1 Introduction to Neural Networks

Learning Outcomes

By the end of this lecture you will be able to

1. Explain why neural networks are of interest in data science
2. Explain in outline how neural networks work

1.1 Why are neural networks of interest?

Neural networks are a tool for machine learning. They have successfully been used to perform a variety of tasks, in some cases outperforming humans:

- recognising faces
- identifying traffic signs in images
- lip reading
- medical diagnosis
- targeting advertisements
- . . . etc.

Neural networks are trained using data. In the news lately are so-called *Deep Neural Networks*, that can perform sophisticated tasks such as recognising faces. They have extremely large numbers of parameters, and hence require huge amounts of data to train them. The advent of Big Data and development of suitable algorithms has enabled the training of such networks. The huge amounts of data available have minimised the need for human knowledge input, such as in *feature engineering* i.e. using domain knowledge to create new attributes by combining existing attributes. Instead the network itself uses the data to construct useful new features.

1.2 What actually *is* a neural network?

A *neural network* is a model for computing answers by combining many simple calculations. The simple calculations are performed by elements called *neurons*. Neural networks are inspired by how the brain works, and should strictly be called artificial neural networks, but we will drop the "artificial" in this course. Neural networks are typically simulated in software using standard computers. There are many different types of neural network — if it has interconnected elements that perform simple calculations, it qualifies — but we will be most interested in the most popular kind of neural network, so-called *feed-forward*[1] *neural networks* that take an input, such as an image, and calculate an answer, such as the class of the object shown.

1.3 What can a neural network do?

Feed-forward neural networks commonly perform two tasks:

- *Classification*: The inputs are attributes of an object, and the output indicates the class of that object e.g. using the attributes of a borrower to predict whether they will default on a loan or not.
- *Regression*: The inputs are attributes of an object, and the output indicates a value associated with that object e.g. using the attributes of a car to estimate its price.

The difference between these two is in the output. For classification, the output can take on one of a discrete number of classes; for regression the output is usually continuous. Neural networks can also be used for other tasks, such as deriving an encoding of the input, or data exploration and clustering.

[1]Feed-forward simply means the calculation in the network proceeds from the input to the output, and there are no feedback loops, as there are in *recurrent neural networks*.

1.4 How does a neural network calculate its output?

Let's see how an extremely simple network with only one neuron could make a decision. Obviously this is hopelessly unrealistic! But our aim here is just to explain how a single neuron operates.

Example

Suppose we wished to decide whether to go to a particular restaurant. The input data we have is three binary values ($0 =$ no, $1 =$ yes):

x_1: is the food good?
x_2: is the ambience good?
x_3: is it expensive?

The decision process is to multiply each of these inputs x_1, x_2, x_3 by a different weight w_1, w_2, w_3 respectively and add up the total. The weights show how important each input is. The answer is yes i.e. the output is 1 if the total reaches a certain threshold T and i.e. the output is 0, if it does not. Let's work through the details with weights $w_1 = 3$, $w_2 = 2$, $w_3 = -1$ and threshold $T = 3$:

The inputs do not have to be binary values — we could instead have given them continuous values between some limits, denoting the quality of the food, the quality of the ambience and the expense. The key thing is the simple nature of the calculation, just a weighted sum and a threshold. Note, we could rearrange our calculation, by instead subtracting the threshold and seeing if the answer was greater or equal to zero. This is exactly the sort of calculation a neuron does:

1. Calculate a weighted sum of its inputs.
2. Add a bias term (in effect, the negative of the threshold value).
3. Apply a function to the result (in this example, the *step function* that is 1 if the input is greater of equal to zero and 0 otherwise)

We will call the output of our “neural network” $\hat{y}$:

$$\hat{y} = f(w_1 x_1 + w_2 x_2 + w_3 x_3 + b)$$

Diagrammatically this calculation can be depicted as shown in Figure 1.1[2].:

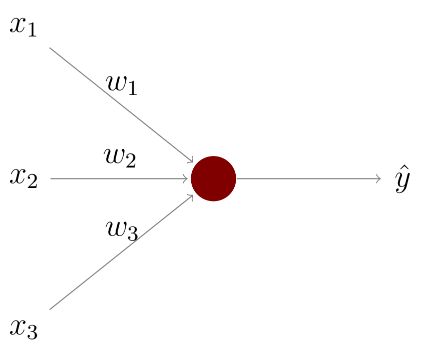


Figure 1.1: A single neuron

In a different example, there might be more input factors, say x_1, x_2, . . . , x_n, and so the equation for the output of a neuron is:

$$\boxed{\hat{y} = f\left(\sum_{k=1}^{n} w_k x_k + b\right)} \tag{1.1}$$

The equation above is fundamental to understanding neural networks. Every neuron calculates its output by calculating the weighted sum, adding a bias term, and applying a function f. (Variation: some authors invent an extra input x_0 that is always 1, and instead write $\hat{y} = f\left(\sum_{k=0}^{n} w_k x_k\right)$, where their w_0 is our b. The difference is only one of notation.)

Terminology

In equation (1.1) the parameters w_k are called *weights*, the parameter b is called the *bias*, and the function f is called the *activation function.*

We will see examples of commonly used activation functions later in the course.

[2]Figures are adapted from one by Kjell Magne Fauske under Creative Commons Attribution 2.5 License.

For a more general decision, the factors for the final decision might be calculated from a set of subfactors, in turn calculated from other subfactors, in turn calculated from a certain number of inputs. This sort of calculation leads a network that looks more like Figure 1.2.

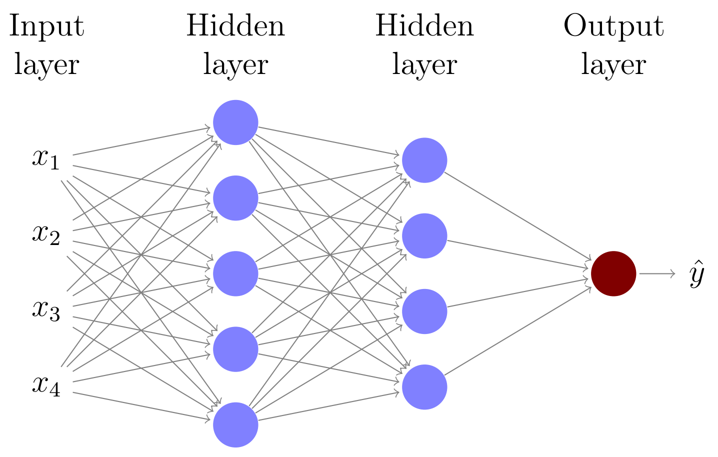


Figure 1.2: A neural network

Terminology

The neurons, as shown in Figure 1.2, are arranged in *layers*. The *input layer* is the list of inputs, the *output layer* is the last layer, and the ones in between are called *hidden layers*. Somewhat counter-intuitively, when counting the number of layers in a network, we don't count the first layer, since no calculation is done there. So the network shown in Figure 1.2 is a 3-layer network.

