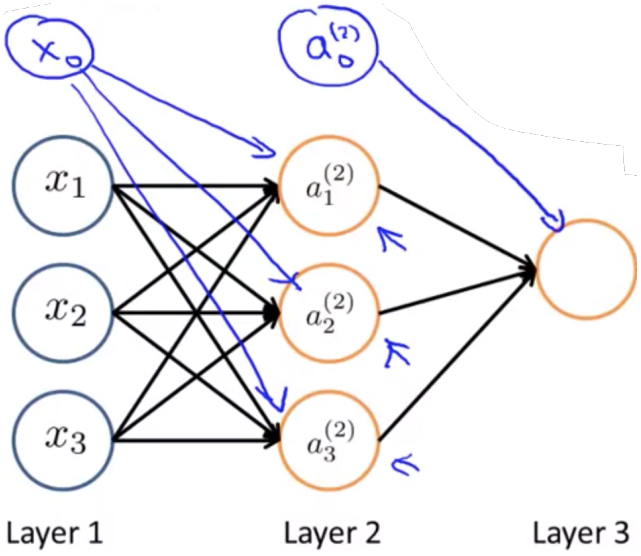

Figure 1.8: Example neural network with 3 layers

Note that in Figure 1.8 bias units are added to each layer (excluding the *output layer*, layer 3) and have the value $x_0 = 1$. Layer 1 is referred to as the *input layer*. If a layer is not an input layer nor an output layer then it is referred to as a *hidden layer*.

1.4.1 Forward-propagation

We shall use the following notation in our work.

- $a_i^{(j)}$ = activation of unit i in layer j;
- $\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$.

Common activation functions include *sigmoid*, *tanh*, *softmax* and *ReLU*, each of which have their own advantages (see here for sigmoid/*tanh* vs. ReLU). Before moving on, let's be explicit in what calculations are being done in Figure 1.8.

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3) \end{aligned} \tag{1.17}$$

$$h_\theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}x_0 + \Theta_{11}^{(2)}x_1 + \Theta_{12}^{(2)}x_2 + \Theta_{13}^{(2)}x_3) \tag{1.18}$$

Note that if our network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(j)} \in \mathbb{R}^{s_{j+1} \times (s_j+1)}$ i.e. we are adding a bias unit to layer j. We denote

$$z_i^{(j)} = \Theta_{i0}^{(j)}x_0 + \Theta_{i1}^{(j)}x_1 + ... + \Theta_{in}^{(j)}x_n. \tag{1.19}$$

That is, $a_i^{(j)} = g(z_i^{(j)})$. We can vectorise Equations 1.17 by

$$\begin{aligned} z^{(j+1)} &= \Theta^{(j)}a^{(j)} \\ a^{(j+1)} &= g(z^{(j+1)}) \end{aligned} \tag{1.20}$$

where $a^{(j+1)}, z^{(j+1)} \in \mathbb{R}^{s_{j+1}}$ and $a^{(1)} = x$. If $a^{(j+1)}$ is not in an output layer, then we must add a bias unit to it, resulting in $a^{(j+1)} \in \mathbb{R}^{s_{j+1}+1}$. The *output value* of the network in Figure 1.8 is $a^{(3)}$.

The above algorithm is called **forward propagation** since we propagate forward through the network. From what we've seen so far, neural networks simply run many regressions in each layer to give the activation units of the next layer. We learn the parameter matrix Θ for each layer in order to train our network.

As we shall see below, having many hidden layers allows us to learn much more complex, non-linear features than we were able to learn in logistic/linear regression. The features are complex because you have layers taking as inputs the activation units of the previous layer. Say if you were looking at the 100^{th} hidden layer of a neural network, then the input values for that layer will have been through 99 hidden layers! That is potentially a highly complicated transformation from the network's input values all the way to the 100^{th} layer.

Multi-class classification

Figure 1.8 is an example of single-class classification. There is only one neuron in the output layer. We can have multiple neurons in the output layer to create a multi-class classification network, with each neuron being an indicator for a certain class. An example neural network with 4 possible outputs can be seen in Figure 1.9.

One would could then predict the class of a given example as the output neuron with the highest value i.e. a binary encoding for each neuron. Another common encoding method seen in machine learning is one-hot encoding.

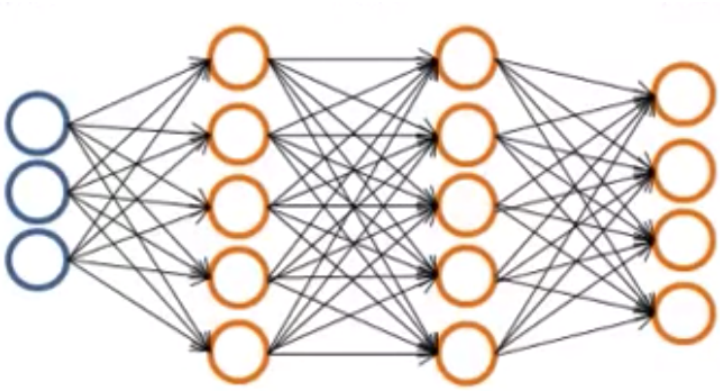

Figure 1.9: Example neural network with 4 outputs

1.4.2 Cost function & Back-propagation

We shall carry on focusing on neural networks applied to classification problems. Let us denote

- $\{(x^{(1)}, y^{(m)}), ..., (x^{(m)}, y^{(m)})\}$ as our training set;
- L as the total number of layers in the network;
- s_l as the number of units (excluding the bias unit) in layer l.

We shall use a generalisation of the cost function that we used for logistic regression, seen in Equation 1.14. Given a neural network with $s_L = K$ (i.e. a multi-class classification problem) where $h_\Theta(x) \in \mathbb{R}^K$, $(h_\Theta(x))_i = i^{th}$ output, we define the cost function as

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\log(h_\Theta(x^{(i)}))_k y_k^{(i)} + \log(1 - h_\Theta(x^{(i)})_k)(1 - y_k^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2 \quad (1.21)$$

We can minimise the cost function using the **back-propagation** algorithm. Similarly to in gradient descent, we need both $J(\Theta)$ and $\frac{\partial}{\partial\Theta}J(\Theta)$. Note that here, Θ is in fact the collection of matrices $\Theta^{(1)}, ..., \Theta^{(L-1)}$.

Say we have just completed one run of the forward propagation algorithm to compute what our hypothesis actually predicts. Now, we shall work backwards through our algorithm "correcting" the errors that our original hypothesis made. For our output layer, we calculate the vector of errors

$$\delta^{(L)} = a^{(L)} - y. \quad (1.22)$$

In the remaining layers, we calculate

$$\delta^{(l)} = (\Theta^{(l+1)})^T \delta^{(l+1)} .* g'(z^{(l)}) \quad (1.23)$$

where ".*" denotes element-wise multiplication. In the case of logistic regression,

$$g'(z^{(l)}) = a^{(l)}(1 - a^{(l)}). \quad (1.24)$$

Of course, there is no $\delta^{(1)}$ term as you cannot have an error for input values. We now have all of the building blocks for the back-propagation algorithm:

1. set $\Delta_{ij}^{(l)} = 0 \ \forall l, i, j$;
2. for $i = 1$ to m:
   - perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, ..., L$;
   - compute $\delta^{(l)}$ for $l = L, ..., 3, 2$ using Equations 1.22 and 1.23;

- set $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$
- calculate

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \frac{1}{m}\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \mathbb{I}\{j \neq 0\} \tag{1.25}$$

You can check that your value of $\frac{\partial}{\partial \Theta} J(\Theta)$ is approximately correct by comparing it to the two-sided difference

$$\frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}. \tag{1.26}$$

Random initialisation

We need to pick some initial value for Θ. If we set Θ to the zero matrix, then we will encounter problems. Consider forward-propagation from the input layer to the next layer. All of our weights are the same and so each of our activation units in the second layer $a^{(2)}$ will be the same too. This will carry on throughout the entire matrix and consequently all of our δ values in each layer of our back-propagation algorithm will be the same too. This means that our updated values of our parameters $\Theta_{ij}^{(l)}$ will also be the same for each l. The above process will then carry on for all iterations of our back-propagation algorithm. We can't exactly compute any interesting functions in this way.

Instead what we will do is initialise $\Theta_{ij}^{(l)} \in [-\epsilon, \epsilon]$ where ϵ is some small random number. This mitigates the problem described above. We have performed *symmetry breaking*.

1.4.3 Putting it all together

The first thing we have to do when building a neural network is decide on the network architecture i.e. how many layers and how many units in each layer? The number of input units (i.e. dimension of $x^{(i)}$) and number of output units (i.e. number of classes) is set. A reasonable starting point is to either begin with one hidden layer or, if using more than one hidden layer, to have the same number of units in each of these layers (usually the more the better). We shall visit this in more detail in Section 1.5.

Training a neural network is done as follows:

1. Randomly initialise weights;
2. implement forward-propagation algorithm to get $h_\Theta(x^{(i)})$ $\forall x^{(i)}$;
3. compute the cost function $J(\Theta)$;
4. implement back-propagation algorithm to compute the partial derivatives $\frac{\partial}{\partial_{jk}^{(l)}} J(\Theta)$;
5. use an optimisation algorithm (e.g. gradient descent) with back-propagation to try and minimise $J(\Theta)$ as a function of parameters Θ.

1.5 Advice on applying machine learning algorithms

Here we shall consider some practical guidelines on developing machine learning models and what to focus on when building them.

1.5.1 Evaluating a learning algorithm

Suppose your learning algorithm is performing poorly. What could you possibly try?

1. get more training examples;
2. try a smaller set of features to avoid overfitting;
3. try getting more features to add more information to your model;

4. try adding polynomial features;

5. try increasing/decreasing the regularisation parameter λ.

and so on. We can use diagnostic tests to gauge what isn't working in our algorithm and which of the strategies above might be most fruitful in improving performance.

The first thing to do is investigate the performance errors. When investigating the performance of our algorithm, we should use a training, validation, and test set.

Why do we need a validation set as well as a test set? If we were just to use a test set to evaluate the error of our model, then we would likely choose the model with the smallest test error. This is problematic as there is a chance that this is an optimistic estimate of the error and our model has performed coincidentally well on this specific test set. To address this problem we use a cross-validation set. We use the cross-validation error to *select* which model we want to use and subsequently evaluate performance using the test error.

1.5.2 Bias vs. Variance

Two of the most common reasons for poor performance are **high bias** (i.e. underfitting) and **high variance** (i.e. overfitting).

One method to diagnose this is to plot the training and validation/test errors against increasingly complex models. See Figure 1.10 for an example where we are increasing the order of the polynomial in our hypothesis.

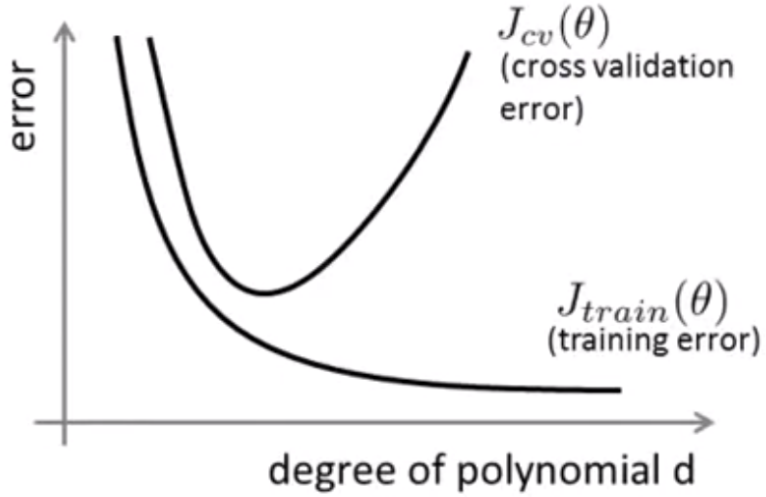

Figure 1.10: Diagnosing high bias and high variance

In Figure 1.10, we see that for

- high bias we have J_{train} large and $J_{train} \approx J_{cv}$;
- high variance we have J_{train} small and $J_{train} \ll J_{cv}$.

We could also draw a similar plot to Figure 1.10 but with our regularisation parameter λ on the x-axis. However now we have

- high bias when λ is large; J_{train} large and $J_{train} \approx J_{cv}$;
- high variance when λ is small; J_{train} small and $J_{train} \ll J_{cv}$.

Learning Curves

Learning curves are useful in determining whether gathering more data will in fact help your model performance. Figure 1.11 is an example of fitting a quadratic hypothesis while adding more data to your training set.

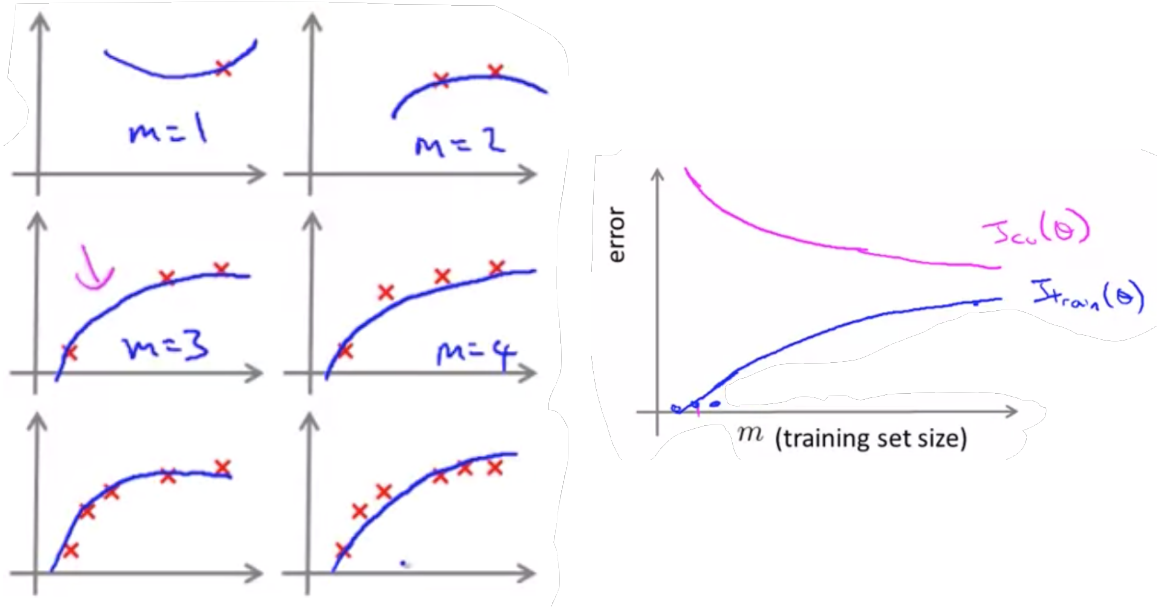

Figure 1.11: An example learning curve

As you can see, adding more examples to the training set, J_{train} becomes larger and J_{cv} becomes comparable to J_{train}. Let's look at some example learning curves. First, Figure 1.12 shows what a learning curve might look like if your model is suffering from high bias.

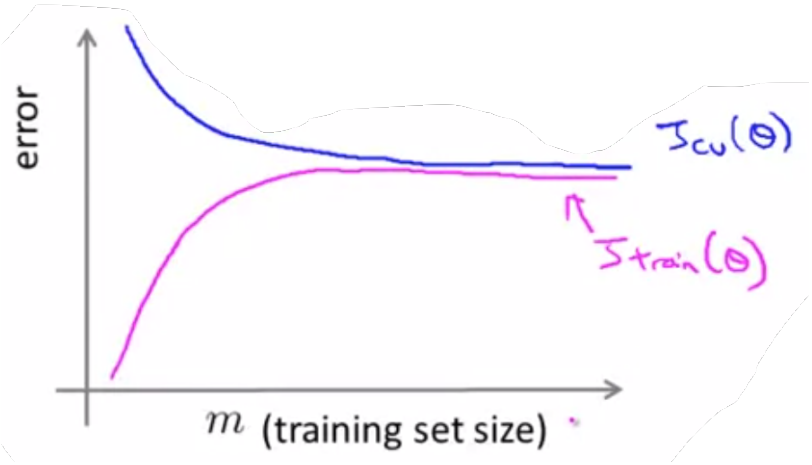

Figure 1.12: Learning curve showing high bias

It is clear here that getting more data is unlikely to help our model perform better. Note that if a model is suffering from high bias then J_{train} and J_{cv} are usually *large*. Figure 1.13 shows what a learning curve might look like if your model is suffering from high variance. Note the large gap between J_{train} and J_{cv}. In the case of high variance, gathering more data will likely improve model performance.

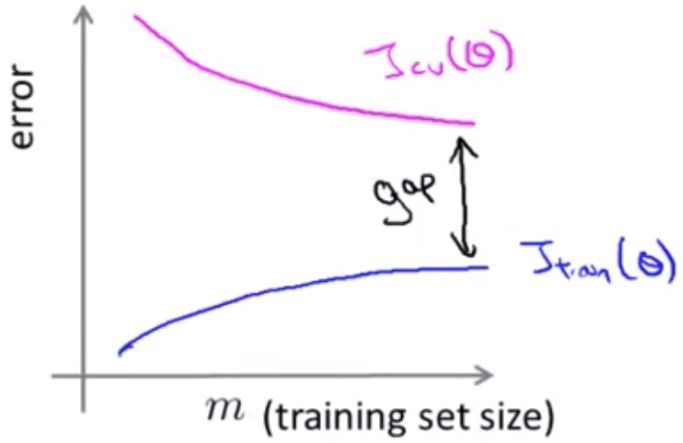

Figure 1.13: Learning curve showing high variance

Summary

To summarise this section, let's revisit our initial list ways to improve model performance and state under which circumstances applying them might be fruitful:

1. get more training examples → fixes **high variance**;
2. try a smaller set of features to avoid overfitting → fixes **high variance**;
3. try getting more features to add more information to your model → fixes **high bias**;
4. try adding polynomial features → fixes **high bias**;
5. try increasing/decreasing the regularisation parameter $\lambda \rightarrow$ fixes **high variance/bias**, respectively.

Finally, let's revisit the problem of choosing a neural network's architecture. A *small* neural network (few neurons and layers) is prone to underfitting. A *large* neural network (potentially many neurons and layers) is prone to overfitting. Using regularisation on a large neural network will help to address the problem of overfitting. This is often a more fruitful approach than simply using a small neural network.

1.5.3 Machine learning system design

To finish this section, we shall discuss the importance of having a **single-value performance** metric for our algorithm as well as the danger of using **skewed data**.

Using a single-value to evaluate the performance of our algorithm means that we can easily rank them and decide which one performs *best*. This is also beneficial when evaluating whether a change in our algorithm was actually useful or not. Feedback is quick and simple.

What is skewed data? This is where there are many more instances of one type of data than the others (e.g. 99% of our data is *positive* and only 1% of our data is *negative*). Why is this bad? If our data is in fact 99% *positive*. We could simply make an algorithm that predicts *positive* always and we would have an accuracy of 99%. Despite this being a naïve algorithm, it has done well, and will likely do better than a lot more sophisticated algorithms. An example of some useful single-value metrics in this case is **precision**

$$\text{Precision} = \frac{TP}{TP + FP} \tag{1.27}$$

(i.e. of all positively predicted records, what fraction were actually positive?) and **recall**

$$\text{Recall} = \frac{TP}{TP + FN} \tag{1.28}$$

(i.e. of all positive records, what fraction did we correctly predict positive?) where If our algorithm

		Actual	
		Positive	Negative
Predicted	Positive	*True Positive* (TP)	*False Positive* (FP)
	Negative	*False Negative* (FN)	*True Negative* (TN)

has both high precision and recall then we can be confident that it is performing well even in the presence of skewed data. Sometimes you may not want both precision and recall to be high. Suppose we want to

- predict $y = 1$ only if very confident (e.g. if someone has a disease or not). Then we will want a *higher recall and lower precision*;
- avoid missing too many cases of the disease (i.e. avoid false negatives). Then we will want *higher recall and lower precision.*

Essentially, this is the same as changing the threshold at which we predict a positive event. That is, $h_\theta(x) >$ threshold, where threshold doesn't have to be 0.5.

One can compare different precision (P)/recall (R) values by putting them into a single value called the F-score:

$$F = \frac{2PR}{P+R}. \tag{1.29}$$

1.5.4 Receiver Operating Characteristic (ROC)

A widely used tool to assess classification error is the Receiver Operating Characteristic (ROC) curve. The ROC plots the TP rate (TPR) against the FP rate (FPR). The ROC curve has certain characteristics:

- Must pass through point $(0, 0)$ and $(1, 1)$;
- The best model has a ROC curve that passes through $(0, 1)$ since this is the case when there are no errors of either type;
- A model that has no discriminatory power is such that $TPR = FPR$. This is represented by a straight line from $(0, 0)$ to $(1, 1)$;

We can compare models using the ROC curve; the ROC curve with the greatest area under the curve (AUC) represents the better model and the one with more predictive power. An AUC= 0.5 corresponds to a model with no classification power (i.e. when $TPR = FPR$). AUC= 1 corresponds to a model with perfect classification power. Figure 1.14 shows an example ROC curve.

1.6 Support Vector Machines

1.6.1 Large Margin Classifiers

SVMs are used widely as supervised classification algorithms to learn complex, non-linear functions. Previously, we have used the *sigmoid* function as the hypothesis for our classification problems. Now, we are going to use the rectified linear unit (ReLU) function. The ReLU function has many benefits e.g. more efficient to compute, avoids vanishing gradient problem that sigmoid/$tanh$ suffer from as their extremes result are relatively flat, making the gradient close to zero, resulting in slow progress of gradient descent.
Figure 1.15 shows the ReLU function against "$-\log(h_\theta(x))$" (left) and "$-\log(1 - h_\theta(x))$" (right). The ReLU function is a piecewise-linear approximation of the log-sigmoid function. Substituting the ReLU function into the cost function we used for logistic regression (see Equation 1.10), we get

$$J(\theta) = C\sum_{i=1}^{m} y^{(i)} ReLU(-\theta^T x^{(i)}) + (1 - y^{(i)}) ReLU(\theta^T x^{(i)}) + \frac{1}{2}\sum_{j=0}^{n} \theta_j^2 \tag{1.30}$$

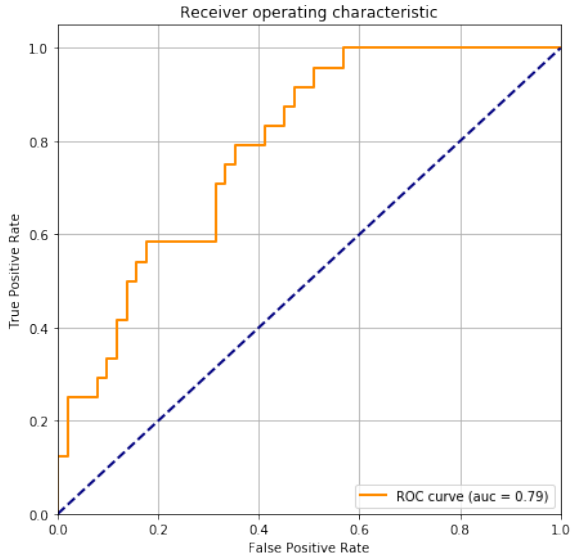

Figure 1.14: Example ROC curve

where we have got rid of the $\frac{1}{m}$ terms as they don't effect the optimisation (and this is the convention for SVMs), and are regularising across all elements of θ. Note that we have used C instead of λ, again because of convention. They play a similar role to one another. Hence, for **large** C (small λ) we will do little regularisation (i.e. prone to overfitting, low bias and high variance). For **small** C (large λ) we will perform heavy regularisation (i.e. prone to underfitting, high bias and low variance). Rather than having a probabilistic output, the hypothesis for a SVM instead gives the direct output

$$h_\theta(x) = \begin{cases} 1, & \theta^T x \geq 0 \\ 0, & \theta^T x < 0 \end{cases} \tag{1.31}$$

The effect of using ReLU compared to the sigmoid function is that we now require $\theta^T x \geq 1$ to predict a positive value (whereas the sigmoid function only required $\theta^T x \geq 0$). This shows that we must be much more confident in our results to predict a positive result. Similar logic applies to predicting a negative result.

There are three main types of SVM decision boundary: separable, non-separable, and non-linear. The key behind building these decision boundaries is the **large margin classifier**.
When we optimise Equation 1.30, we are in fact finding the line(s) that maximises the distance (or *margin*) between the different classes. An example of this is Figure 1.16.

