# Assignment 1

*Priyanka Singhvi and Fleur Petit*

*15 May 2019*

## Exercise 1: Identifying handwritten numbers

### Question 1: Can you think of another application where automatic recognition of hand-written numbers would be useful?

An application where automatic recognition of hand-written numbers would be useful is in sorting mail/posts at central or sub-central postal facilities which allow faster sorting of packages and posts. Another application is for audits where the documents to be audited are mostly handwritten. It would be easier to use a digit recognizer to convert them to computable digits without much human effort.

### Download mnist

```
mnist <- dataset_mnist()
```

### Data preparation

New variables called x_train and x_test, y_train and y_test to avoid overwriting the original images.

```
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test  <- mnist$test$x
y_test  <- mnist$test$y
```

Flattening training and test data to remove spatial relationships.

```
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <-  array_reshape(x_test, c(nrow(x_test), 784))
dim(x_train)
```

```
## [1] 60000   784
```

```
dim(x_test)
```

```
## [1] 10000   784
```

Rescaling the above from grayscale to floating point between 0 and 1.

```
x_train <- x_train /255
x_test  <- x_test/255
```

One-hot encoding the vectors in y into binary classes

```
y_train <- to_categorical(y_train, 10)
y_test  <- to_categorical(y_test, 10)

# Check it out
y_train %>% as_tibble %>% head(5) %>% kable
```

| V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | V10 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

| V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | V10 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

## Model definition

Defining, compiling and fitting the model.

```
model <- keras_model_sequential()

model %>%
  layer_dense(units = 256, input_shape = c(784)) %>%
  layer_dense(units = 10, activation = 'softmax')

summary(model)
```

```
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense (Dense)                (None, 256)               200960
## _________________________________________________________________________
## dense_1 (Dense)              (None, 10)                2570
## =========================================================================
## Total params: 203,530
## Trainable params: 203,530
## Non-trainable params: 0
## _________________________________________________________________________
```

```
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
```

```
history <- model %>% fit(
  x_train, y_train,
  batch_size =128,
  epochs = 12,
  verbose = 1,
  validation_split = 0.2
)
```

## Question 2: In the output text in your console, how long did each epoch take to run?

There are 12 epochs as defined in our model and on average each epoch takes about 5s to run.

```
## Train on 48000 samples, validate on 12000 samples
## Epoch 1/12
## 48000/48000 [==============================] - 5s 104us/sample - loss: 0.4022 - acc: 0.8852 - val_loss: 0.2989 - val_acc: 0.9146
## Epoch 2/12
## 48000/48000 [==============================] - 4s 78us/sample - loss: 0.3074 - acc: 0.9133 - val_loss: 0.2970 - val_acc: 0.9201
## Epoch 3/12
## 48000/48000 [==============================] - 4s 79us/sample - loss: 0.2939 - acc: 0.9180 - val_loss: 0.2922 - val_acc: 0.9197
## Epoch 4/12
## 48000/48000 [==============================] - 4s 93us/sample - loss: 0.2879 - acc: 0.9189 - val_loss: 0.2660 - val_acc: 0.9271
```

```
## Epoch 5/12
## 48000/48000 [==============================] - 4s 81us/sample - loss: 0.2801 - acc: 0.9222 - val_loss: 0.2795 - val_acc: 0.9218
## Epoch 6/12
## 48000/48000 [==============================] - 4s 83us/sample - loss: 0.2771 - acc: 0.9233 - val_loss: 0.2706 - val_acc: 0.9278
## Epoch 7/12
## 48000/48000 [==============================] - 4s 80us/sample - loss: 0.2735 - acc: 0.9243 - val_loss: 0.2924 - val_acc: 0.9178
## Epoch 8/12
## 48000/48000 [==============================] - 4s 80us/sample - loss: 0.2700 - acc: 0.9247 - val_loss: 0.2798 - val_acc: 0.9238
## Epoch 9/12
## 48000/48000 [==============================] - 4s 85us/sample - loss: 0.2677 - acc: 0.9255 - val_loss: 0.2790 - val_acc: 0.9220
## Epoch 10/12
## 48000/48000 [==============================] - 5s 111us/sample - loss: 0.2664 - acc: 0.9253 - val_loss: 0.2762 - val_acc: 0.9237
## Epoch 11/12
## 48000/48000 [==============================] - 4s 85us/sample - loss: 0.2651 - acc: 0.9262 - val_loss: 0.2831 - val_acc: 0.9222
## Epoch 12/12
## 48000/48000 [==============================] - 4s 85us/sample - loss: 0.2640 - acc: 0.9265 - val_loss: 0.2807 - val_acc: 0.9240
```