The activation function used by the neurons, the number of neurons in each layer, and the number of layers, are design decisions taken by the data scientist constructing the neural network. The output layer might have several neurons — in the case of digit recognition we would usually have 10, one for each possible digit. Note in Figure 1.2, most neurons appear to have several outputs — in fact they each have only one output, but it is copied to all the neurons in the next layer, and easier to draw as separate lines rather than a line that splits.

1.5 How is a neural network trained?

The weights and biases in a neural network are parameters that can be adjusted. They need to be adjusted to the best values for a particular task. This is done by using training data i.e. it's an example of *supervised learning*: we have examples of input vector x, and the corresponding correct output value y.

Notation

In this course we will suppose we have m examples of input data and the corresponding correct output, and denote them as follows

$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \dots, (x^{(m)}, y^{(m)})$$

In general $x^{(i)}$ for $i = 1, \dots, m$ will be a vector of values. We denote the components of the vector using subscripts, and suppose there are n of them e.g. $x^{(1)}$ has components

$$x_1^{(1)}, x_2^{(1)}, x_3^{(1)}, \dots, x_n^{(1)},$$

The output produced by the network on input $x^{(1)}$ will be denoted $\hat{y}^{(1)}$.

The key to supervised learning in neural networks is to have some *error function* that calculates how bad the difference is between the calculated output $\hat{y}$ and the desired output y. We can then calculate how to adjust the weights and bias to reduce this error function. This idea is the key to training neural networks. We will look at the details of suitable functions and how exactly the weights and bias are modified later in this course.

1.6 Conclusion

Neural networks are machine learning tools that can outperform humans on some recognition tasks. They are made from small interconnected elements called *neurons*. The neurons perform simple calculations, typically a weighted sum of inputs, followed by application of an activation function, as shown in Equation (1.1). The parameters of the network are adjusted using training data and a suitable error function.

2 Logistic Regression as a Neural Network

Learning Outcomes

By the end of this lecture you will be able to

1. Explain and implement the process of training a single neuron neural network

2.1 Introduction

In this lecture we will explore the detailed working of a single neuron neural network. We will implement this in the practical sessions, *not* because you will need to do this in your future work life, but because it is a good way for you to gain a clear understanding of how neural networks work.

2.2 Recap on Logistic Regression

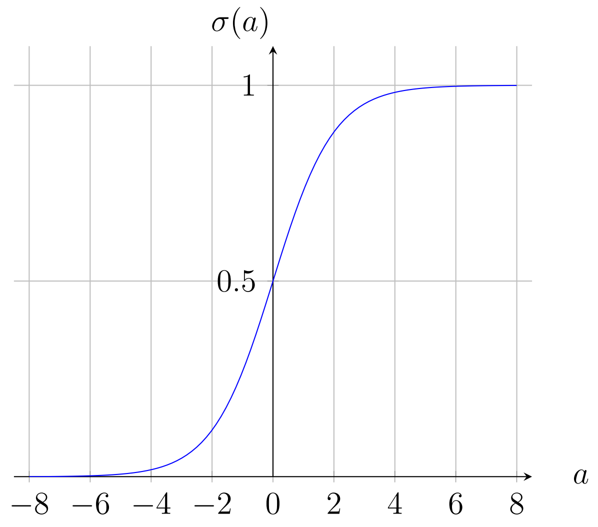
Suppose we have a *binary classification* task where we would like to predict the class of an object, given its attributes. Typical examples are: predicting whether someone has cancer, on the basis of a set of tests, or whether someone will default on a loan, given information about them. In our training data $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \ldots, (x^{(m)}, y^{(m)})$ the outputs $y^{(i)}$ will be 1 if the example belongs to one class, and 0 if it belongs to the other. A standard statistical approach to this task is *logistic regression*, in which the output $\hat{y}$ from our model gives the probability that an example, with given vector of attributes x, belongs to class 1. So $\hat{y}$, like all probabilities, should be a value between 0 and 1.

Recall from last handout Equation (1.1), the formula for calculating the output of a single neuron is

$$\hat{y} = f\left(\sum_{k=1}^{n} w_k x_k + b\right)$$

Here w_k and b are parameters that need to be fitted using the data, and the x_k are the components of the input vector x. If we choose f to be the *logistic* function

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$



then the output will always lie between 0 and 1. This is a so-called logistic regression model[1].

To fix our ideas, we will focus on a logistic model with two inputs x_1 and x_2, hence two weights w_1 and w_2, and a bias b. So the output from the network is

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b) \tag{2.1}$$

To use this as a classifier, we will say that if the model gives an output of 0.5 or greater when applied to a given input, then model has assigned the input example to class 1. If the output is less than 0.5 then we will say the model has assigned the input example to class 2.

2.3 Error functions

Remember, we are using training data to train the network, (i.e. “fit the model”). For the i^{th} example $x^{(i)}$ we know that the target answer is $y^{(i)}$. And we will denote our actual output on this i^{th} example as $\hat{y}^{(i)}$, this answer depending on our choice of parameters w_i and b. We would like to measure of how bad the error is with the current choice of parameters. A common way to measure the error on this i^{th} example would be by calculating

$$(\hat{y}^{(i)} - y^{(i)})^2$$

[1]You may encounter this model presented in other forms in statistical text books.

We could then get an *overall* measure of error on all the examples in the training set, by adding these values up and dividing by m to get the mean value:

$$E = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2$$

However for logistic regression we prefer a different function. Although it looks complicated, it turns out that the function

$$L^{(i)}(w,b) = -\left(y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right) \tag{2.2}$$

is a natural way to measure the error on the i^{th} example $x^{(i)}$. We have written (w, b) after the function, to emphasize that its value depends on the choice of the weights w and bias b.

We can at least convince ourselves that minimising the value of the expression in (2.2) will push the network in the correct direction: we know the target value $y^{(i)}$ is either 0 or 1. If $y^{(i)} = 1$, then the right hand term disappears and (2.2) reduces to $-\log(\hat{y}^{(i)})$. So to minimise it we want to make $\log(\hat{y}^{(i)})$ as big as possible, which is when $\hat{y}^{(i)}$ is as big as possible, which is when it's 1. Alternatively if $y^{(i)} = 0$, then the left hand term disappears and (2.2) reduces to $-\log(1-\hat{y}^{(i)})$. So to minimise it we want to make $\log(1-\hat{y}^{(i)})$ as big as possible, which is when $\hat{y}^{(i)}$ is small as possible, which is when it's 0. So the error function is at least plausible.

