5 Further Practicalities of Using Neural Networks

Learning Outcomes

By the end of this lecture you will be able to

1. understand the effect of step size and other variations to training algorithms
2. understand data normalization

5.1 Introduction

Last session we looked at how to avoid overfitting, a key danger for machine learning algorithms. This session we fill in many of the other small but important details. The following sections will look at the effect of step size, batch size, using momentum or RMS scaling, weight initialization, and data normalization.

5.2 Step size

In Section 2.4 we introduced the idea of gradient descent as a way to train a parameterised model:

1. we have some measure of the error E that the current network produces, when it is applied to the training data.
2. we calculate the best "direction" in parameter space i.e. the direction that will cause the error to decrease at the maximum rate.
3. we step the parameters a small amount, the step size, in this best direction.

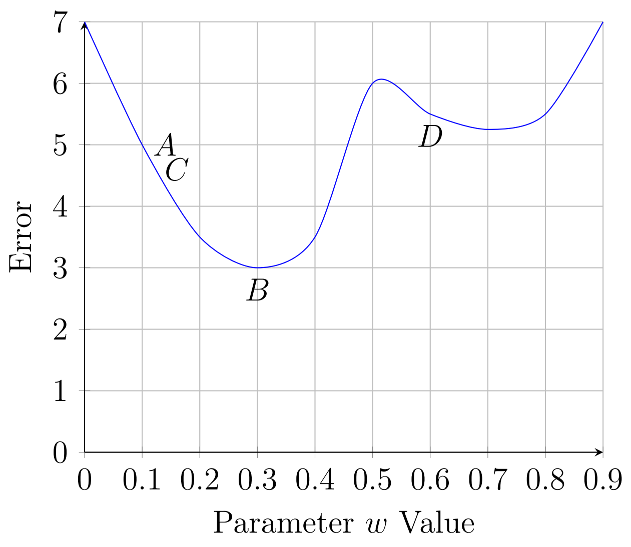

Figure 5.1: Example error "landscape"

We can think about what would happen in 1 dimension by considering the made-up error function "landscape" of a model with 1 parameter, shown in Figure 5.1.

If the parameter value is currently 0.1 then from reading the graph, the error function value is 5, and the model can be thought of as being at point A. The gradient of the error function is -20 (note: in this 1 dimensional case the gradient can only give us a direction in the sense of positive or negative). What this gradient means is that if we increase the parameter by a small amount h, then the error function will decrease by about $20h$.

Our algorithm for gradient descent in (2.4) was[1]

$$w := w - \eta \frac{dE}{dw}$$

where η is the *step size*, and $\frac{dE}{dw}$ Let's consider what happens for different step sizes, given the gradient $\frac{dE}{dw} = -20$

With a step size of 0.001, the increase in w is $0.001 \times 20 = 0.02$, the system will move to $w = 0.12$, position C on the curve. Continuing with this step size, the system will gradually converge on B, the minimum energy value. If the step size was 0.00001, the system would still converge to B but 100 times more slowly. So *too small a step size leads to slow convergence.*

[1]Note, when we have only one parameter we write $\frac{\partial E}{\partial w}$ as $\frac{dE}{dw}$. This is just a mathematical convention — it still means "the amount E increases, relative to a small change in w".

With a step size of 0.01, the increase in w is $0.01 \times 20 = 0.2$, the system will move to $w = 0.3$, position B on the curve. So it will step directly to the minimum energy position, rather lucky. If the step size was 0.025, the increase in w is $0.025 \times 20 = 0.5$, the system will move to $w = 0.6$, position D on the curve. So the error has actually increased to 5.5. Furthermore the next step will take the system further to the right, away from the minimum error position at B. So *too large a step size prevents convergence* leading to the error value oscillating and jumping around.

To see this behaviour in a neural network, consider the first neural network we developed in Section 3.6, which could recognise the two moon data set. We created it using the code:

```
## 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 will modify step 2, to allow us to explicitly set the step size, and also to switch to the most basic gradient descent algorithm.