## Question 3: Plot the training history and add it to your answers

See "model linear", Figure 1.

| epoch | data | acc | loss |
|---|---|---|---|
| 1 | training | 0.8852500 | 0.4021841 |
| 1 | validation | 0.9145833 | 0.2988635 |
| 2 | training | 0.9133334 | 0.3073706 |
| 2 | validation | 0.9200833 | 0.2970077 |
| 3 | training | 0.9179792 | 0.2938989 |
| 3 | validation | 0.9197500 | 0.2922458 |
| 4 | training | 0.9189375 | 0.2879153 |
| 4 | validation | 0.9270833 | 0.2659858 |
| 5 | training | 0.9221875 | 0.2801069 |
| 5 | validation | 0.9218333 | 0.2795259 |
| 6 | training | 0.9232708 | 0.2771453 |
| 6 | validation | 0.9278333 | 0.2706185 |
| 7 | training | 0.9243125 | 0.2734818 |
| 7 | validation | 0.9178333 | 0.2923905 |
| 8 | training | 0.9246666 | 0.2699873 |
| 8 | validation | 0.9238333 | 0.2797627 |
| 9 | training | 0.9255208 | 0.2677428 |
| 9 | validation | 0.9220000 | 0.2790304 |
| 10 | training | 0.9253125 | 0.2664042 |
| 10 | validation | 0.9237500 | 0.2761676 |
| 11 | training | 0.9261875 | 0.2650807 |
| 11 | validation | 0.9222500 | 0.2830916 |
| 12 | training | 0.9265417 | 0.2639824 |
| 12 | validation | 0.9240000 | 0.2806991 |

## Question 4: Describe how the accuracy on the training and validation sets progress differently across epochs, and what this tells us about the generalisation of the model.

In Figure 1 we can see that model starts off with a better accuracy on the validation set than the training set. But the training set gradually catches up to the validation set by the $4^{th}$ epoch and shows a gentle increase. The similarity of the validation and training data indicates that the model will perform similar on out of training data as it performs on the training data. This means that the model generalises well.

## Question 5: What values do you get for the model's accuracy and loss?

| loss | acc |
|---|---|
| 0.2844012 | 0.9232 |

### Question 6: Discuss whether this accuracy is sufficient for some uses of automatic hand-written digit classification.

The accuracy is not high enough for sensitive applications which require high performance and accuracy. It can be safely said though, that it would not be a bad choice to use the digit recogniser for in reducing less risk-sensitive manual work such as sorting postal codes. Because in postal codes there are various levels at which this recognizer can be checked such as the postal code, street number, house number and tied to a validator system which will raise an alarm if the letter is wrongly sorted. Approximately 1 mistake in every 10 digits might not be an issue in this scenario.

## Changing the model parameters

### Question 7: How does linear activation of units limit the possible computations this model can perform?

With a linear activation function it is not possible to perform non-linear mappings from inputs to outputs. If a non-linear decision boundary is required for classification, this can not be achieved with linear activation units. If we use linear activations here, we do not get enough information to classify the outputs correctly. All we get is a weighted average at the end [another linear function], so we may end up losing specific information.

```
modelRelu <- keras_model_sequential()
modelRelu %>%
  layer_dense(units = 256, activation ='relu', input_shape = c(784)) %>%
  layer_dense(units = 10, activation = 'softmax')

summary(model)
```

```
## ________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ========================================================================
## dense (Dense)                    (None, 256)                   200960
## ________________________________________________________________________
## dense_1 (Dense)                  (None, 10)                    2570
## ========================================================================
## Total params: 203,530
## Trainable params: 203,530
## Non-trainable params: 0
## ________________________________________________________________________
```

```
modelRelu %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)

historyRelu <- modelRelu %>% fit(
  x_train, y_train,
  batch_size =128,
  epochs = 12,
  verbose = 1,
```