We again get an *overall* measure of error on all the examples in the training set, by adding these values up and dividing by m to get the mean value:

$$\boxed{\begin{aligned} E(w,b) &= \frac{1}{m}\sum_{i=1}^{m} L^{(i)} \\ &= -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log(\hat{y}^{(i)}) + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right) \end{aligned}} \tag{2.3}$$

This is known as the *cross-entropy* error function. The key takeaway here is not the exact form of the error function, but the very fact that *there is an error function*, which we can then use to adjust the weights and bias to improve performance. So don't get worried about the exact mathematical details.

2.4 Minimising the error to train the network

The idea of gradient descent

Health warning: this section may contain some unfamiliar mathematical symbols. However the ideas are ok, so don't let the symbols faze you.

Once we have an error function we can now ask the question:

> *What effect does increasing w_1 have on the error?*

If increasing w_1 *increases* the error E, then we should *decrease* w_1; conversely if increasing w_1 *decreases* error E then we should *increase* w_1. Mathematics gives us a precise way of calculating the effect of increasing w_1, denoted:

$$\frac{\partial E}{\partial w_1}$$

This is called a *partial derivative*, and the process of calculating its value is called *partial differentiation*. In words the *partial derivative* means

> *how much the error function E changes, if we just increase w_1 by a tiny bit (and keep all the other weights and bias fixed).*

It is measured relative to the increase in w_1 i.e. if we increased w_1 by 0.001, it gives the increase in E divided by 0.001. (You may have encountered more precise definitions of partial differentiation in previous courses).

So our big idea here is:

1. Calculate $\frac{\partial E}{\partial w_1}$ (in some as yet unspecified way).
2. Subtract an amount proportional to $\frac{\partial E}{\partial w_1}$ from w_1, in symbols $w_1 := w_1 - \eta\frac{\partial E}{\partial w_1}$. Here η (the Greek letter eta) is a positive amount called the *step size*.

Note this does exactly what we want, since if $\frac{\partial E}{\partial w_1}$ is *positive* (i.e. the error increases when we increase w_1), then we want to *decrease* w_1, which is exactly what subtracting $\eta\frac{\partial E}{\partial w_1}$ achieves. Conversely, if $\frac{\partial E}{\partial w_1}$ is *negative* (i.e. the error decreases when we increase w_1), then we want to *increase* w_1, which again is exactly what subtracting $\eta\frac{\partial E}{\partial w_1}$ achieves. So we now have a way to improve the performance of the network, by adjusting parameter w_1.

We can apply this approach to all the weights and biases. This approach is called *gradient descent*:

$$\boxed{\begin{array}{cccl} w_k & := & w_k & -\eta\frac{\partial E}{\partial w_k} \qquad \text{for } k = 1, \ldots, n \\ b & := & b & -\eta\frac{\partial E}{\partial b} \end{array}} \tag{2.4}$$

So we now have one way to attempt to train a neural network. We will look at improvements to this later, but for now our aim is to apply this approach to a logistic regression model.

Gradient Descent for Logistic Regression

To apply the update equations of (2.4) to logistic regression, we need to decide on a value for the *step size* η (e.g. 0.01); we also need know how to calculate $\frac{\partial E}{\partial w_i}$ and $\frac{\partial E}{\partial b}$ for the cross-entropy error function E of (2.3). Since "the derivative of a sum is the sum of the derivatives" we have

$$\begin{aligned}\frac{\partial E}{\partial w_k} &= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L^{(i)}}{\partial w_k}\\ \frac{\partial E}{\partial b} &= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L^{(i)}}{\partial b}\end{aligned} \tag{2.5}$$

It can be shown, using the chain rule for differentiation, that for our choice of $L^{(i)}$ in (2.2), the values we need are:

$$\begin{aligned}\frac{\partial L^{(i)}}{\partial w_k} &= (\hat{y}^{(i)} - y^{(i)})\; x_k^{(i)}\\ \frac{\partial L^{(i)}}{\partial b} &= \hat{y}^{(i)} - y^{(i)}\end{aligned} \tag{2.6}$$

Note that these are actually pleasantly simple things to calculate: for the i^{th} example $x^{(i)}$, we subtract the target value $y^{(i)}$ from our calculated answer $\hat{y}^{(i)}$ from . If we are updating the bias, this is all we need; if we are updating the k^{th} weight w_k, we additionally need to multiply by the value of the k^{th} component of the vector $x^{(i)}$.

So we now have all we need to train our single neuron model of (2.1):

The algorithm

We present the training examples and

1. For each training example, calculate the values on the left-hand side of (2.6).
2. Average these values to calculate the left-hand side of (2.5).
3. Use the left-hand side of (2.5) to update the weights and biases using (2.4).

We repeat this until we are satisfied. Notice we do not actually need to calculate the error E explicitly to update the parameters. However, it does give us a useful indication of how training is going, we will calculate this too.

2.5 Implementing Logistic Regression in Python

In a lab session, we will implement the algorithm at the end of Section 2.4 in Python, so train the single neuron logistic regression model of (2.1):

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b)$$

In pseudocode it is:

```
obtain the training data
visualise the training data

initialise weights, bias, stepsize, num_epochs
for 1 to num_epochs do:
   for each training example do:
      calculate the output
      calculate its contribution to error
      calculate its contribution to the training steps
   calculate overall error
   calculate training steps
   update the weights and bias by the training steps

evaluate performance of trained model
```

In the lab, we can use this code to learn to distinguish between two Gaussian clusters, centered at $(1, 3)$ and $(2, 0)$. The results are shown in Figure 2.1.

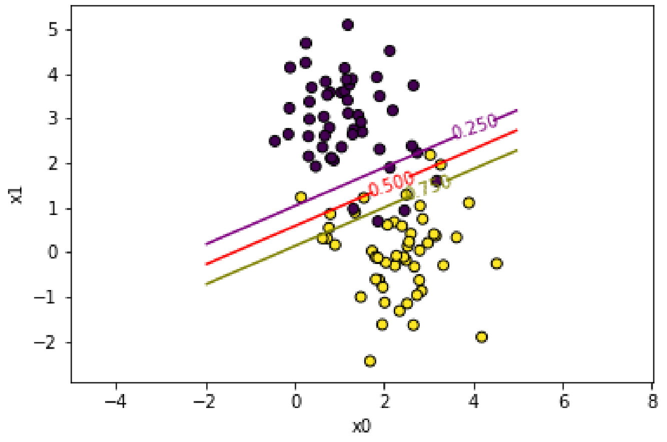


Figure 2.1: Logistic regression to distinguish two Gaussian clusters, centered at $(1, 3)$ and $(2, 0)$.

2.6 Conclusion

Logistic regression can be implemented as a single neuron neural network. The error function commonly used in logistic regression is the cross-entropy function. Using this, we can calculate in which way the weights and bias should be changed to decrease the error on the training data. The simplest way to do this is to repeatedly take small steps in the direction that reduces the error.