```
## 2 ## error and optimizer
opt = optimizers.SGD(0.001)
network.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
```

If we repeat this for 4 different step sizes and plot the error function values during training for each step size, we get Figure 5.2. It can be seen (for this particular example!) that a step size of 0.01 is too small as convergence is too slow. A step size of 0.2 looks good. A step size of 0.5 is too large, as it causes the training to oscillate.

Of course, there is no rule that says we need to use the same step size through out training.

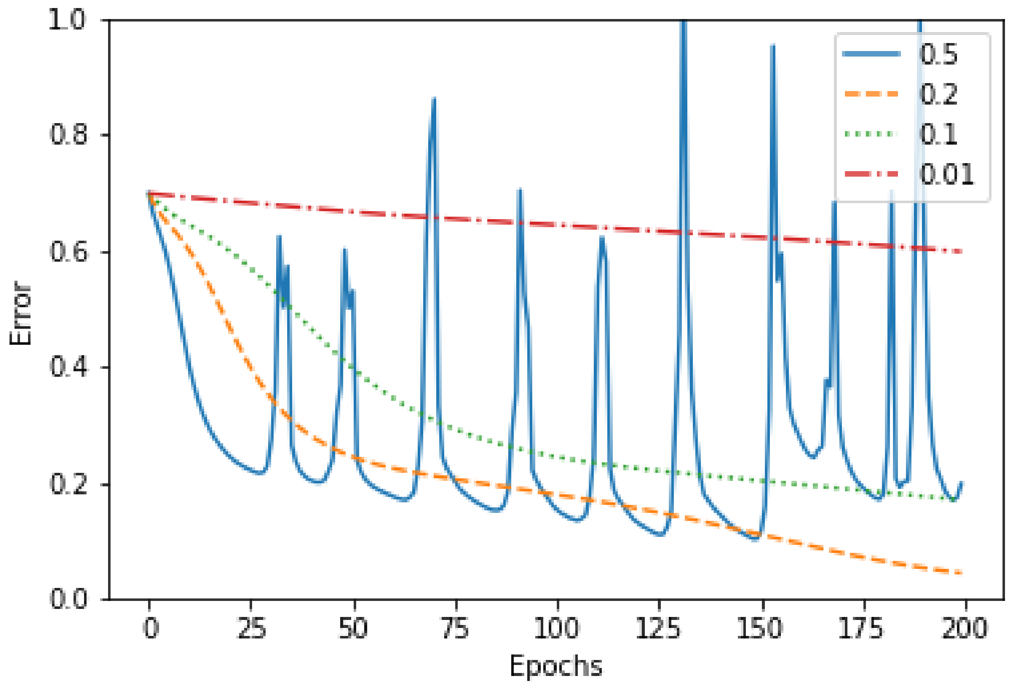

Figure 5.2: The training error plots for different step sizes

5.3 Batch Size

Our existing model of gradient descent in Equation (2.4) involved calculating the gradient of the error function with respect to the paramater values, *where the error function was evaluated over all the examples in the training set.* This is termed *batched gradient descent.*

Batched gradient descent is slightly wasteful in that, we can get a good estimate of the correct direction in which to adjust the parameters, using just a sample from the training set. This sample is called a *mini-batch* and this approach is called *mini-batch gradient descent.* Its advantage is that this is likely to be faster overall, though the step directions will be slightly more "noisy", as they are based on samples rather than the entire training data set. The gain of using a sample to set the direction is offset slightly by the fact processing a bigger batch using a matrix is much faster than looping over several smaller batches. The batch size used varies, often a power of 2 is used — the default value in Keras is 32. Setting different batch sizes in Keras is done using the `batch_size` keyword parameter:

