EARL
ENTERPRISE APPLICATIONS OF THE R LANGUAGE

Deep Learning with Keras in R

Workshop 3 at the EARL Conference

Date: 11 September 2018
Time: 10am – 1pm
Room: Tower 3

@ earl-team@mango-solutions.com
+44 (0)1249 705 450
mango-solutions.com

Chapter 1
Introduction to Deep Learning in R

1.1 Introduction to the Training

1.1.1 Course Aims

This course has been designed to provide a practical introduction to building deep neural network models in R. The theory of machine learning neural networks will be touched upon, but the main focus is on the implementation in R, specifically with the Keras interface for R. It is assumed that you will have either attended an Introduction to R course or have equivalent experience with the R language.

1.1.2 Course Materials

Items appearing in this material are sometimes given a special appearance to set them apart from regular text. Here's how they look:

```
> This is a section of code                    # This is a comment
```

🔴	A warning, typically describing non-intuitive aspects of the R language
🟠	A tip: additional features of R or "shortcuts" based on user experience
🟢	Exercises to be performed during (or after) the training course

1.1.3 Course Script and Exercise Answers

A great deal of code will be executed within R during the delivery of this training. This includes the answers to each exercise, as well as other code written to answer questions that arise. Following the course, you will be sent a script containing all the code that was executed.

1.2 What is Deep Learning?

Deep learning sits within the larger field of machine learning as a method for solving problems by abstracting data through a series of sequential transformations. That is to say: If we start with measurements that are easy to understand, such as the brightness of a pixel, a sensor measurement, or a grouping, and then we apply multiple transformations. These transformations gradually turn the data into something that better represents the problem we are trying to solve but is no longer easy to recognise the input.

In the case of images this can be easy to describe. We do not recognise a cat because of pixels. It is features, such as the particular point on an ear, or the placement of the eyes. These kinds of abstract features need to be learnt from experience, and once learnt can be reused for a number of tasks. This reusability is a key feature of deep learning above other machine learning methods.

The vast majority of deep learning models are neural networks, and these are the only models we will consider in this course.

1.3 What Problems Does It Solve?

Deep learning has some significant strengths and weaknesses when compared to more traditional machine learning techniques. When choosing the right approach to a new problem it's important to think about whether deep learning is the best choice.

1.3.1 Unstructured

According to the founder of Kaggle, the machine learning competition and open data sets website, Anthony Goldbloom, the winning approaches in their competitions fall into two broad categories:

Structured problems, with well organised tables of data, perhaps extracted from a relational database. These competitions are typically won by a combination of careful feature engineering and ensemble learners, such as gradient boosting.

Unstructured problems, such as images, natural language, or raw sensor data, tend to be best solved by deep neural networks. For these problems it is extremely difficult for humans to build meaningful features, but for a neural network this is exactly what it does. Neural networks learn the features, as well as the relationships between them.

1.3.2 Big

The downside to the extra freedom to learn your own features is that deep learning typically needs a lot of data to be effective. Compared to a traditional machine learning algorithm you may require 10 to 100 times more data before you begin to realise the benefits above careful feature engineering.

Modern deep learning frameworks are well placed for handling large amounts of data. With highly optimised, GPU accelerated, numerical libraries, and mini-batching to process data well beyond the bounds of RAM.

1.3.3 Familiar

If you don't have lots of data deep learning may still have something to offer. This is down to the reusability mentioned in the previous section. It is typical for practioners of deep learning to spend a lot of time learning to distinguish pictures of cats from pictures of dogs. After training on tens of thousands of labelled images your neural network will have learnt useful features for this task.

It turns out that these features can also be useful for other tasks, beyond the original problem. For example, you may wish to count animals in a scene, or to distinguish domestic cats from big cats. By using the features from a *pre-trained network* one can reduce the amount of data required to start seeing results with the new problem.

Pre-trained networks are beyond the scope of this course, but we will briefly touch on them in a later chaper.

1.4 Why Now?

Neural networks have been around since the mid-20th century, but their impact has not been felt until relatively recently. This is mostly down to a number of factors coming together at the same time.

1.4.1 Algorithmic

A breakthrough in the underlying algorithm (backpropagation), behind training neural networks allowed networks to grow deeper for the same computational cost.

1.4.2 Computational

The continued massive increase in computer power, further boosted by the use of graphical processing units (GPUs) in research, has made neural networks solvable in an acceptable amount of time.

1.4.3 Big Data

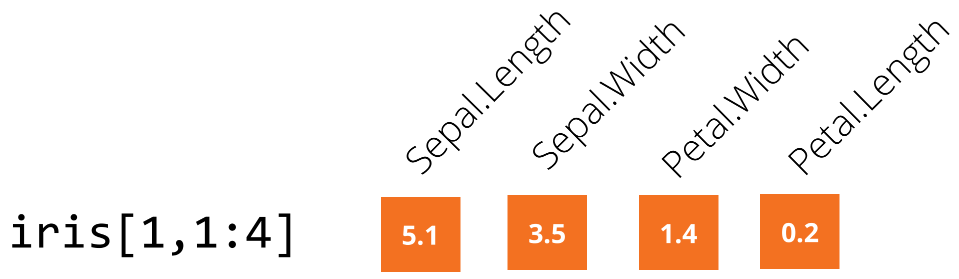
As previously mentioned, neural networks need a lot of data, and we are collecting more data than ever before. This is deep learning's raw material.

1.5 Neural Networks

Inspired by biological neurons, the basic idea of a neural network is to connect a large number of simple units (neurons), that each do a simple thing, together in such a way that rich behaviour can emerge.

1.5.1 Input Layer

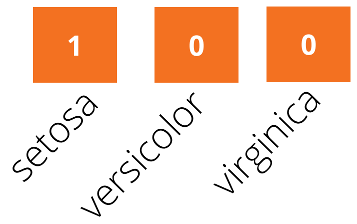
In an artificial neural network (ANN) each neuron is a node in a network and each node carries a single number. The neurons are arranged in layers and at the start we would put a single observation of our feature variables. For the famous `iris` data set this would look something like the following diagram:



Note that we have not yet connected these neurons to anything yet, we have simply assigned each feature variable to an input neuron.

1.5.2 Output Layer

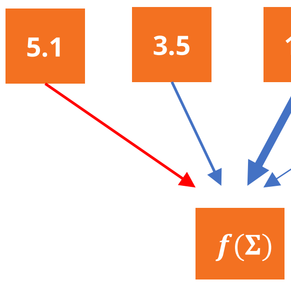
The target variable, Species in the case of `iris`, is what we want to see in the output layer. The input above was for a "setosa" iris, so the output we would like to see would look like the layer below.



This way of representing categorical variables across multiple neurons is known as "dummy variables" or "one hot encoding", we will cover more on this later.