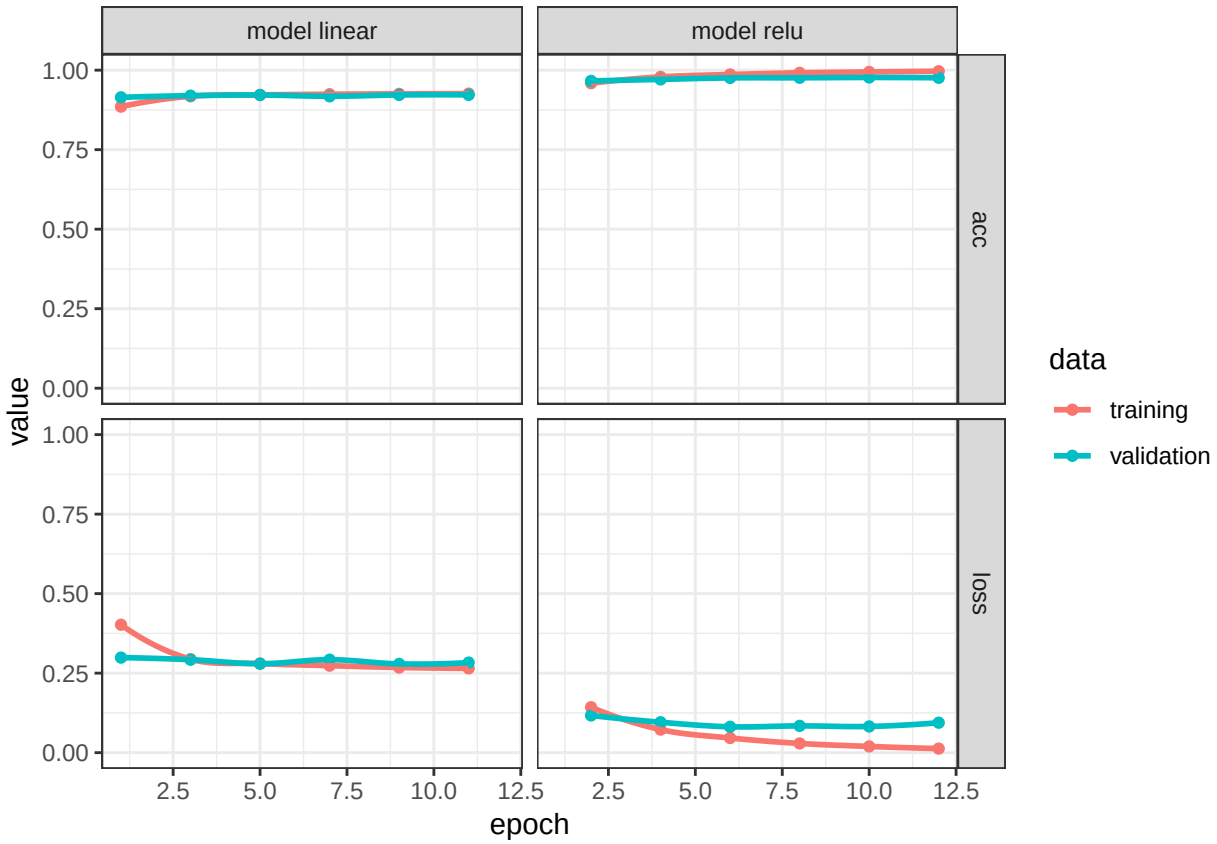


Figure 1: Comparison of the model with a linear activation for the first layer and the model with relu activation.

```
  validation_split = 0.2
)
```

## Question 8: Plot the training history and add it to your answers

See "model relu", Figure 1.

## Question 9: How does the training history differ from the previous model, for the training and validation sets? What does this tell us about the generalisation of the model?

In Figure 1 we can see that the loss of the model with the linear activation function was consistently higher than the loss of the model with the relu activation. The loss on the training and the validation sets converge after approxiametly the $3^{rd}$ epoch in the model with linear activiation. The loss of the model with relu activation diverges after the $3^{rd}$ epoch. The accuracy of the model with relu activation is a bit higher overall than that of the model with linear activation. Yet the difference is small.