In the next lectures we will look at why we need *multilayer* networks, and then consider the practical details of building and evaluating such models.

Historical Note

Early interest in artificial neural networks began in the 1940s, as a way for neuroscientists to model brain function; later there was interest in them as alternative models of computation, and as a way for computers to "learn" without being explicitly programmed. The aim here was to get computers to perform tasks successfully, rather than accurately model the human brain.

In the late 1950s and 1960s, there was considerable interest in a single neuron neural model called the *Perceptron*, created by Frank Rosenblatt. As with much research in Artificial Intelligence there was considerable hype over its abilities. Research in this area dramatically dropped off when Minsky and Papert in 1969 delivered a sober assessment of its capabilities and limitations.

Interest in neural networks revived again in the mid-1980s when Hopfield showed that a different architecture of neural networks could solve minimisation problems; and Rumelhart and McClelland rediscovered the *backpropagation algorithm*, an efficient way to calculate the gradient of the error function with respect to the weights and biases, needed for training multilayer neural networks. These networks provided a solution to the problems highlighted by Minsky and Papert.

Neural networks were gradually surpassed in the 1990s, at least for classification tasks, by Support Vector Machines. However another leap forward for neural networks happened in the mid 2000s when Hinton and others developed ways to train neural networks with large numbers of layers. This, coupled with increased computing power obtained by harnessing GPUs, and architectures such as convolution neural networks, enabled neural networks to learn complicated and sophisticated features. In particular, so called Deep Neural Networks are currently the most successful approach to a number of image classification tasks.

3 Multilayer Neural Networks

Learning Outcomes

By the end of this lecture you will be able to

1. Explain why multilayer neural networks are needed
2. Describe typical activation functions used
3. Implement a neural network using Keras

3.1 Introduction

We have seen that logistic regression can be an effective tool for binary classsification. Why go further? The fundamental limitations of a single neuron are demonstrated in the next section, and from there we go on to look at possible solutions.

3.2 Linear Separability

Recall that a single neuron produces its output according to the formula (1.1)

$$\hat{y} = f\left(\sum_{k=1}^{n} w_k x_k + b\right)$$

If we are using this neuron for binary classification then we typically choose an output level threshold to distinguish between the two classes (in the lab session for the logistic neuron, we chose 0.5). What possible shapes can this boundary be? The output of function f is constant if its argument is held constant, say equal to c, i.e.

$$\sum_{k=1}^{n} w_k x_k + b = c \tag{3.1}$$

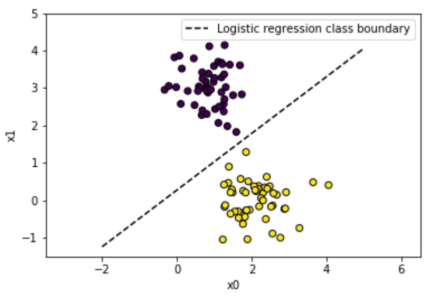


Figure 3.1: Linearly Separable Classes

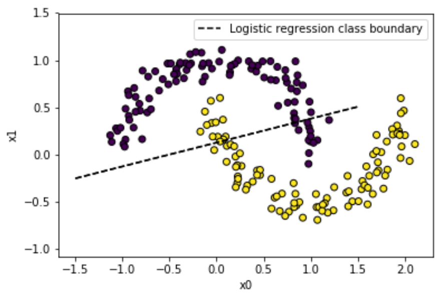


Figure 3.2: Linearly Inseparable Classes

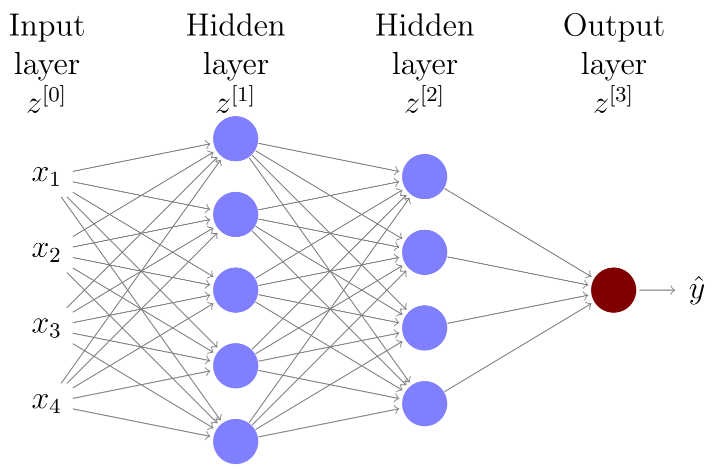
When $k = 2$ we have

$$w_1x_1 + w_2x_2 + b = c$$

which is the equation of a straight line. For larger k, the decision boundary given by (3.1) is a plane or hyperplane. The implication of this is that a single neuron can only successfully perform binary classification if the classes are *linearly separable*. Examples of this are seen in Figures 3.1 and 3.2.

3.3 Multiple Layer Networks

One solution to this is to build neural networks of multiple layers as was depicted in Figure 1.2.



Notation

We will number the layers and use an index in square brackets to indicate the number of the layer. So output of the 2nd layer will be denoted $z^{[2]}$. This is a vector consisting of the outputs $z_1^{[2]}, z_2^{[2]}, \dots z_{n^{[2]}}^{[2]}$ of the individual neurons in that layer.
The output of, say, the third neuron in the second layer $z_3^{[2]}$ is still calculated according to our fundamental neuron equation (1.1), adapted so the weights and biases are labelled according to the layer:

$$z_3^{[2]} = f\left(\sum_{k=1}^{n^{[1]}} w_{3k}^{[2]} z_k^{[1]} + b_3^{[2]}\right)$$

The expression inside the bracket i.e. the argument of the function f is called the *activation* of the third neuron in the second layer and denoted $a_3^{[2]}$. The function f itself is called the *activation function*. The activation function f is usually the same for all neurons within a layer, and often the same for all neurons in the network other than perhaps those in the output layer.
In general, the output of the r^{th} neuron in the t^{th} layer is calculated as

$$z_r^{[t]} = f\left(\sum_{k=1}^{n^{[t-1]}} w_{rk}^{[t]} z_k^{[t-1]} + b_r^{[t]}\right) \tag{3.2}$$

The activation of the r^{th} neuron in the t^{th} layer is denoted $a_r^{[t]}$. If there are N layers in total then we can also write $\hat{y}$ for $z^{[N]}$ and x for $z^{[0]}$. If we want to denote the output on the i^{th} example, we write $\hat{y}^{(i)}$ or $z^{[N](i)}$

Although this notation looks complicated, it gives us a precise way to talk about specific neuron outputs.