1.5.3 Connecting Layers

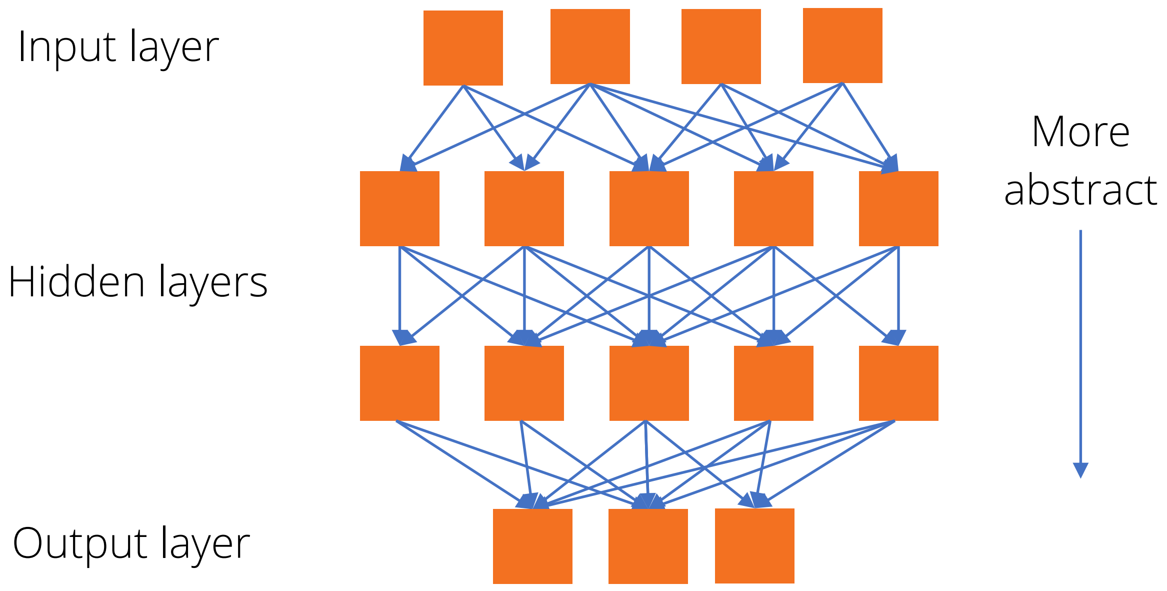
To turn the neurons into a neural network we need to add connections. Or in network language, edges. Each neuron in the input layer is connected to neurons in the next layer with a weighted edge. In the picture below the width of the arrow represents the magnitude of the weight, and the colour whether it is positive (blue) or negative (red).



The value on the bottom neuron comes from taking the weighted sum, Σ, of the input and passing that into an *activation function*, f. We will discuss activation functions later in the course. For now, it is a simple function that can be anything from the identity (so just return the weighted sum) to a step function where the neuron is off or on (firing) depending whether the input passes a threshold.

1.5.4 The deep in deep learning

The power of a neural network comes from its connectivity. From the simple picture above, we build larger networks with many more connections and more layers. These middle layers are called *hidden layers*. Each hidden layer allows us to transform its input into a more abstract representation that better relates to the target output.



There are many ways to connect a neural network. The above example is known as a “feed forward” network. The choice of architecture will depend on the type of problem. A neural network is “deep” if it has more than one hidden layer.

1.5.5 The learning in deep learning

The key components in a neural network are the connections, specifically the weights. It is the weights that we need to learn to make good predictions on new data. In the picture above, each blue line represents a weight and we need to learn a value for each one. This means that neural network models typically have a very large number of parameters.

The details of how the weights are computed are beyond the scope of this course. An algorithm, known as back-propogation, allows for the efficient computation of weights. As a high-level operator of a neural network this computation is handled for us by lower level software. In the case of this course that software is Google’s TensorFlow and the Keras package.

1.6 Tensorflow

TensorFlow, https://www.tensorflow.org, was developed at Google and released as open source software in 2015. TensorFlow is a general-purpose library that allows the user to write down a series of mathematical equations through its Python interface, which it then turns into a dataflow graph. It can run on CPU and multiple GPUs, as well as new TPU (Tensor Processing Units), and can be distributed across a cluster.

TensorFlow is extremely fast at performing matrix operations which makes it perfect for use in neural networks. Many specialised functions have been added to TensorFlow specifically for the purposes of deep learning, and indeed this is its primary use case.

While the main interface to TensorFlow is written in Python, RStudio created the tensorflow R package, https://tensorflow.rstudio.com, to provide an R interface (API). This works using the reticulate R package that connects R to Python. All functionality that is available in the Python API is available to R users and the extra R layer does not add any significant overhead, especially for larger computations.

The tensorflow package provides what RStudio calls the “Core API” and for most of this course we will not interact with it directly. Instead TensorFlow will serve as the “backend” to a higher-level library, Keras, built specifically for building neural networks.

1.7 Keras

Keras is an open source library, https://keras.io, developed by François Chollet and released in 2015. Keras is a high-level library designed to make it quick and easy to build neural networks without getting bogged down in computational details. It states:

> *"Being able to go from idea to result with the least possible delay is the key to doing good research".*

Unlike TensorFlow, Keras does not perform the computations behind the neural networks. Instead it defers to a backend provider. Currently supported are TensorFlow (described above), Microsoft's CNTK (recently rebranded Microsoft Cognitive Toolkit), and Theano (soon to be retired).

Instead Keras provides high-level APIs specifically tailored for neural networks. This includes building, training, and evaluating your model. As well as numerous helper functions for converting and reshaping data for modelling.

Keras is written in Python, and once again RStudio have built the keras package for R, https://keras.rstudio.com, that provides an R interface. After installing keras from CRAN in the usual way, we need to install the Keras, and TensorFlow Python packages. This only needs to be done once.

```
> library(keras)
> install_keras()
Creating r-tensorflow conda environment for TensorFlow
installation...
```

If you do not already have it, this will download and install the Anaconda Python distribution which may take a while, then it installs Keras and TensorFlow into a special environment. For custom setup, such as GPU support (Nvidia graphics cards) see the RStudio documentation at https://keras.rstudio.com. We can check it installed correctly with:

```
> is_keras_available()
[1] TRUE
```

1.8 Alternatives

There a a few options for building deep learning models in R.

MXNet, https://mxnet.incubator.apache.org/api/r/index.html is a standalone deep learning framework. It provides both the backend (equivalent to TensorFlow) and the front end (equivalent to Keras) and fully supports R.

H2O Deep Water, https://www.h2o.ai/deep-water/, is part of the H2O ecosystem and, like H2O itself, is a separate application from R. However, an R interface is provided that allows building neural networks on top of supported backends, such as MXNet and TensorFlow.

The machine learning CRAN task view has a more comprehensive list of available packages. See https://cran.r-project.org/web/views/MachineLearning.html.

1.9 Further Reading

If you want to know more about deep learning we recommend the book "Deep Learning with R" by Francois Chollet and J.J. Allaire, which focuses specifically on the R implementation of keras.

Chapter 2 Getting Started with Keras

2.1 Introduction

The **keras** package for R is actually an interface to a Python library. For the rest of this chapter we will assume that all of the correct Python libraries are available to you and **keras** is installed and loaded

```
> library(keras)
```

2.2 Getting Data Ready for Deep Learning

Like many machine learning techniques, getting our data ready to go into our model is often the trickiest part and can take a large amount of our time. Throughout this chapter we are going to assume that our data is already reasonably structured. In this section we will overview a few final steps you need to take to get your data ready for modelling.

To keep things simple and focus on the technique we are going to work with the `iris` data.

2.2.1 Training and Test Sets

So that we can test our model we need to split the data into training and test sets. There are many ways to do this in R but here we are going to use functions from the **rsample** package.

```
> library(rsample)
>
> data_split <- initial_split(iris, strata = "Species", prop = 0.8)
>
> fullData <- list(train = analysis(data_split),
+                  test = assessment(data_split))
```

This will put 80% of the data into the training set, and 20% into the test set.

As we have split our data we need to remember that any manipulations must be applied to both the training and test sets. Creating a list makes this easier as we can use one of the many apply functionalities in R.

2.2.2 Pre-processing

There are a few pre-processing steps that we need to do in order to get our data ready for putting into a neural network. In general, for machine learning we must be careful that any pre-processing does not pass information from the training set to the test set. For example, using a mean or a median in the pre-processing will be affected by values in the test set.