We can expect the model with relu activation to perform better on the test set than the model with linear activation, with a lower loss, and perhaps a slightly higher accuracy. The difference between the test and the training loss may be a bit higher than in the previous model, as can be expected from the higher difference between validation and training loss. Because the training and the validation performance are still very close for the model with the relu activation, we can expect the model the perform similar on out of training data

as on training data. Hence it generalises well.

### Question 10: How does the new model's accuracy on test set classification differ from the previous model? Why do you think this is?

This model scores better on the test data than the previous model. This is unsurprising, the performance on the validation set was consistently better than that of the previous model (see Figure 1).

| loss | acc |
|---|---|
| 0.0841106 | 0.9763 |

## Deep convolutional networks

```
x_train_new = mnist$train$x
x_test_new = mnist$test$x
y_train_new = mnist$train$y
y_test_new = mnist$test$y

x_train_new <- array_reshape(x_train_new, c(nrow(x_train_new), 28, 28, 1))
dim(x_train_new)
```

```
## [1] 60000    28    28     1
```

```
x_test_new <- array_reshape(x_test_new, c(nrow(x_test_new), 28, 28, 1))
dim(x_test_new)
```

```
## [1] 10000    28    28     1
```

```
x_train_new = x_train_new/255
x_test_new = x_test_new/255

y_train_new <- to_categorical(y_train_new)
y_test_new  <- to_categorical(y_test_new)

modelDeep <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),activation = 'relu',
                input_shape = c(28,28,1)) %>%
  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 = 128, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')

summary(modelDeep)
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## conv2d (Conv2D)                  (None, 26, 26, 32)            320
## ___________________________________________________________________________
## conv2d_1 (Conv2D)                (None, 24, 24, 64)            18496
## ___________________________________________________________________________
## max_pooling2d (MaxPooling2D)     (None, 12, 12, 64)            0
## ___________________________________________________________________________
```

```
## flatten (Flatten)                (None, 9216)                0
## ________________________________________________________________________________
## dense_4 (Dense)                  (None, 128)                 1179776
## ________________________________________________________________________________
## dense_5 (Dense)                  (None, 10)                  1290
## ================================================================================
## Total params: 1,199,882
## Trainable params: 1,199,882
## Non-trainable params: 0
## ________________________________________________________________________________
```

```
modelDeep %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)
```

```
historyDeep <- modelDeep %>% fit(
  x_train_new, y_train_new,
  batch_size = 128,
  epochs = 6,
  verbose = 1,
  validation_split = 0.2
)
```

### Question 11: Plot the training history and add it to your answers

See "deep model", Figure 2.

### Question 12: How does the training history differ from the previous model, for the training and validation sets? What does this tell us about the generalisation of the model?

Overall, the performance of the deep model improved quicker than that of the simpler model (see Figure 2). The performance on the training set and the validation set converged quickly for the deep model, and end up being very similar, so the model is likely to perform similar on out of training data and training data. The validation loss is a bit higher than the training loss after the $2^{nd}$ epoch, but it does not differ much. The difference is smaller than that of the previous model. It probably generalises a tiny bit better, i.e. training and test difference might be tiny bit smaller than it was for the simpler model.

### Question 13: What values do you get for the model's accuracy and loss?

| loss | acc |
|---|---|
| 0.0316117 | 0.9895 |

### Question 14: Discuss whether this accuracy is sufficient for some uses of automatic hand-written digit classification.

The accuracy is fairly sufficient for automatic hand-written digit classification in applications where 1 mistake in a 100 digits is doable and can be checked manually further on. For example: in postal codes there are multiple levels of check involved where there is not just a postal code, but also house number, street number etc.