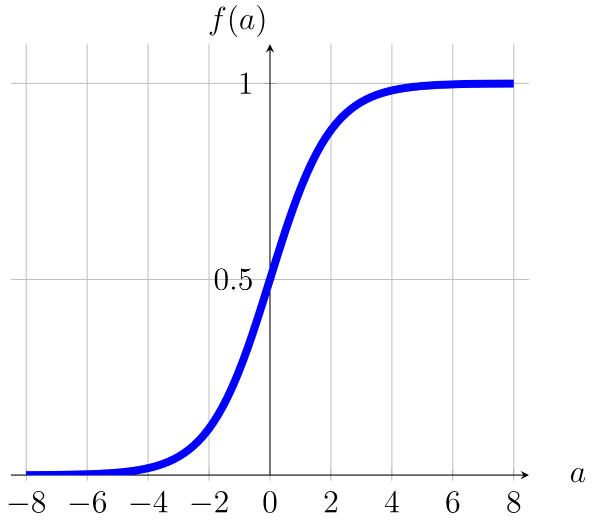
3.4 Activation functions

To get any benefit from multiple layers we need to have some of our activation functions non-linear. (It can be shown that otherwise the multiple layers are equivalent to a large single layer, and so still unable to handle classes that are not linearly separable). Common activation functions are:

3.4.1 The Logistic Function

We have already seen this function. It is *sigmoid* in shape, and so has the property that the output "saturates" i.e. the output level tends to a limiting value as the input increases or decreases without bound. This is similar to biological neurons in the brain where the output, given by the rate of firing pulses, has an upper bound. Constraining the output to lying between 0 and 1 is useful if we want the final output to represent a probability, or a degree of class membership.

$$f(a) = \frac{1}{1+e^{-a}}$$



The logistic function has the useful property that its derivative is expressable in terms of the original function.

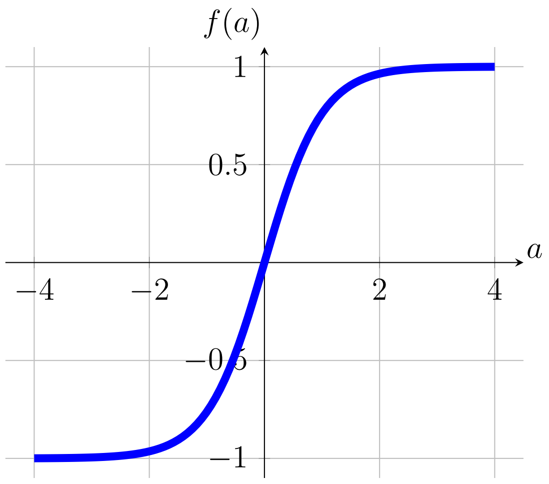
$$f'(a) = f(a)(1 - f(a))$$

Sigmoid functions become flat for large input and output values i.e. the gradient of the curve is small. In neural networks with many layers using sigmoid activation functions, training can encounter the so-called *vanishing gradient* problem. This means that the information available to train the network rapidly declines as we consider layers further away from the output. To avoid this, most deep neural networks use ReLU activation functions (see below) in their hidden layers.

3.4.2 The Tanh Function

The hyperbolic tangent tanh is also *sigmoid* in shape, but between -1 and 1.

$$f(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

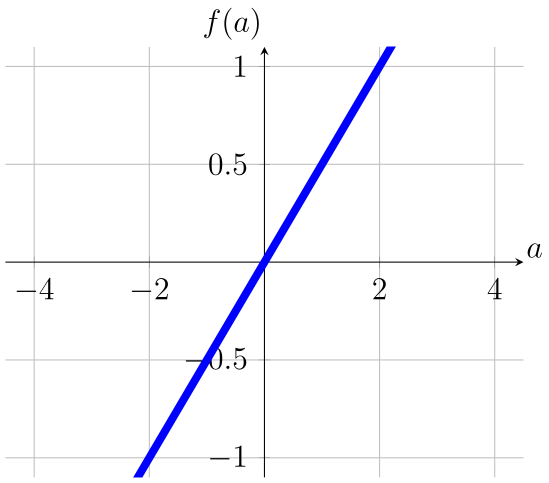


The function tanh can be expressed in terms of the logistic function σ since $\tanh(a) = 2\sigma(2a) - 1$.

3.4.3 Linear Functions

If we are performing a regression task, such as predicting the value of a house, then we do not want the output to be constrained by 0 and 1, or -1 and 1. In this case we might choose a linear activation function:

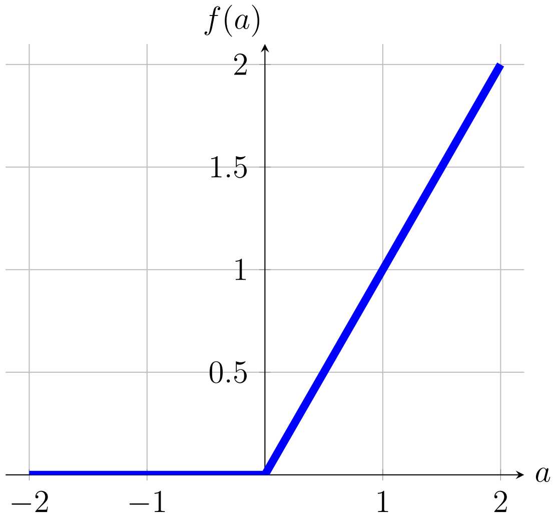
$f(a) = c\,a$
with c constant.



3.4.4 ReLU and Leaky ReLU

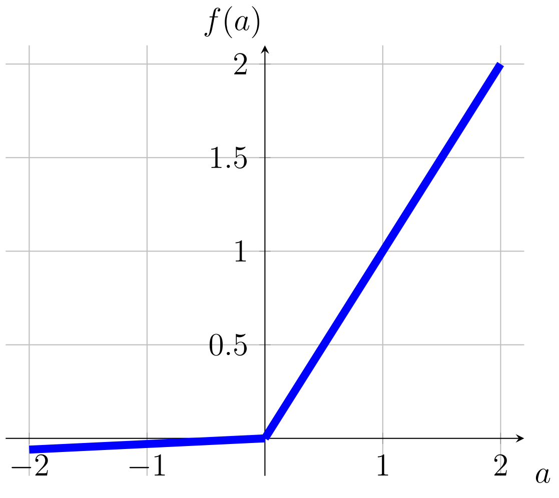
As said earlier, to get benefit from multiple layers in a network, we need a non-linear function. However as also noted, in networks with many layers, sigmoid functions encounter a problem with vanishing gradients. So the most common activation function in these networks is Rectified Linear Unit function or ReLU for short:

$$f(a) = \begin{cases} 0 & \text{for } a \leq 0 \\ a & \text{for } a \geq 0 \end{cases}$$



Although the gradient for the function is constant for positive a, it is zero for negative values. A consequence of this is that once the activation value of a ReLU neuron becomes negative, there is no gradient information available to train the neuron parameters (remember that the gradient investigates how changing the weights a little affects the error — here if we change the neuron weights so the activation becomes -1.9 instead of -2, there is no effect on the output, so no effect on the error, so the weights are unchanged). This can in theory result in neurons that “die” and take no further part in the training. One way to avoid this is the Leaky Rectified Linear Unit function:

$$f(a) = \begin{cases} 0.01\,a & \text{for } a \leq 0 \\ a & \text{for } a \geq 0 \end{cases}$$



3.4.5 Other activation functions

There are other activation functions that are sometimes used. For example step functions like the so-called Heaviside step function, or the *abs* function that is 1 if its argument is positive, and -1 if it is negative, and 0 if its zero. These were used in early neural networks, notably Rosenblatt's Perceptron. However backpropagation does not work with them, so they are less commonly used nowadays.