The **recipes** package is designed to help us with common pre-processing steps in a reusable way. There are three steps:

1. Define a "recipe" describing the pre-processing you want to do.
2. "prep" using the training data to compute values.
3. "bake" the recipe, which applies the pre-processing to the data.

We'll first create an empty recipe and divide variables into output (target) and predictors (features).

```
> library(recipes)
> empty_recipe <- recipe(Species ~ ., data = fullData$train)
> empty_recipe
Data Recipe

Inputs:

      role #variables
   outcome          1
 predictor          4
```

We'll then add "steps" to the recipe in the following sections.

2.2.3 Creating Dummy Variables

Unlike most R based modelling techniques that will handle factor variables for us, **keras** expects us to have pre-created our dummy variables. While there are a number of ways to do this in R, **recipes** has a pre-built step, `step_dummy`, to create the dummy variable model matrix. The option, `one_hot = TRUE`, ensures that every factor level gets a column with no reference level.

```
> dummy_recipe <- empty_recipe %>%
+   step_dummy(Species, one_hot = TRUE, role = "outcome")
```

On its own this does not perform the pre-processing. We need to "prep" and "bake".

```
> dummy_recipe %>%
+   prep(fullData$train) %>%
+   bake(fullData$train, all_outcomes()) %>%
+   head()
# A tibble: 6 x 3
  Species_setosa Species_versicolor Species_virginica
           <dbl>              <dbl>             <dbl>
1           1.00                  0                 0
2           1.00                  0                 0
3           1.00                  0                 0
4           1.00                  0                 0
5           1.00                  0                 0
6           1.00                  0                 0
```

The `all_outcomes()` selector restricts the output to just target variables.

2.2.4 Centering and Scaling Numeric Data

Neural networks are a class of model that are particularly sensitive to the range of values in our data. If two variables are on very different scales this can significantly impact the ability of the model to learn. For this reason, we typically rescale all of the data so that they represent a standard normal distribution i.e. mean of zero and standard deviation of one.

```
> scale_recipe <- empty_recipe %>%
+   step_center(all_predictors()) %>%
+   step_scale(all_predictors())
>
> scale_recipe %>%
+   prep(fullData$train) %>%
+   bake(fullData$train,
+        all_predictors()) %>%
+   head()
# A tibble: 6 x 4
  Sepal.Length Sepal.Width Petal.Length Petal.Width
         <dbl>       <dbl>        <dbl>       <dbl>
1       -0.875      1.02          -1.33       -1.31
2       -1.34       0.331         -1.38       -1.31
3       -1.45       0.0996        -1.27       -1.31
4       -0.991      1.26          -1.33       -1.31
5       -0.528      1.95          -1.16       -1.05
6       -1.45       0.793         -1.33       -1.18
```

2.2.5 Missing Values

Many machine learning techniques in R will automatically handle missing data for us. However, when building neural networks, we need to have pre-handled any missing data. A simple strategy for this is to set any missing values to 0, effectively the mean of the data once scaled. There are many options in R for substituting missing values including the `replace` function or the `replace_na` function from the **tidyr** package.

In recipes there are several steps available for imputing missing data. From a simple mean with `step_meanimpute()`, to a nearest neighbours model with `step_knnimpute()`. There are no missing values in `iris` so we will leave this for now.

2.2.6 Chaining Steps

Putting everything together we can now create, and prep, our final recipe for the `iris` data using the pipe operator to chain multiple steps together.

```
> iris_recipe <- recipe(Species ~ ., data = fullData$train) %>%
+   step_dummy(Species, one_hot = TRUE, role = "outcome") %>%
+   step_center(all_predictors()) %>%
+   step_scale(all_predictors()) %>%
+   prep(training = fullData$train)
```

2.2.7 Feature and Target Matrices

A final step to prepare our data is to convert to a matrix or array. Whilst not strictly necessary when our data is in data frame format, as named lists can be used when working with **keras**, we will often be working with data that doesn't come from a data frame and as such must be in this format. Here we are using map from the **purr** package to apply our recipe to training and test and ensuring a matrix output.

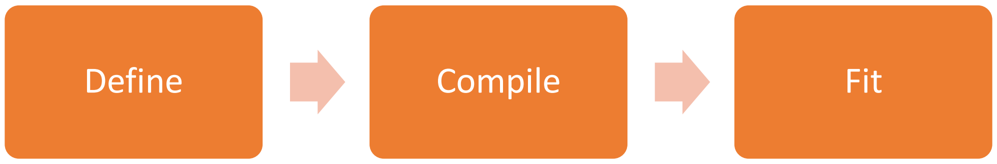
```
> xIris <- map(fullData, ~ bake(object = iris_recipe,
+                               newdata = .x,
+                               all_predictors(),
+                               composition = "matrix"))
> yIris <- map(fullData, ~ bake(object = iris_recipe,
+                               newdata = .x,
+                               all_outcomes(),
+                               composition = "matrix"))
```

●

1. Load the BreastCancer data
2. Split the data so that 80% is used for training and 20% for testing
3. Remove the Id column from both data sets
4. Create dummy variables for the class variable
5. Scale the numeric variables
6. Replace all missing values with 0
7. Convert the target and feature data frames to matrices

2.3 Building a Simple Model

When we build a model in Keras there are a number of elements that we have to specify before we can run the model. The workflow for building a model can be summarised as:



2.3.1 Define the model

A neural network can take any of a number of structures but the simplest, and most common structure, is a linear stack of layers. This simply means that we start from our input, moving through the layers in a linear fashion to reach the output. There is no possibility to branch in multiple directions or take inputs from other sources.

To define this basic, linear structure for our **keras** model we can use the function `keras_model_sequential`, creating a modifiable model object.

```
> model <- keras_model_sequential()
```

As R users the main thing that we need to know (and remember) about this object is that, unlike most other R objects, it is modified in place. This means that we do not have to reassign when we apply other functions, they will simply modify the model object.

● We can also define this same model with the functional API, which is required for structures beyond the linear form. However, both the functional API and non-linear structures are beyond the scope of this course.

2.3.2 Adding Layers

Having defined that we will have a linear structure to our layers we now need to add those layers to define our pipeline of data transformations.

2.3.2.1 Layer Compatibility

Layers must be connected with other compatible layers. Compatibility is defined based on the input and output shape. If one layer returns a tensor of shape 4, the next layer must allow an input of shape 4.

When we talk about shape we do not define the number of observations or samples. Think about a rectangular data set that has 150 rows and 5 columns. We would say that this has a shape of 5. This means that we can always provide more samples without having to re-define our model.

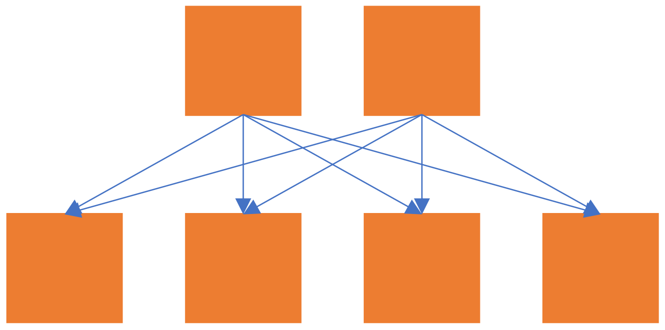
When we define layers with keras we do not always have to specify the input shape as it will determine what it should be based on the output of the previous layer. The one exception to this is the very first layer. When we define our first layer we do need to define the input shape, however this can be easily calculated automatically from the data.