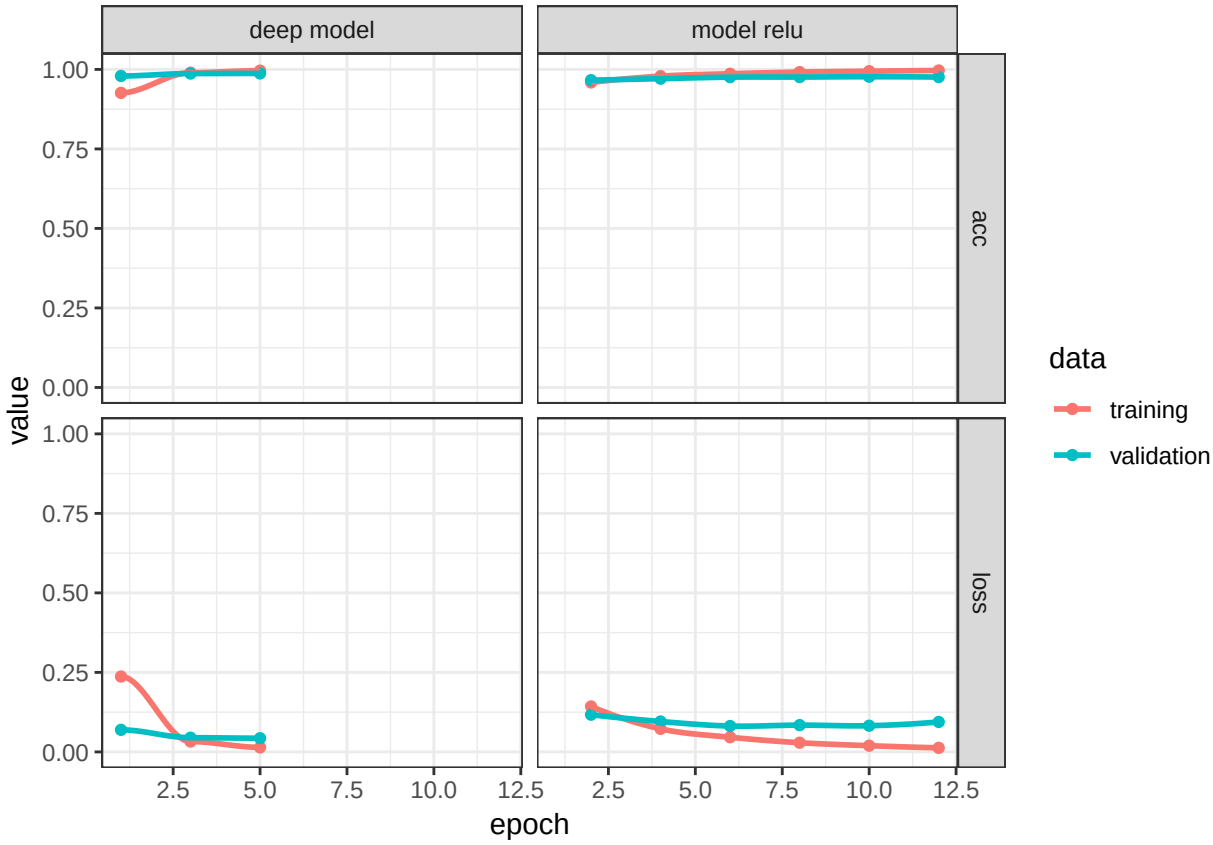


Figure 2: Comparison of the model with relu activation and the same model with extra 2 extra convolutional layers, and pooling and flattening.

### Question 15: Describe the principles of overfitting and how dropout can reduce this.

Large neural nets trained on relatively small datasets can overfit the training data. This may result in the model learning the statistical noise in the training data, which results in poor performance when the model is evaluated on new data, e.g. a test dataset. Dropout prevents overfitting due to a layer's "over-reliance" on a few of its inputs. Because these inputs aren't always present during training (i.e. they are dropped at random), the layer learns to use all of its inputs, improving generalization.

```
modelDeepDrop <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),activation = 'relu',
                input_shape = c(28,28,1)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3),activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')

summary(modelDeepDrop)
```

```
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## conv2d_2 (Conv2D)                (None, 26, 26, 32)            320
## ___________________________________________________________________________
## conv2d_3 (Conv2D)                (None, 24, 24, 64)            18496
## ___________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)   (None, 12, 12, 64)            0
## ___________________________________________________________________________
## dropout (Dropout)                (None, 12, 12, 64)            0
## ___________________________________________________________________________
## flatten_1 (Flatten)              (None, 9216)                  0
## ___________________________________________________________________________
## dense_6 (Dense)                  (None, 128)                   1179776
## ___________________________________________________________________________
## dropout_1 (Dropout)              (None, 128)                   0
## ___________________________________________________________________________
## dense_7 (Dense)                  (None, 10)                    1290
## ===========================================================================
## Total params: 1,199,882
## Trainable params: 1,199,882
## Non-trainable params: 0
## ___________________________________________________________________________
```

```
modelDeepDrop %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)
```

```
historyDeepDrop<- modelDeepDrop %>% fit(
  x_train_new, y_train_new,
  batch_size =128,
  epochs = 6,
```