For other neural network architectures, such as radial basis function networks, the activation function is a Gaussian. We will consider this later in the course. Also later in the course we will consider how to handle multiclass classifications. In this case, we would ideally like one output layer neuron per class, and an output which reflects the probability of the example input belonging to each particular class. In this case the neuron outputs must sum to 1, as per all probabilities. This can be achieved using the so-called *softmax* function, that takes a vector of non-zero values and produces another vector whose elements are between 0 and 1, and sum to 1. The j^{th} component of this output vector is given by

$$f(z_j^{[N]}) = \frac{e^{z_j^{[N]}}}{\sum_{k=1}^{n^{[N]}} e^{z_j^{[N]}}}$$

3.5 Training Multilayer Neural Networks

The problem of efficiently training multilayer neural networks was a problem that halted work on neural networks in the 1970s. In the mid-1980s Rummelhart and McClelland and others rediscovered the *backpropagation algorithm* that allowed partial derivatives of the error with respect to the weights and bias to be efficiently calculated. These can then be used in training the network. The mathmatical details use the chain rule of differentiation, to work out the derivatives, starting at the output layer, and working backwards, hence the name. We will not go through the details on this course — the thing to know is that the algorithm exists.

3.6 Building and Training Neural Networks in Python

Many packages provide the ability to build and train neural networks. We will use the package Keras, which is a freely available Python module. We will first apply a neural network to the linearly inseparable data of Figure 3.2, which logistic regression was unable to classify correctly. The process comprises three steps:

1. specify the architecture in terms of layers, interconnection and activation functions
2. specify the error (aka loss) function and optimization routine

3. train the network (aka fit the model)

in Python this becomes

```
## 1 ## architecture
network = models.Sequential()
network.add(layers.Dense(20, activation='relu', input_dim=2))
network.add(layers.Dense(5, activation='relu'))
network.add(layers.Dense(5, activation='relu'))
network.add(layers.Dense(1, activation='sigmoid'))

## 2 ## error and optimizer
network.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

## 3 ## train
network.fit(X_train,y_train, epochs=300)
```

We can visually inspect the decision boundary obtained in Figure 3.3. The choice of the network architecture is rather arbitrary — experiment with different numbers of neurons and activation functions. There are a large number of practical issues that we have not addressed e.g. assessing model fit. We will start tackling these issues in the next lecture. But it is clear that multilayer feedforward neural networks can classify some linearly inseparable clusters[1]. In the lab session we will consider some more practical examples e.g. image recognition for digit classification.

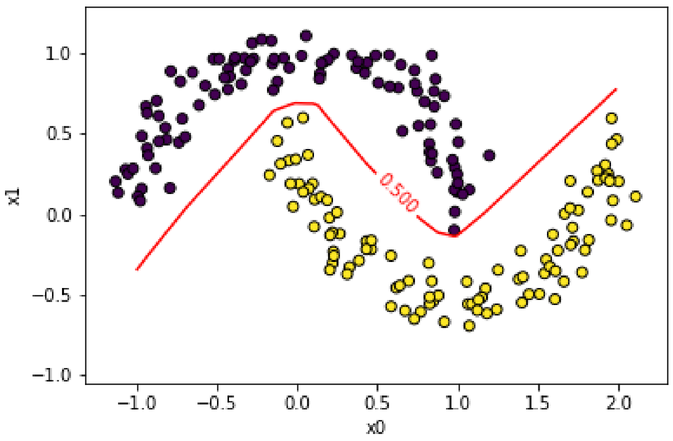


Figure 3.3: Linearly Inseparable Classes with class boundary learnt by a multilayer feedfoward neural network

[1]It can in fact be shown mathematically that such a network can approximate a continuous function, on bounded inputs, arbitrarily accurately given sufficient neurons, and some minor conditions on the activation functions.

3.7 Conclusion

Single layer networks are limited in that they cannot successfully classify classes that are linearly inseparable. Multilayer feedforward networks can achieve this, and the *backpropagation* algorithm gives us a way to efficiently train them. Common activation functions used are *sigmoid*, such as the logistic function, and *ReLU*.

In next lecture we will address some of the practicalities of using neural networks.

4 Overfitting and Underfitting

Learning Outcomes

By the end of this lecture you will be able to

1. understand techniques to evaluate and control overfitting

4.1 Introduction

Now that we have seen how a simple multilayer feedforward network works, we can start to tackle some of the practicalities of using neural networks. The most important of these, as with many machine learning algorithms, is the issue of *overfitting*.

4.2 Overfitting and Underfitting

A general and very important issue when fitting models to the data is that, perhaps surprisingly, it is not the model that best fits the training data that will be best at fitting new data. The issue is that a model that has many parameters can be adjusted to fit the idiosyncrasies of a particular sample of training data, capturing properties that are not representative of the population from which the training data was obtained. This is called *overfitting*. We want a model that has, as far as possible, just learned from the training data properties that are true of the population as a whole. The opposite of overfitting is (tada!) *underfitting*. This can be caused by a model with too few parameters, or a model that is inapproprate for the use to which it is being put. We have already seen an example of underfitting when we applied logistic regression to the two moons data of Figure 3.2.

For an example of overfitting, consider training a neural network on the two cluster data below. (We use 2D data i.e. data with 2 attributes, since it can be easily visualized. However overfitting is an issue, whatever the dimensionality of the data). Notice that

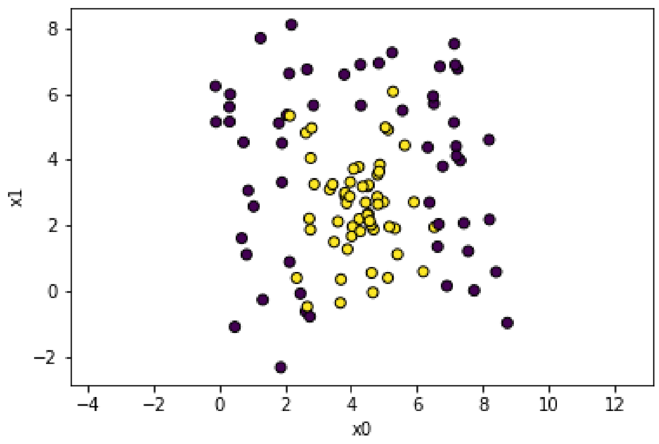


Figure 4.1: Two Classes

the two classes overlap. A neural network of 3 hidden layers of 20 relu neurons each, trained for 200 epochs, yields the decision boundary shown in Figure 4.2. This seems to be a reasonable model, even though it is only 95% accurate i.e. there are data points that are mis-classified.