There are a variety of different types of layers that are suited to different types of problem. When working with two dimensional data (like the iris dataset), we generally work with densely connected layers. We will see other types of layer later in this course.

2.3.2.2 Densely Connected Layers

A densely connected layer joins two layers together through a functional transformation that has some corresponding weights that are to be calculated through the model fitting processes.

As a simplified example of what we are trying to achieve, suppose we have a layer that has shape 2 (a dataset with 2 columns) and we want to transform this a layer that has shape 4 (a dataset with 4 columns). We can think of this as each of the existing columns will contribute some amount to each of the new columns.



This is represented in the image above, with each box being a column and these columns being joined by the arrows. When we are fitting the model, we are trying to estimate the amount that each column contributes to the new columns, or the thickness of the arrows.

In reality its not as simple as a direct arrow and we often use a function to make some transformation to the data as we move from one layer to the next.

As an example, let's add two layers to our classification model for the iris data.

```
> model %>%
+   layer_dense(units = 10, input_shape = 4) %>%
+   layer_dense(units = 3, activation = 'softmax')
```

In the first layer we have defined an input shape of 4, as iris has 4 columns. We have defined the units (output shape) to be 10. We will talk more later about how we define the number of units.

2.3.2.3 Output Layers

In the example above our output layer determines its input size from the previous layer. The number of units for the output is, however, determined by the number of outputs that we expect.

For classification of the iris data we expect to be returned a probability that an observation would be in each of the categories. As such we expect to have a shape of three, or one column for each category.

The final element that we have used in this layer is an activation function. This is the specific function that defines the transformation from one layer to the next. The exact function that you should use in the output layer depends on the type of output that is expected.

2.3.2.4 Activation Functions

As you have seen in the output layer, instead of having a simple linear transformation between the layers we can instead make the transformation some function of the inputs. For the output layers there are specific functions that we should use, but for other layers we are free to try a variety of functions.

2.3.3 Compiling Models

Now that we have defined the architecture of our model, or the structure that the model will have, we need to consider the techniques that will be used to estimate the parameters in our model. We define these in the compile step for our model.

```
> model %>% compile(
+   optimizer = 'rmsprop',
+   loss = 'categorical_crossentropy',
+   metrics = 'accuracy'
+ )
```

2.3.3.1 The Loss Funtion

When we start to fit out model we are going to repeatedly try to estimate the best values for the weights, or the parameters that define the transformation from one layer to the next. Ultimately, we want to find the values for the weights that minimise the difference between what the model would predict and the true observed values. We do this using a loss function.

Firstly, a loss function helps us to quantify the difference between the observed and predicted values. More importantly it helps us to understand how we should change the values of the weights to minimise the difference.

The mathematical details of this are beyond the scope of this course and thankfully this is all handled for us by **keras**, we just need to define the function that we want to use. This is simplified further for us as there are simple rules that we need to follow, based on the type of output that we are generating, as to which loss function we should use. The table below defines the most common outputs and the corresponding loss functions that should be used.

Output	Loss Function
Binary Classification	binary_crossentropy
Multi-class Classification (single label)	categorical_crossentropy
Multi-class Classification (multiple labels)	binary_crossentropy
Regression	mse

2.3.3.2 The Optimizer

When it comes to the detail of the mathematical approach that is used to identify how we update the weights, this is defined by the optimizer. There are a number of different approaches that can be taken but in general it is fine to stick to the RMSProp algorithm with its default values.

2.3.3.3 Metrics

The final thing that we can provide as we build our model are metrics to observe as we move through the model fitting process. As with other machine learning approaches, we can observe a variety of different metrics, many are available through **keras**, but we will often simply use the accuracy, as in this example.

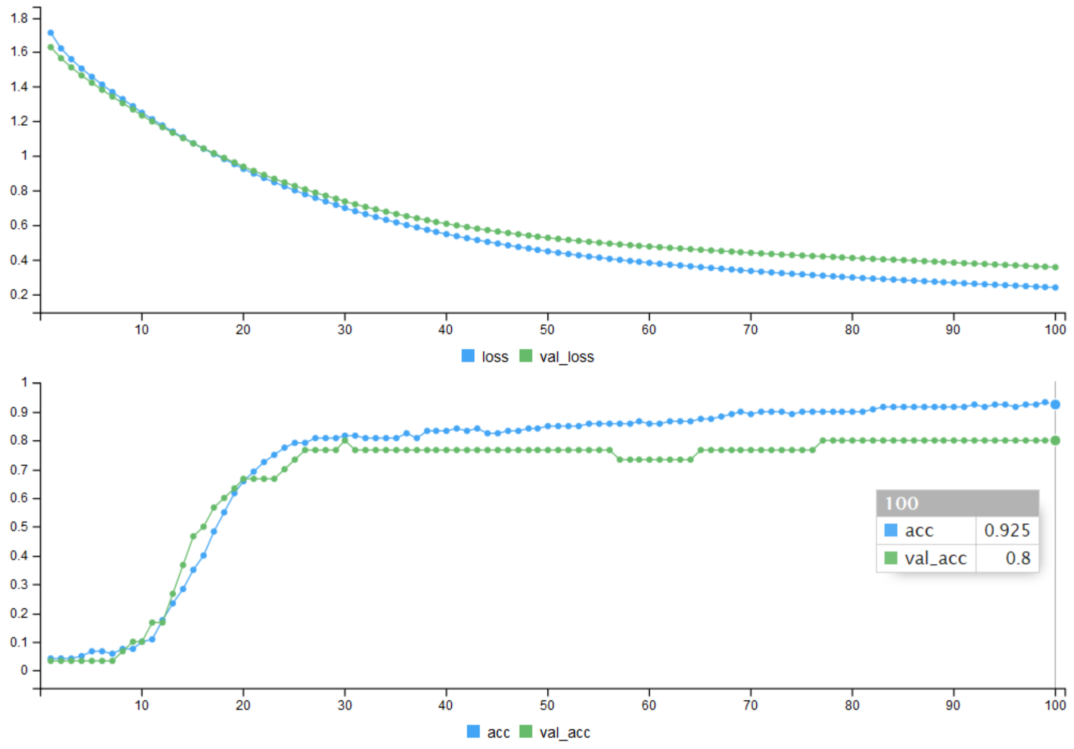
2.3.4 Fitting the Model

Having defined the architecture and the functions to be used for estimating the weights we can now fit the model with the data that we have available, that we prepared earlier.

In its simplest form we just need to provide our training data and the number of epochs (iterations through the data).

```
> history <- model %>% fit(xIris$train,
+                          yIris$train,
+                          epochs = 100,
+                          validation_data = list(xIris$test,
+                                                 yIris$test))
Train on 120 samples, validate on 30 samples
Epoch 1/100
120/120 [==============================] - 1s 4ms/step - loss: 1.6167
- acc: 0.0750 - val_loss: 1.5486 - val_acc: 0.0333
Epoch 2/100
120/120 [==============================] - 0s 1ms/step - loss: 1.5039
- acc: 0.1083 - val_loss: 1.4682 - val_acc: 0.1000
Epoch 3/100
120/120 [==============================] - 1s 5ms/step - loss: 1.4280
- acc: 0.1333 - val_loss: 1.4041 - val_acc: 0.1333
...
```

In this example we have also provided our test set for validation. This means for each epoch we will also obtain the metrics that we specified for our validation data to help us monitor the progress of the fit. In this case, as the iris data is small, we have used the test data that we previously created as the validation set. However, we could instead provide the argument `validation_split` to define a proportion of the data that should be used at each epoch as validation data. During the fitting process an interactive graphic is generated to allow us to monitor performance of both the training and validation data.