1.6.2 Kernels

We can adapt support vector machines so that they work on non-linear datasets too. We do this with the use of different *kernels*. Is there a better way to describe the data rather than using a high order polynomial? Instead what we will do is, given x, compute features based on the proximity of x to some landmarks $l^{(1)}, ..., l^{(n)}$. We will do this using the **Gaussian** or **Radial Basis Function** (RBF) kernel

$$f_i = \exp\left(-\frac{\left\|x - l^{(i)}\right\|^2}{2\sigma^2}\right) = \exp\left(-\gamma\left\|x - l^{(i)}\right\|^2\right) \tag{1.32}$$

where f_i is the i^{th} feature of our hypothesis e.g. $h_\theta = \theta_0 + \theta_1 f_1 + ... + \theta_n f_n$. If x is close to $l^{(i)}$, then $f_i \approx 1$. Similarly if x is far from $l^{(i)}$, then $f_i \approx 0$. We can increase/decrease the value of σ in order to be less/more strict on how close x must be to $l^{(i)}$ to have f_i close to 1 or 0. A larger value of σ will cause the features f_i to vary more smoothly, and we are therefore likely to have a higher bias and lower variance (i.e. underfitting). Similarly for smaller values of σ. Look at Figure 1.18 and make sure you understand why the magenta point was classified as $y = 1$ and why the cyan point

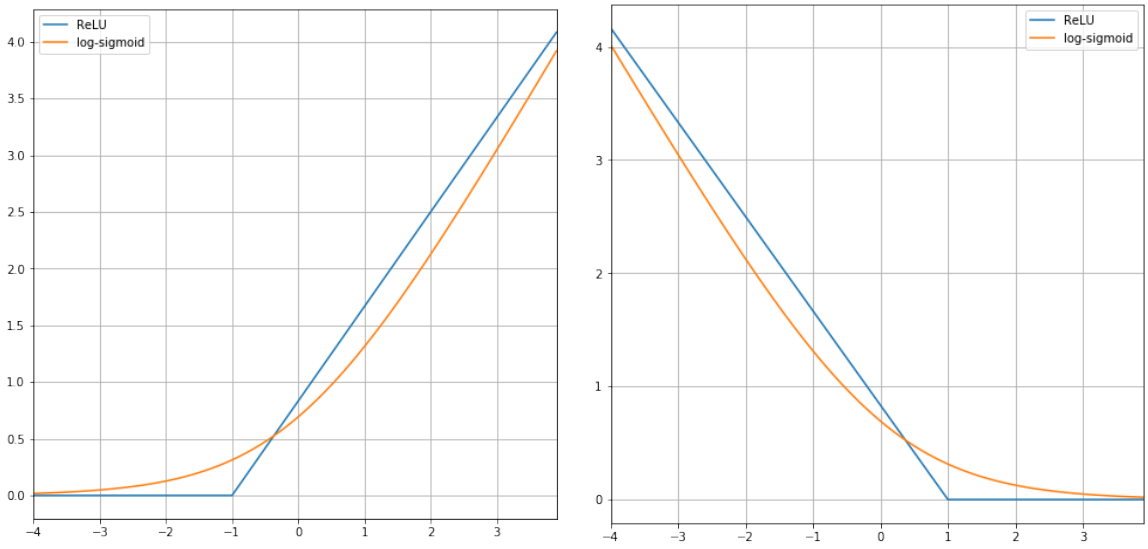

Figure 1.15: ReLU vs. log-sigmoid function

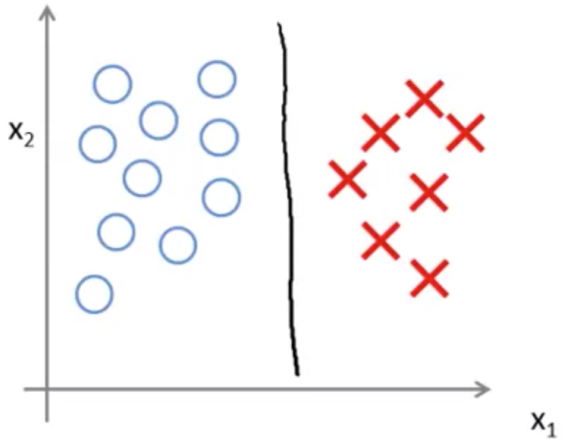

Figure 1.16: Large margin classifier on separable data

was classified as $y = 0$. Given our landmarks, we are therefore able to build decision boundaries wherein we can classify new points.

How do we choose $l^{(i)}$. One method is to choose landmarks equal to $x^{(i)}\ \forall i$. We will then end up with landmarks $l^{(1)}, ..., l^{(m)}$. We can then run each point in our training set through our kernel against each of these landmarks. You can then group the results together into a feature vector $f^{(i)} \in \mathbb{R}^{m+1}$ (i.e. $f_0^{(i)} = 1$) for the training example $(x^{(i)}, y^{(i)})$. We will therefore predict $y = 1$ if $\theta^T f \geq 0$.

Something to note is that often the final term in Equation 1.30 is $\theta^T M \theta$ rather than $\theta^T \theta$. The matrix M is dependent on the chosen kernel and performs some rescaling of the parameters allowing it to scale much better to large datasets in practice.

Finally, it would be worth investigating the other kernels that are commonly used here. You should note that it is important to perform feature-scaling prior to using the gaussian kernel. Looking at the numerator of the kernel, it is clear why this is the case:

$$\|x - l\|^2 = (x_1 - l_1)^2 + (x_2 - l_2)^2 + ... + (x_n - l_n)^2. \tag{1.33}$$

If our features are on vastly different scales, then the value in Equation 1.33 may be skewed massively by one feature in particular.

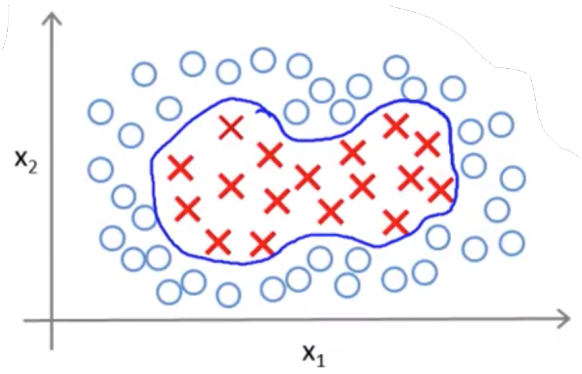

Figure 1.17: Non-linear classification problem

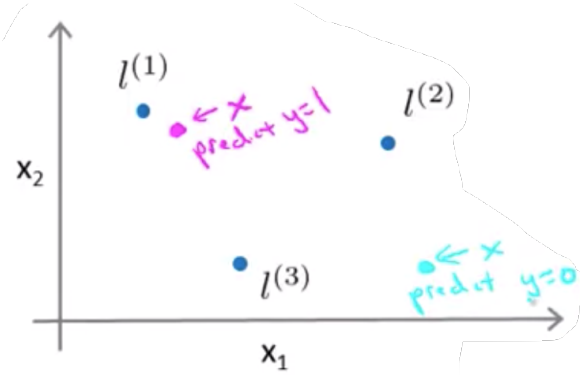

Figure 1.18: Simple example of prediction using Gaussian kernel

One should use **logistic regression** or an **SVM with no kernel** if n is large/small and m is small/large. You should use an **SVM** if n is small and m is intermediate. **Neural networks** are appropriate for any of these situations.

1.7 Unsupervised Learning

Unsupervised learning is where we learn from unlabelled data. We give the unlabelled dataset to an algorithm and ask it to find structure in the data.

1.7.1 Clustering: K-means

Here are some examples of applications of clustering:

- market segmentation;
- social network analysis;
- organise compute clusters;
- analysing astronomical data.

Let's say we take an unlabelled dataset and want to group the data into K sets. First, we randomly initialise K points in the space called the **cluster centroids**. We then perform the following two steps:

1. cluster assignment:
2. move centroid.

Cluster assignment is where the algorithm cycles through the training set and assigns each point a label depending on which cluster centroid the point is closest to. Figure 1.19 shows this for the case $K = 2$.

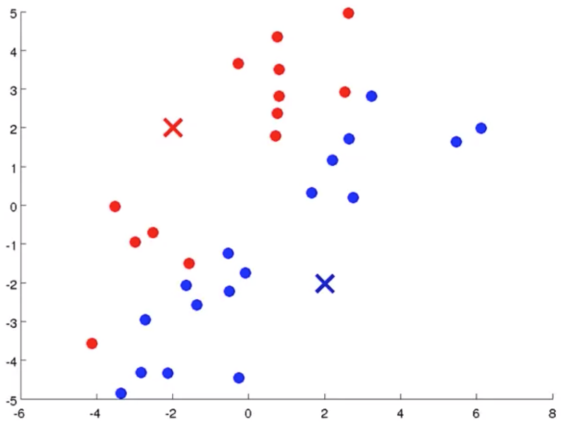

Figure 1.19: cluster-assignment

The *move centroid* step takes each of the cluster centroids and moves them to the average of the points coloured the same colour. Now, we iteratively perform the *cluster assignment* and *move centroid* steps until the algorithm converges. Eventually, Figure 1.19 will look like Figure 1.20.

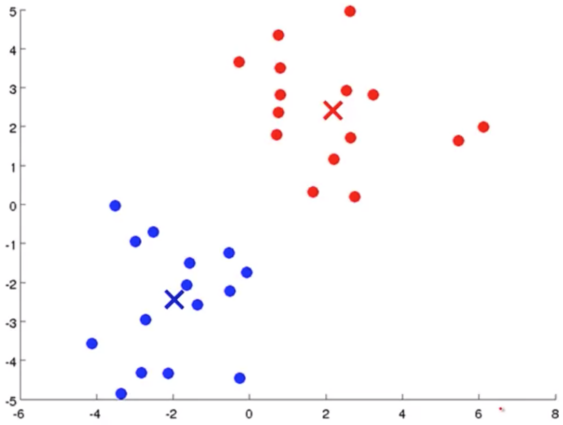

Figure 1.20: Converged k-means

One can choose the value of K in a smart way (which we will look at later). K-means can also be applied to non-separable clusters just as it has been in the separable example above.

Optimisation objective

First, let's specify some useful notation for ourselves going forward.

- $c^{(i)}$ = index of cluster $(1, 2, ..., K)$ to which example $x^{(i)}$ is currently assigned;
- μ_k = cluster centroid k ($\mu_k \in \mathbb{R}^n$, $k \in \{1, 2, ..., K\}$);
- $\mu_{c^{(i)}}$ = cluster centroid of cluster to which example $x^{(i)}$ has been assigned.

The optimisation objective for the K-means algorithm is

$$\min_{c^{(1)},...,c^{(m)},\mu_1,...,\mu_K} J(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K) \tag{1.34}$$

where

$$J(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K) = \frac{1}{m} \sum_{i=1}^{m} \left\| x^{(i)} - \mu_{c^{(i)}} \right\|^2 \tag{1.35}$$

J is commonly referred to the *distortion* of the K-means algorithm. It turns out that *cluster assignment* minimises J over $c^{(1)}, ..., c^{(m)}$, holding $\mu_1, ..., \mu_K$ constant. Similarly, the *move centroid* step minimises J over $\mu_1, ..., \mu_K$, holding $c^{(1)}, ..., c^{(m)}$ constant. It is **not** possible for J to increase; if it does then there is likely a bug in the code.

Randomly initialising cluster centroids

One possible initialisation for the K-means algorithm is randomly picking K training examples and setting $\mu_1, ..., \mu_K$ equal to these K examples. You can run into problems with this method, as exemplified in Figure 1.21

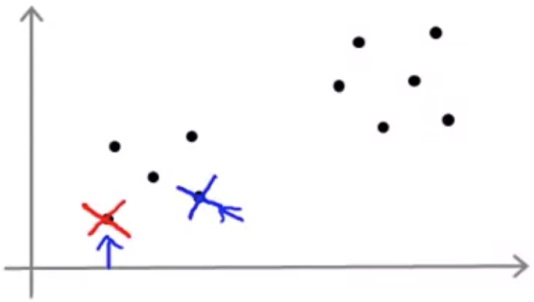

Figure 1.21: Possible failure of initialisation

K-means can, therefore, result in different solutions depending on the initialisation. Specifically, K-means can end up at local optima as seen in Figure 1.22. The local optima, of course, refers to J getting stuck.

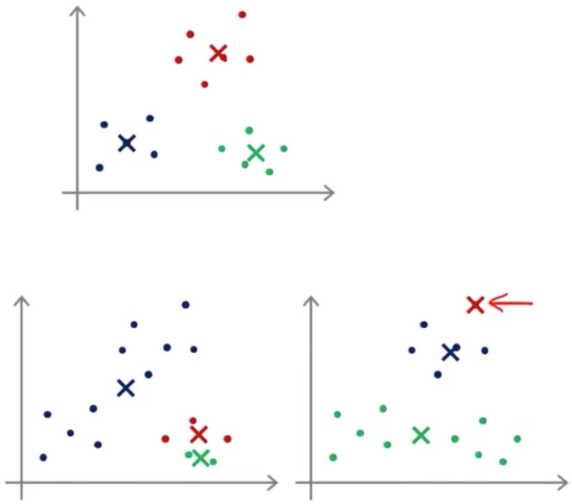

Figure 1.22: More failures

To mitigate the risk of getting stuck in a local optima, we can try multiple random initialisations and pick the clustering that gives the lowest value of our cost function J.

Choosing K

One great way is just to look at the data (if possible) and estimate how many clusters there are! One should bear in mind that there is no *correct* value for K. Another possible way to do this is the *elbow method*, as seen in Figure 1.23. Choosing 3 clusters here is a good choice for K as you gain little by adding more clusters. Often however, one is not so lucky and the decrease in J with increasing K is much more smooth, making it harder to pick K using this method. Finally, there may be some practical value of K that you could use (e.g. if you want to cluster people into S, M, or L clothes, then you would pick $K = 3$).

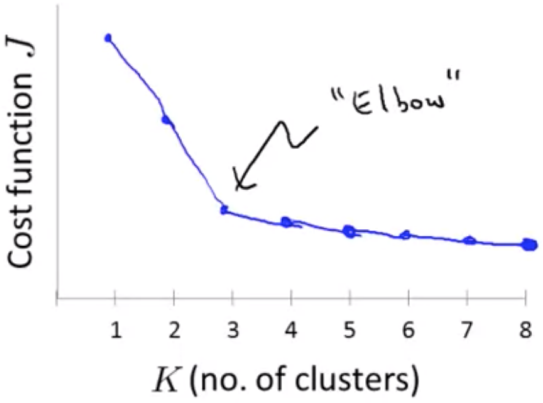

Figure 1.23: Elbow method for choosing K

1.7.2 Dimensionality Reduction: PCA

There are a couple of reasons why one might want to perform dimensionality reduction:

1. data compression;
2. visualisation.

Dimensionality reduction seeks to discard redundant features in your feature space in order to increase interpretability of results, reduce space required for storing data etc. For example, why would you keep a feature describing the length of something in inches as well as a feature describing something in centimetres.

One should note that dimensionality reduction does not necessarily mean that you discard a feature entirely. It is instead better practice to perform some kind of mapping from your old space onto a new one with lower dimension. In the length example above, this would look like

$$(x_1^{(i)}, x_2^{(i)}) \mapsto z^{(i)} \in \mathbb{R}. \tag{1.36}$$

A common problem here is that the meaning of the variable z may be unclear. Discarding a feature altogether runs the risk of one losing information held in that feature.

Principal Component Analysis (PCA)

PCA attempts to find a lower dimensional surface onto which to project the data such that the sum of squares of the distance of the data points onto the surface is minimised. This sum of squares is often referred to as the *projection error*. It is standard practice to perform normalisation before applying PCA (we will discuss this further below).

Formally, if we want to reduce n-dimensional data to k-dimensional data, we seek k vectors $u^{(1)}, ..., u^{(k)}$ onto which to project the data so as to minimise the projection error. The data is projected onto the linear subspace spanned by $u^{(1)}, ..., u^{(k)}$.

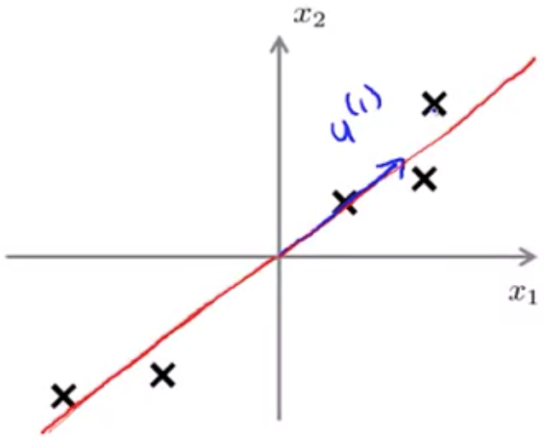

Figure 1.24: Example PCA for $n = 2$, $k = 2$

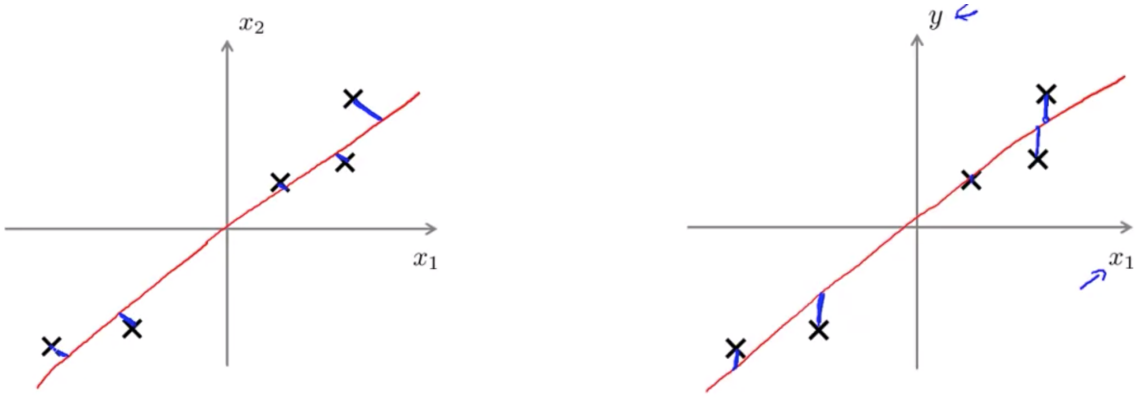

Figure 1.25: PCA (left) v.s. linear regression (right)

Figure 1.24 shows the vector $u^{(1)}$. Note that it doesn't matter if the result is $u^{(1)}$ or $-u^{(1)}$ as the two lines span the same space (i.e. describe the same line).

One should note that PCA is **not** linear regression and they are quite different algorithms. PCA is a *projection* whereas linear regression is for *prediction* This is made clear by Figure 1.25.

PCA Algorithm

Here is the algorithm to compute PCA. Say we wish to reduce some data from n dimensions to k dimensions.

1. Compute the covariance matrix

$$\Sigma = \frac{1}{m}\sum_{i=1}^{n}(x^{(i)})(x^{(i)})^T; \tag{1.37}$$

2. Compute the eigenvectors U of Σ and extract the first k eigenvectors

$$U_{reduce} = \begin{pmatrix} | & & | \\ u^{(1)} & \cdots & u^{(k)} \\ | & & | \end{pmatrix} \in \mathbb{R}^{n\times k}; \tag{1.38}$$

3. Finally, calculate

$$z = U_{reduce}^T x \in \mathbb{R}^k. \tag{1.39}$$

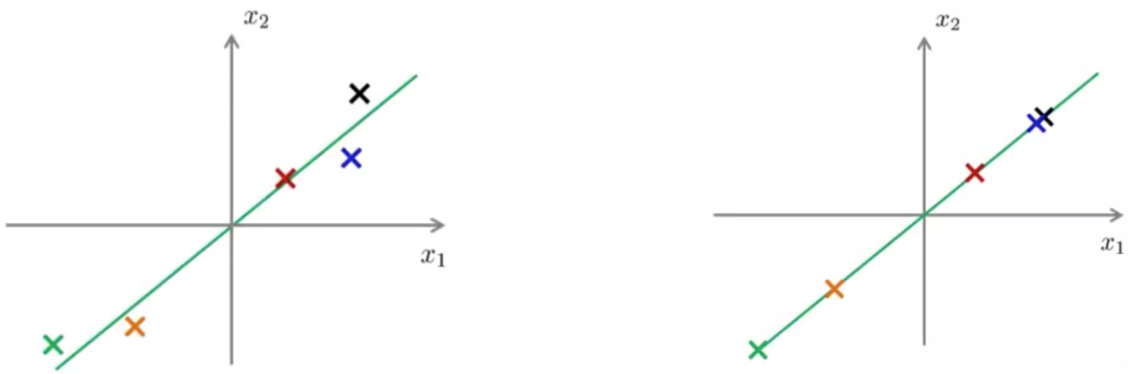

Figure 1.26: x (points left) $\mapsto z \mapsto x_{approx}$ (points right)

You can retrieve an approximation of x from z using

$$x_{approx} = U_{reduce} z \in \mathbb{R}^n \tag{1.40}$$

since U is unitary (i.e. all columns have length 1) $\Rightarrow U^T = U^{-1}$. This is only an approximation of x as we have *projection error* baked into z. The smaller the projection error, the closer x_{approx} is to x. x_{approx} lies on the plane U_{reduce} in $\mathbb{R}^n$, as shown in Figure 1.26.

How to choose the number of principal components, k

PCA attempts to minimise the *average squared projection error*

$$ASPE = \frac{1}{m} \sum_{i=1}^{m} \left\| x^{(i)} - x^{(i)}_{approx} \right\|^2. \tag{1.41}$$

We define the *total variation* in the data to be

$$TV = \frac{1}{m} \sum_{i=1}^{m} \left\| x^{(i)} \right\|^2 \tag{1.42}$$

which describes how far, on average, our points are from the origin.

Typically, one chooses k to be the smallest value such that

$$\frac{ASPE}{TV} \leq 0.01 \tag{1.43}$$

i.e. 99% of the variance is retained. More generally, one chooses k such that α% of the variance is retained. There is, however, a much simpler way of calculating the left-hand side of Equation 1.43. The singular value decomposition of Σ is

$$\Sigma = USV^T \tag{1.44}$$

where

$$S = \begin{pmatrix} s_{11} & 0 & \dots & 0 \\ 0 & s_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & s_{nn} \end{pmatrix} \tag{1.45}$$

It can be shown that the left-hand side of Equation (1.43) is equivalent to

$$1 - \frac{\sum_{i=1}^{k} s_{ii}}{\sum_{i=1}^{n} s_{ii}} \tag{1.46}$$

and therefore we seek the smallest k such that

$$\frac{\sum_{i=1}^{k} s_{ii}}{\sum_{i=1}^{n} s_{ii}} \geq \alpha. \tag{1.47}$$

Given we are likely computing S while computing U, this is the simpler computation.

One of the most common uses of PCA is speeding up supervised learning algorithms. Reducing the dimensionality of the input data results in less computation required. Although we are losing information here, we will still be retaining $\alpha\%$ of the variance.

Of course, the mapping $x \mapsto z$ should be defined on the *training* set only. After defining the mapping from the training set, we can then use it in the cross-validation and test sets also.

One bad implementation of PCA is in an attempt to prevent overfitting i.e. use z instead of x to reduce the number of features to $k < n$. Thus, fewer features and less likely to overfit. This method might actually work okay, but it is not a good way to address overfitting as PCA doesn't use the output in its calculations - how can it be used to reduce overfitting if it doesn't know what the outputs are? One should use regularisation instead here.

1.8 Anomaly Detection

Given a dataset $\{x^{(1)}, ..., x^{(m)}\}$, we want to assess whether the point x_{test} is anomalous or not.

One can plot the density of the given dataset and work under the assumption that if $p(x_{test}) < \epsilon$ then we will flag x_{test} as anomalous, for some $\epsilon \in (0, 1)$. For example, we could take the normal distribution

$$p(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{1.48}$$

One can estimate μ and σ from a given dataset using

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}, \qquad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2 \tag{1.49}$$

These are the maximum likelihood estimators.

Anomaly detection can be used in

- fraud detection;
- identification of faulty manufactured goods;
- monitoring computers in a data centre.

We can estimate the density of some vector of features $x \in \mathbb{R}^n$ by

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) \tag{1.50}$$

where we assume that $p(x_j; \mu_j, \sigma_j^2) \sim N(\mu_j, \sigma_j^2) \, \forall j$.

Anomaly detection algorithm

Putting the above together, we have the following anomaly detection algorithm:

1. Choose features x_i that you think might be indicative of anomalous examples.
2. Fit parameters $\mu_1, ..., \mu_n, \sigma_1^2, ..., \sigma_n^2$ using

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}, \qquad \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2 \tag{1.51}$$

3. Given a new example x, compute

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp^{-\frac{(x_j-\mu_j)^2}{2\sigma_j^2}} \tag{1.52}$$

4. x is an anomaly if $p(x) < \epsilon$.

Supervised Approach

The above algorithm was *unsupervised* - we were given unlabelled data and were tasked with identifying anomalous data. We can also train a supervised algorithm.

Assume we have some labelled data of anomalous ($y = 1$) and normal ($y = 0$) data. We can then, as before, train our algorithm on a training set $x^{(1)}, ..., x^{(m)}$ and evaluate it using a cross-validation set $(x_{cv}^{(1)}, y_{cv}^{(1)}), ..., (x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})$ and a test set $(x_{test}^{(1)}, y_{test}^{(1)}), ..., (x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$.

Here is the equivalent supervised learning algorithm:

1. Fit the model $p(x)$ on training set $\{x^{(1)}, ..., x^{(m)}\}$
2. On a cross=validation/test example x, predict

$$y = \begin{cases} 1, & p(x) < \epsilon \quad \text{(anomaly)} \\ 0, & p(x) \geq \epsilon \quad \text{(normal)} \end{cases} \tag{1.53}$$

3. Evaluate performance using e.g. F-score, precision/recall, confusion matrix.

The training data is likely highly skewed (i.e. you have many more *normal* examples than you do *anomalous* ones) and therefore it is unwise to use classification accuracy as an evaluation metric since estimating $y = 0$ always will give a high accuracy. We can also use ϵ as a variable here, maximising your evaluation metric on your cross-validation set.

The takeaway here is that you should aim to use a *single-number* evaluation metric. It is then easy to pick which model is 'best'.

Anomaly detection vs Supervised Learning

When should you use an anomaly detection algorithm over a supervised learning algorithm (note: here we are referring to a *supervised learning algorithm* as an algorithm such as linear regression or a neural network and *not* the supervised approach we have taken above to anomaly detection)?

Anomaly detection methods are generally more fruitful when

- we have a small number of positive examples (e.g. 0-20 anomalous examples) and a large number of negative examples;
- many different types of anomaly. It is hard for any algorithm to learn what the anomalies look like. Future anomalies may be nothing like those seen previously;
- examples: fraud detection, manufacturing, monitoring machines in a data centre.

Supervised learning algorithms are better suited when

- we have a large number of both positive and negative examples;
- we have enough positive examples for the algorithm to get a sense of what positive examples are like. Future positive examples likely to be similar to the ones in training set;
- examples: spam classification, weather prediction, cancer detection.

Feature selection in anomaly detection algorithms

Firstly, your input data should be *approximately* gaussian. This is one of our assumptions in our anomaly detection algorithm. If our data is not gaussian, then maybe we can apply some transformation to the data so that it is gaussian (e.g. log-returns of a stock).

One can also perform *error analysis* for anomaly detection. That is, we look at the examples our algorithm gets wrong in the cross-validation set and see if we can't add a feature that might help explain these anomalies better.

An obvious choice for parameters would be those that take on unusually large/small values in the event of an anomaly.

1.8.1 Multivariate gaussian distributions

We can use the multivariate gaussian distribution to extend the algorithms described above and hopefully improve their performance.