If instead we use a network with 6 hidden layers of 100 relu neurons each, and train for 2000 epochs, we can get 99% classification accuracy on the training data. However the decision boundaries, shown in Figure 4.3, demonstrate this is a clear case of overfitting the training data — the decision boundary in the top right has a spike in order to correctly classify a point, and weaves its way between the points in the lower left. The shape of this region is a result of the random sample of training data, not due to any underlying feature of the data source. How do we avoid overfitting? This is the subject of the next section.

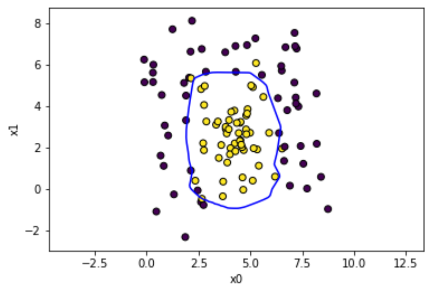


Figure 4.2: Fitting the training data

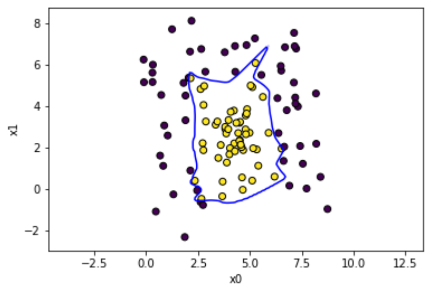


Figure 4.3: Overfitting the training data

4.3 Avoiding Overfitting

One technique to reduce overfitting is simply to collect and prepare more training data. In this way, the training data is more representative of the population as a whole. But often collecting more training data may not be a viable option. (Sometimes even though we cannot collect more data, we can simulate new data. For example if we are training a network to recognise images of cars, we can take each training data image, and crop it differently.)

Another technique is to use a simpler model. Having fewer parameters makes it harder for the model to fit the precise details of the training set and instead the parameters are used to capture general properties, ones which tend also to be present in the population as a whole. But this is also slightly unsatisfactory — in order to judge whether the model is overly complex, we have to be able to spot overfitting in the first place.

Therefore the two usual techniques are:

1. Use *validation* sets to detect overfitting.
2. Use *regularization* to penalise overly complex models

4.4 Validation Sets

4.4.1 Single Validation Set

One approach is to split the data we have into three sets:

1. the *training* set
2. the *validation* set
3. the *test* set

The test set is kept aside, to evaluate the network performance once we have finished training it. The training set is used for training the network and the validation set is used to detect overfitting. How much data should be in each set? We need to have a sufficient number of data points in the validation and test sets to have a reliable estimate of performance, and then want to use all the rest to train the network. The elements of the sets should be selected at random.

When we initially start training the network, the error function value on the training set goes down (unsurprisingly, since we are training the network by determining how to reduce the error on the input data), and so does the error function value on the validation set. However once the network starts overfitting, the error function continues to decrease on the training set, but starts to rise on the validation set. This can be seen on the toy example below: points inside the unit circle are one class, and those outside are the other. We have added a little noise to each radius measurement. Figure 4.4 shows the training and validation sets. Figure 4.5 shows the the error function value on the training and validation sets. What can be seen is that the training error pretty much always

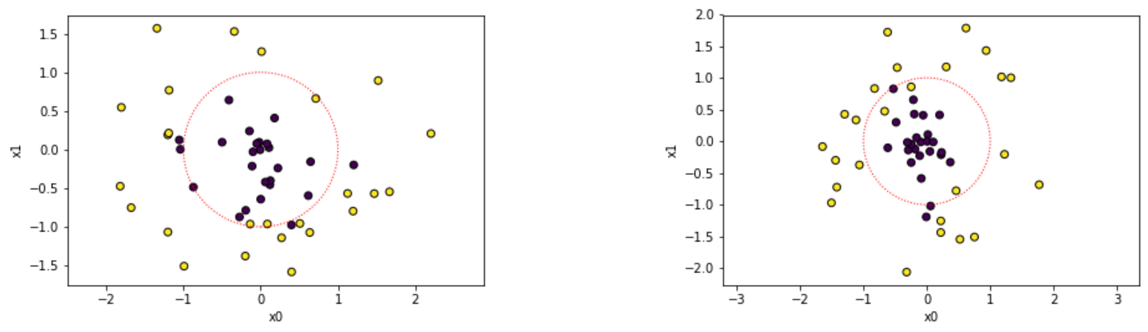


Figure 4.4: Training and Validation sets

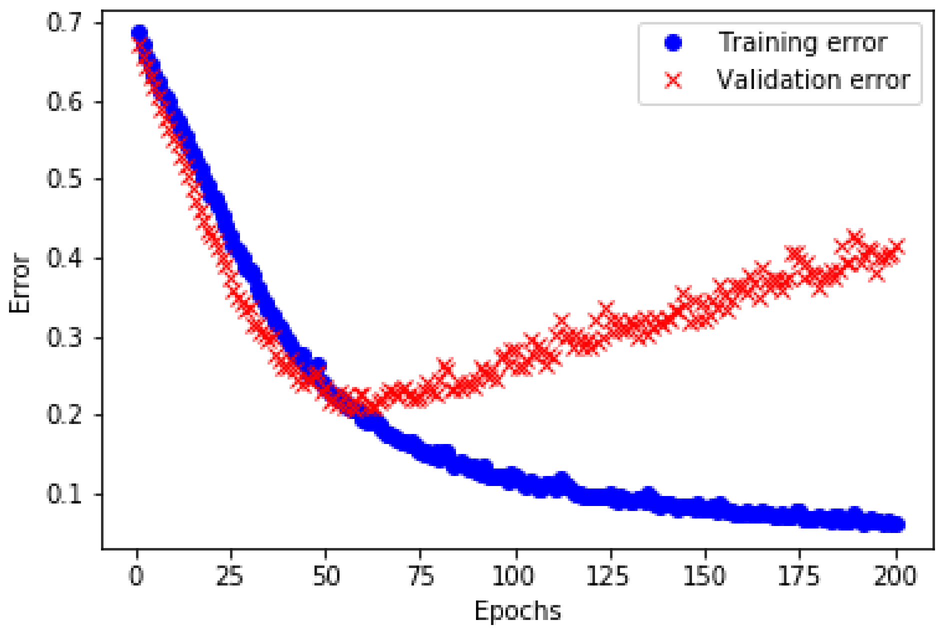


Figure 4.5: Training and Validation errors

decreases (the places where it does not decrease are where the stepsize is too large — we will consider this later). By contrast the validation error is much more noisy — it decreases until about epoch 60, but then starts to increase. This is the point at which the network is starting to overfit. For this particular toy example, where we actually know the class boundary, and where the data is 2D, we can plot the learned class boundary at various stages and see this overfitting in action in Figure 4.6.