In this example we can see that we obtain a reasonable accuracy with our very basic model, the final accuracy of the training data being 92.5%. We can also see that there seems to be some overfitting of the data, with the accuracy of the training data being only 80%. A **ggplot2** version of this graphic can be obtained through the `plot` method for the history object created from running the fit function.

1. Load the pre-cleaned BreastCancer data
2. Create a model with:
    a. A dense layer with 5 hidden units
    b. A dense, output layer using the "sigmoid" activation function
3. Compile the model using "binary_crossentropy" as the loss function
4. Fit the model over 20 epochs

Extension Questions

5. Change the activation function in the first dense layer to "relu", what effect does this have?
6. Increase the number of hidden units in the dense layer, does this have any impact?
7. What effect does adding additional layers to your model have?

2.4 Evaluation and Prediction

Once we have fitted the model we can start to evaluate the performance and predict new observations. The evaluate and predict functions make this very simple.

```
> model %>%
+   evaluate(xIris$test, yIris$test)
30/30 [==============================] - 0s 33us/step
$loss
[1] 0.3548475

$acc
[1] 0.8
>
> model %>%
+   predict(xIris$test)
              [,1]         [,2]         [,3]
 [1,] 0.9936997890 0.0062513999 4.879060e-05
 [2,] 0.9642422795 0.0356328636 1.248152e-04
 [3,] 0.9980422258 0.0019066988 5.107183e-05
 [4,] 0.9977123737 0.0022543857 3.315854e-05
 [5,] 0.9752730131 0.0243754424 3.515764e-04
 [6,] 0.9893329144 0.0105852988 8.188351e-05
...
```

For classification examples we can also use the predict_classes function to return the specific class that an observation is expected to belong to, although note that categories are numbered in the order they were input starting from zero, not named.

```
> model %>%
+   predict_classes(xIris$test)
[1] 0 0 0 0 0 0 0 0 0 0 2 2 1 1 2 1 2 1 1 1 2 1 2 2 2 2 1 2 2 2
```

●

1. Using the model that you built in the last exercise and the pre-cleaned test breast cancer data evaluate the performance of your model
2. Predict the classes for the test data

2.5 Controlling the Model with Layers

Once we have got to grips with the basics of building a neural network with keras we can start to think about how we can build models that generalise well and don't overfit the data. With our networks we control the fit of the model through the layers that we add. As we will see later, there are many other types of layers for working with different types of data, but here we will consider some general principles.

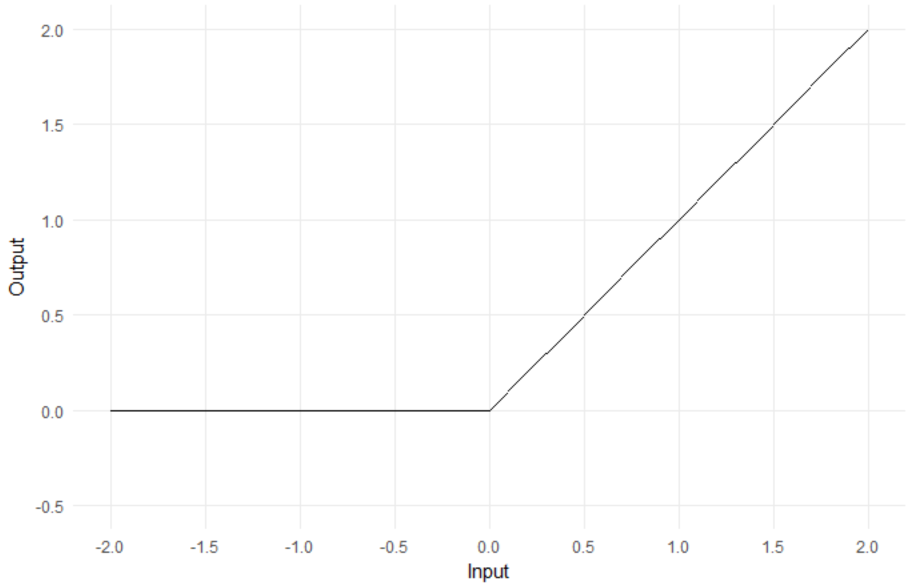
2.5.1 Activation Functions

We saw earlier a very simple example using the iris data where we used a dense layer with no specified activation function. In this case the transformation that was applied was simply a linear transformation from one layer to the next.

```
> model %>%
+   layer_dense(units = 10, input_shape = 4) %>%
+   layer_dense(units = 3, activation = 'softmax')
```

Whilst this made it easier to think about the general process we were following, in practice such a simple transformation doesn't allow us to extract all the features from the data that we can.

One of the most commonly used transformations in neural networks is the relu function, or rectified linear unit function. The graphic below shows what this function looks like.



Essentially the function sets any negative values to zero and all remaining values remain the same. We can think of this as only allowing a feature to become active if its contributions reach a certain threshold. This gives us a much finer level of granularity in the features that can be identified.

Just as with the output layer we specify this function with the activation argument.

```
> model %>%
+   layer_dense(units = 10, input_shape = 4, activation = 'relu') %>%
+   layer_dense(units = 3, activation = 'softmax')
```

Remember that the model object modifies in place, so if we actually want to change our model we also need to re-run the initial model definition to reset the model. Just running the section of code above would in fact add these layers to the existing model.

2.5.2 Output Layers

As we have seen above we can specify a different activation function to map our data onto a particular range of values. When it comes to the output layer this is particularly important to ensure that the output makes sense, you wouldn't want your output to take any numeric value when it should define a probability of belonging to a particular class.

Just like with the loss functions there are specific activation functions that we can use as standard for particular types of problem.

Output	Activation Function
Binary Classification	sigmoid
Multi-class Classification (single label)	softmax
Multi-class Classification (multiple labels)	sigmoid
Regression	-

Note that for regression we do not need to use an activation function, as in this case we do want to allow our data to take any value.

The other element that we have to consider for the output layer is the output shape, or number of units. This is again directly related to the type of problem. In our iris example we have been using 3 units as we want to return a probability for each of the three possible classes. In a binary classification example we would have an output of 2 units and for regression problems this would be simply 1 unit.

As an example, suppose that we were using 5 columns of the Boston House price data to estimate the price of homes in Boston. In this case, we want to return a single estimated house price. Our model might look something like this:

```
> model <- keras_model_sequential()
> model %>%
+   layer_dense(units = 64, input_shape = 5, activation = 'relu') %>%
+   layer_dense(units = 1)
> model %>% compile(optimizer = 'rmsprop',
+                   loss = 'mse',
+                   metrics = 'mae')
```

2.5.3 Hidden Units

One of the main parameters that we control in a neural network is the number of hidden units. Unfortunately, there are no rules to help us to select the number of units that we include or the number of layers that we should add.

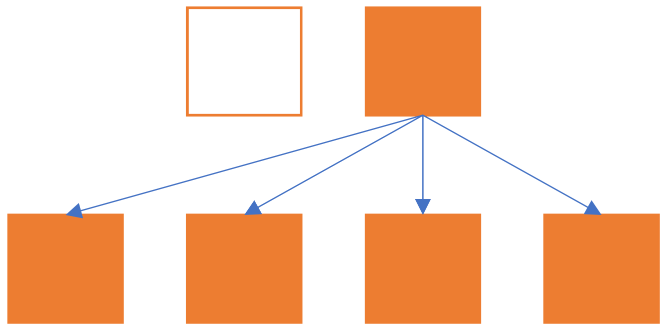
Just as with any model, if we include too many parameters our model will start to overfit to the training data. The model starts to memorise the features that lead to a particular outcome. However, if we don't include enough parameters our model won't be able to identify the features of the data that lead to particular outcomes.