```
## 3 ## train
network.fit(X_train,y_train, epochs = 300, batch_size=128)
```

If gradient descent is done with a batch size of 1, it is called *stochastic gradient descent (SGD)*. This will tend to be slower than mini-batch gradient descent, since gains of using a sample for the direction, are outweighed by the inefficiency of using a loop rather than a matrix. However if the training data is being generated in real-time, SGD may be the best option.

The performance of various different batch sizes on the two moons problem is shown in Figure 5.3. We have used the step size 0.2, and it can be seen that the fastest convergence in Figure 5.2 can be improved upon by using mini-batches. Smaller batch sizes tend to take fewer epochs. They also tend to be noisier, to the extent that the algorithm fails to converge to a minimum error, with batch sizes of 1 or 8.

However the epochs for smaller batch sizes take longer as we switch to using loops rather than matrix multiplication. Looking at the actual run times on a particular computer for this particular problem, we see that the batch sizes of 64 and 32 have similar performance, better than the batch sizes of 8 or 200.

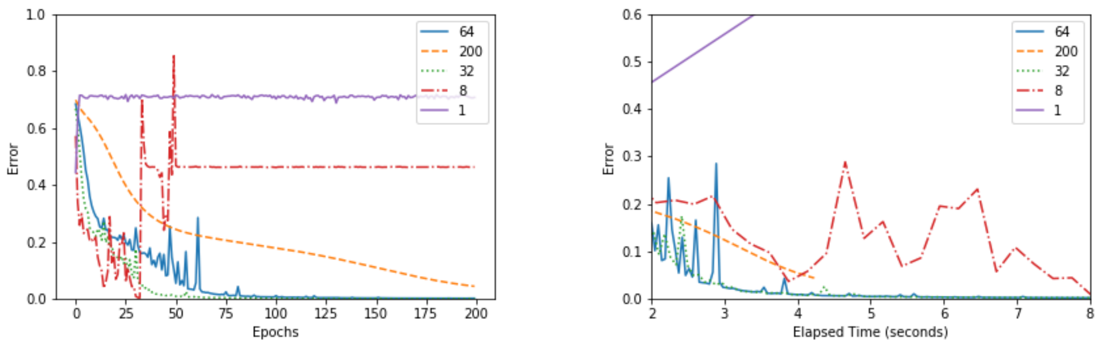

Figure 5.3: Error curves for different batch sizes

5.4 Algorithms

5.4.1 First order and second order algorithms

Training a neural network is an optimization problem: we are in essence trying to find the optimal set of parameter values that minimise a given function, the error function. Optimization is a heavily researched field and there are many algorithms available. As well as methods that just use the gradient (the first derivative), called *first order methods*, there are methods that also use the matrix of second derivatives, called the Hessian, resulting in so-called *second order methods*.

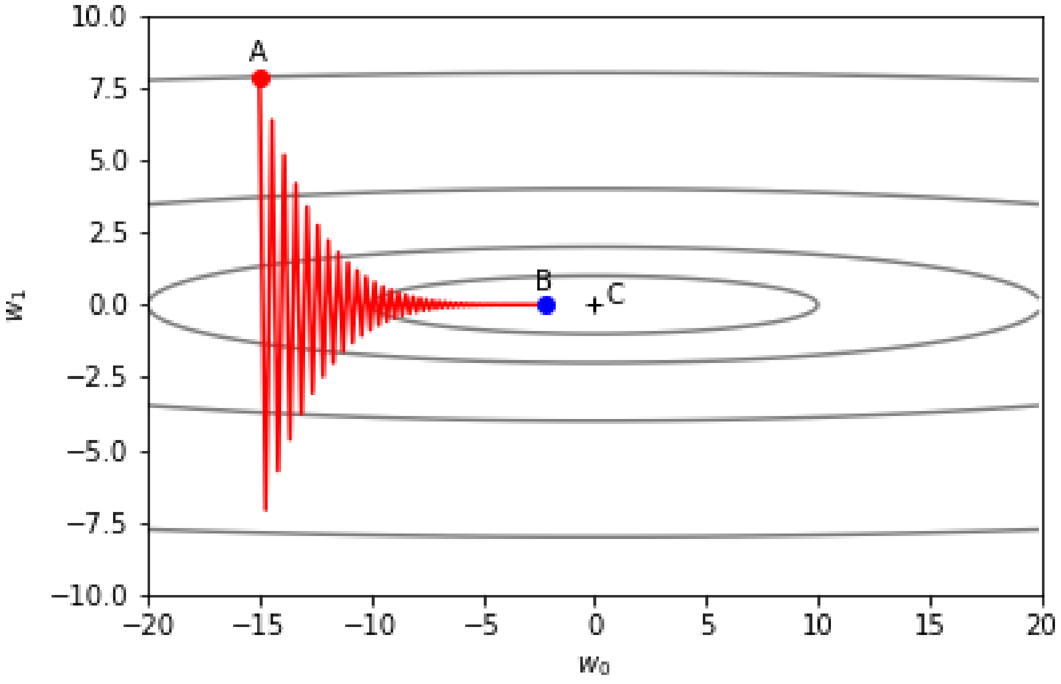

Figure 5.4: 100 steps of gradient descent in a narrow basin