After 10 epochs the learned class boundary is still too small. After 58 epochs the learned boundary approximates the true circular boundary. After 200 epochs the learned boundary has moved away from the shape of the true boundary, although it fits the training data better.

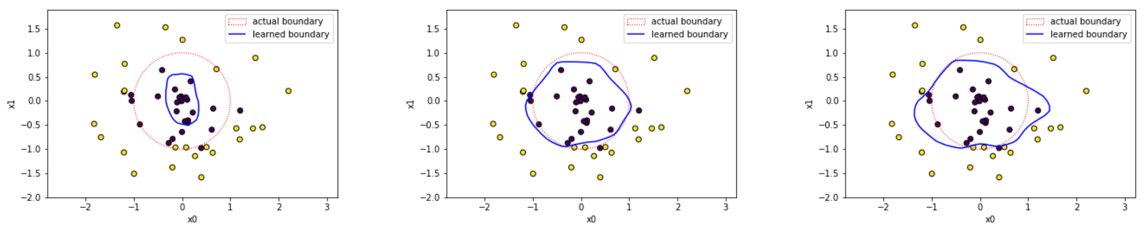


Figure 4.6: Learned class boundary after 10, 58 and 200 epochs

So to use a validation set, we observe at what point the network starts overfitting the training data. We then train our model to that number of epochs i.e. here 58, using all the data apart from the test set. This yields an error function value of 0.23 and 90.3% accuracy (whereas if we had trained to epoch 200 we would have achieved an error function value of 0.27 and accuracy 89.7%. Note too, that the best possible accuracy of a classifier on this noisy data is actually 92%.)

4.5 Regularization

The other main way of stopping overfitting is to prevent the network from learning an overcomplicated model.

4.5.1 Weight regularization

One way to do this is to minimise the number and size of parameters used, as well as minimising the error $E(w, b)$. Practically this is done by adding another function to the error function, and minimising their total value. So we minimise $J(w, b)$ where

$$J(w, b) = E(w, b) + \alpha\, \Omega(w, b)$$

Here α is a constant that we choose e.g. 0.01. It is a *hyperparameter* i.e. a parameter that shapes the model, not a parameter of the model like w or b.

- If $\Omega(w, b)$ is equal to the sum of the *absolute values* of the parameters of the model, then we are using *L1 regularization* (when applied to regularize a linear regression model rather than a neural network, it is sometimes called Lasso regression).

- If $\Omega(w, b)$ is equal to the sum of the *squares* of the parameters of the model, then we are using *L2 regularization* (when applied to regularize a linear regression model rather than a neural network, it is sometimes called Ridge regression).

Often regularization is applied just to the weights and not the biases. To achieve this in Keras, we use the keyword argument `kernel_regularizer` when defining the network layers e.g.

```
network = models.Sequential()
alpha = 0.0
network.add(layers.Dense(50,
                         activation='relu',
                         input_dim=2, kernel_regularizer=regularizers.l2(alpha)))
...
```

The effect on the class boundary for different values of α is shown in Figure 4.8.

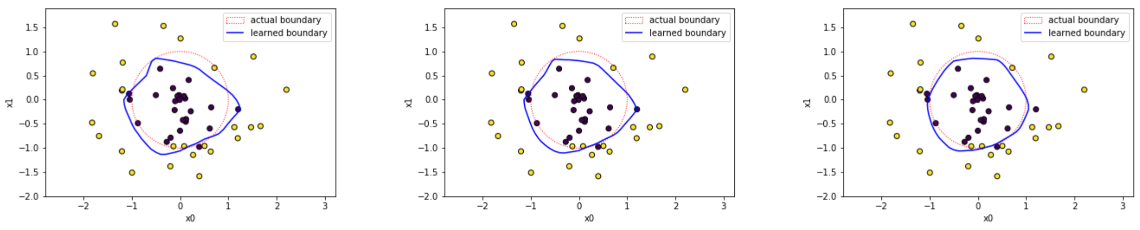


Figure 4.7: Learned class boundaries with $\alpha = 0.0,\ 0.001,\ 0.01$

4.5.2 Dropout regularization

An alternative method of regularization is so-called dropout regularization. This was proposed by Geoffrey Hinton and others in 2012, and is a popular method, particularly with Deep Neural Networks. It consists, during training, of zeroing a random proportion (say 0.2) of the outputs of a layer, and increasing the other outputs by a corresponding factor (1/(1-0.2)). It is not obvious that this will have the desired effect, but it does. In Keras, dropout can be achieved by inserting a dropout pseudolayer after each given layer:

```
network = models.Sequential()
alpha = 0.0
network.add(layers.Dense(50,
                         activation='relu',
                         input_dim=2)
network.add(layers.Dropout(0.2))
...
```

The effect on the class boundary for different values of dropout proportion is shown in Figure **??**.

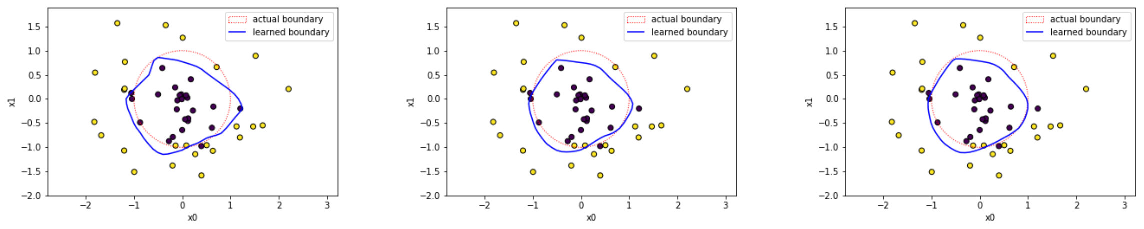


Figure 4.8: Learned class boundaries with dropout proportions 0.0, 0.1, 0.2

4.6 Conclusion

A problem encountered by many models in machine learning is that of overfitting. It can be avoided by gathering more training data, using simpler models i.e. ones with fewer parameters, using validation sets, and by regularization.

In next lecture we will continues addressing practicalities of using neural networks: the effect of stepsize, variations on batch gradient descent, initialization of weights, normalization of input data.