In general, the more data we have the more layers and hidden units we can add, making our network larger. A good strategy for determining the number of layers and units is to start with a smaller network and increase the size, through additional layers and increased hidden units. By working with validation sets you can monitor the performance of the model, through the loss function and defined metrics, and start to establish where there is no further benefit to adding parameters.

2.5.4 Dropout

A technique that we can use to help prevent overfitting is to add random noise to our data through dropout. The idea behind random dropout is to prevent the model from identifying patterns that don't really exist in the data.

We can add dropout to any layer in our network using the `layer_dropout` function. This will randomly dropout a proportion of the points in the layer that precedes it. We can think of this as randomly setting to zero a proportion of the data in a layer so that it doesn't contribute to the next layer.



For our iris example this might look something like this:

```
> model %>%
+   layer_dense(units = 10, input_shape = 4, activation = 'relu') %>%
+   layer_dropout(rate = 0.3) %>%
+   layer_dense(units = 3, activation = 'softmax')
```

1. Using the pre-cleaned Boston House Price data, build a model from scratch to predict the house price deciding:
    a. An initial number of layers
    b. The number of hidden units
    c. The activation function(s) to use
2. Add a dropout layer to your model, does this improve performance on the test data?

Chapter 3
Networks for Spatial Data

3.1 Introduction

The `iris` example in the previous chapter was useful for learning how to prepare our data for Keras, and to build a model. However, it is the kind of problem for which traditional machine learning is better suited. The data is small and highly structured, with a low number of features. An area that is more difficult for traditional machine learning is problems involving spatial data. This includes classifying:

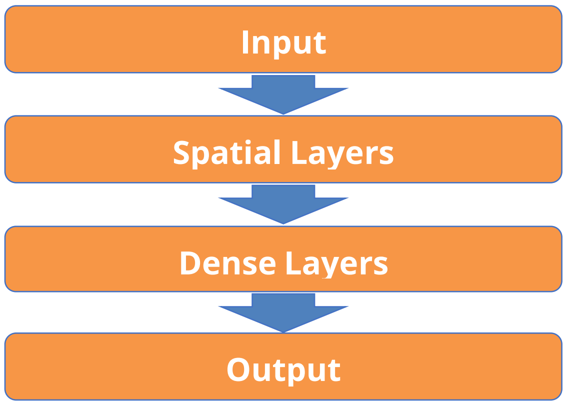
- Images (x and y)
- Time series (time)
- Audio (time)
- Video (x, y, time)

Spatial data is much less structured than tabular data like `iris`. Before neural networks, significant amounts of hand crafting, convolving, and fourier transforming would often go into solving these types of problems. The big breakthrough came with the advent of Convolutional Neural Networks (CNNs or ConvNets).

3.1.1 Convolutional Neural Networks

In 2010 the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was launched with the task of classifiying images from a library of millions of labelled images. Many different methods were used, but once CNNs arrived in 2012 with the AlexNet, they quickly took over. By 2015 the ResNet CNN had dropped the error rate to 3% —matching human level errors.

While there are many variants, the basic structure is the same for all CNNs. Instead of connecting all our data immediately into densely connected layers, as we did in the previous chapter, we instead send it through a series of layers that preserve the spatial nature of the data. These layers learn to see the data through gradually increasing abstraction, before we eventually combine into the output.



3.2 Walking Data

To illustrate how to work with spatial data we will be using the walking data set. This dataset was created from the accelerometer data from the UCI machine learning repository[1]. The full dataset provides wristband accelerometer data for 15 people while they perform several types of activity: walking, sitting, climbing stairs etc.

We have filtered it to only the walking activity and chopped it into 5 second chunks to attempt to answer the question: "Can we recognise someone from their gait in 5 seconds?".

Load the data into your environment using:

```
> walking <- readRDS("Data/walking.rds")
> names(walking)
[1] "x"         "y"         "labels"
```

The data has been split into two parts. The `labels` are an integer label referring to which person each time chunk refers to. The `y` matrix is a one-hot-encoded version of this. The `x` array carries the measurements, which have been scaled within each chunk.

3.2.1 *Measurements Data*

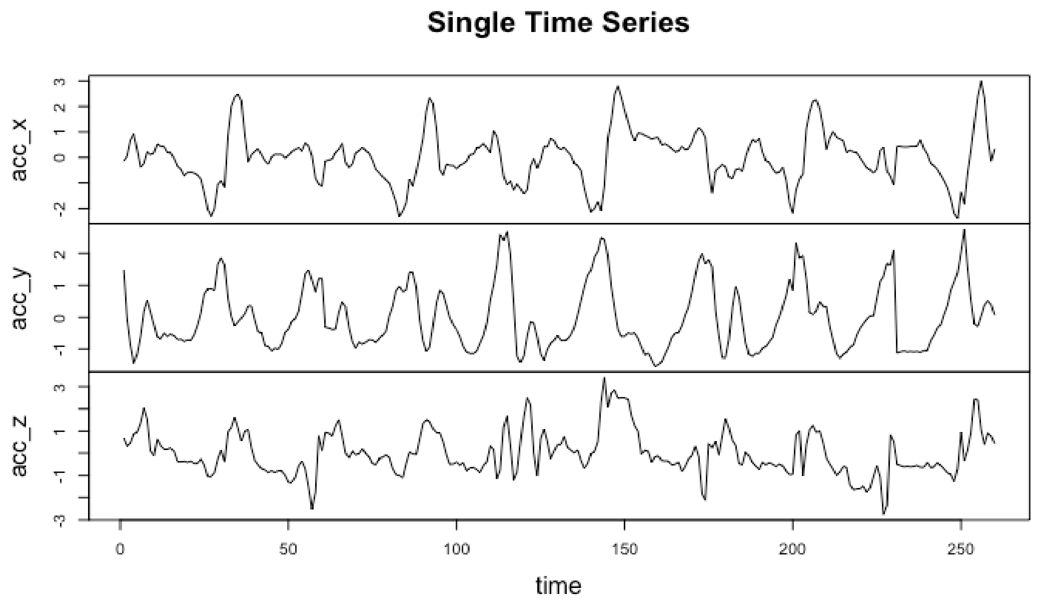
We can see how the `x` array is structured using the `dim` function.

```
> dim(walking$x)
[1] 6792  260    3
```

This output tells us that we have 6792 samples, each with 260 time points, across three channels (x, y, and z). We can plot one to get a feel for how it looks. As it's time series data we can build a `ts` object to quickly get a nice plot.

```
> walkTs <- ts(walking$x[50,,])
> plot(walkTs, xlab = "time", main = "Single Time Series")
```

[1] https://archive.ics.uci.edu/ml/datasets/Activity+Recognition+from+Single+Chest-Mounted+Accelerometer



This is a single chunk for person number, 1. Without deep learning we might compute various features based on these time series. However, we will let the network learn the right features.

