# Class 2: From simple to deep neural networks

Andrew Parnell
andrew.parnell@mu.ie


PRESS RECORD

https://andrewcparnell.github.io/intermediate_ML

# Learning outcomes

- Understand what a keras model is doing in the background
- Fit some more advanced type deep learning models
- Start learning about transformers

# Reminder: structure of a keras model

```
library(keras)
model <- keras_model_sequential() %>%
  layer_dense(units = 5, activation = 'relu',
              input_shape = 4) %>%
  layer_dense(units = 3, activation = 'softmax')
model %>% compile(
  optimizer = optimizer_adam(),
  loss = "categorical_crossentropy",
  metrics = c("mean_absolute_error")
)
history <- model %>% fit(
  x_train, y_train,
  epochs = 50, # Number of passes through the data set
  batch_size = 16, # Samples used in each gradient update
  validation_data = list(x_test, y_test)
)
```

# What do all these functions and arguments mean?

- `keras` (meaning 'horn') is the package we use: it provides an interface to Python and calls other (lower level) neural network packages
- `keras_model_sequential` sets up a neural network pipeline with successive (and often quite elaborate) layers
- There then follows a list of all the layers (more on this later)
- `compile` sets up how the model will work (loss function, optimisation routine, etc)
- We then `fit` the model and tell is how many epochs, and how much data is used in each pass

# What if we were to run this without the `keras` package?

Suppose we had one hidden layer. This means we have two sets of weights and biases. First from the inputs to the hidden layer ($W_1$ and $b_1$), then from the hidden layer to the outputs ($W_2$ and $b_2$).

If we run this on the `iris` data set (classifying species) we have 4 inputs and 3 outputs. So the weights to the hidden layer $W_1$ will have dimension $4 \times 5$, with bias $b_1$ having dimension $5 \times 1$. The weights from the hidden layer to the outputs $W_2$ will have dimension $5 \times 3$ with bias $b_2$ having dimension $3 \times 1$.

# Remember the steps:

1. Data Prep: training/test split, feature extraction
2. Model Initialization: guess initial parameter values
3. Forward Propagation: pass input data through the network to get a prediction
4. Loss Computation: work out how badly you did
5. Backpropagation: calculate how you need to change the weights to reduce the loss
6. Weight Update: use an optimisation algorithm to estimate new weight values
7. Repeat!

# 1 Data Prep

Let's assume we have the data all set up into `train_data` and `test_data` containing the features, and `train_labels` and `test_labels` converted into one-hot encoded matrices

```
head(train_labels_onehot)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    1    0    0
## [3,]    0    0    1
## [4,]    1    0    0
## [5,]    0    0    1
## [6,]    0    0    1
```

# 2a Model initialisation

Have some values that are set by the model, and some that we choose:

```
input_size <- ncol(train_data)
output_size <- ncol(train_labels_onehot)
hidden_size <- 5 # These ones are set by us
learning_rate <- 0.001
epochs <- 200
```

# 2b Model initialisation

Set up the weights in the correct dimensions

```
W1 <- matrix(rnorm(input_size * hidden_size),
             nrow = input_size)
b1 <- matrix(0, nrow = hidden_size)
W2 <- matrix(rnorm(hidden_size * output_size),
             nrow = hidden_size)
b2 <- matrix(0,
             nrow = output_size)
str(W1)
```

```
##  num [1:4, 1:5] -1.22072 0.1813 -0.13889 0.00576 0.38528 ...
```

# 2c Some functions we need

```
relu <- function(x) {
  return(matrix(pmax(0, x), nrow=nrow(x), ncol=ncol(x)))
}
softmax <- function(x) exp(x) / rowSums(exp(x))
```

# 3 Forward propagation

The forward pass function takes the inputs (x) and the weights and outputs the value of the two nodes (the hidden and the output):

```
forward <- function(x, W1, b1, W2, b2) {
  z1 <- x %*% W1 + matrix(rep(t(b1), nrow(x)),
                          nrow(x), length(b1), byrow=TRUE)
  a1 <- relu(z1)
  z2 <- a1 %*% W2 + matrix(rep(t(b2), nrow(a1)),
                           nrow(a1), length(b2), byrow=TRUE)
  a2 <- softmax(z2)
  return(list(a1=a1, a2=a2))
}
activations <- forward(train_data, W1, b1, W2, b2)
```

# 4 Loss computation

Let's work out how badly our initial guesses did

```
# Categorical cross entropy loss
compute_loss <- function(y_pred, y_true) {
  -sum(y_true * log(y_pred)) / nrow(y_true)
}
# Get predictions
y_pred <- activations$a2
loss <- compute_loss(y_pred, train_labels_onehot)
print(loss)
```

```
## [1] 6.351131
```

# 5 Backpropagation

First work out how to change the weights, working from the end node back to the beginning:

```
dz2 <- y_pred - train_labels_onehot
dW2 <- t(activations$a1) %*% dz2
db2 <- colSums(dz2)
dz1 <- dz2 %*% t(W2) * (activations$a1 > 0)
dW1 <- t(train_data) %*% dz1
db1 <- colSums(dz1)
```

These all come from differentiating the loss function with respect to each of the weights and biases.