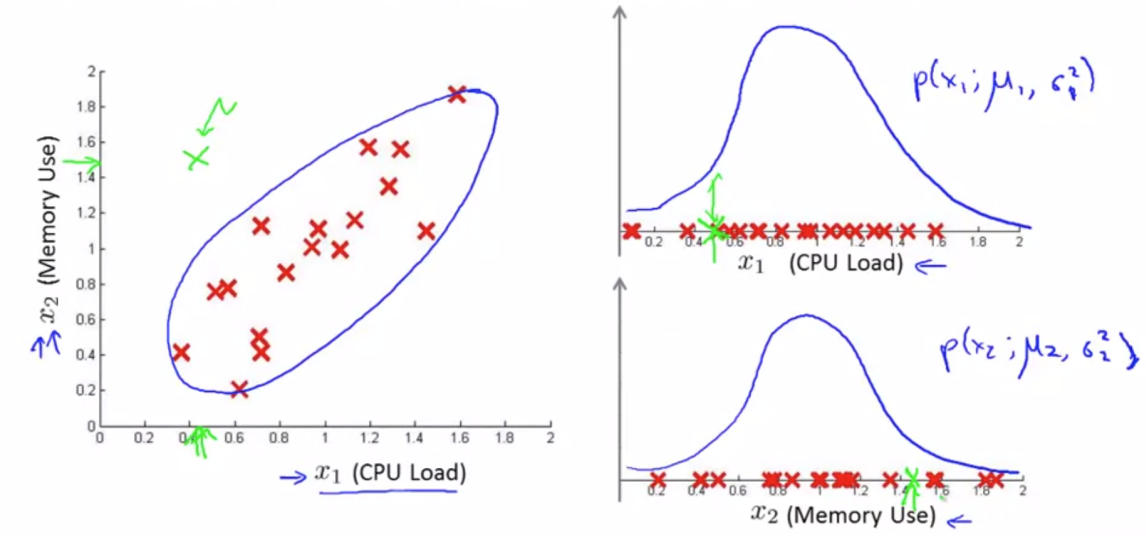

Figure 1.27: Potential application of multivariate normal

Figure 1.27 shows how a multivariate normal could be used to detect an anomalous data point that is otherwise considered normal in the univariate cases.

We now have

$$p(\boldsymbol{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1} (\boldsymbol{x}-\boldsymbol{\mu})} \tag{1.54}$$

where $\boldsymbol{x}, \boldsymbol{\mu} \in \mathbb{R}^n$, $\Sigma \in \mathbb{R}^{n \times n}$. Similarly to above, given a training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ we can estimate $\boldsymbol{\mu}$ and Σ using

$$\boldsymbol{\mu} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{x}^{(i)}, \qquad \Sigma = \frac{1}{m} \sum_{i=1}^{m} (\boldsymbol{x}^{(i)} - \boldsymbol{\mu})(\boldsymbol{x}^{(i)} - \boldsymbol{\mu})^T \tag{1.55}$$

and can flag a new example $\boldsymbol{x}$ as anomalous if $p(\boldsymbol{x}; \boldsymbol{\mu}, \Sigma) < \epsilon$.

1.8.2 Recommender Systems

Given a user's purchase/viewing/reading history, we attempt to recommend them new items/shows/books to look at. For example, we would attempt to estimate the ratings of the unseen movies given the ratings history of each user.

Movie	Alice(1)	Bob(2)	Carol(3)	Dave(4)
Love at Last	5	5	0	0
Romance Forever	5	?	?	0
Cute Puppies of Love	?	4	0	?
Nonstop Car Chases	0	0	5	4
Swords vs Karate	0	0	5	?

What we could do is add features to this table in order to better predict which movies people will (dis)like. One such feature set would be *the extent to which a movie can be described as being from genre A*.

Movie	Alice(1)	Bob(2)	Carol(3)	Dave(4)	x_1 (romance)	x_2 (action)
Love at Last	5	5	0	0	0.9	0
Romance Forever	5	?	?	0	1.0	0.01
Cute Puppies of Love	?	4	0	?	0.99	0
Nonstop Car Chases	0	0	5	4	0.1	1.0
Swords vs Karate	0	0	5	?	0	0.9

We therefore have the following feature matrix

$$X = \begin{pmatrix} 1 & 0.9 & 0 \\ 1 & 1.0 & 0.01 \\ 1 & 0.99 & 0 \\ 1 & 0.1 & 1.0 \\ 1 & 0 & 0.9 \end{pmatrix} \tag{1.56}$$

For each user j, learn a parameter $\theta^{(j)} \in \mathbb{R}^3$. Predict user j as rating movie i with $(\theta^{(j)})^T x^{(i)}$ stars. For example, we predict Alice would rate *Cute Puppies of Love* as

$$\begin{pmatrix} 0 & 5 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0.99 \\ 0 \end{pmatrix} = 4.95 \tag{1.57}$$

Problem formulation

We denote

- $r(i,j) = 1$ if user j has rated movie i (0 otherwise);
- $y^{(i,j)}$ = rating by user j on movie i (if defined);
- $\theta^{(j)}$ is the parameter vector for user j;
- $x^{(i)}$ is the feature vector for movie i; $m^{(j)}$ = number of movies rated by user j.

For user j, movie i, we learn the parameter $\theta^{(j)} \in \mathbb{R}^{n+1}$ and predict the rating using $(\theta^{(j)})^T(x^{(i)})$, where $x_0^{(i)} = 1$ is our intercept term. This is essentially a least-squares regression problem. Learning $\theta^{(j)}$,

$$\min_{\theta^{(j)}} \frac{1}{2} \sum_{\substack{i \\ \text{s.t. } r(i,j)=1}} \left((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (\theta_k^{(j)})^2 \tag{1.58}$$

We want to learn for $\theta^{(1)}, ..., \theta^{(n_u)}$:

$$\min_{\theta^{(1)},...,\theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{\substack{i \\ \text{s.t. } r(i,j)=1}} \left((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2 \tag{1.59}$$

This system is called a *Content Based Recommendation System*. Here, we have assumed that someone has given us the features for all of the movies in our feature set.

Collaborative Learning

Here we have an algorithm that can learn for itself which features to use. It performs *feature learning*. We assume that the users tell us their preferences. Using our example above, we might have

Movie	Alice(1)	Bob(2)	Carol(3)	Dave(4)	x_1 (romance)	x_2 (action)
Love at Last	5	5	0	0	?	?
Romance Forever	5	?	?	0	?	?
Cute Puppies of Love	?	4	0	?	?	?
Nonstop Car Chases	0	0	5	4	?	?
Swords vs Karate	0	0	5	?	?	?

where the users have provided us with their preferences

$$\theta^{(1)} = \begin{pmatrix} 0 \\ 5 \\ 0 \end{pmatrix}, \quad \theta^{(2)} = \begin{pmatrix} 0 \\ 5 \\ 0 \end{pmatrix}, \quad \theta^{(3)} = \begin{pmatrix} 0 \\ 0 \\ 5 \end{pmatrix}, \quad \theta^{(4)} = \begin{pmatrix} 0 \\ 0 \\ 5 \end{pmatrix} \tag{1.60}$$

Using these vectors, we can infer the values for $x^{(1)}$ and $x^{(2)}$ by solving

$$(\theta^{(1)})^T x^{(1)} \approx 5, \quad (\theta^{(2)})^T x^{(1)} \approx 5, \quad (\theta^{(3)})^T x^{(1)} \approx 0, \quad (\theta^{(4)})^T x^{(1)} \approx 0. \tag{1.61}$$

We therefore have the following algorithm. Given $\theta^{(1)}, ..., \theta^{(n_u)}$, we learn $x^{(1)}, ..., x^{(n_u)}$ using

$$\min_{x^{(1)},...,x^{(n_u)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{\substack{j \\ \text{s.t. } r(i,j)=1}} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 \tag{1.62}$$

We have the following scenarios:

- Given $x^{(1)}, ..., x^{(n_m)}$ (and movie ratings), we can estimate $\theta^{(1)}, ..., \theta^{(n_u)}$;
- Given $\theta^{(1)}, ..., \theta^{(n_u)}$, we can estimate $x^{(1)}, ..., x^{(n_m)}$;

Let's put this into a *collaborative filtering algorithm*. Combining the two optimisation equations above together, we get

$$J(x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}) = \frac{1}{2} \sum_{\substack{i,j \\ \text{s.t. } r(i,j)=1}} \left((\theta^{(j)})^T (x^{(i)}) - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 \tag{1.63}$$

where we minimise over $x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}$. Given that we are now learning *all* of the features, we will have $x, \theta \in \mathbb{R}^n$ i.e. no intercept term. The algorithm now has the flexibility to add this in itself. We therefore don't have to worry about accidentally regularising the intercept term.

Here is our final algorithm:

1. Initialise $x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}$ to small random values;
2. Minimise J above using some optimisation algorithm (e.g. gradient descent);
3. Predict the star rating of movie i by user j by $(\theta^{(j)})^T x^{(i)}$.

We initialise $x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}$ to small random values as this serves as symmetry breaking (similar to the random initialisation of a neural network's parameters) and ensures the algorithm learns features $x^{(1)}, ..., x^{(n_m)}, \theta^{(1)}, ..., \theta^{(n_u)}$ that are different from each other.

Mean normalisation

Suppose we have a new user, Eve:

Movie	Alice(1)	Bob(2)	Carol(3)	Dave(4)	Eve(5)
Love at Last	5	5	0	0	?
Romance Forever	5	?	?	0	?
Cute Puppies of Love	?	4	0	?	?
Nonstop Car Chases	0	0	5	4	?
Swords vs Karate	0	0	5	?	?

The first step is to initialise Eve's preference as $\theta^{(5)} = \big(0, 0\big)$. Using J above, however, we would predict that Eve rates every movie 0 stars. This isn't very helpful. Instead, we calculate the average rating from all other users for each movie and use that as Eve's ratings instead. This is sensible as we don't have any information on Eve's tastes; we expect her to have average preferences.

$$\theta^{(5)} = \mu = \begin{pmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{pmatrix} \tag{1.64}$$

1.9 Large Scale Machine Learning

One of the best ways to get a high performance learning system is to take a low-bias learning algorithm and train it on a lot of data. Learning with large datasets, however, comes with computational problems.

1.9.1 Gradient Descent

Recall that so far we have been using gradient descent to minimise our cost function and help us to optimise our parameters. For large datasets, this can become a very computationally expensive procedure. We call this algorithm *Batch Gradient Descent* as we look at all examples. We will look at more computationally efficient methods here.

Let's remind ourselves of what the batch gradient descent does. Given

$$J_{train}(\theta) := \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2, \tag{1.65}$$

calculate

$$\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j} = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}, \quad \forall j \in \{0, ..., n\} \tag{1.66}$$

Here, we have to sum over all m examples for every iteration of the algorithm. If m is large, then this could be very computationally expensive.

Stochastic gradient descent

Here we only use one example for every iteration of the above algorithm. Let's define

$$cost(\theta, (x^{(i)}, y^{(i)})) := \tfrac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2. \tag{1.67}$$

We define the stochastic gradient descent algorithm as follows:

1. Randomly shuffle dataset;
2. Repeat the following some number of times: For $i \in \{1, ..., m\}$,

$$\theta_j := \theta_j - \alpha \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}, \quad \forall j \in \{0, ..., n\} \tag{1.68}$$

Because we only look at one example at a time, the gradient descent algorithm won't converge as efficiently as in the batch case. However, the time to compute each iteration is potentially *much* smaller. Stochastic gradient descent doesn't actually converge to the global minimum as batch gradient descent does. Instead, it will walk around the global minimum but never converge, since only one example is considered in each iteration. So long as the parameter estimate is in some acceptable range of the global minimum then this is okay.

Mini-batch gradient descent

Mini-batch gradient descent will use b examples in each iteration - this is somewhere in-between stochastic and batch gradient descent. This should solve some of the problems seen in both batch and stochastic gradient descent.

For example, for $b = 10$ examples $(x^{(i)}, y^{(i)}), ..., (x^{(i+9)}, y^{(i+9)})$ and $m = 1000$, calculate

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)} \tag{1.69}$$

for $i = \{1, 11, 21, 31, ..., 991\}$. You can then repeat this for different samples of size 10.

Similarly to stochastic gradient descent, mini-batch gradient descent improves on batch gradient descent by not using all m examples to update θ_j each time. You can also improve on stochastic gradient descent by taking advantage of vectorisation. Calculating each of the summations above in one go will potentially reduce the computation time dramatically - in this example you reduce the number of calls to the vector $(h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$ 10-fold. One downside of mini-batch gradient descent is that you have another parameter b that you might want to fiddle with.

Stochastic Gradient Descent Convergence

How should you check that the algorithm is converging okay and how do you tune the learning rate, α?

When the training set size was small, we were able to plot J_{train} against *number of iterations of gradient descent*. If our training set is large, however, then this might not be feasible. In stochastic gradient descent, let's instead calculate $cost(\theta, (x^{(i)}, y^{(i)}))$ before updating θ. We can then plot the average value of $cost(\theta, (x^{(i)}, y^{(i)}))$ over the last 1000, say, examples processed by the algorithm. Examples of these plots are shown in Figure 1.28.
Typically, the learning rate α is held constant. If you want θ to converge, you can slowly decrease α over time e.g. use $\alpha = \frac{constant_1}{iterationNumber + constant_2}$. You can then expect to see a transformation similar to that seen in Figure 1.29.

1.9.2 Map-reduce and data parallelism

A quick note on map-reduce. Say we have a large dataset, so large that it cannot fit in memory. How should we handle it?

Let's take computing the batch gradient descent

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \tag{1.70}$$

as an example. Say we have $m = 400,000,000$ examples. We can split the sum across n machines such that the summation is only across $\frac{m}{n}$ values per machine rather than all m values being added together in one machine. We can then add the results together and get the value of our original sum.

A map-reduce program is composed of a **map procedure** which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a **reduce method** which performs a summary operation (such as counting the number of students in each queue,

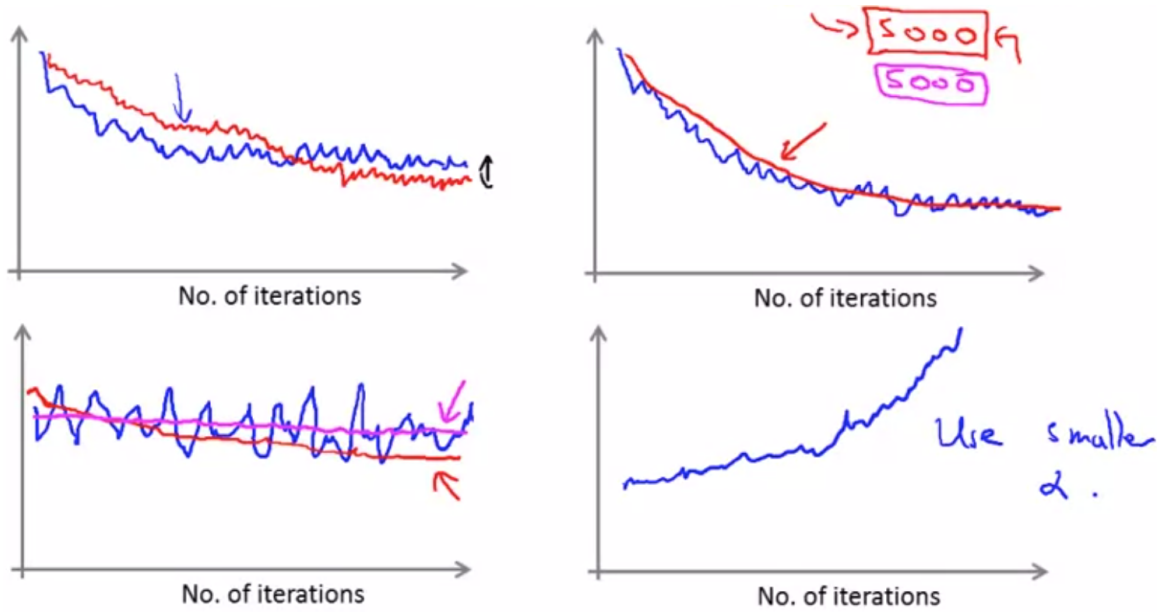

Figure 1.28: (TL) If you use a smaller learning rate; (TR) If you average over more examples; (BL) Two possibilities if you average over more examples; (BR) when you should use a smaller learning rate.

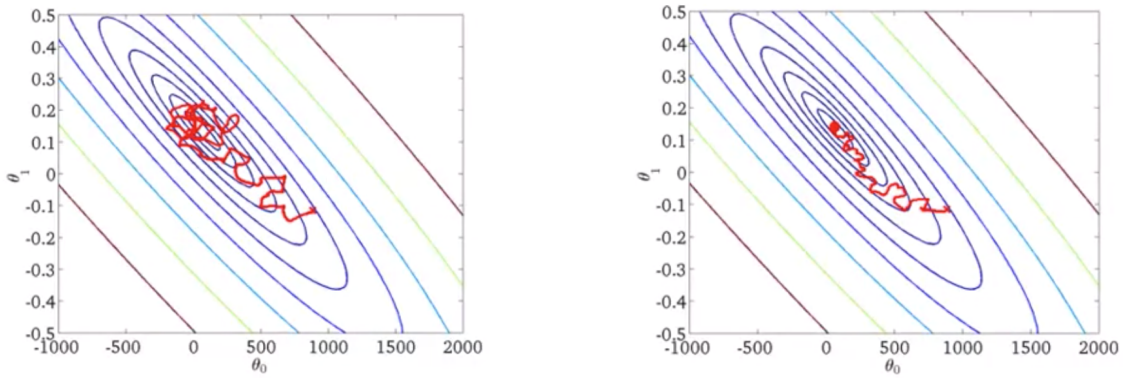

Figure 1.29: Effect on θ when you reduce α over time

yielding name frequencies). Apache Spark is a good example of map-reduce and the advantages it brings. Try to think which operations in your algorithm could be parallelised (e.g. summations) and therefore which operations could be sent to Spark, say.

1.10 Ensemble Learning

A significant portion of this section's content has come from the scikit-learn documentation.

Ensemble learning is a general term for when multiple algorithms are used together in order to improve predictive performance. The term *ensemble* is usually reserved for methods that generate multiple hypotheses using the same base learner. The broader term of *multiple classifier systems* also covers hybridization of hypotheses that are not induced by the same base learner. An example of an ensemble method is the use of decision trees to build a random forest, see Section 1.11.1. Generally, ensemble learning is applied to supervised learning algorithms although it can also be applied to unsupervised learning algorithms too (e.g. in anomaly detection).

Two families of ensemble methods are usually distinguished:

- In **averaging methods**, the driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced. Examples include *bagging* (bootstrap aggregating) methods and *random forests*;
- By contrast, in **boosting methods**, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble. Examples include *AdaBoost* (adaptive boosting) and *Gradient Boosting Machine* (GBM).

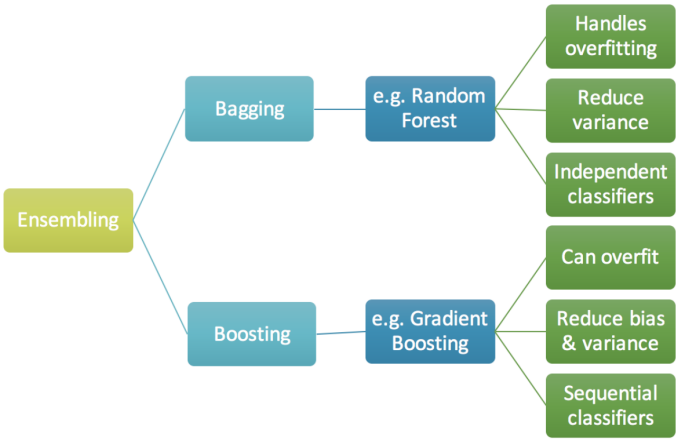

Figure 1.30: Ensembling

1.10.1 Boosting

Boosting is used primarily for reducing bias (underfitting) and variance (overfitting) in supervised learning, and are a family of machine learning algorithms that convert *weak* learners to *strong* ones. A **weak learner** is defined to be a classifier that is only slightly correlated with the true classification (it can label examples better than random guessing). In contrast, a **strong learner** is a classifier that is arbitrarily well-correlated with the true classification.

The strong learner G comprises many weak learners G_i weighted by their associated alpha weight $\alpha_i \in \mathbb{R}$. Equation 1.71 shows an additive strong classifier.

$$G(x) = \text{sign}\left(\alpha_1 G_1(x) + \alpha_2 G_2(x) + ...\alpha_M G_M(x)\right) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) \tag{1.71}$$

AdaBoost & Gradient Boosting Machine

AdaBoost and GBMs learn regression and classification problems and produce a prediction model in the form of an ensemble of weak prediction models, typically decision trees. Both AdaBoost and GBMs initialise a strong learner (usually a decision tree) and iteratively create a weak learner that is added to the strong learner. They differ on how they create the weak learners during the iterative process.

At each iteration, **AdaBoost** changes the sample distribution by modifying the weights attached to each example in the dataset. It increases the weights of the wrongly predicted instances and decreases the weights of the correctly predicted instances. The weak learner thus focuses more on the difficult instances. In each so-called boosting iteration, we apply weights $w_1, ..., w_n$ to each of the training samples. Initially, those weights are all set to $w_i = \frac{1}{N}$. After being trained, the weak learner is added to the strong one according to its performance (so-called **alpha weight**). The

higher it performs, the more it contributes to the strong learner.

On the other hand, **GBMs** don't modify the sample distribution. Instead the weak learner trains on the remaining errors (so-called **pseudo-residuals**) of the strong learner. It is another way to give more importance to the difficult instances. At each iteration, the pseudo-residuals are computed and a weak learner is fitted to these pseudo-residuals. The contribution of the weak learner to the strong learner isn't computed according to its performance on the new distribution sample but using a gradient descent optimisation process. The computed contribution is the one minimising the overall error of the strong learner.

1.10.2 Bagging

Bagging (bootstrap aggregating) methods form a class of algorithms which build several instances of a *black-box estimator* on random subsets of the original training set and then aggregate their individual predictions to form a final prediction. Bagging is useful for reducing variance (overfitting).

These methods are used as a way to reduce the variance (overfitting) of a base estimator (e.g. decision tree) by introducing randomisation into its construction and then making an ensemble out of it. In many cases, bagging methods constitute a very simple way to improve with respect to a single model, without making it necessary to adapt the underlying base algorithm. As they provide a way to reduce overfitting, bagging methods work best with strong and complex models (e.g. fully developed decision trees), in contrast with boosting methods which usually work best with weak models (e.g. shallow decision trees). Perhaps a good idea might be to first apply boosting to create a strong model, and then apply bagging to that strong model!

Specifically, given a standard training set D of size n, bagging generates m new training sets D_i, each of size n', by sampling from D uniformly *with* replacement. This kind of sample is known as a bootstrap sample. Then, m models are fitted using the above m bootstrap samples and combined by averaging the output (for regression) or voting (for classification).

Bagging leads to improvements for unstable procedures which include, for example, neural networks, classification and regression trees, and subset selection in linear regression.

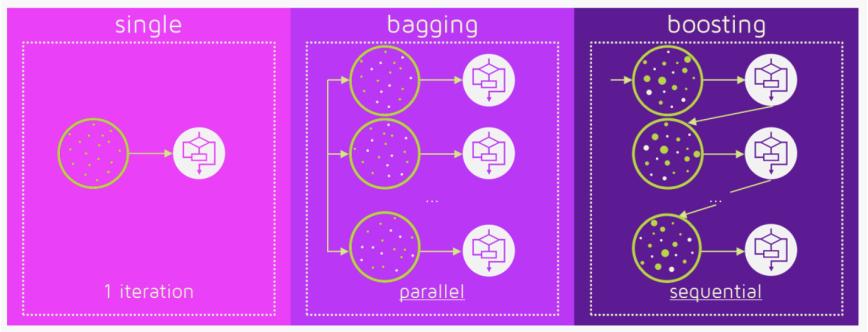

Figure 1.31: Bagging vs. Boosting

1.11 Decision Trees

A decision tree uses a series of *True or False* questions about a given data sample in order to make a prediction. Before getting into any details, let's get some terminology under our belts:

- **Root Node**: represents the entire sample and this gets divided into two or more homogeneous sets;
- **Splitting**: a process of dividing a node into two or more sub-nodes;
- **Decision Node**: a node that splits into sub-nodes;

- **Leaf/Terminal Node**: nodes that do not split;
- **Pruning**: when we remove sub-nodes of a decision node, we are pruning the tree;
- **Branch/Sub-Tree**: a subsection of the tree;
- **Parent and Child Node**: a node which is divided into sub-nodes is called a parent node of sub-nodes, whereas sub-nodes are the children of parent nodes;
- **Classification tree**: the predicted outcome is the class to which the data belongs;
- **Regression tree**: the predicted outcome is a real number (e.g. the price of a house).

Classification And Regression Tree (CART) is a term used to refer to both of the above types of trees.

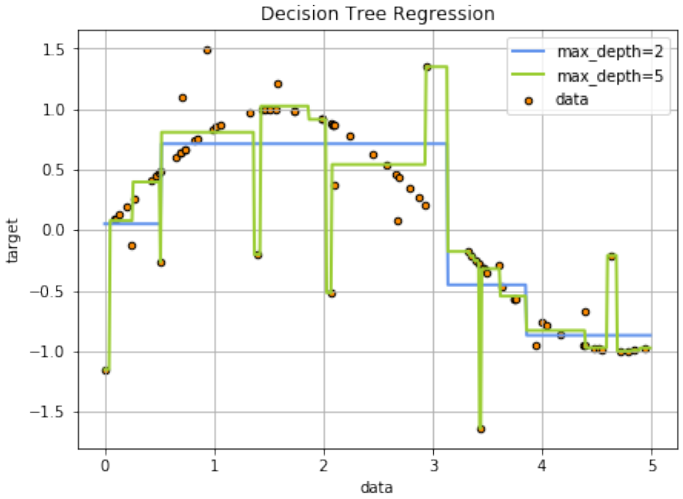

Figure 1.32: Example regression tree

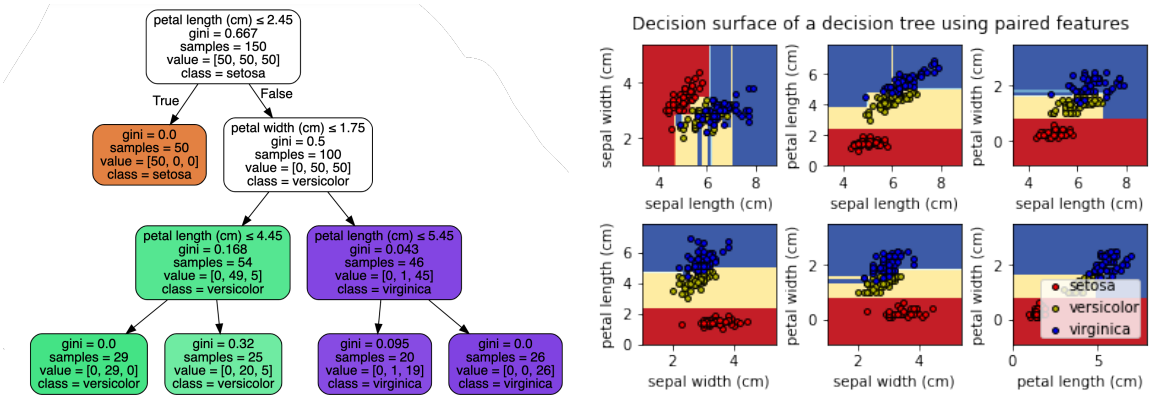

Figure 1.33: Example classification tree

Figure 1.32 plots a decision tree on the iris dataset. Nodes are coloured according to their level of purity and which class they belong to (i.e. low impurity ⇒ coloured node). Figure 1.33 shows the decision surface of the decision tree using paired features.