3.3 Preparing Spatial Data

Spatial data brings a few new challenges in preparing our data for machine learning. We now have three types of dimension to consider:

1. **Samples**: These are our observations. For `iris` this is the rows, for `walking` it is the first dimension. In Keras it is always *has* to be the leading dimension.
2. **Space**: The next dimensions are spatial. Spatial dimensions can be anything for which the ordering matters. For an image this would be our x and y dimensions. Time is equivalent to space in this context. Although for sequence problems time can sometimes be thought of as the samples dimension (albeit still ordered).
3. **Channels**: These dimensions do not have specific ordering. For `iris` the column order had no baring on the problem. For images this would be our red, green, blue (RGB) colour channels. For `walking` these are our `acc_x, acc_y, acc_z` channels. We can swap the order of these without any change to the problem. With Keras channels are put at the end.

● For higher dimensional arrays in R we would expect the slow dimension (samples) to be at the end. However, this is one of the consequences of the R to Python interface.

In this problem each time series has exactly 260 points. If your data has varying lengths then you will need a strategy to reshape, or pad the data, to fit into the array.

3.3.1 Labels Data

The labels are supplied as an integer vector with values 1 to 15.

```
> walking$labels
   [1] 1 1 1 1 1 1 1 1 ...
```

However, we have already one-hot-encoded this and added it to a y variable, which is in the right shape for training a model.

```
> walking$y[1:3, 1:10]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    0    0    0    0    0    0    0    0     0
[2,]    1    0    0    0    0    0    0    0    0     0
[3,]    1    0    0    0    0    0    0    0    0     0
```

●

1. Load the walking data.
2. Create two lists, `xWalk` and `yWalk`, each with an 80:20 split of train and test sets for `x` and `y` data respectively.

   (hint) `nr <- nrow(walking$y)`
   `ids <- sample(nr, size = nr*0.8)`

3.4 Layers for Spatial Data

A Convolutional Neural Network (CNN) refers to a network architecture that introduces a few new types of layers that are specialised for spatial data. The term CNN refers to the combination of these layers, rather than just one layer type.

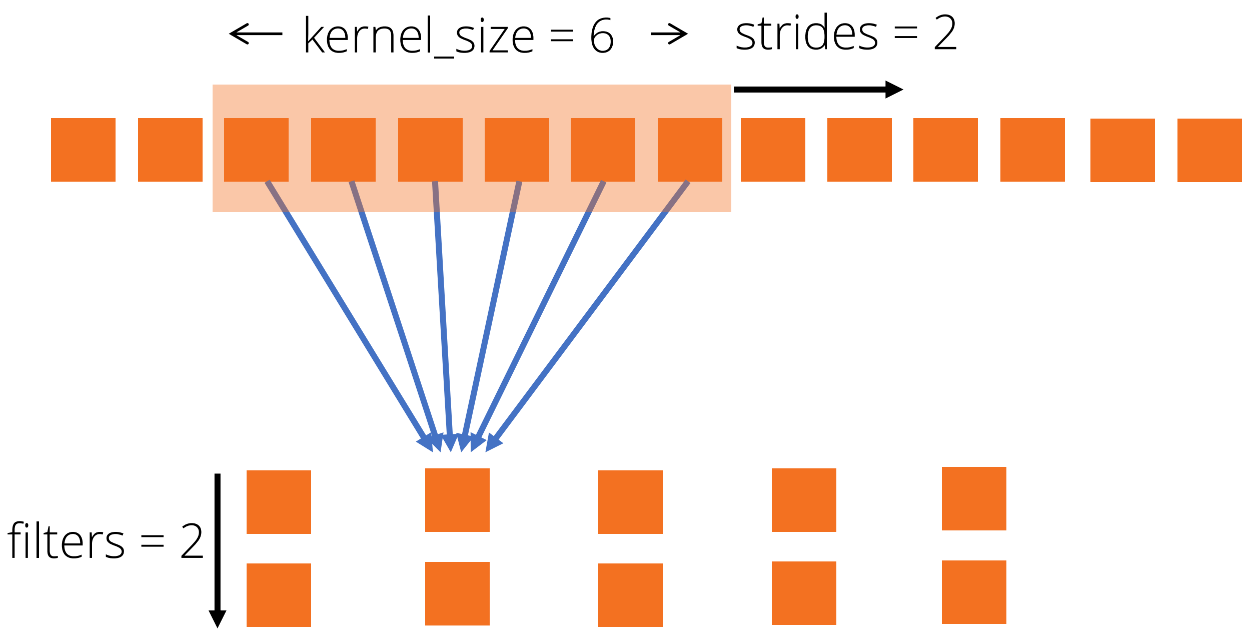
Let's start by creating a new model to hold our CNN.

```
> model <- keras_model_sequential()
```

3.4.1 Convolution Layer

For the `iris` data set the ordering of the columns did not matter. This is because, in neural network terms, these are *channel* dimensions. It makes sense to connect them up equally and as such a dense layer is ideal. For spatial data, a dense layer will remove all sense of space in one go. Instead we would like to build space-aware features. This is where a convolution layer comes in.

A convolution layer reduces the connections between layers so that only nearby nodes are connected and the sense of position in space is maintained. In keras there are a family of `layer_conv_` functions, and because our time series data is 1d we will use the 1d version.



There are three important components:

3.4.1.1 Kernel Size

The kernel is a pattern of weights, or a *mask*, that we apply to a small window of nodes and connect to a single node in the layer below. In the diagram above this is a line of 6 points. In Keras we set this with the `kernel_size = 6` argument.

Deciding what kernel size to use depends on the problem at hand. For image classification it tends to be set quite small, 2 or 3, and then many convolution layers are used to gradually reduce the spatial dimensions.

3.4.1.2 Strides

The next step is to begin sliding our kernel across the points (*convolving* the kernel). To reduce the number of nodes in our network (and reduce the risk of overfitting) we would like to start reducing the size of our layers. This is done by moving the kernel across in jumps. In the diagram we reduce the number of nodes in the output layer by setting the argument `strides = 2`.

3.4.1.3 Filters

The final argument is the `filters` argument. This tells Keras how many passes across our input layer we would like to do. The idea is that each filter will encode a different spatial feature— a different mask. So perhaps one layer is good at spotting sharp peaks, while another responds to troughs. In the diagram above we have set `filters = 2`, although typically we would want more than this.

Filters have a similar function to channels after going through a convolution layer, in that their ordering does not matter. Any channels in the input layer are brought together and their affect is distributed among the new filter layers.

Because this is the first layer we will set the input shape based on our input array of 260 time points and 3 channels.

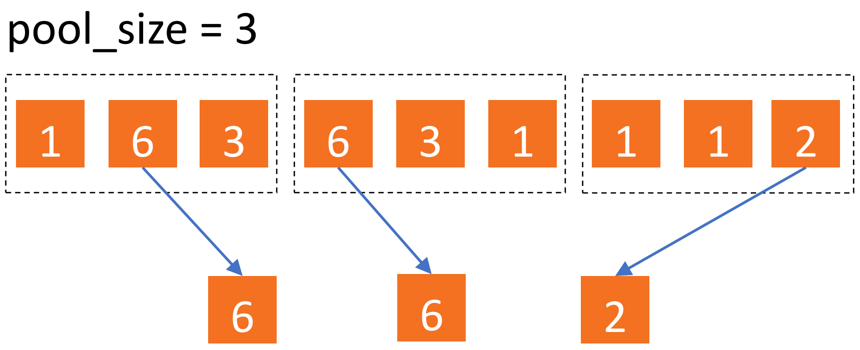
```
> model %>%
+   layer_conv_1d(filters = 40, kernel_size = 40, strides = 2,
+                 activation = "relu", input_shape = c(260, 3))
> model
Model
________________________________________________________________
Layer (type)                 Output Shape              Param #
================================================================
conv1d_10 (Conv1D)           (None, 111, 40)           4840
================================================================
Total params: 4,840
Trainable params: 4,840
Non-trainable params: 0

________________________________________________________________
```

Note the the spatial dimension is less, due to the striding, and the number of channels (now called filters) has increased due to the filters argument. As a rule, always use `activation="relu"` for convolution layers.