For some problems second order methods can be more efficient — you may have come across the second order Newton-Raphson method when solving problems with one parameter. However there is an issue that for a model with N parameters, the Hessian will have N^2 terms, and when N is large it may be impractical to calculate this. Neural networks may have millions of parameters, and so we will restrict our attention to improved first order methods. We will investigate a couple of simple ways for improving our gradient descent algorithm of Equation (2.4). To do that we will first need to understand the possible problems.

5.4.2 Momentum

Again considering a 2D problem, so we can visualize it easily, consider Figure 5.7. The grey ellipses are contour lines of the function we are trying to minimise. If our initial parameter values are at A, then 100 gradient descent steps reaches B, approaching the minimum are C. As can be seen, a lot of time is spent stepping backward and forward across the narrow valley.

Let us write our gradient descent method as

$$w^{t+1} = w^t - \eta\,\nabla E$$

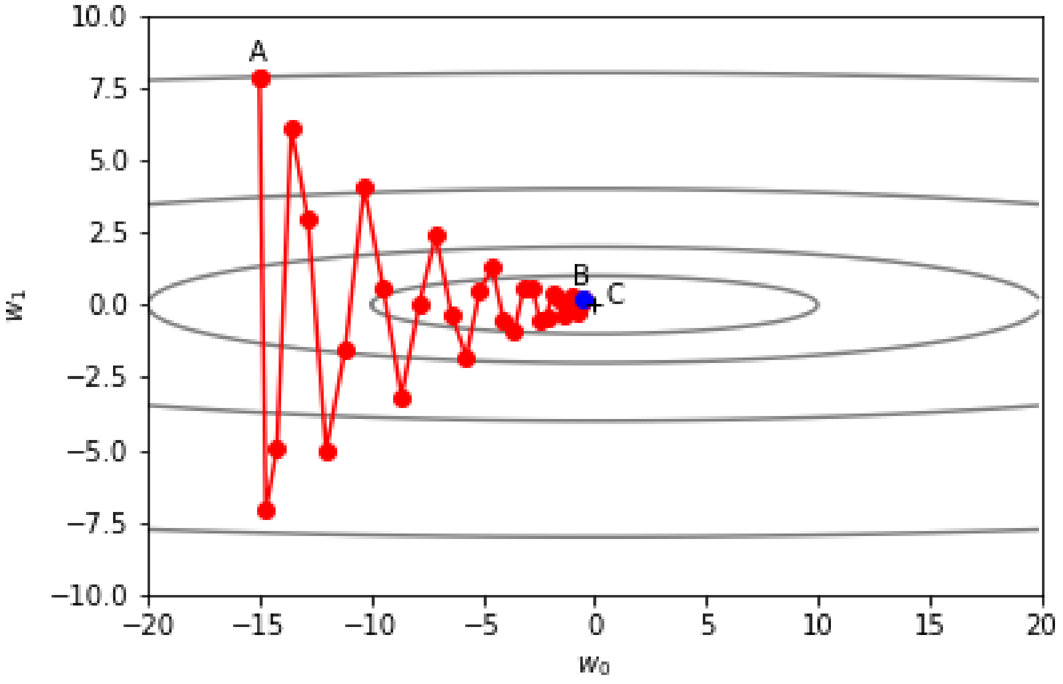

Figure 5.5: 30 steps of gradient descent with momentum in a narrow basin

meaning that the parameters at time $t + 1$ are those at time t plus a small step in the direction opposite to that of greatest increase of E. We can modify gradient this algorithm by keeping a proportion of the previous step each time:

$$w^{t+1} = w^t - \eta \, \nabla E + \mu(w^t - w^{t-1})$$

In effect we have a decaying average of past steps, and this average tends to attenuate the high frequency oscillations, and emphasize the lower frequency components: if steps consistently have a component in a particular direction then this component will build up. Figure 5.5 shows that adding a momentum term, here with $\mu = 0.8$, means that the minimum can be approached more closely than before with only 30 steps.

Applying this idea to the two moons problem, we can set the momentum coefficient in the SGD optimizer using a keyword parameter:

```
opt = optimizers.SGD(0.2, momentum = 0.8)
network.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
```

Looking at the resultant plots in Figure 5.6 for three different momentum coefficients, we can see that, in this particular case, a value of 0.8 outperforms one of 0.5 or 0. Momentum will tend to add noise, since the step size may now sometimes be too big.

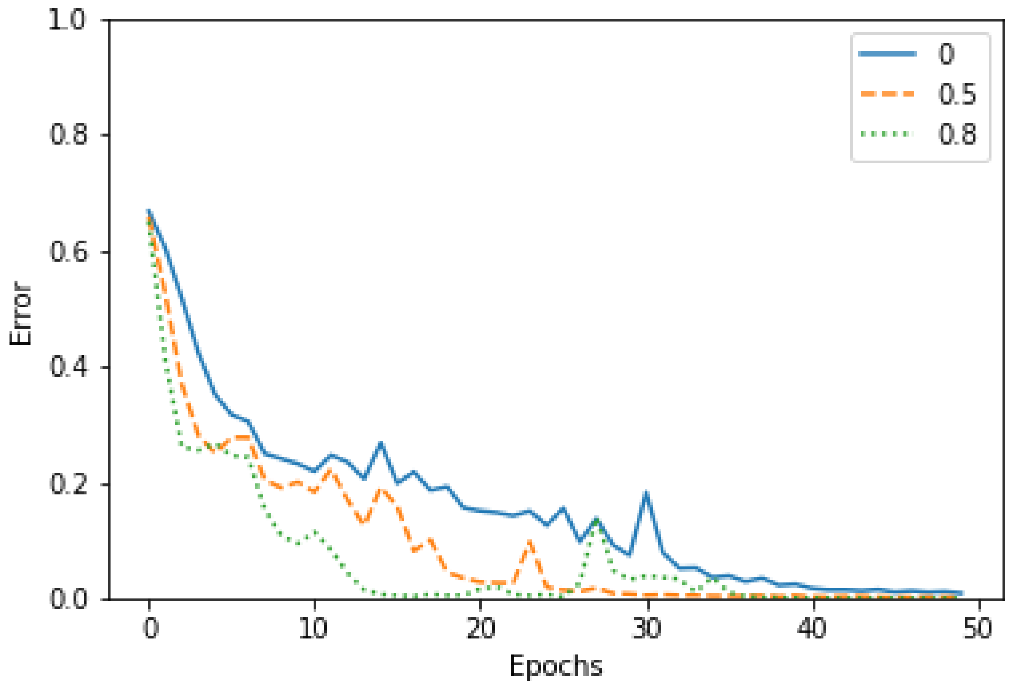

Figure 5.6: Error value trajectories for three different coefficients of momentum

5.4.3 RMSprop and Adam

RMSprop is an idea proposed by neural network researcher Geoffrey Hinton. We do not spell out full details here. In essence it is about trying to scale the step size separately for each parameter. Referring to Figure **??**, parameters with small gradient such as w_0 have their step size increased, and parameters with large gradient such as w_1 have their step size decreased. The scaling factor used is a weighted average of something called the root mean square of the step sizes, hence the RMS in the name. It only takes 5 steps of gradient descent with RMS scaling, with parameter value 0.99, to approach the minimum i.e. a 20 times speed up over basic gradient descent.

Using momentum and RMS scaling is usually better still. This in essence gives the Adaptive Moment Estimation method, known as the Adam method. It can be selected in Keras via