So how are decision trees actually built? Algorithms for constructing decision trees usually work top-down by choosing a variable at each step that *best splits* the set of items. There are many metrics one can use to decide what constitutes *best*. Here we look at the **Gini Impurity** (used in CARTs), not to be confused with the Gini coefficient. Another popular metric is information gain

(used in ID3 and C4.5).

Gini Impurity is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset. It reaches its minimum (zero) when all cases in the node fall into a single target category (see Figure 1.33).

To compute Gini impurity for a set of items with J classes, suppose $i \in \{1, 2, ..., J\}$, and let p_i be the fraction of items labeled with class i in the set.

$$I_G = 1 - \sum_{i=1}^{J} p_i^2 \tag{1.72}$$

You split the node on whichever split value results in the lowest Gini impurity. For example, suppose $I_G = 0.5$ when you split on class $A > 1$ and $I_G = 0.1$ when you split on class $A > 2$. Then you would split on class $A > 2$.

One can use ensemble methods to construct more than one decision tree and hopefully improve performance:

- **Boosted trees**: training each new instance to *correct* the training instances previously mismodelled. A typical example is **AdaBoost**. These can be used for regression-type and classification-type problems and are helpful in reducing variance and bias of your tree;
- **Bagging**: builds multiple decision trees by repeatedly resampling training data with replacement, and taking the mode of the trees' predictions for a consensus prediction. A **random forest** classifier is a specific type of bagging method. Bagging is useful in reducing the variance of your tree.

1.11.1 Random Forests

A random forest is an example of a bagging method (more information in Section 1.10.2). As you know by now, this is where the decision tree is trained on multiple resampled (with replacement) training sets, the output being the average (regression) or mode (classification) of all of the individual trees.

Random forests differ in only one way from this general scheme: they use a modified learning algorithm that selects, at each candidate split in the learning process, a *random subset of the features* (i.e. *feature bagging*). The reason for doing this is if one or a few features are very strong predictors, these features will be selected in many of the individual trees, causing them to become correlated.

Chapter 2

Deep Learning

2.1 Neural Networks and Deep Learning

First thing's first; if you haven't done so already, read Section 1.4. There will be some overlap in material between the beginning of this chapter and Section 1.4.

As a motivating example, let's say we are trying to predict the price of a house given the house's size, number of bedrooms, zip code, and the wealth of the area. We can represent our neural network as in Figure 2.1.

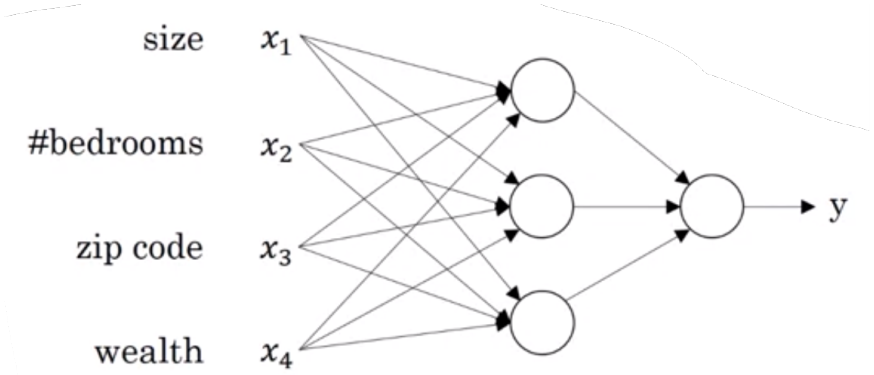

Figure 2.1: Predicting house prices using a neural network

To train our neural network, all we have to give it is a set of (x, y) pairs. We then run an algorithm to *learn* from these examples with the hope that our neural network will then be able to accurately predict y_{new} for some unseen x_{new}. Given enough data, neural networks become remarkably good at predicting/classifying new examples. Note that bias units with value 1 are often added to each non-output layer. A great explanation of why we include bias units can be seen here. Bias units are not regularised and always have a value of 1.

Neural networks are most commonly used as *supervised learning* algorithms; that is we give the neural network labelled data to learn from. Here are some examples of supervised learning applications using neural networks

- real estate: predict housing price (standard NNs);
- facial recognition: photo tagging (CNNs);
- translation: Google translate (RNNs);
- autonomous vehicles: position of other cars (hybrid of potentially many types of NNs);

A fantastic summary of the use-cases for each type of neural network can be found here. Data can be split into **structured** and **unstructured** data. Structured data is what you might find in a typical database e.g. the input data for Figure 2.1. Unstructured data, on the other hand, has no such structure to each example, examples of which include audio data, images and text.

2.1.1 Logistic Regression as a Neural Network

Here we will look at the details of a neural network using classification as our motivating example. As mentioned previously, we feed a feature vector x into our neural network and receive some predicted output y. If our input is a RGB image (i.e. matrices $M_1, M_2, M_3 \in \mathbb{R}^{p \times q}$), then we will *unroll* these 3 matrices to form one long matrix $x \in \mathbb{R}^{3pq}$.

We will have m training examples $x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$ i.e. $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$. We can write the training examples as a matrix $X \in \mathbb{R}^{n_x \times m}$. Similarly, we create a vector of y values $Y \in \mathbb{R}^m$.

Recall that for logistic regression, given some input x, we want to predict $a = \mathbb{P}(y = 1 \mid x) \in (0, 1)$. Since we have bounded a, we calculate it using the sigmoid function

$$a = \sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}. \tag{2.1}$$

The purpose of our algorithm is to learn the parameter values for our weights w and bias term b. For large input z, $\sigma(z) \approx 1$ (i.e. we predict a positive output). Similarly for small z, $\sigma(z) \approx 0$ (i.e. we predict a negative output). See Figure 1.4 for a plot of the sigmoid function. We are keeping w and b separate to clearly denote the bias term and also make regularisation easier to implement (we don't want to regularise b). For brevity, we shall denote

$$z^{(i)} = w^T x^{(i)} + b \tag{2.2}$$

We can learn the parameter values w and b by minimising some cost function across w and b. We shall use the **loss function** seen in Equation 2.3 for our logistic regression model. This is commonly known as the **cross-entropy** loss function.

$$\mathcal{L}(a, y) = -\left(\log(a)y + \log(1 - a)(1 - y)\right) \tag{2.3}$$

We can see that when y and a are vastly different (e.g. 0 and 1, respectively), then we have a *large* loss. For similar values of y and a (e.g. 1 and 1, respectively) we have a *small* loss. The loss function, however, only gives us a metric for a single training example. We want a metric of the performance of our algorithm on the training set as a whole. We use the loss function to calculate the cross-entropy **cost function** across all of our training examples

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} \log(a^{(i)})y^{(i)} + \log(1 - a^{(i)})(1 - y^{(i)}). \tag{2.4}$$

See Section 1.3.2 for some more explanation on why we use Equation 2.4 as the cost function for logistic regression.

Gradient Descent

Here we shall look at the gradient descent algorithm for minimising the cost function J seen in Equation 2.4. See Section 1.2.1 for an introduction to gradient descent. Again, it is important to note that gradient descent does not necessarily guarantee finding the global minimum for J; it will find a local minimum. Mostly, we don't need to worry about this.

Gradient descent iterates over Equation 2.5 until J converges.

$$\theta := \theta - \alpha \frac{\partial}{\partial \theta} J(w, b) \tag{2.5}$$

where α is our *learning rate* and θ is one of w or b. The learning rate is a variable that we can change. Details on the effect of α on gradient descent are discussed in Section 1.2.1. As an implementation note, you should update all of your parameters at the same time (i.e. at the end of each loop). If you update parameters as you go, then this will create unwanted side-effects.

2.1.2 Shallow Neural Networks

A *shallow neural network* is a neural network with only one hidden layer. A quick notational point: we shall denote layer i by the superscript $[i]$. For example, the activations of the nodes in the hidden layer of Figure 2.2 will be denoted $\left[a_1^{[1]}, a_2^{[1]}, a_3^{[1]}, a_4^{[1]}\right]$. For training example j in layer i, we would write $a^{[i](j)}$. Figure 2.2 is a 2-layer neural network (we do not count the input layer).

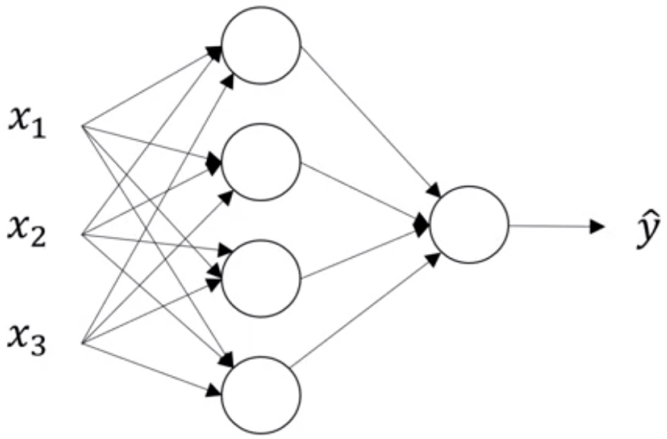

Figure 2.2: Shallow neural network

At each node of the neural network, we compute two quantities: $z = w^T x + b$ and $a = \sigma(z)$. We can vectorise the calculations done in each layer of the neural network by

$$z^{[i]} = \begin{pmatrix} — & w_1^{[i]} & — \\ — & \vdots & — \\ — & w_{n^{[i]}}^{[i]} & — \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{n_x} \end{pmatrix} + \begin{pmatrix} b_1^{[i]} \\ \vdots \\ b_{n^{[i]}}^{[i]} \end{pmatrix}; \quad a^{[i]} = \sigma(z^{[i]}) \tag{2.6}$$

where $n^{[i]}$ is the number of nodes in layer i. This implementation only calculates $z^{[i]}$ and $a^{[i]}$ for one training example, however. In order to calculate the above over *all* training examples at once, we calculate

$$Z^{[i]} = \begin{pmatrix} | & | & | \\ z^{[i](1)} & \cdots & z^{[i](m)} \\ | & | & | \end{pmatrix} = \begin{pmatrix} — & w_1^{[i]} & — \\ — & \vdots & — \\ — & w_{n^{[i]}}^{[i]} & — \end{pmatrix} \begin{pmatrix} | & | & | \\ x^{(1)} & \cdots & x^{(m)} \\ | & | & | \end{pmatrix} + \begin{pmatrix} b_1^{[i]} \\ \vdots \\ b_{n^{[i]}}^{[i]} \end{pmatrix} \tag{2.7}$$

which can then be used to calculate, using $a^{[i](j)} = \sigma(z^{[i](j)})$,

$$A^{[i]} = \begin{pmatrix} | & | & | \\ a^{[i](1)} & \cdots & a^{[i](m)} \\ | & | & | \end{pmatrix} \tag{2.8}$$

2.1.3 Activation Functions

So far, we have only used the sigmoid function. This is not necessarily the best activation function for us to use. More generally, we have some activation function g that we apply to $z^{[i]}$ i.e. $g(z^{[i]})$. We denote the activation function for layer i as $g^{[i]}(z^{[i]})$. Another common activation function that people use is the hyperbolic tangent (i.e. **tanh**) function, which is technically just a shifted version of the sigmoid function:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{2.9}$$

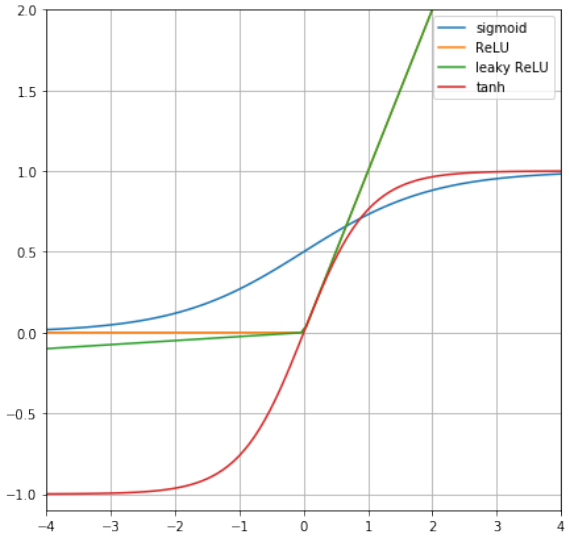

Figure 2.3: Some common activation functions

tanh almost always works better than the sigmoid activation function as the mean of the activations that come out of the hidden layer are closer to having a zero mean since $-1 \leq \tanh(z) \leq 1$. The sigmoid function has activations with mean approximately 0.5. Having activations for a given layer with mean close to zero actually makes learning easier for the next layer. We will revisit this idea in more detail in Section 2.2.

The one exception where it generally makes sense to use the sigmoid function over the tanh activation function is when used in the output layer of a binary classification problem. If $y \in \{0, 1\}$, then it makes sense for our prediction $\widehat{y}$ to be in this range too.

One of the downsides to *both* the tanh and sigmoid activation functions is that if z is large (either positive or negative) then the gradient of each becomes very small (approximately zero). This can slow down gradient descent massively. To solve this problem, we use the **Rectified Linear Unit** (ReLU) activation function

$$\text{ReLU}(x) = \max(0, x). \tag{2.10}$$

This is efficient to compute and solves the problem of **vanishing gradients** described above. One disadvantage to the ReLU activation function is that it has a gradient of zero for $z < 0$. This is where the **leaky ReLU** (see Figure 2.3) comes into play. This usually works better in practice than the ReLU function. Nevertheless, if you are not sure which activation function to use in your hidden layer, a safe default is (leaky) ReLU!

Finally, the **softmax** activation function is a type of sigmoid function and is handy when we are trying to handle multi-class classification problems. The softmax function scales the outputs for each class between 0 and 1 and divides by the sum of the outputs. This essentially gives the probability of the input being in a particular class. It is defined as

$$\text{softmax}(z^{[i]}) = \frac{e^{z^{[i]}}}{\sum_{k=1}^{K} e^{z^{[i](k)}}}. \tag{2.11}$$

where K is the number of classes in our output layer. The loss function used here is

$$\mathcal{L}(\widehat{y}, y) = -\sum_{j=1}^{K} y_j \log(\widehat{y}_j) \tag{2.12}$$

with cost

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\widehat{y}^{(i)}, y^{(i)}). \tag{2.13}$$

For our neural network to calculate interesting functions, it's important to use non-linear activation functions like the ones described above. A combination of linear activation functions is itself linear, meaning the transforms we are applying to our data will be no more than linear. This restricts the performance of our network dramatically. One place where you may want to use a linear activation function (e.g. $g(z) = z$) is when in the output layer of a regression network i.e. $y \in \mathbb{R}$. Your hidden units should still be non-linear, though.

2.1.4 Gradient descent for neural networks

We find optimal values of our parameters $w^{[i]}$ and $b^{[i]}$ using gradient descent. As you can see in Equation 2.5, this requires the derivative of J w.r.t. $w^{[i]}$ and $b^{[i]}$. To get this value, we must perform the **back-propagation** algorithm (previously described in Section 1.4.2). The back-propagation algorithm is essentially just a repeated application of the chain rule to get these derivatives.

Say we have just completed one run of the forward propagation algorithm to compute what our hypothesis predicts. Now, we shall work backwards through our algorithm "correcting" the errors that our original hypothesis made. For our output layer, we calculate the vector of errors

$$\frac{\partial \mathcal{L}}{\partial z^{[L]}} = a^{[L]} - y. \tag{2.14}$$

In the remaining layers, we calculate

$$\frac{\partial \mathcal{L}}{\partial z^{[l]}} = (w^{[l+1]})^T \frac{\partial \mathcal{L}}{\partial z^{[l+1]}} .* g'(z^{[l]}) \tag{2.15}$$

where ".∗" denotes element-wise multiplication. We can then use the *chain rule* to calculate

$$\frac{\partial \mathcal{L}}{\partial w^{[l]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}} \frac{\partial z^{[l]}}{\partial w^{[l]}}; \quad \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}} \frac{\partial z^{[l]}}{\partial b^{[l]}}. \tag{2.16}$$

Looking at Equation 2.4, we see that we can calculate the derivative of our cost function using

$$\frac{\partial J(w^{[l]}, b^{[l]})}{\partial \theta^{[l]}} = -\frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}(a^{(i)}, y^{(i)})}{\partial \theta^{[l]}} \tag{2.17}$$

We are now ready to use Equation 2.5 to update our parameter values. **Random initialisation** of our parameters is important and is discussed in Section 1.4.2.

2.1.5 Deep Neural Networks

A deep neural network is one with many hidden layers. A deep neural network is often better at learning complex functions that a more shallow network cannot learn. One often uses the number of hidden layers as a hyperparameter for their network. Forward propagation for a deep neural network is the same as in a shallow neural network except now we just iterate the process over each of the hidden layers.

Let's take an example to see why using deep neural networks might work better than using shallow neural networks. In facial recognition, you can think of the first layer of the neural network as perhaps being a *feature* detector i.e. where are the *edges* in this picture. Now that we know where the edges are, we can group them together to form parts of the face. For example, you may have one neuron to find a part of the eye, and another to find ears. The next layer will then compose these features to try and detect a specific face. Will shall revisit this example in greater depth in Section 2.4.
Another example could be speech recognition. Given an audio file, the network may first attempt to recognise low-level audio waveforms. Using these, it may then extract phonemes in the waveform (e.g. 'C', 'A', 'T'), which can be composed to form words, and furthermore sentences. It is possible to compute sufficiently complicated functions using shallow networks, but you will require *exponentially* more hidden units to do so. It therefore makes sense to use deep neural networks as they can be much more computationally efficient.

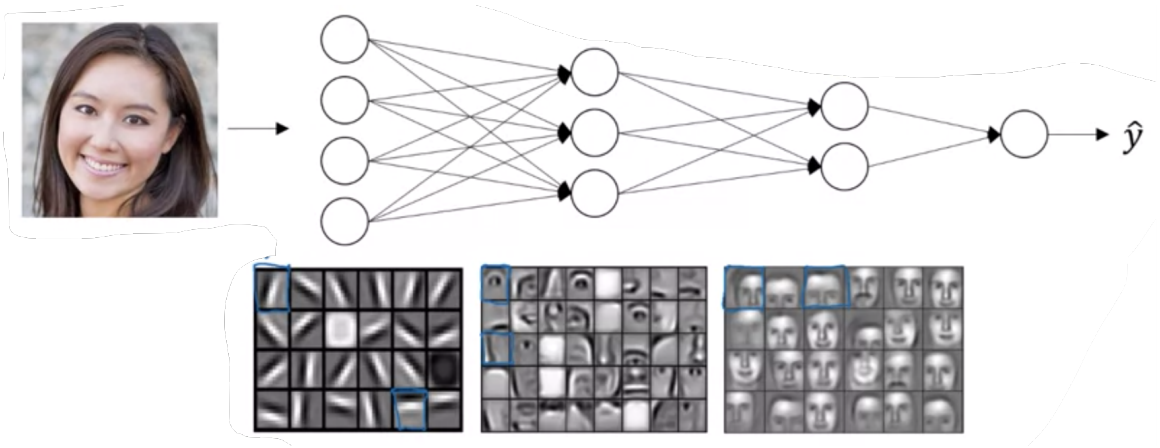

Figure 2.4: Facial recognition in deep neural networks

Parameters vs. Hyperparameters

The parameters of our model are w and b (for each layer). These values are learned by the algorithm. **Hyperparameters**, on the other hand, are parameters whose values are set before the learning process begins. For example,

- learning rate, α;
- number of iterations of forward/back propagation;
- number of hidden layers, L;
- number of hidden units;
- choice of activation function $g^{[l]}$.

These hyperparameters are the parameters that control the values of w and b. We will see other hyperparameters later such as *momentum*, *mini-batch size* and *regularisations*. Tuning of hyperparameters is often a very manual, iterative process.

2.2 Improving Deep Neural Networks: Hyperparameter tuning, Regularisation and Optimisation

2.2.1 Practical aspects of Deep Learning

First let's look into how best to split up our data into train, validation, and test sets. This can make a huge difference in the effectiveness of our research. It was common practice to split the data in a 60/20/20 split. In the era of big data, however, the cross-validation and test sets have become much smaller. Remember, the more data we have to learn on, the better our performance is likely to be! We don't need *that* much data to check the performance of the algorithm itself. The more data we can assign to the training set, the better! One problem that can be encountered here is that the validation and test sets come from a different distribution to that of the training data i.e. you have trained your algorithm on differently distributed data to the data you are testing it on. Do your best to make sure this doesn't happen.

2.2.2 Bias & Variance

It is essential that one has a solid understanding of what bias and variance are. We shall investigate it further here, focusing on its applications in deep learning.
Figure 2.5 shows examples of algorithms that predict with high bias and high variance. It is convenient that this data is only in 2 dimensions so that we can plot it. We can still analyse bias and variance if our data is in some high dimensional space, however. For example, if your training

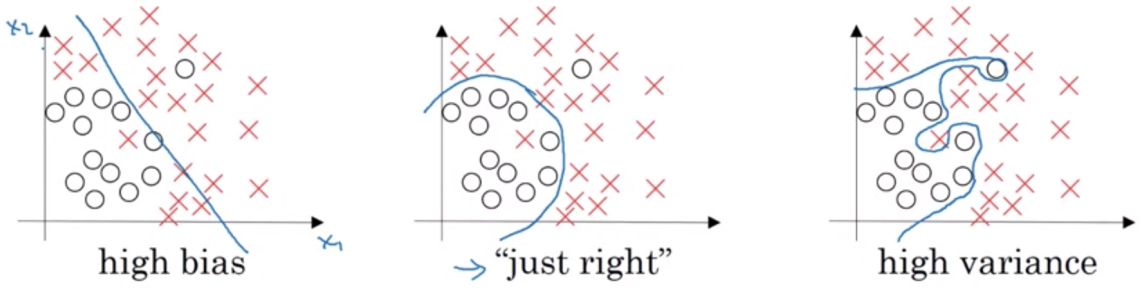

Figure 2.5: 2D example of fits with bias (left) and variance (right)

error is low and your testing error is high, then this would imply overfitting (i.e. high variance). If your training error is higher than expected, but similar to your testing error then this implies underfitting (i.e. high bias). Unexpectedly high training error and even higher testing error can imply both high bias **and** high variance (i.e. underfitting some parts of the data and overfitting in other parts of the data). What an 'expected' error value (a.k.a. Bayes error) is is difficult to estimate.

There are some approaches we can take to reducing bias and variance.

- **High bias**?
  - bigger network;
  - train longer;
  - network architecture search.
- **High variance**?
  - get more data;
  - regularisation;
  - network architecture search.

You can loop over the above until you have an acceptable model. It is key to first do some analysis using that training and cross-validation set in order to correctly diagnose where your time would be best spent in further development. You don't want to collect more data if you have a high bias problem, for example. One of the reasons deep learning has been so effective in practice is because there is little **bias-variance tradeoff** - reducing one tends not to increase the other. As long as you have a well-regularised network then having a bigger network almost never hurts. The main downside is increased computation time.

2.2.3 Regularisation

If your network is suffering from high variance (overfitting) then one of the first things you should investigate is using **regularisation**. In the case of logistic regression, we minimised the following:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\widehat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{j=1}^{n_x} w_j^2 \tag{2.18}$$

where the *regularisation* parameter λ is set using the cross-validation set. In the case of a neural network, we instead use the cost-function

$$J(w^{[1]}, b[1], ..., w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\widehat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}\left(w_{ij}^{[l]}\right)^2 \tag{2.19}$$

where our parameter update is now

$$\begin{aligned}\frac{\partial J}{\partial w^{[l]}} &= (\text{result from backprop}) + \frac{\lambda}{m} w^{[l]} \\ w^{[l]} &= w^{[l]} - \alpha \frac{\partial J}{\partial w^{[l]}} \\ &= \left(1 - \alpha \frac{\lambda}{m}\right) w^{[l]} - \alpha(\text{result from backprop})\end{aligned} \tag{2.20}$$

The final line of Equation 2.20 shows how with each iteration of gradient descent, the value of $w^{[l]}$ shrinks (i.e. $1 - \alpha \frac{\lambda}{m} < 1$). Regularisation is often aptly referred to as *weight decay*.

So why does regularisation reduce overfitting? Essentially what we are doing is setting certain weights in our network *close* to zero, reducing the impact of those nodes on the final result. An example can be seen in Figure 2.6.

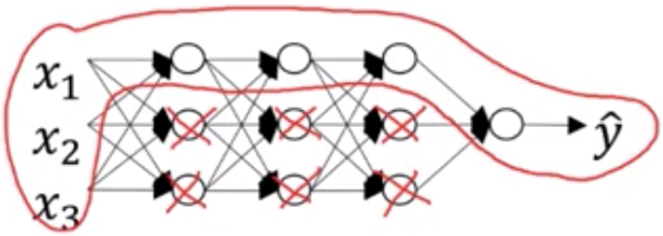

Figure 2.6: Regularisation of a neural network

Dropout regularisation

Dropout regularisation is another regularisation method. Given a network, we set some probability of eliminating each node in the neural network. We then remove nodes at random according to this probability and run our network. Repeating this, we will train our network on many different reduced versions of our original network.

One implementation of dropout regularisation is **inverted dropout**. Here, we select at random to keep each node in a given layer l with probability p (setting the activation value $a^{[l]}$ of those that we don't want to keep to 0). After this, we scale the $a^{[l]}$ by dividing it by p. This is an attempt to avoid changing the expected value $a^{[l]}$ and therefore $z^{[l+1]} = w^{[l+1]}a^{[l]} + b^{[l+1]}$ too. At test time, we do not use dropout. This is because here we do not want our output to be random! This would just add noise to our predictions, which is not ideal.

But why does dropout work? The intuition here is that we can't rely on any one feature in our network, and so we spread out the weights. Note that we can have different dropout probabilities for each layer (perhaps we want to regularise larger layers more than smaller layers). Dropout is particularly common in *computer vision* applications as the input layer is often so big and you almost never have enough data.

One downside of dropout regularisation is that the cost function J is no longer well-defined. On every iteration you are randomly removing nodes. This makes it harder to calculate the cost of your network. What you should do is first train your network without dropout and make sure J is decreasing over time. When you are happy that everything is working fine then you can turn dropout back on.