# 6 Weight updates

Now update the weights according to the learning rate

```
W1 <- W1 - learning_rate * dW1
b1 <- b1 - learning_rate * db1
W2 <- W2 - learning_rate * dW2
b2 <- b2 - learning_rate * db2
print(W1)
```

```
##                      [,1]       [,2]        [,3]       [,4]       [,5]
## Sepal.Length -1.220717712  0.4173289 -0.37132562 1.2253780 -0.8975128
## Sepal.Width   0.181303480 -0.3501013  0.69326001 1.0240182  1.2174034
## Petal.Length -0.138891362  0.6564600  0.07636455 0.6140075 -0.7684220
## Petal.Width   0.005764186 -0.2177527 -0.42419779 0.2602737  2.1360912
```

```
print(b1)
```

```
##              [,1]
## [1,]  0.000000000
## [2,]  0.006242131
## [3,] -0.129513031
## [4,]  0.012000035
## [5,] -0.047282668
```

# 7 Repeat

- In reality we would wrap this in a loop of epochs that would constantly update the weights and improve the fit of the model
- See the `iris_nopackages.R` example for the full script, or the `iris_keras.R` version for the proper `keras` model run
- See the `mtcars_nopackages.R` and `mtcars_keras.R` scripts for equivalent versions using regression rather than classification
- Remember there are lots more fancy bits in the real code but this gives you the main idea

# Reminder of the tunable hyper-parameters

- **Learning rate**: the degree to which each parameter is moved at the end of each iteration. Making this too big might cause the model to overshoot, whereas making it too small will lead to slow optimisation
- **Epochs**: the number of passes through the data. If too small will not reach the optimum. If too big you might over-fit or waste computation time
- **Batch size**: the number of training examples used in iteration. Using all of the training data in each iteration will slow calculations down

There are lots more but these tend to be the main ones we need to set

# Activation functions

- Many of the hidden layers we will use involve *activation functions* which are applied to the output of that neuron
- There are lots of different types, but in general to try and stabilise the values to a range, be differentiable (for the back-propagation step) and be simple to compute
- The activation function for the final layer in the network needs to match the format of the output (e.g. softmax for probabilistic classification)
- You can find a list of activation functions at https://en.wikipedia.org/wiki/Activation_function

# From simple NN to Deep NN

- Moving from a 'simple' NN to a 'deep' NN is just a matter of adding more hidden layers to the model
- The inputs and output sizes are fixed, but the number, type and size of the hidden layers are all within your control
- Some types of layers are known to work well with certain types of data (e.g. convolutional layers with image data)
- But a lot of the modelling here is a mixture of art and tuning (more on this later)

# Changing the layers

It's very easy to add more layers using keras. Here's a much more complicated one

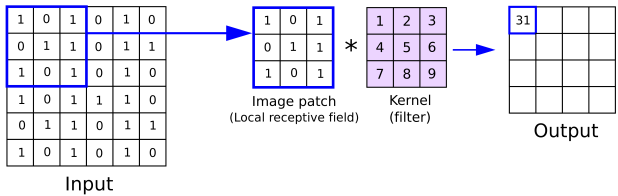
```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = 'relu',
                input_shape = c(32, 32, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filters = 64,
                kernel_size = c(3,3),
                activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')
```

# Common types of layers

- **Dense (or fully connected)**: These are what we have been using already. Every node/neuron has a weight associated with it and a bias. An *activation function* is used to transform the values after they have been computed
- **Flatten**: these simply take a matrix of weights and flatten it into a vector. They're often used after complicated layers that might have multiple dimensions (see the `conv_2d` ones in the previous slide)
- **Dropout**: these drop a random fraction of the inputs with the idea of preventing over-fitting
- **Convolutional**: these are commonly used in image processing to search for common structures in an image. (More on this later)
- **Pooling**: these reduce the dimension of previous layers to take a summary statistic across a dimension, e.g. taking the max (max pooling) or average (average pooling) across the rows of the previous inputs.

# Focus on convolutional layers

- For image data it is very common to have convolutional layers inside the model:




From https://anhreynolds.com/blogs/cnn.html

# Hyper-parameters for convolutional layers

- We often have multiple different shapes we want to match our image to. `keras` calls these `filters`
- The multiple filters produce 3D matrices (tensors!) of weights, which is why sometimes afterwards they are flattened or pooled
- Each filter has a `kernel_size` which determines the size of the filter
- The `stride` specifies if there are jumps or areas where the filter overlaps or not
- These layers also have an activation function

# A bigger example of image analysis

- If time, we will go through the `cifar_keras_cnn.R` file which runs a much bigger image analysis example

# Summary

- We have gone through how to fit one of these models by hand, but we could have done it very simply with `keras` ...
- ... but it's useful to know the basics of what's happening in the background
- The key challenge in creating these models is choosing the right structure for the network. We will talk more about this later
- Once we have got the hang of creating these models we can fit a wide variety of models that can work really well