```
opt = optimizers.Adam(0.2)
network.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
```

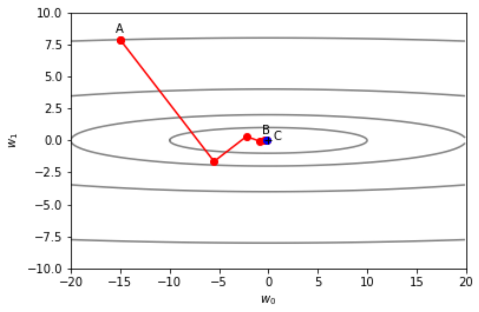

Figure 5.7: Five steps of gradient descent with RMS scaling, in a narrow basin

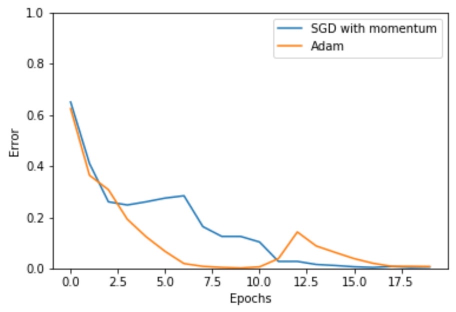

Figure 5.8: Performance of Adam optimization on the two moons problem

We can compare Adam's performance on the two moons problem with our previous best (SGD with a 0.8 coefficient for momentum), as shown in Figure 5.8. In this particular case their performance is not dissimilar. In general RMSprop or Adam are suitable general purpose algorithms for training neural networks, and usually parameter tuning of the step size is the most important.

5.5 Parameter initialization

In a neural network model, if all the weights and biases are initially zero, then the output is zero. But furthermore, changing any one weight by a small amount has no effect on the output and hence no effect on the error. This means that in our gradient descent algorithm of Equation (2.4), $\frac{\partial E}{\partial w}$ *is zero*. And this means that none of the weights will ever change!

In Keras, parameter initialization is done automatically behind the scenes. By default, in densely connected layers all the weights are set to small random values using according to a scheme called *Glorot initialization*, named after Xavier Glorot, the researcher who first proposed it. The biases *are* set to zero, but this is ok, since with non-zero weights, changing the biases does effect the output and hence error. Hence the gradient descent algorithm will adjust them effectively. We can see what happens when we initialize all weights to zero, using the `kernel_initializer` keyword parameter:

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

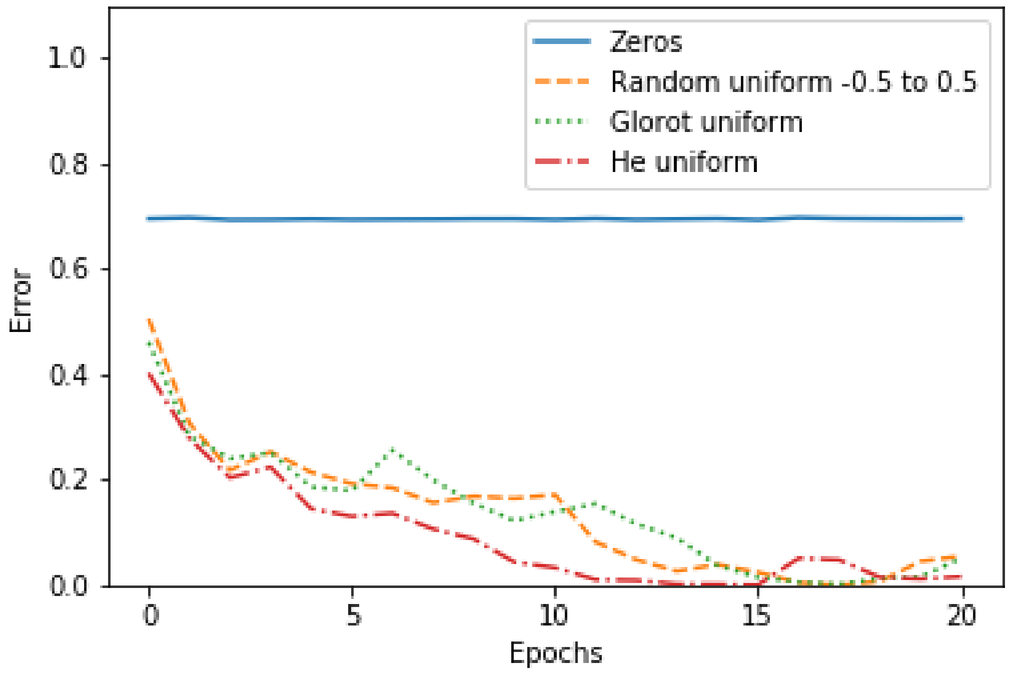

Figure 5.9: The result of different weight initialization strategies