Other regularisation

One method used to reduce overfitting is **data augmentation**. That is where you use the data that you already have and create new examples from it. For example, you could take the mirror image of each image in your training set, such as in Figure 2.7.

Figure 2.7: Data augmentation

Of course this is not as good as collecting more independent training examples, however it is much more time-efficient to carry out. There are many different transformations you can apply to a single image, not just mirroring.

Another method is **early stopping**. This is where we stop training our network at some 'optimal' point. This is most commonly done at the iteration where the cross-validation error starts increasing as in Figure 2.8. There is no point in training any further than this point as we are just increasing the amount of overfitting being done.

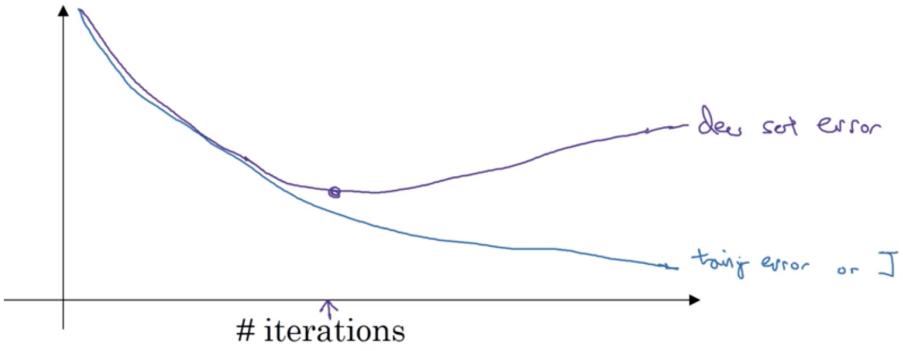

Figure 2.8: Early stopping

One downside to early stopping is that we are combining two separate processes into one here:

1. optimise the cost function J;
2. reduce overfitting.

We should be thinking about only one of the above at a time and working on each one independently. They are separate processes with separate objectives. This method of separating processes in your work is called **orthogonalisation**.

2.2.4 Setting up your optimisation problem

One of the techniques that will speed up training is **normalisation** of your input values. That is

$$x^{(i)'} = \frac{x^{(i)} - \mu}{\sigma} \tag{2.21}$$

It is important to use the same values of μ and σ for your train, cross-validation, and test sets. You want to be comparing like with like. Without normalisation, small bumps in one variable can lead to massive changes in the cost function. This makes gradient descent run much slower. An illustration of this can be seen in Figure 2.9.
Performing normalisation almost never does any harm, so it probably isn't a bad idea to at least try it by default when building a neural network.

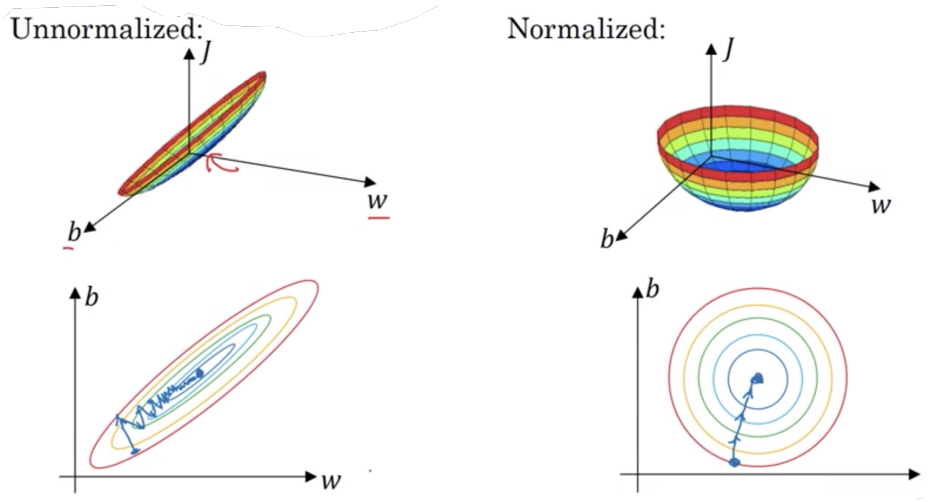

Figure 2.9: The importance of normalisation

Vanishing/exploding gradients

When you are training a very deep neural network, your derivatives can sometimes get very very big/small. Using careful choices of random weight initialisation can significantly reduce the effect of this problem. Having very large/small values for our gradients can result in it taking a long time for the gradient descent algorithm to find the optimum parameter values.

If the value of $\widehat{y}$ is very large/small, then this will feedback into the gradients too. Take a single layer network as an example. We have

$$z = w_1 x_1 + ... + w_n x_n \tag{2.22}$$

and $a = g(z)$, so for *large* n it makes sense that we have *small* w_i, otherwise z will be large, and so will a, and so will $\widehat{y}$! Similarly for *small* n we want *large* w_i. We do this by setting

$$\text{Var}(w_i) = \frac{1}{n} \tag{2.23}$$

For layer l, we have $\text{Var}(w^{[l]}) = \frac{1}{n^{[l-1]}}$. For ReLU activation, it is common to multiply this value by two as the space of non-zero values is half as big as that of the tanh activation function, for example. This doesn't *solve*, but rather *reduces* the effect of the vanishing gradient problem. Initialising using a variance of $\frac{1}{n^{[l-1]}}$ is known as **Xavier initialisation**.

2.2.5 Optimisation algorithms

Gradient descent

Here we shall look at a range of optimisation algorithms that will enable us to train our models much faster. First we shall look at **mini-batch gradient descent**. In the usual **batch gradient descent**, we process all m examples in our dataset before being able to take one step of gradient descent. If m is large then this can become computationally infeasible. We can split our training set into smaller sets (i.e. mini-batches) if m is large. For example, we can split a large training set into mini-batches of size 1000 as in Equation 2.24.

$$X = [\underbrace{X^{(1)}, X^{(2)}, X^{(3)}, ..., X^{(1000)}}_{X^{\{1\}}}, \underbrace{X^{(1001)}, ..., X^{(2000)}}_{X^{\{2\}}}, ..., X^{(m)}] \in \mathbb{R}^{n_x \times m} \tag{2.24}$$

where we have used the superscript $\{t\}$ to denote mini-batch t. Each **epoch** (i.e. pass through the training set) using gradient descent allows you to take one step. Each epoch using mini-batch gradient descent allows you to take $\frac{m}{\text{mini-batch size}}$ steps.

You should expect the cost to be monotonically decreasing when using batch gradient descent. If a plot of cost vs. iteration number is noisy then something is wrong. Mini-batch gradient descent, on the other hand, is likely to be noisy and not monotonically decreasing. This is because on every iteration of the mini-batch gradient descent algorithm we are training on a different dataset $X^{\{t\}}$, unlike normal gradient descent. This results in a noisy plot of cost vs. mini-batch number.

So how do we choose our mini-batch size? If the mini-batch size is m then this is equivalent to batch gradient descent. If, on the other hand, the mini-batch size is 1 then this is what's known as **stochastic gradient descent** i.e. each batch is a single training example. Stochastic gradient descent is of course very noisy and will never reach the optimum value. We also cannot take advantage of vectorisation here. To avoid the problems associated with each of these mini-batch sizes, practitioners often pick somewhere in-between a mini-batch size of 1 and m. If the training set is small enough, then just use batch gradient descent. Otherwise it is common to use a mini-batch size of some power of 2 (say 512 or 1024).

Gradient descent with momentum

This algorithm almost always works faster than vanilla gradient descent. The basic idea is to compute an exponentially weighted average of your gradients and to use that gradient to update your weights instead. The details are as follows. On each iteration t, for some $\beta_1 \in \mathbb{R}$,

- compute $\frac{\partial J}{\partial w}$, $\frac{\partial J}{\partial b}$ on current mini-batch;
- compute $v_{dw} = \beta_1 v_{dw} + (1 - \beta_1)dw$, $v_{db} = \beta_1 v_{db} + (1 - \beta_1)db$;
- update parameters using $w = w - \alpha v_{dw}$ and $b = b - \alpha v_{db}$.

What this does is smooth out our gradient descent steps as can be seen in Figure 2.10. In Figure 2.10, you can see that gradient descent with momentum (red) has increased the directness of the gradient descent algorithm.

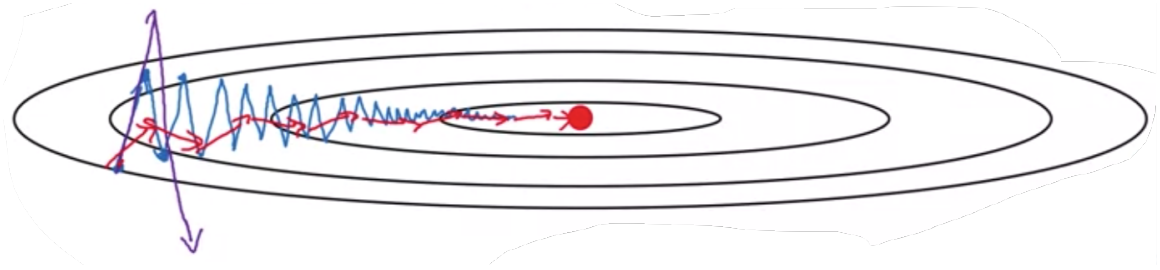

Figure 2.10: Gradient descent with (red) and without (blue) momentum

Note that we now have two hyperparameters α and β. A common value for our momentum parameter is $\beta = 0.9$ (this is because the approximate number fo timesteps smoothed over given β is $\frac{1}{1-\beta}$). So $\beta = 0.9$ smooths across approximately 10 timesteps.

RMSprop

Similarly to gradient descent with momentum, **root-mean-square propagation** seeks to increase the directness of the gradient descent algorithm to the optimum. This is done as follows. On each iteration t, for some $\beta_2 \in \mathbb{R}$,

- compute $\frac{\partial J}{\partial w}$, $\frac{\partial J}{\partial b}$ on current mini-batch;
- compute $s_{dw} = \beta_2 s_{dw} + (1 - \beta_2)dw^2$, $s_{db} = \beta_2 s_{db} + (1 - \beta_2)db^2$ (squaring is element-wise);
- update parameters using $w = w - \alpha \frac{dw}{\sqrt{s_{dw}+\epsilon}}$ and $b = b - \alpha \frac{dw}{\sqrt{s_{db}+\epsilon}}$, for small ϵ.

The updates in RMSprop look similar to the red line in Figure 2.10.

Adam optimisation

We will now combine gradient descent with momentum and RMSprop to form the **Adam optimiser** (ADAptive Moment estimation). That is, on each iteration t, for $\beta_1, \beta_2 \in \mathbb{R}$,

- compute $\frac{\partial J}{\partial w}$, $\frac{\partial J}{\partial b}$ on current mini-batch;
- compute v_{dw}, v_{db}, s_{dw} and s_{db} as above;
- implement bias correction for $\theta = \{v_{dw}, v_{db}, s_{dw}, s_{db}\}$ and their respective β_i, $i = \{1, 2\}$,

$$\theta^{corrected} = \frac{\theta}{1 - \beta_i^t}; \tag{2.25}$$

- update parameters using $w = w - \alpha \frac{v_{dw}^{corrected}}{\sqrt{s_{dw}^{corrected}} + \epsilon}$ and $b = b - \alpha \frac{b_{dw}^{corrected}}{\sqrt{s_{db}^{corrected}} + \epsilon}$, for small ϵ.

The authors of the Adam optimiser paper recommended the following values for our hyperparameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. α needs to be tuned.

Learning rate decay

One thing that can help speed up our algorithm is to slowly reduce our learning rate over time. This is known as **learning rate decay**. As we mentioned previously, running mini-batch or stochastic gardient descent results in the algorithm not reaching the global optimum. The noise in each update results in our learning algorithm moving about the optimum, but not tending to it. This can be fixed by reducing our learning rate (and therefore step size) over time.

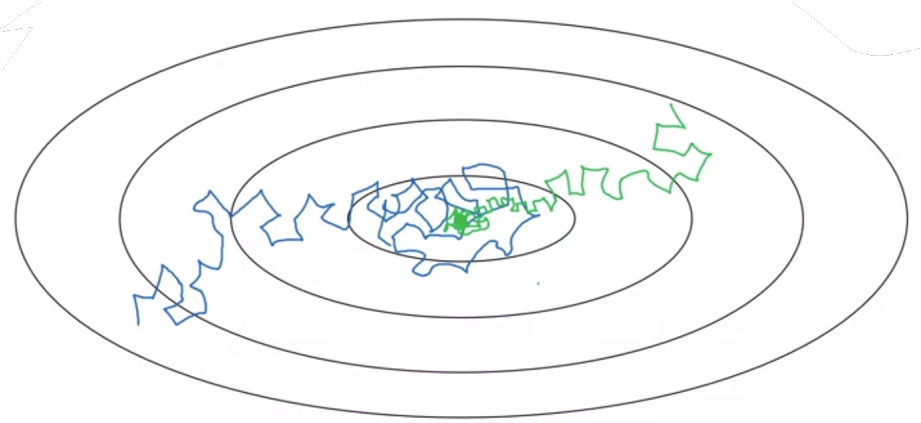

Figure 2.11: Learning rate decay (green)

You can implement learning rate decay as follows. By setting

$$\alpha = \frac{\alpha_0}{1 + \text{decay-rate} \times \text{epoch-number}} \tag{2.26}$$

where α_0 and *decay-rate* are hyperparameters. Other examples are

$$\alpha = 0.95^{\text{epoch-num}} \alpha_0, \quad \alpha = \frac{k\alpha_0}{\sqrt{\text{epoch-num}}}, \quad \alpha = \frac{k\alpha_0}{\sqrt{t}}. \tag{2.27}$$

Local optima

Figure 2.12 illustrates how local optima may cause issues in us trying to minimise the cost function and find the optimum values of w and b.

It turns out that in deep learning most local optima are not as in the right hand figure in Figure 2.12, but instead are saddle points as in the left hand figure. In very high-dimensional spaces, it is very

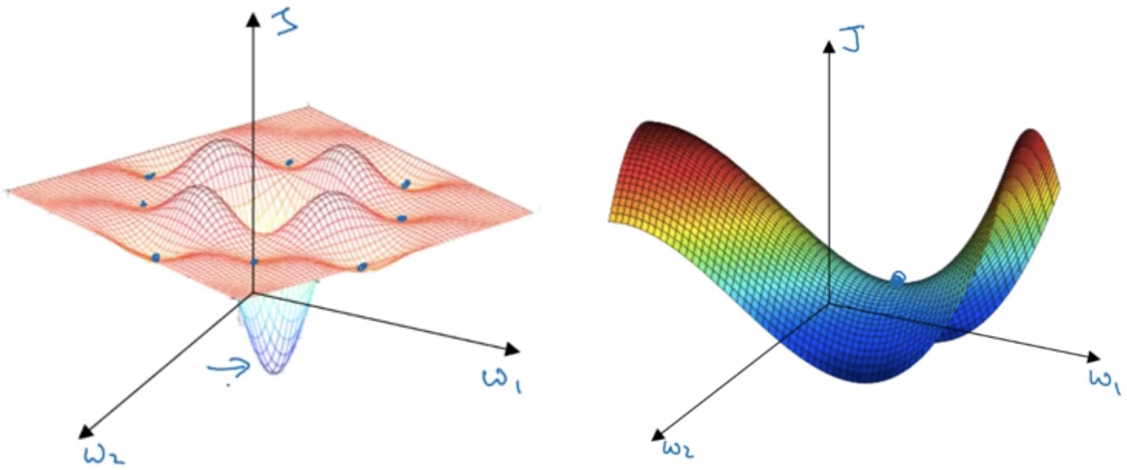

Figure 2.12: Local optimas

unlikely that you will have all dimensions being either convex or concave. Hence it is much more likely to be a saddle point if the gradient is zero.

Another problem that we can encounter in our cost function is regions where the derivative is close to zero everywhere (a *plateau*) as can be seen in Figure 2.13.

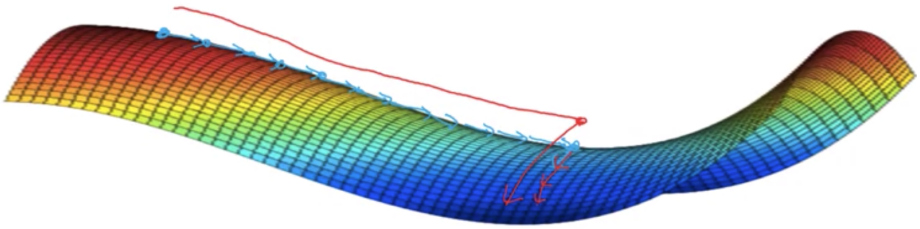

Figure 2.13: It may take a long time to reach the point of zero gradient

Using algorithms like Adam or gradient descent with momentum can help in managing plateaus.

2.2.6 Hyperparameter tuning

So far we have seen many hyperparameters that we could possibly tune. In practice, we'll often be more concerned about tuning certain hyperparameters first, though, as they often make the biggest difference to the performance of our algorithm:

- α - **high importance**;
- β - **medium importance**, often set to 0.9;
- β_1, β_2, ϵ - **very low importance**, often set to 0.9, 0.99, 10^{-8};
- number of layers - **low importance**;
- number of hidden units - **medium importance**;
- learning rate decay - **low importance**;
- mini-batch size - **medium importance**.

But which values should we choose to explore? It is, of course, not recommended to simply create a grid of parameter values for each of our parameters and simply loop over these. For high-dimensional hyperparameter space this could cause serious problems. Instead what is commonly done is we choose

n random points in our hyperparameter space and test how these perform. If there is a particular region of the hyperparameter space that seems to be performing well, then *zoom in* on this region and test more points within it (this is known as **coarse to fine** search). Iterating this process you'll be able to efficiently tune your hyperparameters.

One problem that can arise is the following: common values for α are between 0.0001 and 1. Sampling at random here would be very inefficient and is unlikely to be effective. What we can do instead is sample on a logarithmic scale between these two values, which will spread our search out nicely. One should note that hyperparameter values deteriorate over time - they may need to be updated in regular intervals.

2.2.7 Batch normalisation

Batch normalisation helps to us to find hyperparameters much more efficiently and train deeper neural networks. Previously, we normalised the our input values only. However, we can do the same with our values z (or a) in each layer too! This will help us train our weights w and bias b more efficiently. One can normalise these values to have mean 0, variance 1. Sometimes we won't want this however and instead scale the normalised values appropriately. The calculation is as follows

$$\tilde{z}^{[l]} = \gamma^{[l]} \frac{z^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta^{[l]} \sim N(\beta^{[l]}, \gamma^{[l]\,2}) \tag{2.28}$$

where μ and σ are the sample mean and variance, ϵ has been added to avoid having a singularity, and $\beta^{[l]}$ and $\gamma^{[l]}$ are extra parameters which should be trained just as you train w and b (i.e. using back-propagation). Note that the fraction term in Equation 2.28 has the effect of making the bias terms b redundant. They will always be subtracted away by the mean. We can therefore ignore bias terms when using batch normalisation. That is,

$$z = wa. \tag{2.29}$$

Batch norm is often used on mini-batches. This will hopefully make mini-batch procedures even more efficient!

But why does batch norm work? First, we consider the problem of **covariate shift**. This is where the distribution of your underlying data has changed since you first changed your model (e.g. you trained a detection model on pictures of black cats only, and your new dataset is of pictures of ginger cats!). We cannot expect our algorithm to generalise well in this case. Covariate shift is a problem in neural networks as the distribution of our hidden unit values shifts around when the distribution of our input values changes. Batch norm combats this by reducing the amount the distribution of our hidden units moves by.

Using batch norm with mini-batch gradient descent also has a *slight* regularisation effect. Each mini-batch is scaled by the mean/variance computed on just that mini-batch. The mean and variance are noisy as they have been computed on a sub-sample of our training data. Therefore, each layer's activation will have some noise which results in a slight regularisation effect.

2.3 Structuring Machine Learning Projects

Machine learning strategy revolves around how you can efficiently improve the performance of your system. Examples of this seen so far include

- collect more data;
- use a larger network;
- train algorithm for longer.

Here we shall look at how to determine which strategy may be most effective in improving the performance of our algorithm.

Orthogonalisation is the attribute of a **given parameter** value affecting the model in a **particular way**. If we know of a weakness in our algorithm then ideally we will know which parameter we need to tune in order to fix this weakness. But how can we tell whether performance is improving? The best way to do this is to use a **single value evaluation metric**. For example, rather than using both precision and recall, we can combine them to generate the F-score

$$F = \frac{2PR}{P+R}. \tag{2.30}$$

Having a single value evaluation metric means we can easily compare algorithms and tell if our changes are having the desired affect on performance.

How should we set up our training, development, and testing datasets? First let's consider the development/test set. Of course, we must ensure that our development and test set come from the **same distribution**. This can often be done by simply shuffling the data. It is not so much of a problem if the training set is from a slightly different distribution to the dev/test set, however we must ensure that the dev and test sets are from the same distributions.

One should split their data into training, development and testing sets according to how much data one has overall. If you have truly big data, then it may suffice to use only 1% of the data for your development set and 1% of your data for the testing set and the rest for training. There is no need to follow the 60/20/20 rule when you have a lot of data (say, +1,000,000 samples). Just make sure that the size of your development/test sets are large enough to give confidence in the result.

Something else to consider is that the users of the model may be evaluating the model on data from a different distribution to the one your model has been trained on. For example, your cat classification app may have had a training/development/testing set of nice pictures of cats from the internet that are high resolution. Your users, however, may be trying to evaluate the algorithm using blurry photos of cats that they have taken themselves. This may cause the algorithm to perform poorly when deployed. Considerations like this are important when developing an algorithm and one should try to include all sorts of data in their training/development/testing set when building the model. In the example given here, you would want to put as much of the user data into the dev and test sets as possible, as this is what you want your model to perform well on upon deployment.

2.3.1 Error analysis

It is important to consider/estimate what human-level performance is on a certain task. With what accuracy can a human label pictures of (not) cats? This will give us a benchmark to consider when developing our own model. Looking at the human-level performance can also help us determine whether our model is suffering from high bias/variance or not. For example, if human-level error is at 5% while

- training error is at 5.1% and development error is at 7%, then we could argue the model is suffering from **high variance**;
- training error is at 6.8% and development error is at 7%, then we could argue the model is suffering from **high bias**.

The **Bayes error** is defined as the minimum theoretical error of a given model.

It can often be worth while investigating which samples from the dev/test set your model is failing on For example, is your cat classification model misclassifying 50% of the pictures of dogs going through the model? If so then this may be worth investigating. This is a relatively manual process however a little bit of effort here can potentially lead to large gains in performance. This can also be useful in determining the effect of incorrectly labeled (not necessarily incorrectly classified) on the performance of your model and whether fixing the labels is worth your while.

So what should you do if your training set comes from a different distribution to that of your dev and testing sets? You could create a **test-dev set** that has the same distribution as the training set,

but is not used for training. Instead you treat it as you would the dev set to evaluate performance. You will then be able to compare the performance of the dev set and the training-dev set to see the affect that the difference in distributions is having on performance.

Transfer learning

This is where you use a network previously trained to perform one task to now perform a different task. For example, you have trained a cat classifier and want to use transfer learning to perform radiology diagnoses. This is done by randomly initialising the weights for the final layer and retraining the network. You can do this by either training using the entire network (performing **fine-tuning**) or by just changing the weights for the final layer (using a **pre-trained** network).

Transfer learning makes sense to use when you have a lot of data for the model you're transferring from and not so much data for the model you're transferring to. The two models should, of course, have the same kind of input. If you think that the two tasks have similar low-level features then perhaps transfer learning would be effective too.

Multi-task learning

This is where you have many models working in parallel. For example, in an autonomous-driving system, you will need to check whether there are cars, people, dogs, stop signs etc. in a given frame. The output layer of this network could be a multi-output binary node with the value 1 signifying that feature being present in the image. As oppose to softmax regression, here we are saying that multiple objects can appear in a given image.

Multi-task learning makes sense to use when

- training on a set of tasks that could benefit from having shared low-level features;
- the amount of data you have for each task is quite similar;
- you can train a big enough neural network to do well on all the tasks.

End-to-end deep learning

This is where a chain of operations are combined into a single network. Let's look at a speech recognition example where we take some audio as input and output the transcript of this audio. This may be done using the following process

$$\text{audio} \xrightarrow{\text{MFCC}} \text{features} \xrightarrow{\text{ML}} \text{phonemes} \rightarrow \text{words} \rightarrow \text{transcript}$$

which we then combine to simply have

$$\text{audio} \rightarrow \text{transcript}.$$

The pros to using end-to-end learning are clear: you are working at a higher level and are required to do less hand-designing of components. The cons, however, are that you need a **large amount of data** to perform end-to-end learning and that it excludes potentially useful hand-designed components. You need a lot of data here as you are trying to do a direct mapping between input and output, which is difficult to learn unless you have a lot of data.

Sometimes, however, the task is too complicated, intricate, or perhaps unsafe to perform end-to-end learning on. Take for example autonomous vehicles. You will need the car to detect objects, plan a route, and then steer. This is a non-trivial task and one that end-to-end learning cannot deal with (yet).

2.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are generally applied to image recognition tasks. This can be classification, detection, or even neural style transfer tasks. A problem with the previously mentioned method of classifying images (i.e. via flattening the image) is that higher quality images result in the network having potentially *many* parameters and taking a long time to train. This method is not feasible for training on large images, hence the use of CNNs!

2.4.1 Introduction to CNNs

Edge detection and convolution

Edges in this setting can be thought of as edges of objects in an image (e.g. the edges of a cat in a picture). If you can detect the edges images then you will be able to more easily detect/classify objects. Let's try and detect the edges in some 6×6 grey-scale image: we use convolve the image on a 3×3 filter (or, kernel)

$$\begin{pmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} -5 & -4 & 0 & 8 \\ -10 & -2 & 2 & 3 \\ 0 & -2 & -4 & -7 \\ -3 & -2 & -3 & -16 \end{pmatrix} \tag{2.31}$$

where the convolution operation is done by overlaying the filter on the input matrix and taking the sum of the element-wise product between the two matrices. For example, the top-left value of the output is calculated by

$$3 \times 1 + 1 \times 1 + ... + 8 \times -1 + 2 \times -1 = -5.$$

The above example is a *vertical* edge detector, but why? Let's take a simpler example:

$$\begin{pmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{pmatrix} \tag{2.32}$$

where we take values > 0 to be white pixels, values $= 0$ to be grey pixels, and values < 0 to be black pixels. We can therefore see that the convolution has detected a vertical edge down the middle of the input image. Taking the transpose of the filter seen in Equation 2.32 we get a *horizontal* edge detector. Of course, we would like to detect edges at any orientation we like (not just vertical and horizontal). Not surprisingly, this is possible.