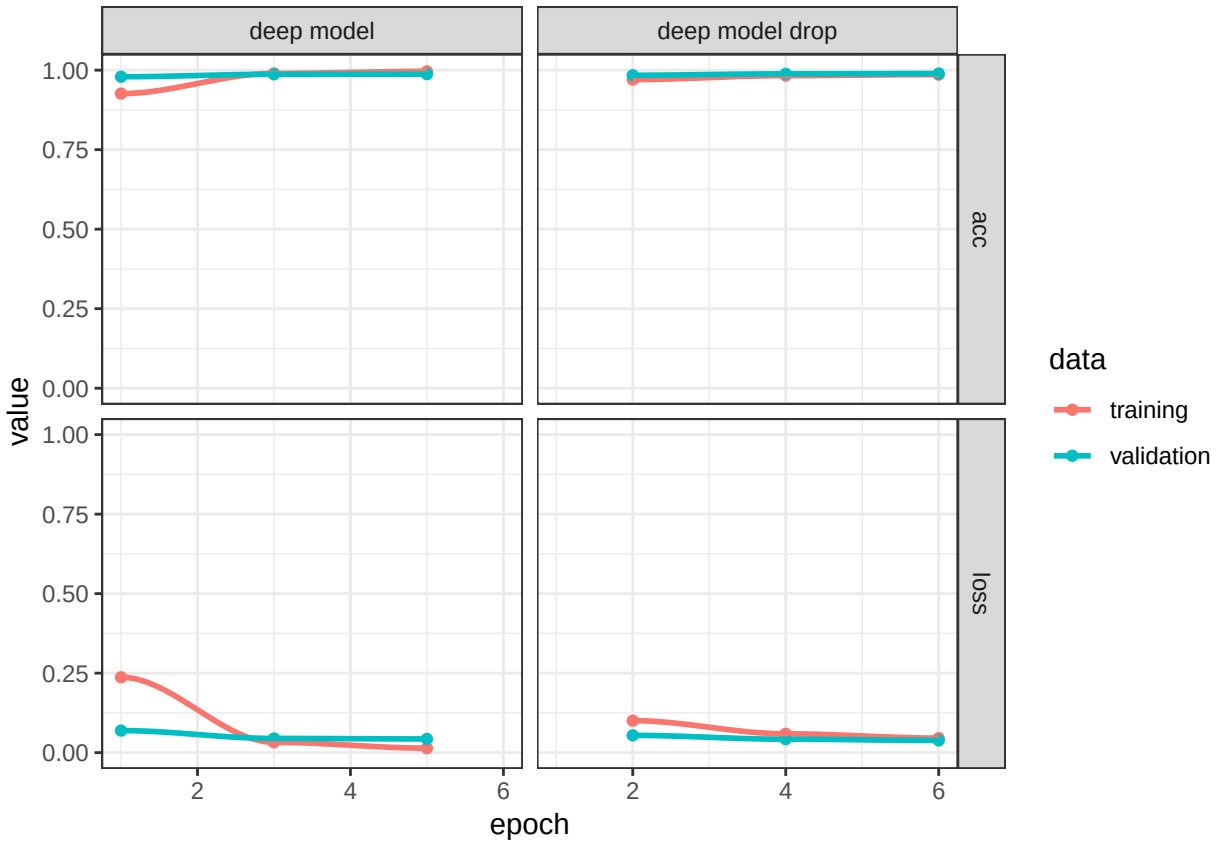


Figure 3: Comparison of the deep model with and without dropout.

```
  verbose = 1,
  validation_split = 0.2
)
```

### Question 16: How does the training history differ from the previous (convolutional) model, for both the training and validation sets, and for the time taken to run each model epoch?