```
opt=optimizers.SGD(0.2)
network.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
history = network.fit(X_train,y_train, epochs=50, verbose = 0)
train_errors = history.history['loss']
```

If we graph the training error with weights initialized to zero, we get the corresponding line in Figure 5.9. We see the error is unchanged, because with the weights set to zero the gradient is also zero. This does the raise two further questions: might we encounter this problem of zero gradient, a bit like a sailing ship becoming becalmed in the doldrums, in the course of training a network? And secondly, what is the best way to initialize the weights?

The answer to the first problem of encountering zero gradient spots during training, is yes. However the noise injected by using mini-batches and momentum mitigates against this. The answer to the second question is more complex. For relatively shallow neural networks, such as the one used in our two moons problem, small random values for initial weights works fine. For deep neural networks, is is desired that the initial activation values of the neurons are all small random numbers. In particular the activation values should not die off to zero, or keep increasing as we move forward through the layers of the network. This is achieved by taking into account the number of inputs that a neuron receives. Both Glorot and He initialization achieve this.

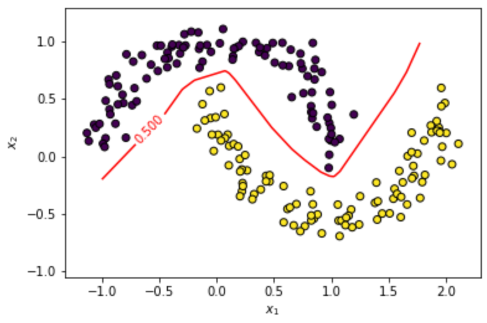

Figure 5.10: Using Normalized Data

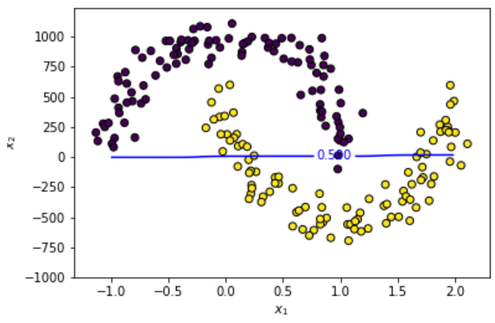

Figure 5.11: Using Non-normalized Data

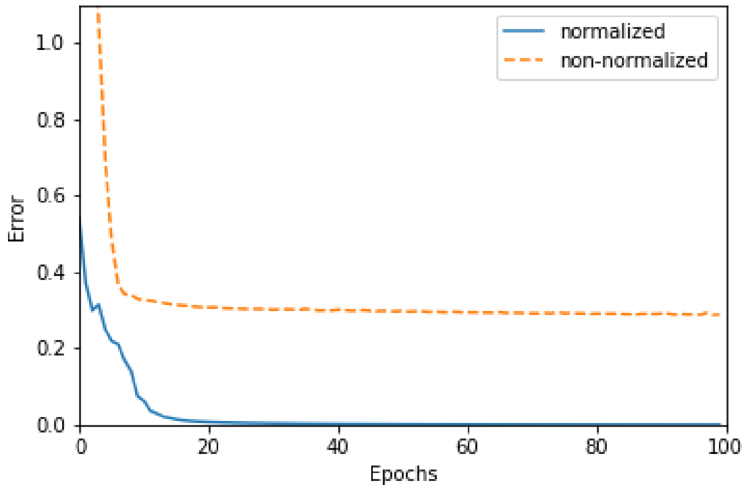

Figure 5.12: Error for normalized and non-normalized data

A takeaway message here is that if you are using Keras, the default initialization is likely to be fine and not need tuning. If however you are for some reason using a tool without sensible or appropriate default options, you will need to ensure initialization happens appropriately.

5.6 Data Normalization

If the input features into a neural network have very different scales, then learning will not work well. For example suppose the first component of x, x_1, ranges uniformly between -1 and 1, and the second x_2 ranges uniformly between -1000 and 1000. Then with random initialization of the weights, the x_2 feature will have a 1000 times more effect on the output and error, so effort will be spent on tuning the parameters relating to x_2, at the expense of tuning the parameters relating to x_1. Furthermore, the step size will have to be reduced to enable convergence to happen. We can see the effect of this in Figure 5.11 if we rescale our two moons problem to make the x_2 range one thousand times bigger than that for x_1. The error function values as training progresses for the two cases is shown in Figure 5.12.

Data normalization is usually effected by subtracting the mean and then dividing by the standard deviation. This results in the features having mean zero and unit variance. A related technique that has proved useful in deep neural networks is so-called batch normalization, where normalization is applied after each layer of the network — we do not go into this further here.

5.7 Conclusion

In order to use neural networks successfully, a number of choices need to be made: the step size for the learning algorithm, the learning algorithm itself and the batch size. The weights need to be initialized away from zero, and the data should be normalized.