Padding

As we can see from the above, an $n \times n$ image convolved on a $f \times f$ filter will result in a $n - f + 1 \times n - f + 1$ output image. If we have many many convolutions in our network then the resulting image will keep on shrinking meaning we are losing more and more information with each convolution. Another issue with convolution is that the filter goes over the centre pixels many more times than it goes over the pixels at the edge of the input image. Using the examples above, the top-left pixel is gone over only once during convolution whereas a pixel near the centre is gone over as many as nine times. Here, we are throwing away a lot of the information at the edge of the image. To fix the above problems, we **pad** the image before convolving it.

Padding is where we add a border of p pixels around the original image, usually with value 0. If we pad the input image of Equation 2.32 with a border of size 1, then the resulting image would be 6×6, therefore we are preserving the size of the input image. **Valid convolution** padding is where one applies **no padding**. **Same convolution** padding is where one applies padding such that the input image size is the same as the output image size (i.e. $p = \frac{f-1}{2}$). By convention, filters

of odd-dimensions are the most common.

Strided convolution is where the filter is applied by skipping over certain positions of the input image. That is, if you convolve an $n \times n$ input image with a $f \times f$ filter using padding p and stride s, the output image is of size $\lfloor \frac{n+2p-f}{s} + 1 \rfloor \times \lfloor \frac{n+2p-f}{s} + 1 \rfloor$.

Convolutions can also be applied over volumes too (i.e. **3D convolution**). This could be on a RGB image. The difference here is that our input image is 3D, but so is our filter. The images are of dimensions height×width×number of channels. The output of a 3D convolution is still 2D and has dimensions as above (e.g. $(n \times n \times n_C) * (f \times f \times n_C) = (n - f + 1 \times n - f + 1)$).

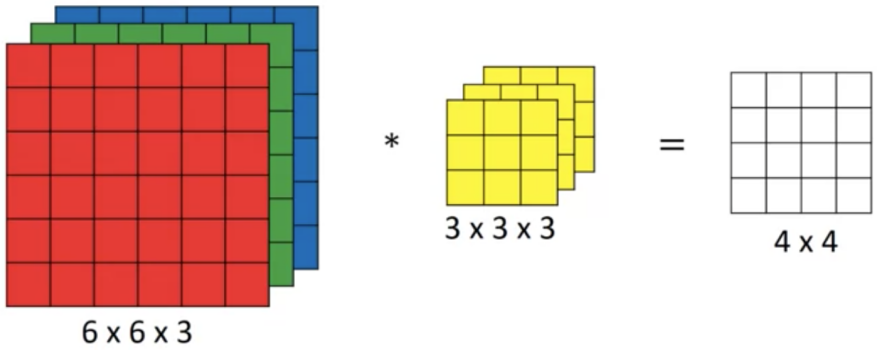

Figure 2.14: 3D Convolution

One can also use **multiple filters** on a single image. For example, say you want to apply both a vertical and a horizontal filter to your input image. You would apply two separate convolutions to the input image and then stack the outputs against each other giving a 2D output image. For example, the dimension of the output seen in Figure 2.14 would be $4 \times 4 \times 2$. Of course, each of the filters applied to the input image must be of the same dimension. The number of channels in the output image is equal to the number of filters applied.

Convolutional Neural Networks

Including a convolution layer in your neural network is not dissimilar to how we have done it before. Recall,

$$\begin{aligned} z^{[i]} &= w^{[i]} a^{[i-1]} + b^{[i]} \\ a^{[i]} &= g(z^{[i]}) \end{aligned} \tag{2.33}$$

where g is some non-linear activation function. In the case of a convolution layer, a is the input image and w is the filter. After convolving, you apply a non-linear function g to the output image plus the bias. In the case of multiple filters, g is applied to each output separately and the results are stacked against each other. The output of the non-linear function is the activation for that layer.

If layer l is a convolution layer, let

- $f^{[l]}$ denote the filter size of layer l;
- $p^{[l]}$ denote the padding of layer l;
- $s^{[l]}$ denote the stride applied in layer l;
- $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$ denote the dimensions of the input image to layer l;
- $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$ denote the dimensions of the output image to layer l.

We can calculate

$$n_{H \text{ or } W}^{[l]} = \left\lfloor \frac{n_{H \text{ or } W}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor. \tag{2.34}$$

For m training examples, we have m activations for that layer i.e. $A^{[l]} = m \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$. Our weights have dimension $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]}$ (that is, the filter size $\times$ number of channels $\times$ number of filters in layer l). Of course, we will have $n_C^{[l]}$ bias values in layer l.

CNNs often comprise a number of convolutional layers (with **pooling** in-between each layer), followed finally by flattening the output of the final convolutional layer and feeding it into a logistic/softmax unit for classification/detection. It is common to include some **dense** layers in CNNs too after flattening. Most of the work here comes in deciding the number of filters, size of padding, stride value, and so on. General hyperparameter tuning here. We will look later at how best to choose the values of these parameters. It is common for the height/width of the image to decrease as it progresses through the network, while the number of channels tends to increase.

The standard CNN architecture is to have a series of

$$\text{convolutional} \rightarrow \text{batch-norm} \rightarrow \text{ReLU} \rightarrow \text{max-pooling layers},$$

followed by some fully-connected layers and finally a softmax output.

Pooling

Pooling layers are often added to CNNs to reduce the size of the computation as well as making feature detection more robust. Let's go through an example of pooling called **max pooling**.

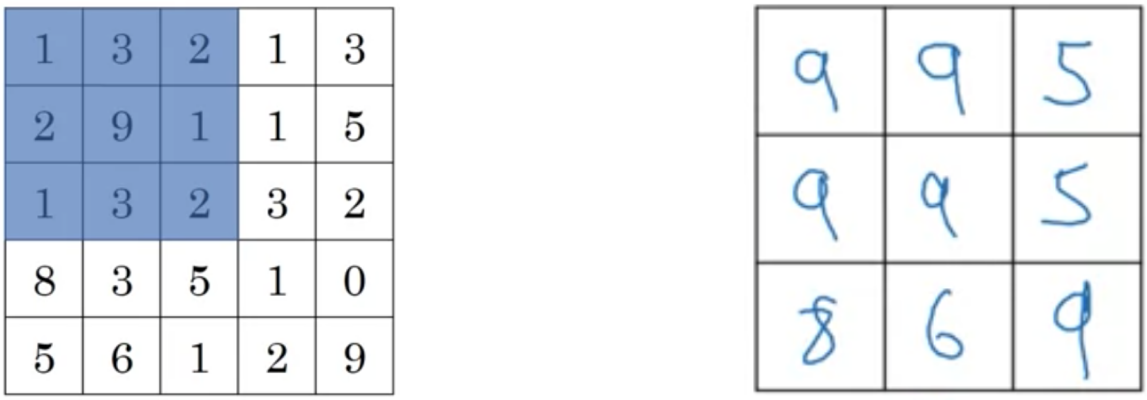

Figure 2.15: Max pooling

Pooling attempts to grab certain features while throwing away less useful information. Perhaps the maximum value of the image pixels is of some importance. Max pooling has two hyperparameters: one for filter size (3 in Figure 2.15) and another for stride (1 in Figure 2.15). Equation 2.34 can be used here too for calculating the size of the output of a max pooling layer. Similarly to max pooling, one can also apply **average pooling**. Max pooling is much more common in CNNs except for when you want to collapse the height and weight of a given layer down to 1×1. Note that there are no parameters to learn in pooling layers; it is just a straightforward computation.

Why convolutions?

There are two main advantages to using convolutional layers over simply using fully connected layers:

- **parameter sharing**: a feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image. That is, the filters we use are often small;

- **sparsity of connections**: in each layer, each output value depends only on a small number of inputs (i.e. the size of the filter being applied).

Note also that CNNs are **translation invariant**. Place the cat anywhere you like in the picture, the features will still be picked up by the filter.

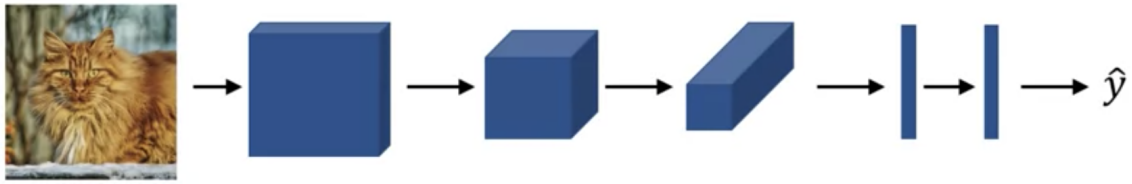

Figure 2.16: Standard CNN: convolutional + pooling layers, followed by some fully-connected layers and a softmax output layer

2.4.2 Case Studies

One of the best ways to gain intuition on how to build conv-nets is to see examples. Often is the case that a given network architecture that works well on one task will work well on another similar task too. We will look at case studies that use the following networks:

- LeNet-5;
- AlexNet;
- VGG-16;
- ResNet;
- Inception.

LeNet-5

This architecture comes from Gradient-Based Learning Applied to Document Recognition, LeCun et al., 1998. LeNet-5 has 7 layers: conv, pool, conv, pool, fully-connected, fully-connected, softmax output. The original network had about 60,000 parameters. The idea here is to decrease the height and width of the activations through the matrix while increasing the number of channels. Due to the age of the LeNet-5, sigmoid/tanh activation functions were used at the time of writing rather than ReLU.

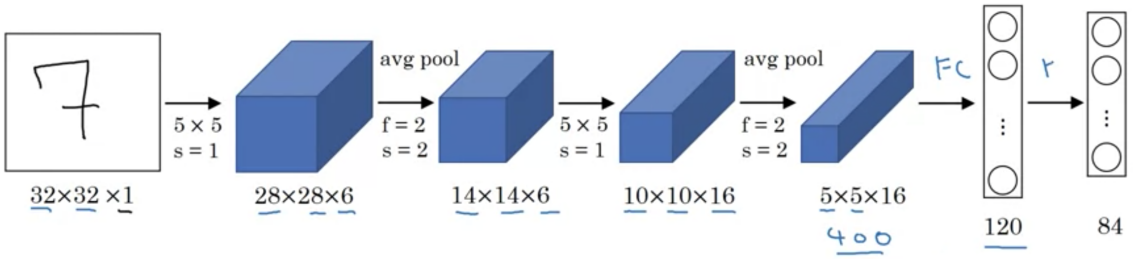

Figure 2.17: LeNet-5 neural network

AlexNet

This architecture comes from ImageNet Classification with Deep Convolutional Neural Networks, Krizhevsky et al., 2012. Although it looks very similar to LeNet-5, there are some key differences:

- much bigger network (about 60 million parameters);
- uses ReLU activations.

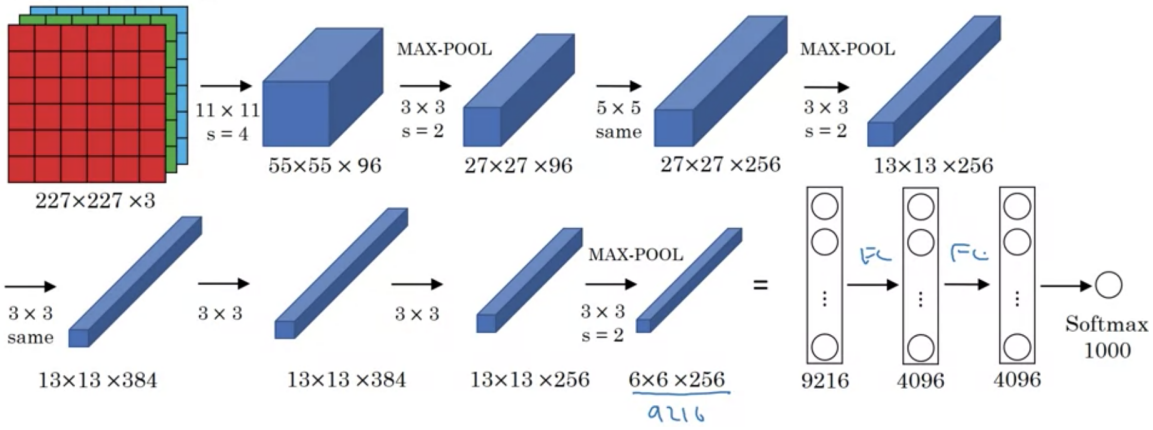

Figure 2.18: AlexNet neural network

VGG-16

This architecture comes from Very Deep Convolutional Networks for Large-Scale Image Recognition, Simonyan & Zisserman, 2015. The notation "[CONV X] × Y" seen in Figure 2.19 denotes Y convolution layers with X filters. Each filter in the network is 3 × 3, has a stride of 1 and is a *same-convolution*. This network has about 138 million parameters! The architecture, however, is simple.

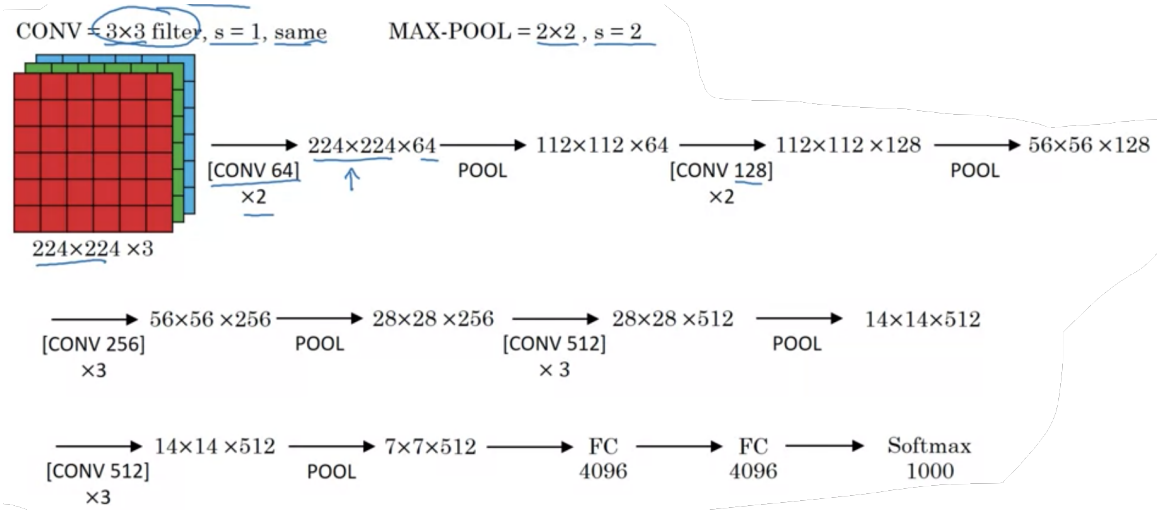

Figure 2.19: VGG-16 neural network

ResNet

Very very deep networks suffer from vanishing/exploding gradients. The ResNet uses **skip-connections** which allows you to take the activation from one layer and feed it into another layer deeper into the network. This speeds learning by reducing the impact of vanishing gradients, as there are fewer layers to propagate through. The network then gradually restores the skipped layers as it learns the feature space. This architecture comes from Deep Residual Learning for Image Recognition, He et al., 2015.

ResNets are built out of a **residual block**. Take 3 activations $a^{[l]}$, $a^{[l+1]}$ and $a^{[l+2]}$. Usually, we have that

$$a^{[l]} = g(z^{[l]}) \tag{2.35}$$
$$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]} \tag{2.36}$$
$$a^{[l+1]} = g(z^{[l+1]}) \tag{2.37}$$
$$z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]} \tag{2.38}$$
$$a^{[l+2]} = g(z^{[l+2]}) \tag{2.39}$$

A residual block will instead take Equation 2.35, skip a few connections, and add it to Equation 2.38. That is, we have the residual block

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]}). \tag{2.40}$$

Note that $a^{[l]}$ has been injected *after* the linear transformation but *before* the non-linear transformation. We can also create skip-connections that skip across many layers at once. You will need to ensure that $a^{[l]}$ and $a^{[l+2]}$ have the same dimensions in order for this to work. To do this, you can multiply $a^{[l]}$ by some matrix W of zeros and ones that pads $a^{[l]}$ to have the same dimension as $a^{[l+2]}$.

If $z^{[l+2]} = 0$ and $a^{[l]} > 0$, then we have the $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$, since g is the ReLU function. It is therefore easy for a residual block to learn the identity function. Hence we are simply passing activations forward through the network in this case by adding skip-connections. At the very least, if your network learns the identity function in one of the layers then it won't hurt performance. Sometimes you'll get lucky and your network will learn a slightly different function that improves performance.

An *identity block* is where $a^{[l]}$ is passed forward without any transformation. Another common type of block is the *convolution block*, which is a convolution (and batch normalisation) is applied to $a^{[l]}$ before passing it into the non-linearity deeper in the matrix.

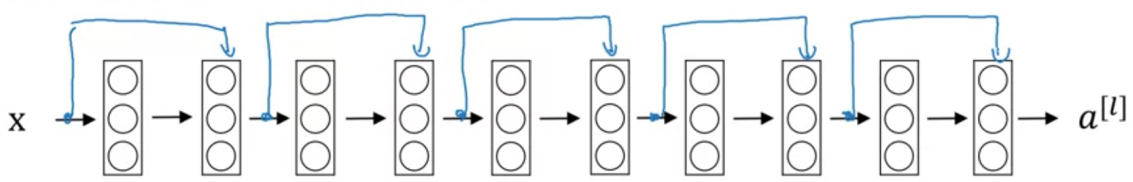

Figure 2.20: ResNet neural network

As mentioned above, ResNets help deeper networks to perform better than their *plain* counterparts. Typical ResNet models are implemented with double- or triple-layer skips that contain nonlinearities (ReLU) and batch normalization in between. An additional weight matrix may be used to learn the skip weights; these models are known as *HighwayNets*. Models with several parallel skips are referred to as *DenseNets*. In the context of residual neural networks, a non-residual network may be described as a plain network.

2.4.3 Network in Network and 1×1 convolutions

What is a 1×1 convolution? Essentially, this is just multiplying the input by the given value. This is useful when we have *many* (e.g. 32) channels. We then apply a ReLU non-linearity to the result of each channel. We can then build this up even further by using more filters. 1×1 convolutions, or *Networks in Networks*, are described in Network in Network, Lin et al., 2013.

The example below is reducing the number of channels for a given input. We apply a $1 \times 1 \times 192$ convolution with 32 filters to get the resulting image. If we used 192 filters instead of 32, then we

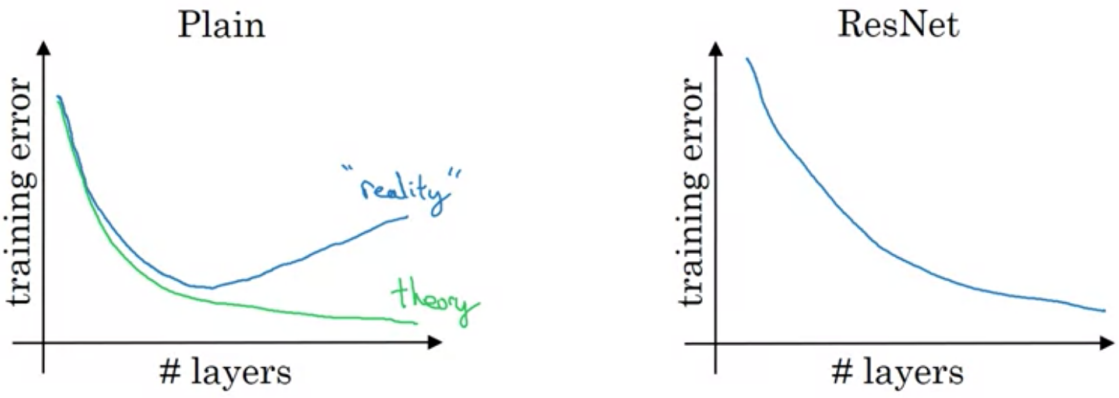

Figure 2.21: ResNet neural network performance

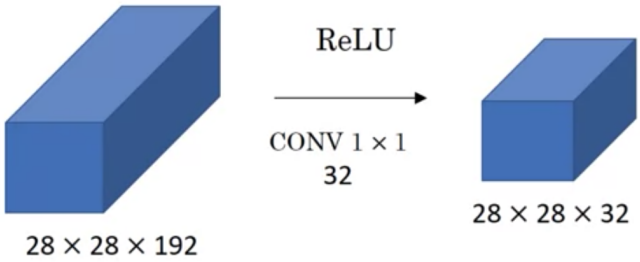

Figure 2.22: Network in Network

would output a $28 \times 28 \times 192$ image, where the affect of this layer would be adding the non-linearity in the ReLU activation.
Therefore, we can use

- pooling to shrink the height and width of an image,
- networks in networks to shrink the number of channels.

2.4.4 Inception Network

When adding a convolution layer to our network, we need to decide whether to use a 3×3 filter, or a 5×5 filter, or perhaps even a pooling layer instead? What the Inception Network does is all of them! This makes the architecture much more complicated but boosts performance!

The architecture comes from Going deeper with convolutions, Szegedy et al., 2014. Here, we apply different filters (same convolutions) and pooling layers (note that padding will have to be applied to the pooling layer in order to keep the output dimensions consistent) each to the input image to the layer. We stack the results and essentially let the network learn how to weight each of these layers. The obvious problem with this network is going to be computational cost. This is where we can use a 1×1 convolution in order to improve efficiency. As an example, if we use a 5×5 same convolution with 32 filters on a $28 \times 28 \times 192$ image, then this will result in $(28 \times 28 \times 32) \times (5 \times 5 \times 192) \approx 120m$ multiplications. The output image is $28 \times 28 \times 32$. On the other hand, if we were to use a 1×1 convolution first (often referred to as a *bottleneck layer*) to reduce the number of channels in the input image to, say, 16 channels, and then run a 5×5 convolution on the resulting $28 \times 28 \times 16$ volume to return our output $28 \times 28 \times 32$ image. This only requires only $12.4m$ multiplications - much better! As long as the bottleneck layer is used *reasonably*, performance will be generally unaffected and computational cost massively reduced.

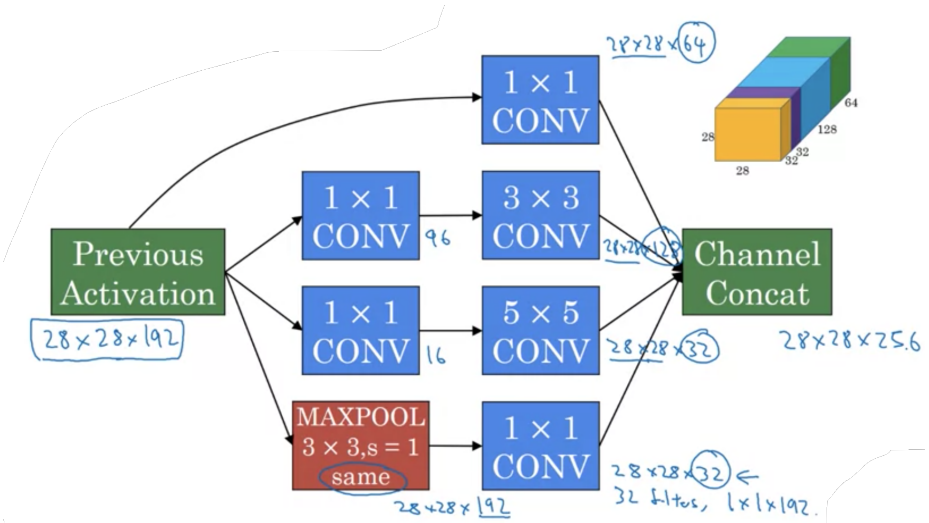

Figure 2.23: Inception Module

Figure 2.23 shows one Inception *module*. To form the Inception *network*, one must combine many such modules together. The Inception network is often referred to as the *GoogLeNet*.

2.4.5 Practical Advice on ConvNets

Transfer Learning

The open-source community is strong within the machine learning world. Many papers have the relevant code posted on GitHub, and there are many pre-trained networks available for the public to use in their own work. Using a pre-trained network means you don't have to spend any time training the network and need only do minor tuning of the final layer for your use case. This is an example of *transfer learning*. The user should consider the weights of each layer of the network as *frozen*, except for the output layer. The output layer is the only one which should be trained when you are using a pre-trained network. Training here should be fast given you are only having to learn one layer. Adaptations to the final layer may be necessary for the use-case also (e.g. changing the number of potential outputs for a softmax output layer).

To make your computation even faster here, the user could pre-compute the result of all of the layers preceding the output layer for each example in your training set, given you know it isn't going to change. This way, the network would only have 2 layers - one hidden layer and one output layer.

Another option for the user is to train not only the output layer, but the final n layers. If you have enough data, then this approach can be fruitful. Furthermore, you could use the weights you download as an initialisation for the network and train over all layers. Again, you will need plenty of data for this to be effective.

Data Augmentation

The most common data augmentation method in computer vision is *mirroring*. Another is taking *random crops* of the image. This is usually not as good as mirroring as you cannot guarantee that your image will contain a clear image of your target (e.g. you could cut out half of a cat). *Colour shifting* is where you add different distortions to each of the RGB channels in the image. In practice, these distortions are often taken from some distribution, say $N(0, 1)$. Alternatively, one can perform *PCAA colour augmentation*, which is where you distort according to the presence/lack of colour in a certain channel. For example, if green was lacking compared to red and blue, then PCA colour augmentation would distort red and blue much more than it would green. This is in an attempt to keep the overall colour/tint of the image the same.

Data augmentation is less about simply having more data in your dataset, and more about artificially exposing your algorithm to pictures in different scenarios. You don't want your algorithm to fail if you flip an image, nor do you want it to fail if your image is instead taken from a sunnier day! The above methods all try to give your algorithm more experience to learn from.

2.4.6 Object Detection

We first look at *object localisation*. We have already looked at image classification, which is where we say what is contained in the image. Object localisation takes this idea one step further and creates a bounding box around the classified item in the image. *Object detection* is where there may be multiple instances of multiple classes of object in a given image, and your algorithm is tasked with localising each object in the image. We will look at object detection later.

We used ConvNets to classify images with a softmax output. To localise the object, we can extend the output layer to contain outputs b_x, b_y, b_h and b_w i.e. the dimensions of the bounding box. We will use the convention that the top-left corner of any image has coordinates $(0, 0)$ and the bottom-right corner has coordinates $(1, 1)$. (b_x, b_y) denotes the centre of the bounding box with height b_h and width b_w.