In the dropout model training loss stays above validation loss (see Figure 3). In the deep model this was not the case. The training and validation loss and accuracy converge a bit quicker in the deep model than in the deep drop model. Training time was about 10 seconds longer for the deep drop model on average.

### Question 17: What does this tell us about the generalisation of the two models?

The models generalise well, training and validation performance are very close for both models.

## Question 18: What code did you use to define the model described here?

```
cifar10 <- dataset_cifar10()
```

```
x_train_cifar <- cifar10$train$x
x_test_cifar <- cifar10$test$x
y_train_cifar <- cifar10$train$y
y_test_cifar <- cifar10$test$y


x_train_cifar <- x_train_cifar/255
x_test_cifar <- x_test_cifar/255
y_train_cifar <- to_categorical(y_train_cifar, 10)
y_test_cifar <- to_categorical(y_test_cifar, 10)


modelDeepDropCifar <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),activation = 'relu',
                input_shape = c(32, 32, 3), padding = 'same') %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),activation = 'relu',
                padding = 'same') %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')

summary(modelDeepDropCifar)
```

```
## ________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ========================================================================
## conv2d_4 (Conv2D)                (None, 32, 32, 32)            896
## ________________________________________________________________________
## conv2d_5 (Conv2D)                (None, 30, 30, 32)            9248
## ________________________________________________________________________
## max_pooling2d_2 (MaxPooling2D)   (None, 15, 15, 32)            0
## ________________________________________________________________________
## dropout_2 (Dropout)              (None, 15, 15, 32)            0
## ________________________________________________________________________
## conv2d_6 (Conv2D)                (None, 15, 15, 32)            9248
## ________________________________________________________________________
## conv2d_7 (Conv2D)                (None, 13, 13, 32)            9248
## ________________________________________________________________________
## max_pooling2d_3 (MaxPooling2D)   (None, 6, 6, 32)              0
## ________________________________________________________________________
## dropout_3 (Dropout)              (None, 6, 6, 32)              0
## ________________________________________________________________________
## flatten_2 (Flatten)              (None, 1152)                  0
## ________________________________________________________________________
## dense_8 (Dense)                  (None, 512)                   590336
## ________________________________________________________________________
## dropout_4 (Dropout)              (None, 512)                   0
```

```
## _________________________________________________________________
## dense_9 (Dense)                 (None, 10)                    5130
## =================================================================
## Total params: 624,106
## Trainable params: 624,106
## Non-trainable params: 0
## _________________________________________________________________
```

```
modelDeepDropCifar %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(lr = 0.0001, decay = 1e-6),
  metrics = c('accuracy')
)
```

```
historyDeepDropCifar<- modelDeepDropCifar %>% fit(
  x_train_cifar, y_train_cifar,
  batch_size =32,
  epochs = 20,
  verbose = 1,
  validation_data = list(x_test_cifar, y_test_cifar),
  validation_split = 0.2,
  shuffle = TRUE
  # The data in the cifar set is not ordered.
  # So setting a shuffle = TRUE is void and
  # has no effect on the accuracy of the
  # said model.
)
```

### Question 19: Execute this model fit command. After your fitting is finished, plot the training history and put it in your answers.

See "model cifar", Figure 4.

| loss | acc |
|---|---|
| 0.8252948 | 0.7166 |

### Question 20: How does the training history differ from the convolutional model for digit recognition? Why do you think this is?

The accuracy and loss function seem to take more epochs before they plateau for the cifar model than for the digit recogniser (see Figure 4). This is because it takes the model longer to figure out the patterns in the data. The training loss is slightly higher than the validation loss.

### Question 21: How does the time taken for each training epoch differ from the convolutional model for digit recognition? Give several factors that may contribute to this difference

Each epoch took roughly 160 seconds on average. The network is more convoluted, and has a deeper layering than the previous models. Hence it takes longer to run.