The number of parameters is roughly (`kernel_size * input_channels) * filters`. This is a very small number compared to a fully connected layer.

3.4.2 Max Pooling Layer

Convolution layers have a tendancy to create a lot of new nodes, thanks to the `filters` argument. A max pooling layer is a crude-but-effective way to reduce the number of nodes. The nodes on the input are put together in groups of size `pool_size` and only the maximum value in the group is passed down to the next layer. So if we have a `pool_size` of 3 we would expect to have a third of the number of neurons on the output.



For our model we'll start with `pool_size = 2`.

```
> model %>%
+   layer_max_pooling_1d(pool_size = 2)
> model
Model
________________________________________________________________
Layer (type)                 Output Shape              Param #
================================================================
conv1d_10 (Conv1D)           (None, 111, 40)           4840
________________________________________________________________
max_pooling1d_15 (MaxPoolin (None, 55, 40)             0
```

Note that we've reduced the size of the spatial dimension, not the filters. Other types of pooling, such as average pooling, are also available.

3.5 Connecting to the Output

By the end of the spatial layers, we have many filters that hold an abstract view of our input data. To make a prediction we need to eventually lose the spatial aspect of the data. The end of a CNN usually has a more standard feed forward network that combines all of the filters into an answer.

3.5.1 Flatten Layer

The last new layer marks the end of our convolutional component. This is the point in which we abandon the spatial dimensions to prepare for our final output. A flatten layer simply reduces the dimensionality.

```
> # Flatten
> model %>%
+   layer_flatten()
> model
Model
________________________________________________________________
Layer (type)                 Output Shape               Param #
================================================================
conv1d_10 (Conv1D)           (None, 111, 40)            4840
________________________________________________________________
max_pooling1d_15 (MaxPoolin (None, 55, 40)              0
________________________________________________________________
flatten_8 (Flatten)          (None, 2200)               0
================================================================
```

3.5.2 Dense Layers

The final stage is exactly the same as for the `iris` problem. We will bring our features together in one or more dense layers and connect that to the output with a softmax dense layer.

```
> model %>%
+   layer_dense(units = 100, activation = "sigmoid") %>%
+   layer_dense(units = 15, activation = "softmax")
```

The first dense layer has dramatically increased the number of parameters.

```
> model
Model
________________________________________________________________
Layer (type)                 Output Shape              Param #
================================================================
conv1d_10 (Conv1D)           (None, 111, 40)           4840
________________________________________________________________
max_pooling1d_15 (MaxPoolin (None, 55, 40)             0
________________________________________________________________
flatten_8 (Flatten)          (None, 2200)              0
________________________________________________________________
dense_15 (Dense)             (None, 100)               220100
________________________________________________________________
dense_16 (Dense)             (None, 15)                1515
================================================================
Total params: 226,455
Trainable params: 226,455
Non-trainable params: 0

________________________________________________________________
```

1. Reproduce the above model and compile it.
2. Train the model with `fit` and assess performance on the validation set over 15 epochs.
3. How does this compare to only using dense layers (you'll still need to flatten)?

3.6 CNN Architectures

Having defined the basic building blocks of a CNN we can combine them together in a number of different ways. There is no one-size-fits all architecture, and finding a good pattern is a matter of trial and error as much as experience. A good starting point is to look at patterns that others are using.

The general strategy is to take an input that has its information spread out across a large space and transform it so that it is instead captured in many filter layers in a much smaller space. A common strategy is to switch between convolution layers, and max-pooling layers, and finish off with a dense "top". For example, the VGG 2014 ImageNet winner[2] was a sequential model with convolution layers mixed with occasional max-pooling layers and finished with dense layers.

3.6.1 Final Walking Model

The final model we will try for the walking problem will be a similar mix of convolutions and max-pooling. This time we're using a kernel size of 30 for the first layer, and maintaining 40 filters all the way down.

```
> model <- keras_model_sequential()
>
> model %>%
+   layer_conv_1d(filters = 40, kernel_size = 30, strides = 2,
+                 activation = "relu", input_shape = c(260, 3)) %>%
+   layer_max_pooling_1d(pool_size = 2) %>%
+   layer_conv_1d(filters = 40, kernel_size = 10, activation =
"relu") %>%
+   layer_max_pooling_1d(pool_size = 2) %>%
+   layer_flatten() %>%
+   layer_dense(units = 100, activation = "sigmoid") %>%
+   layer_dense(units = 15, activation = "softmax")
```

As usual we can take a look at the model:

[2] http://www.robots.ox.ac.uk/~vgg/research/very_deep/

```
> model
Model
______________________________________________________________________
Layer (type)                   Output Shape                  Param #
======================================================================
conv1d_43 (Conv1D)             (None, 116, 40)               3640
______________________________________________________________________
max_pooling1d_31 (MaxPooling1D (None, 58, 40)                0
______________________________________________________________________
conv1d_44 (Conv1D)             (None, 49, 40)                16040
______________________________________________________________________
max_pooling1d_32 (MaxPooling1D (None, 24, 40)                0
______________________________________________________________________
flatten_14 (Flatten)           (None, 960)                   0
______________________________________________________________________
dense_31 (Dense)               (None, 100)                   96100
______________________________________________________________________
dense_32 (Dense)               (None, 15)                    1515
======================================================================
Total params: 117,295
Trainable params: 117,295
Non-trainable params: 0
______________________________________________________________________
```

Compiling and training gives us the final performance:

```
> history <- model %>% fit(xWalk$train, yWalk$train,
+                           epochs = 15,
+                           batch_size = 128,
+                           validation_split = 0.3,
+                           verbose = 1)
Train on 3803 samples, validate on 1630 …
.. val_acc: 0.9540
```

We have managed 95% accuracy on the validation set.

3.6.2 Predicting on New Data

How about the test set that we put to one side? We can use this as an independent measure of performance, and a way to show how we would produce scores on new data. First, we'll use the `evaluate` function to test performance:

```
> model %>% evaluate(xWalk$test, yWalk$test, verbose = 0)
$loss
[1] 0.1877462

$acc
[1] 0.9470199
```

To get the actual scores we can use the `predict_classes` function. This function will unroll the one-hot-encoding and predict a class label. Note that the predictions are in Python 0-index format so we'll need to add 1.

```
> yPred <- model %>% predict_classes(xWalk$test) + 1
> yPred
   [1] 13  2  1  2 14 12  1 12 13  7 ...
```

Once we have predicted scores and the known, real scores, we can use other R packages that evaluate performance. `caret` has the `confusionMatrix` function for just this purpose. Note that we're using the `max.col` function to undo the one-hot-encoding here.

```
> caret::confusionMatrix(yPred, max.col(yWalk$test))
Confusion Matrix and Statistics

          Reference
Prediction   1   2   3   4   5   6   7   8   9  10  11  12
         1  86   1   0   2   0   0   0   0   0   0   0   0
         2   0  66   0   0   0   0   0   1   0   1   0   0
         3   1   1  53   0   1   0   2   0   0   2   0   0
         4   0   0   0 121   0   2   1   0   0   0   0   0
         5   0   0   0   0  71   0   0   0   0   0   0   0
...
```

1. How do further epochs affect performance?
2. Try changing the kernel size and number of filters. How does this affect your results?
3. Try adding more dense layers. How does this affect training time and model performance?
4. Try adding a dropout layer.