We define the target label y as

$$y = [p_c, b_x, b_y, b_h, b_w, c_1, c_2, ...] \in \mathbb{R}^n \tag{2.41}$$

where $p_c = \mathbb{P}[\text{image contains object}]$ and $c_i = \mathbb{P}[\text{bounded object is of type } i]$. If $p_c = 0$ then the rest of the vector y can be filled with `NaN`. We can use the loss function

$$\mathcal{L}(\hat{y}, y) = \begin{cases} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2, & y = 1 \\ (\hat{y}_1 - y_1)^2, & y = 0 \end{cases} \tag{2.42}$$

where we are starting indexing at 1. Equation 2.42 is a simplified loss function. We could instead use, perhaps, a binary cross-entropy loss for p_c, a squared error loss for the bounding box, and a softmax loss for the softmax output.

Landmark detection

We may want to detect the location of certain landmarks in the image e.g. the locations of the corners of someone's eyes within an image. Perhaps you even want to locate all of the key landmarks of a face e.g. the location of the persons ears, nose, edges of the mouth and face. We can use a ConvNet with a softmax output for landmark detection. This could be useful for detecting someone's mood, or adding a Snapchat filter to a photo, or even detecting the pose type of a photo e.g. is someone running in this photo?

Sliding Window Detection

Here, we attempt to detect objects within an image. We can do this using the *sliding window* detection algorithm. This is where you input a window of an image into your ConvNet and classify whether the window contains an object of interest or not. You then proceed to *slide* this window through the entire image, detecting in each window whether or not it contains an object or not. After running this for a given sized window, you run the algorithm again and again each time using different sized windows. The hope is that you will be able to catch each of the objects in the image. Any detected image will then be bounded.

Of course, there is a massive computational cost involved in this technique, as you have to run many windows through your ConvNet many many times. There is a way to fix this by applying the sliding window algorithm *convolutionally*. Let's investigate what this means.

We can turn fully-connected layers into convolution layers by using *filters* rather than flattening an image into a fully-connected layer. This can be seen in Figure 2.24.

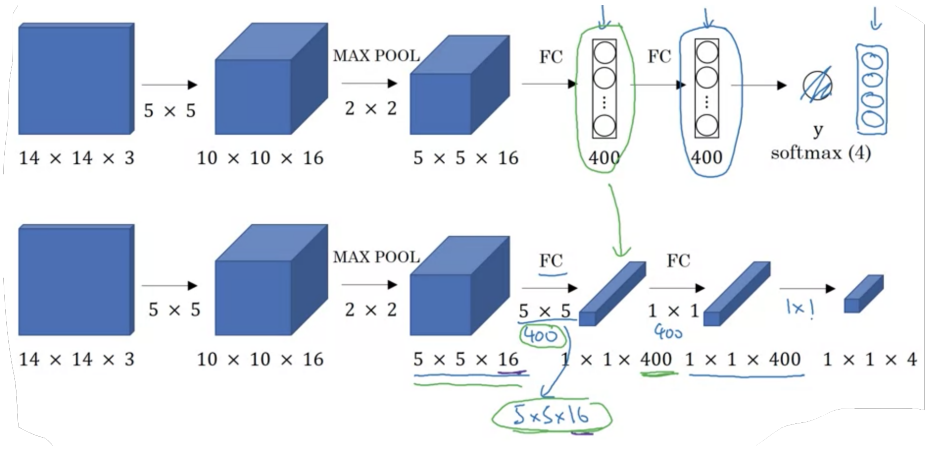

Figure 2.24: Fully-connected layer → convolutional layer

Figure 2.25 shows how we can use this idea to efficiently run our sliding window algorithm on an image. That is, we are taking a window that we slide four times on the original image. After running the ConvNet, we are left with a $2 \times 2 \times 4$ image, where each pixel of the 2×2 face of the output is the result of running the sliding window algorithm. We are left with a 2×2 face since we have used a 2×2 max pooling in the ConvNet i.e. we have run the neural network on the original image with a stride 2. Doing this has allowed us to make all of the predictions at the same time through one forward pass of our ConvNet, rather than running each one sequentially.

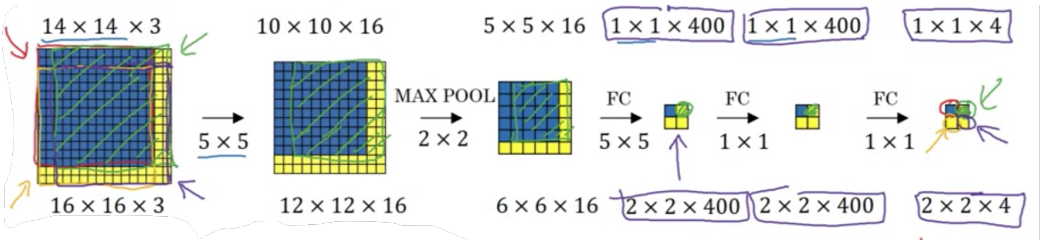

Figure 2.25: Classifying using sliding window and fully-connected layer → convolutional layer

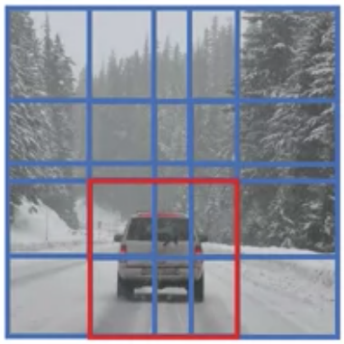

Figure 2.26: Sliding window algorithm detecting a car in the image

2.4.7 YOLO (You Only Look Once) Algorithm

We have the problem that we still cannot guarantee accurate bounding boxes. What if our window doesn't align nicely with the position of the car, say, in the image? We can use the YOLO algorithm (see You Only Look Once: Unified, Real-Time Object Detection, Redmon et al., 2015) to help us here.

This is where you place a grid on the image and apply an image classification localisation algorithm to each of the cells in the grid. For each of the cells, you specify a label y as in Equation 2.41. The YOLO algorithm takes the midpoint of any object found in the image and assigns the object to the cell of the grid that contains the midpoint of that object.

Take the example where we are searching for one of a car, motorbike or person in an image. Therefore $y \in \mathbb{R}^8$. If we apply the YOLO algorithm with a 3×3 grid then we will be left with a $3 \times 3 \times 8$ output i.e. one label vector $y \in \mathbb{R}^8$ for each of the 9 cells in the grid. See Figure 2.27.

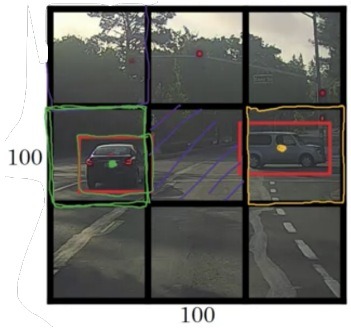

Figure 2.27: YOLO algorithm

Using the example seen in Figure 2.27, our algorithm takes a $100 \times 100 \times 3$ input image and runs it through a ConvNet (convolutions, max pooling...) such that we end up with a $3 \times 3 \times 8$ image. We then use backprop to train the algorithm to map from some input image to our output volume. The advantage to this algorithm is that the output contains precise bounding boxes for each of our objects/cells. We will address the issue of multiple objects per cell later. In practice, people tend to use much finer grids (e.g. 19×19) in the YOLO algorithm which reduces the chances of a given cell containing multiple objects. Again, here we only have to run an image through our ConvNet once - hooray!

How do we encode b_x, b_y, b_h and b_w in our output label? The convention is to take each of our cells as a sub-image of our input image. That is, we take the top-left corner of any cell in our image as having position $(0, 0)$ and the bottom-right as having position $(1, 1)$. b_x and b_y will, of course, always be between 0 and 1. b_h and b_w, on the other hand, could be greater than 1. There are other more complicated parameterisations possible here, but we will use this one.

Intersection Over Union

How can we tell if our object detection algorithm is working well? *Intersection Over Union* (IoU) is a function we can use to both evaluate our object detection algorithm as well as to add as another component to our algorithm to improve its performance.

In object detection, we are expected to both classify and locate (i.e. bound) objects within an image. How can we tell whether our bounding box has been well-placed or not? We can look at the intersection area divided by the union area of our bounding box v.s. the ground-truth bounding box. In general, computer vision tasks will judge that the answer is correct if IoU ≥ 0.5.

Non-max suppression

A problem with our approach so far is that there is a high chance that our algorithm will detect a single object more than one time upon completion. *Non-max suppression* is a way to make sure our algorithm detects each object only once. For example, if we use a very fine grid for object detection in Figure 2.27, then we may have many cells within the grid claiming they detect the blue car. We want just one detection per object.

Non-max suppression calculates the probability that the detected object is in fact in this cell. The cell with the highest probability is taken as containing the object, and those cells surrounding this

one with a high IoU with this cell (e.g. IoU ≥ 0.5) will be suppressed. Cells with probability of object being in that cell less than some threshold (e.g. 0.6) will be discarded. If you have multiple output classes, then non-max suppression should be run across each of these classes.

Anchor Boxes

So far, each of our grid cells have only been able to detect one object at a time? What if we wanted a cell to be able to detect multiple objects? This is where we can use *anchor boxes*. Here, we extend our output vector y depending on the number of anchor boxes we have. For example, if take the image seen in Figure 2.28, both the person and the car are centred in the same cell (bottom middle).

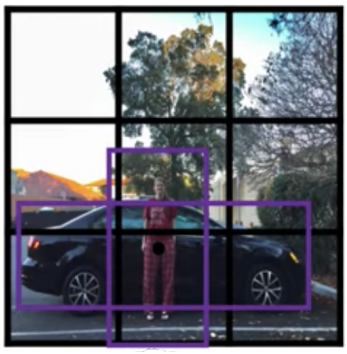

Figure 2.28: The person and the car are centred in the same cell

The purple boxes are the anchor boxes with centres at the black dot. Let's say anchor box 1 is the vertical box (i.e. for the pedestrian) and anchor box 2 is the horizontal box (i.e. for the car). If our algorithm is detecting pedestrians, cars, and motorcycles (i.e. c_1, c_2, and c_3, respectively), our y vector now looks like

$$y = [\underbrace{p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3}_{\text{anchor box 1}}, \underbrace{p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3}_{\text{anchor box 2}}] \tag{2.43}$$

That is, we encode anchor box 1 to detect the pedestrian, and anchor box 2 to detect the car. The anchor box that results in the highest IoU with the ground-truth bounding box is assigned to the object. We therefore have that objects are assigned to the pair (grid cell, anchor box) and that is how objects are encoded in the target label. Our output for Figure 2.28 is therefore $3 \times 3 \times 2 \times 8$. Carrying on with our example, we expect

$$y = [\underbrace{1, b_x, b_y, b_h, b_w, 1, 0, 0}_{\text{anchor box 1}}, \underbrace{1, b_x, b_y, b_h, b_w, 0, 1, 0}_{\text{anchor box 2}}]. \tag{2.44}$$

It is down to the researcher to choose the size of the anchor boxes that they think will best capture the objects likely to come up in the images they are looking at. An advanced way of choosing these dimensions is to use the K-means clustering (see Section 1.7.1) algorithm to group together the types of object shapes you tend to see in your image set. We are not restricted in the number of anchor boxes we use.

YOLO Algorithm

We shall now combine all of the techniques seen above to form the *YOLO algorithm*. We shall use the example of detecting a pedestrian, car, and motorcycle (i.e. c_1, c_2, and c_3, respectively) here. Again, we have $y \in \mathbb{R}^{3\times3\times2\times8}$ (or $y \in \mathbb{R}^{3\times3\times16}$ if you flatten the output).
In Figure 2.29, we have

$$y = [\underbrace{0, ?, ?, ?, ?, ?, ?, ?}, \underbrace{0, ?, ?, ?, ?, ?, ?, ?}]. \tag{2.45}$$

for the top-left cell, for example, and

$$y = [\underbrace{0, ?, ?, ?, ?, ?, ?, ?}, \underbrace{1, b_x, b_y, b_h, b_w, 0, 1, 0}]. \tag{2.46}$$

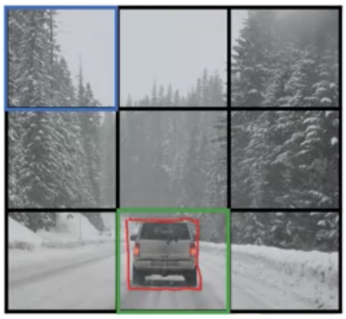

Figure 2.29: YOLO algorithm example

for the bottom middle cell.

We train a ConvNet that takes in a $100 \times 100 \times 3$ image, for example, and outputs a $y \in \mathbb{R}^{3\times3\times16}$. For each grid cell, we get 2 predicted bounding boxes. We now apply non-max suppression by getting rid of the low probability predictions, then for each class use non-max suppression to generate final predictions.

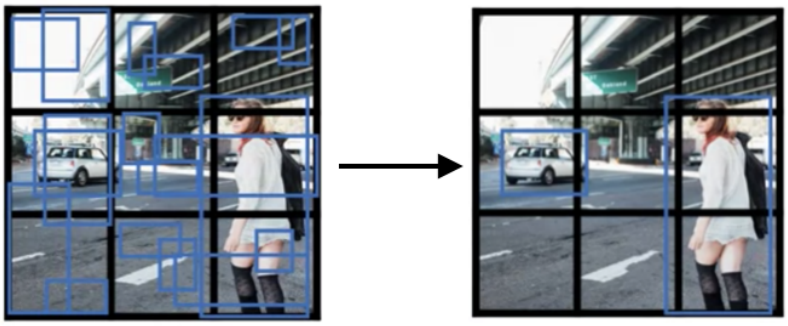

Figure 2.30: Non-max suppressed outputs

I implore readers to check out the YOLO website. It is an experience.

2.4.8 Facial Recognition

Facial Recognition

Here we shall look into developing a face-recognition algorithm. An interesting extension of this idea is liveness detection, detecting whether we are looking at a live image of a face, or a photo.

Face verification is where, given an image and name/ID, the algorithm checks whether the input image is that of the claimed person. *Face recognition*, on the other hand, is much harder. Given a database of K persons and an input image, the algorithm outputs the ID of the person if contained in the database (or *not recognised*, otherwise). As K gets larger, this increases the chances of us making a mistake and requires a higher accuracy rate compared to a database with smaller K in order to trust the algorithm.

One of the main challenges in facial recognition is the fact that we have to solve the one-shot learning problem. That is, we have to learn what the person looks like given potentially only one photo of that person! For example, we might want to build a facial recognition system for allowing entry to a building based on an employee's photo stored in the company database. Building a CNN with a softmax output doesn't make sense here. If we have n people working at the company, then we will train a CNN with a $n + 1$-dimensional output. What if someone joins/leaves the company? We would then need to re-train our network. This is not sustainable. Instead, we aim to learn a

similarity function.

A similarity function is of the form

$$d(img1, img2) = \text{degree of difference between images.} \tag{2.47}$$

Hence, for some threshold $\tau \in \mathbb{R}$, if $d(img1, img2) \leq \tau$ we predict the two images are of the same person. This can be used for the *face verification* problem. For *face recognition*, we would calculate d for each of our database images against the input image. Hopefully, we will have at most one match such that $d \leq \tau$ (zero matches if person is not in database). Using Equation 2.47 allows us to easily add/subtract persons from our database. But how do we train d? We can do this using a *Siamese network* (see DeepFace: Closing the Gap to Human-Level Performance in Face Verification, Taigman et al., 2014).

We are used to having ConvNets as in Figure 2.31. That is, we have a series of convolution → batch-norm → ReLU → max-pooling layers, followed by some fully-connected layers and finally a softmax output.

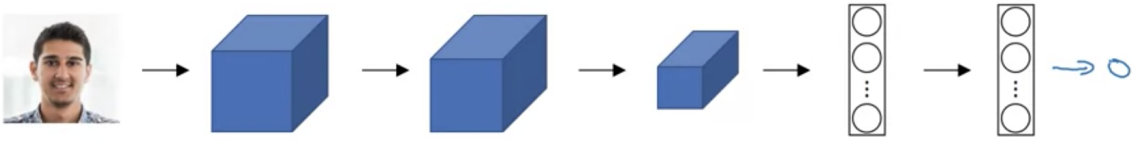

Figure 2.31: Common ConvNet architecture

Here, we take a network without a softmax output and instead take the final fully-connected layer as our output. Let's take the example where this layer has 128 units and we shall call this output $f(x^{(1)})$ (i.e. encoding of input image $x^{(1)}$). Given an image $x^{(2)}$ that we want to compare to $x^{(1)}$, we feed $x^{(2)}$ through the same network with the same parameters and compare $f(x^{(1)})$ to $f(x^{(2)})$. We can then define

$$d(x^{(1)}, x^{(2)}) = \left\| f(x^{(1)}) - f(x^{(2)}) \right\|_2^2. \tag{2.48}$$

The process of running two images through the same neural network and comparing their outputs leas to the name *Siamese network*. We train our network by learning parameters such that if $x^{(i)}$ and $x^{(j)}$ are of the same/different person then Equation 2.48 is small/large.

Triplet Loss

Let's be more specific. We can learn a Siamese network using the *triplet loss function*. See FaceNet: A Unified Embedding for Face Recognition and Clustering, Schroff et al., 2015. Here, we look at three images at one time; an *anchor* (ground-truth), *positive* (picture of same person as in anchor) and *negative* image, denoted A, P and N, respectively . We want

$$d(A, P) = \|f(A) - f(P)\|^2 \leq \|f(A) - f(N)\|^2 = d(A, N) \tag{2.49}$$

Equation 2.49, however, can result in us learning poorly (e.g. we could just learn equality in Equation 2.49 for all images). We therefore include a *margin* parameter $\alpha \in \mathbb{R}$

$$\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha \leq 0. \tag{2.50}$$

We define our loss function as

$$\mathcal{L}(A, P, N) = \max(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0) \tag{2.51}$$

and our cost function as

$$J = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)}). \tag{2.52}$$

Of course, for the purpose of training the system we require multiple images of the same person(s) in the training set in order to be able to learn what features reflect similarities best. When predicting,

we do not require this, of course (i.e. we use one-shot predictions).

One of the problems with choosing A, P and N randomly when training the model is that Equation 2.50 is very easily satisfied. What we instead want to do is choose triplets that are *hard* to train on i.e. choose people that look similar but are in fact different people. That way our algorithm will be able to differentiate between people much more accurately.

Face Verification and Binary Classification

Another way to learn a Siamese network is simply by framing face verification as a binary classification problem.

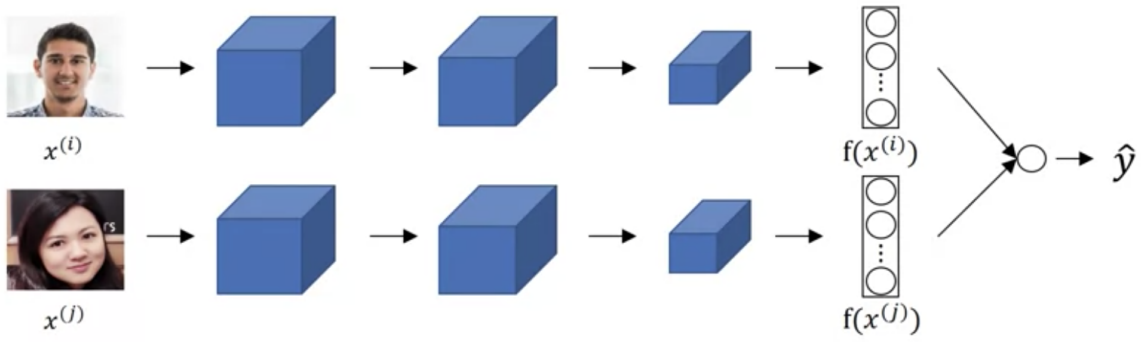

Figure 2.32: Siamese Network and Binary Classification

Here, our output layer is a logistics regression unit returning 1 if $x^{(i)}$ and $x^{(j)}$ are images of the same person, and 0 otherwise. Specifically, we predict, using the sigmoid function σ,

$$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_k \left| f(x^{(i)})_k - f(x^{(j)})_k \right| + b\right) \tag{2.53}$$

training the weights w_k and bias b. Instead of $\left|f(x^{(i)})_k - f(x^{(j)})_k\right|$, practitioners sometimes use the *chi-squared function/similarity/distance* too:

$$\frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k}. \tag{2.54}$$

Again, our networks are Siamese networks i.e. they each have the same parameters. As oppose to when we use the triplet loss, here we only need pairs of labelled images for training.

2.4.9 Neural Style Transfer

Neural style transfer is where you regenerate a given image in a specific style. For example, we can take a landscape image and regenerate it in the style of The Scream, as can be seen in Figure 2.33. Specifically, we take some *content* image (C), a *style* image (S), and *generate* a new image (G). To help us in our quest for neural style transfer, let's take a deeper dive into what deep ConvNets are really learning.

What are deep ConvNets learning?

Let's start with an example. Figure 2.34 shows an AlexNet-like network, and suppose we want to visualise what the hidden units in each layer are computing. The following is based off of the work in Visualizing and Understanding Convolutional Networks, Zeiler and Fergus, 2013.

We start by picking a unit in layer 1. We find the n (e.g. 9) image patches that maximise the unit's activation. We repeat this for other units in the layer. Figure 2.35 visualises this for 9 example image patches for 9 different neurons in 5 different layers. As you can see, each neuron seems to

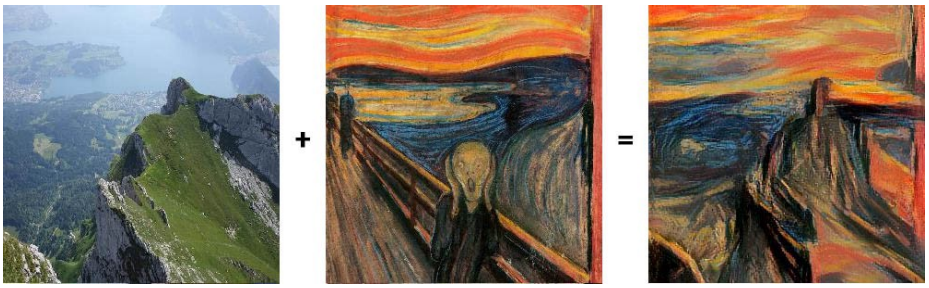

Figure 2.33: Example of Neural Style Transfer

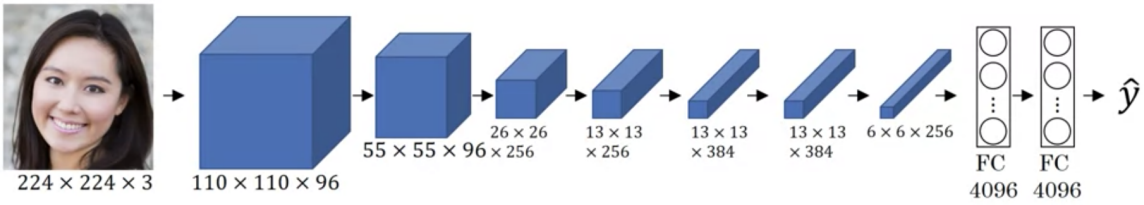

Figure 2.34: What are deep ConvNets learning?

be picking out different kinds of features from the image, such as edges of different orientations and colours.

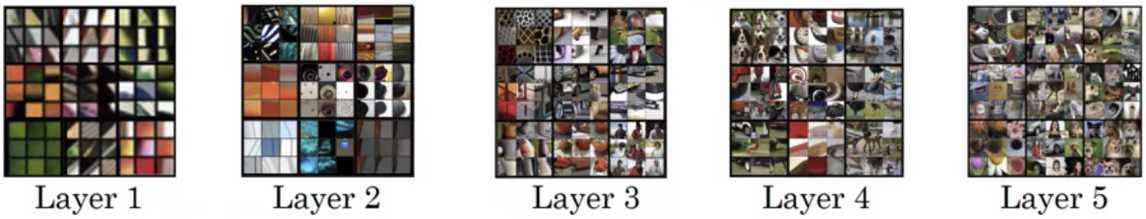

Figure 2.35: 9 image patches from 9 example neurons in a 5 different layer

Looking at Figures 2.34 and 2.35, it is obvious that the neurons in the deeper layers of the network will be looking at larger patches of the input image (due to the transformation of dimensions). Here, a single pixel will have a much smaller affect on the value of the output than a single pixel in the input layer.

Cost function for Neural Style Transfer

We define the cost function here as

$$J(G) = \alpha J_{\text{content}}(C, G) + \beta J_{\text{style}}(S, G) \tag{2.55}$$

which we minimise using gradient descent (or some other optimisation algorithm). $J_{\text{content}}(C, G)$ measures how similar the content of image G is to the content of image C, and $J_{\text{style}}(S, G)$ measures how similar the style of image G is to the content of image S. The neural style algorithm that we look at below is from the paper A Neural Algorithm of Artistic Style, Gatys et al., 2015.

The nueral style algorithm is as follows.

1. Initiate G randomly e.g. $100 \times 100 \times 3$;
2. Use gradient descent to minimise $J(G)$ using

$$G = G - \frac{\partial}{\partial G} J(G).$$

Let's look at how to define J_{content}. Say you use a hidden layer l to compute the content cost. If l is a small number then it would force the generated image to have very similar pixel values to that of the content image. On the other hand, if l is closer to the end of the network, then the content requirements will be much less strict e.g. if there's a dog in the content image then make sure there's a dog *somewhere* in the generated image. In practice, l is usually chosen to be somewhere in the middle of the layers of the network. Using a pre-trained ConvNet (e.g. VGG network), we look at how similar the activations $a^{[l](C)}$ and $a^{[l](G)}$ are to check how similar the images are in content. Hence, we define

$$J_{\text{content}}(C,G) = \frac{1}{2}\left\|a^{[l](C)} - a^{[l](G)}\right\|^2 \tag{2.56}$$

where $a^{[l](C)}$ and $a^{[l](G)}$ have been unrolled into vectors.

What about defining J_{style}? Say we are using layer l to measure the *style* of a given image. We define the style as *the correlation between activations across different channels*. But why do these values capture style? Looking at Figure 2.35, this is made clear; similar parts of the image are grouped together in each of the layers' activations.

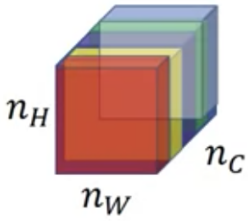

Figure 2.36: Each channel corresponds to a different feature detector. $G^{[l]}$ measures the degree to which the activations of different feature detectors in layer l correlate

Let $a^{[l]}_{ijk}$ denote the activation at (i,j,k), where (i,j,k) correspond to dimensions (H,W,C) of the image. Define $G^{[l]} \in \mathbb{R}^{n_C^{[l]} \times n_C^{[l]}}$ (not to be confused with our generated image G) where, letting $k,k' = 1,...,n_C^{[l]}$,

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](S)}_{ijk} a^{[l](S)}_{ijk'} \tag{2.57}$$

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](G)}_{ijk} a^{[l](G)}_{ijk'} \tag{2.58}$$

where G is commonly referred to as the Gram matrix. The cost function is defined to be

$$J^{[l]}_{\text{style}}(S,G) = \frac{1}{(2n_H^{[l]} n_W^{[l]} n_C^{[l]})}\left\|G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'}\right\|^2_F \tag{2.59}$$

where F denotes the Frobenius norm. Often, better results are given when combining the cost function from multiple layers. This results in the new cost function

$$J_{\text{style}}(S,G) = \sum_{l=1}^{L} \lambda^{[l]} J^{[l]}_{\text{style}}(S,G) \tag{2.60}$$

where $\lambda^{[l]}$ is a set of hyper-parameters. It makes sense that using results from multiple layers would yield better results. As discussed above, earlier layers tend to look at lower level features, whereas later layers tend to look at higher level features. Combining the two together should create a more complete analysis of the images. We are now ready to use Equation 2.55 as we please.

CNNs on 1D and 3D data

The extension to data of different dimensions is pretty intuitive. What changes here is the dimension of the filters used. As seen above, a 2D image is convolved using a 2D filter. Similarly, 1D data (e.g. a time series) or 3D data (e.g. a CAT scan of a human body, or a video) use 1D/3D filters, respectively. As we'll see in the next section, RNNs are commonly used on 1D data, rather than using a CNN.

2.5 Sequence Models

2.5.1 Recurrent Neural Networks

Here we shall learn about *Recurrent Neural Networks* (RNNs). This type of model has been proven to perform extremely well on temporal data. It has several variants including LSTMs, GRUs and bi-directional RNNs, which we will look at in this section. Sequence models are useful for both natural language processing and music generation.

Let's take a motivating example for defining our notation. The sentence

Harry Potter and Hermione Granger invented a new spell.

is our input, denoted x. x is a sequence of words, each word indexed as $x^{\langle i \rangle}$. The output of our algorithm is denoted y. If we were running a *named-entity recognition* algorithm, then we would have $y = (1, 1, 0, 1, 1, 0, 0, 0, 0)$. Similarly, we denote the elements of y using $y^{\langle i \rangle}$. We use T_v to denote the length of the vector v e.g. $T_x = T_y = 9$. As above, we denote the i^{th} training example by superscript (i).

We can represent words using a *vocabulary/dictionary*. This is a list of all of the words we know. It is not uncommon to have a dictionary of +100,000 words in length! We can then use a one-hot encoding for each word in our sentence i.e. if *Harry* were 4075^{th} in our dictionary, then we would have a list of zeros everywhere except for in position 4075 where we would have a 1. If we come upon a word that is not in our dictionary then we create a new word in our dictionary called $\langle$UNK$\rangle$ to represent these. We shall discuss what to do in this event later.

Recurrent Neural Networks

Why doesn't a standard network work well for this kind of task? Some problems include

- inputs and outputs can be different lengths for different examples;
- standard networks don't share features learned across different positions of the text. That is, there is no memory.

Let's now look at the recurrent neural network, which fixes these problems. Figure 2.37 shows an illustration of a RNN, where $a^{\langle 0 \rangle}$ is a vector of zeros.

The diagram shows how the activations from the previous input are injected into the calculation of the current input 's activation. We train the weights W_{ax}, W_{aa} and W_{ya}. These weights are the same across each timestamp in the network. One limitation of the structure seen in Figure 2.37 is that at prediction time, we only use information from previous timestamps and do not look forward. Depending on the use-case, this could weaken the performance of the model. Bi-directional RNNs (BRNNs) solve this problem.

The calculations done in a standard RNN are as follows:

$$a^{\langle 0 \rangle} = \underline{0} \tag{2.61}$$

$$a^{\langle t \rangle} = g_1(W_{aa}a^{\langle t-1 \rangle} + W_{ax}x^{\langle t \rangle} + b_a) = g_1(W_a[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_a) \tag{2.62}$$

$$y^{\langle t \rangle} = g_2(W_{ya}a^{\langle t \rangle} + b_y) = g_2(W_y a^{\langle t \rangle} + b_y) \tag{2.63}$$

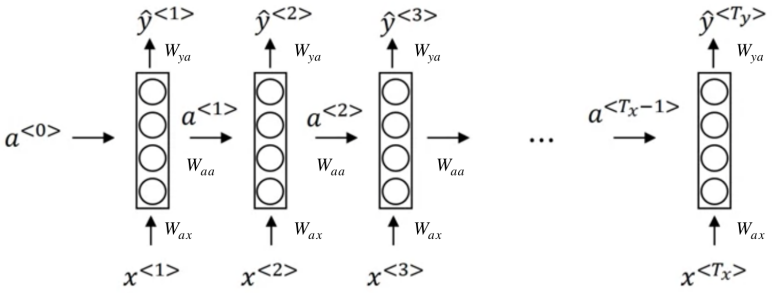

Figure 2.37: Recurrent Neural Network

where we have used the convention that W_{ij} is multiplied by some j-like quantity to give some i-like quantity. tanh (or, less commonly, ReLU) activations are most common for g_1 while, depending on the application, sigmoid activations are a common choice for g_2. The right-hand side of Equation 2.62 denotes W_a as W_{aa} and W_{ax} side-by-side in a matrix, and $[a^{\langle t-1\rangle}, x^{\langle t\rangle}]$ as stacking $a^{\langle t-1\rangle}$ and $x^{\langle t\rangle}$ on top of each other in a vector. Similarly, we have that $W_{ya} = W_y$.

Backpropagation Through Time

Forward propagation is performed as shown in Figure 2.37 where we calculate all of the activations for a given input at a specific timestep. Let's define our element-wise loss function as the standard binary cross-entropy loss fucntion

$$\mathcal{L}^{\langle t\rangle}(\hat{y}^{\langle t\rangle}, y^{\langle t\rangle}) = -y^{\langle t\rangle} \log(\hat{y}^{\langle t\rangle} - (1 - y^{\langle t\rangle}) \log(1 - \hat{y}^{\langle t\rangle}) \tag{2.64}$$

and the overall loss as

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{\langle t\rangle}(\hat{y}^{\langle t\rangle}, y^{\langle t\rangle}). \tag{2.65}$$

Unsurprisingly, backpropagation works backwards through Figure 2.37 in order to update the weightings appropriately. The most important dimension in an RNN is the horizontal (time) dimension, hence why backpropagation with RNNs is commonly referred to as *backpropagation through time*.

RNN Architectures

Here we briefly look at the different RNN architectures commonly used by researchers and practitioners. Figure 2.37 is an example of a *many-to-many* architecture in that it has many inputs and many outputs for each training example. If, on the other hand, we were performing *sentiment classification* (i.e. given a sentence, what is its sentiment?), this would be an example of a *many-to-one* architecture. We can also have a *one-to-many* architecture, commonly used in music generation (i.e. given just the starting note as input, generate the rest of the music). Here, the output of the previous timestep is often fed into the next timestep (e.g. to give musical context of the previous notes).

So far, we have assumed that the input and output lengths of our many-to-many RNN are the same. This is not always true. For example, if we are performing *machine translation* from English to French then we may well have a different number of words in the input and output. In this case, we first *encode* the input and then *decode* the output, as can be seen in Figure 2.38.

Language Modelling and Sequence Generation

Context is essential in performing speech recognition. For example, we are much more likely to have the sentence *The apple and pear salad* rather than the sentence *The apple and pair salad*. A good speech recognition system would pick the first sentence, and would do so using a language model

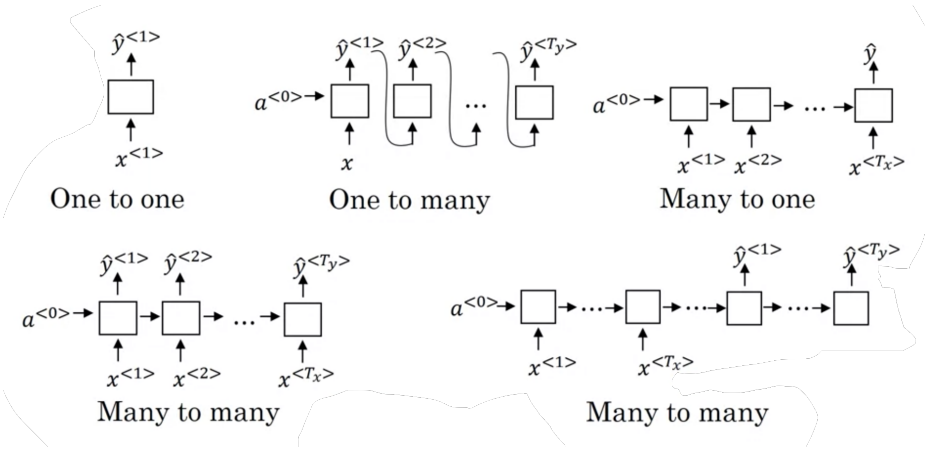

Figure 2.38: RNN architectures. Think of each of the activations as sitting in the boxes.

that outputs some probabilities as to the likelihood of the given speech being a particular sentence. That is, a language model outputs

$$\mathbb{P}[\text{sentence}] = p_{\text{sentence}}. \tag{2.66}$$

The sentence is modelled as outputs $y^{\langle 1\rangle}, ..., y^{\langle T_y\rangle}$ and, specifically, the language models the probability of that particular sequence of words

$$\mathbb{P}\left[y^{\langle 1\rangle}, ..., y^{\langle T_y\rangle}\right]. \tag{2.67}$$

To build a language model, we first need a training set, such as a large corpus of English text. The first thing we do is *tokenise* each sentence (i.e. form a vocabulary/dictionary) in the corpus. A common addition to the dictionary is to add a ⟨EOS⟩ token that encodes the end of a sentence. Punctuation is also commonly added to the dictionary. As mentioned above, words that we come across during prediction that are not in our dictionary are assigned the ⟨UNK⟩ token. After tokenising the sentence, we run the sentence through a RNN.

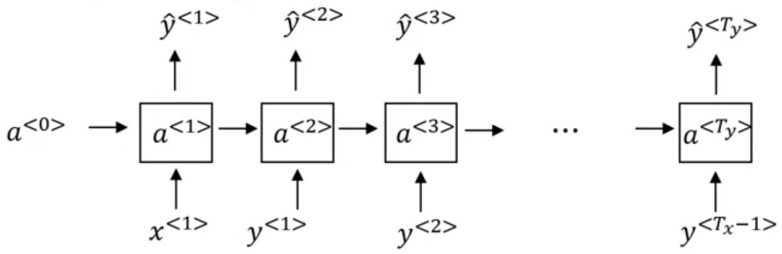

Figure 2.39: Language Model Architecture

Here we use a standard many-to-many architecture for our network, with a few minor edits. Let's take the example sentence

Cats average 15 hours of sleep a day. ⟨*EOS*⟩

That is, $T_y = 9$ (we ignore punctuation). We initialise the model with $a^{\langle 0\rangle} = x^{\langle 1\rangle} = 0$ and use a softmax output for $\hat{y}^{\langle 1\rangle}$ i.e. what is the probability of any word in our dictionary appearing. We then set $x^{\langle 2\rangle} = y^{\langle 1\rangle} = \textit{Cats}$ and calculate the activation and yield a softmax output $\hat{y}^{\langle 2\rangle}$, which is essentially

$$\hat{y}^{\langle 2\rangle} = \mathbb{P}[\text{all words in dictionary} \mid \textit{Cats}], \tag{2.68}$$

where ideally *average* has the highest probability here. This carries on through the sentence, calculating at each timestep

$$\hat{y}^{\langle t\rangle} = \mathbb{P}[\text{all words in dictionary} \mid \hat{y}^{\langle 1\rangle}, ..., \hat{y}^{\langle t-1\rangle}], \tag{2.69}$$

where it is key that the right-hand side of the probability is *ordered* i.e. sentence order matters. To train the network, we use the *softmax loss function*

$$\mathcal{L}^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}) = -\sum_{i=1}^{T_y} y_i^{\langle t \rangle} \log(\hat{y}_i^{\langle t \rangle}) \tag{2.70}$$

and an overall loss of

$$\mathcal{L}(\hat{y}, y) = \sum_t \mathcal{L}^{\langle t \rangle}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle}). \tag{2.71}$$

Given a new sentence, say, $(y^{\langle 1 \rangle}, y^{\langle 2 \rangle}, y^{\langle t3 \rangle})$, we can now calculate the probability of this entire sentence as

$$\mathbb{P}\left[y^{\langle 1 \rangle}, y^{\langle 2 \rangle}, y^{\langle 3 \rangle}\right] = \mathbb{P}\left[y^{\langle 1 \rangle}\right] \mathbb{P}\left[y^{\langle 2 \rangle} \mid y^{\langle 1 \rangle}\right] \mathbb{P}\left[y^{\langle 3 \rangle} \mid y^{\langle 1 \rangle}, y^{\langle 2 \rangle}\right]. \tag{2.72}$$

Sampling Novel Sequences

One of the most interesting things to do given a trained language model is to sample from it! This will give us an informal sense of what the network has learned. We do this by sampling from the output probability distribution of our trained model.

To sample, we use a slightly different RNN than the one seen in Figure 2.39. Here, we sample from our distribution in order to get $\hat{y}^{\langle t-1 \rangle}$, which we then use as $x^{\langle t \rangle}$ for calculating $a^{\langle t \rangle}$ and sampling from $\hat{y}^{\langle t \rangle}$. If the $\langle$EOS$\rangle$ token is in our vocabulary, then we can keep running the RNN until we sample an $\langle$EOS$\rangle$ token. Otherwise, we can have a set number of words to run our RNN for before stopping.

So far, we have only looked at *word-level* language models. We could also develop a *character-level* language model, which is a simple extension of the ideas described above. Here, $y^{\langle t \rangle}$ would be a character rather than a word. These models, however, are much more computationally expensive and have a much harder time *remembering* what characters they have already seen (e.g. you may have to look back 20 characters rather than just 2 words). This has resulted in research focusing on word-level models instead. As computers get faster, perhaps character-level models will become more popular.

Vanishing Gradients in RNNs

Basic RNNs aren't great at capturing very long-term dependencies. For example, the two sentences

1. The cat, which already ate an apple, banana,..., and zucchini, was full.
2. The cats, who already ate an apple, banana,..., and zucchini, were full.

are grammatically correct but potentially require a long-term dependency to have the tuples (cat, was) and (cats, were) correct. This is made challenging by the *vanishing gradient* problem where it is difficult for the errors associated with the later timesteps to affect the computations that are earlier in the network if the network is *very deep*. Because of this problem, the basic RNN has mainly local influences i.e. $y^{\langle t \rangle}$ is mostly affected by what happened at $t-1$, and less so by what happened at $t-2$, and even less so at what happened at $t-3$ etc.

Vanishing gradients is much more common in RNNs than *exploding gradients*, however exploding gradients can also be catastrophic to performance. The exploding gradient problem is often easier to spot as you'll see exploding parameter values in your network. A solution to this is to perform *gradient clipping* throughout your network to set maximum/minimum values that the gradient can take.

Gated Recurrent Unit (GRU)

The *Gated Recurrent Unit* (GRU) is a modification to the RNN hidden layer which makes it much better at capturing long-range connections and helps a lot with the vanishing gradient problem. The

basic RNN calculates the activation using

$$a^{\langle t \rangle} = g(W_a[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_a) \tag{2.73}$$

where g is commonly tanh. We shall use the *cat* sentences above as an ongoing example.

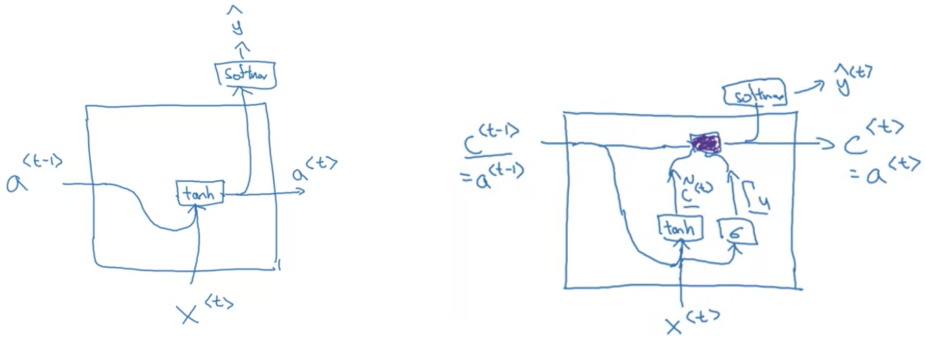

Figure 2.40: Basic RNN unit (left) and simplified GRU (right). The purple box is where we evaluate Equation 2.76.

The GRU uses a *memory cell* $c^{\langle t \rangle}$ that provides memory, for example, to remember whether the noun in a sentence is singular or plural (e.g. *cat* or *cats*). The GRU will in fact output $c^{\langle t \rangle} = a^{\langle t \rangle}$ (this notation is useful later when we look at LSTMs where $c^{\langle t \rangle} \neq a^{\langle t \rangle}$). At each timestep, we consider overwriting the value of $c^{\langle t \rangle}$ with the value

$$\widetilde{c}^{\langle t \rangle} = \tanh(W_c[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c). \tag{2.74}$$

We also include an *update gate* Γ_u where

$$\Gamma_u = \sigma(W_u[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u). \tag{2.75}$$

Think of the update gate Γ_u as either being 0 or 1 i.e. closed or open. Of course the sigmoid function is never 0 nor 1, but for the majority of its support it is close enough. The gate will decide whether we actually update $c^{\langle t \rangle}$ with $\widetilde{c}^{\langle t \rangle}$ or not. If the gate is open (has value approximately 1) then we update $c^{\langle t \rangle}$ and if the gate is closed then we do nothing. Specifically,

$$c^{\langle t \rangle} = \Gamma_u * \widetilde{c}^{\langle t \rangle} + (1 - \Gamma_u) * c^{\langle t-1 \rangle}. \tag{2.76}$$

For example, we may have that c denotes the context of a singular (1) or plural (0) noun in our sentence e.g. *cat* or *cats*. Our GRU will then open the gate when it sees *cat* and therefore update c to $\widetilde{c} = 1$ for that timestep. c will then remain equal to 1 for the remainder of the sentence and we will have the context to use *was* rather than *were* at the end of our sentence.

Because Γ_u can take on such small values, this means the value of c is often maintained across many many timesteps of our sequence. This helps significantly with the vanishing gradient problem as we often have c (i.e. our activations, in this case) represented by an almost-identity function. Our network can now handle long-range dependencies much better.

$c^{\langle t \rangle}$ has the same dimensions as $\widetilde{c}^{\langle t \rangle}$ and Γ_u and can also be a vector. In this case, the $*$ in Equation 2.76 denotes an element-wise multiplication. Using $n-$dimensional vectors here means we can have n different long-term rules that we want our network to keep track of over time, and not just whether we are talking about singular or plural noun.

The above is in fact a *simplified* GRU, however the extension to a *full* GRU is not large. We extend the ideas described above to include one more gate Γ_r and edit Equation 2.74 to include our additional gate:

$$\Gamma_r = \sigma(W_r[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_r) \tag{2.77}$$

$$\widetilde{c}^{\langle t \rangle} = \tanh(W_c[\Gamma_r * c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c). \tag{2.78}$$

Over the years, researchers have experimented with different GRUs to best model long-range connections and fix the vanishing gradient problem, and it turns out the full GRU tends to perform very well.

Long Short-Term Memory (LSTM) Unit

The *long short-term memory* (LSTM) unit is similar to the GRU in that it captures long-term dependencies well, and it often performs better than the GRU too. The LSTM was first introduced in Long Short-Term Memory, Hochreiter and Schmidhuber, 1997.

The equations for the LSTM unit are similar to the GRU. As mentioned above, $c^{\langle t \rangle} \neq a^{\langle t \rangle}$ in the case of a LSTM.

$$\tilde{c}^{\langle t \rangle} = \tanh(W_c[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c) \tag{2.79}$$

$$\Gamma_u = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u) \quad \text{(update)}$$

$$\Gamma_f = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f) \quad \text{(forget)}$$

$$\Gamma_o = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o) \quad \text{(output)}$$

$$c^{\langle t \rangle} = \Gamma_u * \tilde{c}^{\langle t \rangle} + \Gamma_f * c^{\langle t-1 \rangle} \tag{2.80}$$

$$a^{\langle t \rangle} = \Gamma_o * \tanh(c^{\langle t \rangle}) \tag{2.81}$$

To create a network here, all we have to do is chain many LSTM units, such as in Figure 2.41, together. Similarly to GRUs, it is clear here that the use of gates allows values of c to be passed through many timesteps with potentially minimal updating of its value, which helps to stop vanishing gradients. This also allows the network to maintain memory of c across potentially many timesteps.

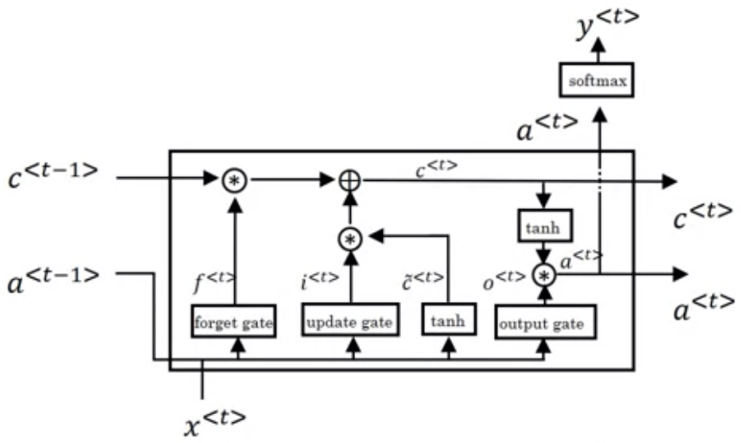

Figure 2.41: Illustration of LSTM unit

We can extend the LSTM to include *peephole connections* which is where $c^{\langle t-1 \rangle}$ is appended to the vector inside the calculation for each of the gates Γ_u, Γ_f and Γ_o.

Bi-directional RNN (BRNN)

The bi-directional RNN (BRNN) allows you to take information from both earlier *and* later in the sequence, allowing our algorithm to "look forward" some timesteps. Consider the sentences

1. He said, "Teddy bears are on sale!"
2. He said, "Teddy Roosevelt was a great President!"

The two sentences begin the same and context is achieved only after the word *Teddy*. It would therefore be useful to be able to look forward in the sentence in order to gain more context.

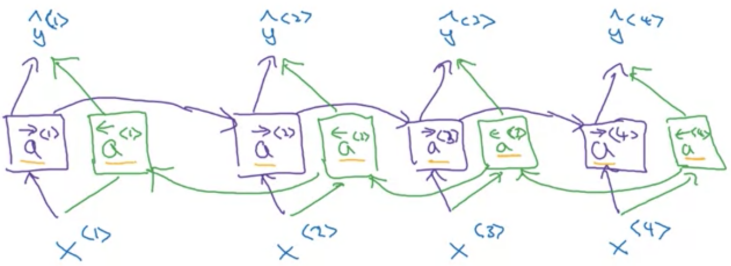

Figure 2.42: Example Bi-directional RNN

The architecture of a BRNN can be seen in Figure 2.42. Our forward propagation calculations go from both left-to-right (purple) as well as right-to-left (green). After computing all activations, you can then make the predictions $y^{\langle t \rangle}$. The prediction is a simple extension to what we have seen already:

$$\hat{y}^{\langle t \rangle} = g\left(W_y[\overrightarrow{a}^{\langle t \rangle}, \overleftarrow{a}^{\langle t \rangle}] + b_y\right). \tag{2.82}$$

Look at the flow of information for $\hat{y}^{\langle 3 \rangle}$, for example. It is receiving information from

- $\overrightarrow{a}^{\langle 1 \rangle}$, $\overrightarrow{a}^{\langle 2 \rangle}$ and $\overrightarrow{a}^{\langle 3 \rangle}$ from the left (i.e. information from the *past*), and;
- $\overleftarrow{a}^{\langle 4 \rangle}$ and $\overleftarrow{a}^{\langle 3 \rangle}$ from the right (i.e. information from the *future*).

The activation blocks in a BRNN are often LSTM units as they tend to perform best on the types of problems the BRNN is applied to. One disadvantage to the BRNN is that you need the entire sequence in order to process the information and make any predictions.

Deep RNNs

It is often useful to stack multiple RNNs together in order to increase performance. Let's start with an example. We shall use the notation $a^{[l]\langle t \rangle}$ to denote the activation associated with timestep t in layer l.

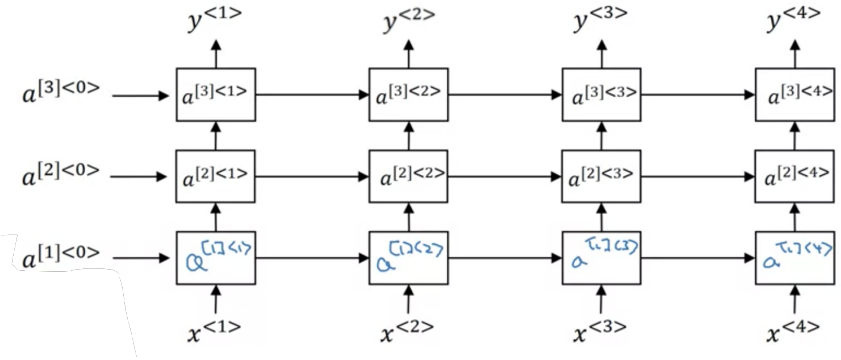

Figure 2.43: Deep RNN with 3 layers

An example activation calculation in Figure 2.43 would be

$$a^{[2]\langle 3 \rangle} = g\left(W_a^{[2]}[a^{[2]\langle 2 \rangle}, a^{[1]\langle 3 \rangle}] + b_a^{[2]}\right). \tag{2.83}$$

As opposed to in standard deep neural networks, a deep RNN with just a handful of layers can be computationally very expensive. A network as in Figure 2.43 is already quite deep. One common addition is to attach a conventional deep network between the final RNN activation and the output value at *each timestep*. We can build deep versions of GRUs, LSTMs and BRNNs too.

2.5.2 Natural Language Processing and Word Embeddings

2.5.3 Sequence Models and Attention Mechanism

Chapter 3

Reinforcement Learning