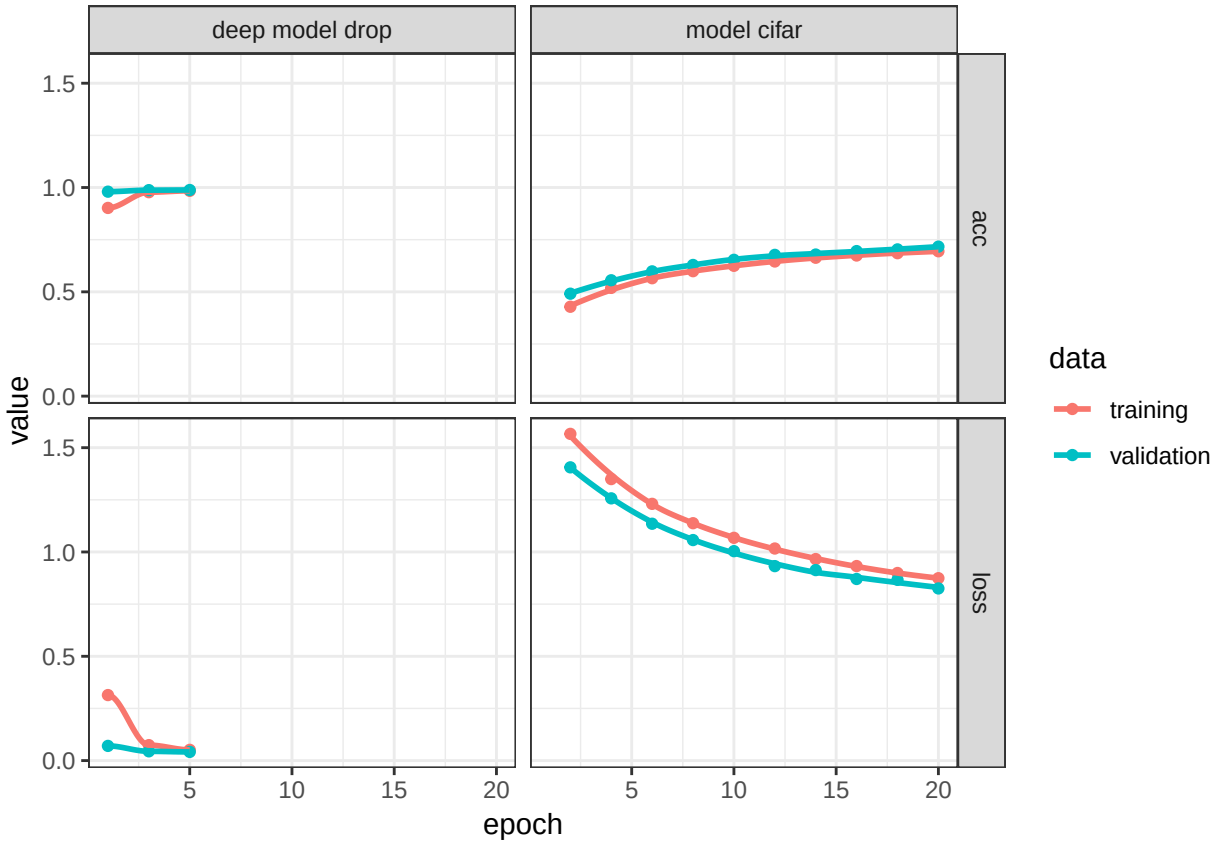


Figure 4: Comparison of the deep convolutional model with dropout for digit recognition and the deep convolutional model for image recognition.

## Question 22: Read the research paper "Performance-optimized hierarchical models predict neural responses in higher visual cortex"

Available from: http://www.pnas.org/content/pnas/111/23/8619.full.pdf. Write a short (~500 word) summary of the experimental approach and results.

**Problem:**

Diverse tunings of neurons in the inferior temporal cortex are difficult to characterize.

**Objective:**

Modelling approach to yield a quantitatively accurate model of the Inferior Temporal Cortex. The task is to find a neural network model that matches or maybe even exceeds human performance on object recognition tasks.

**Approach:**

Measurement of Neural ITC responses are made on an image set consisting of 5760 photorealistic 3d images in cluttered backgrounds, which would be otherwise difficult for a vision system to recognize. This is done using electrode arrays from 168 ITC neurons. Then high throughput computation was used to evaluate other neural network models on the same image set measuring categorization performance as well as ITC neural predictivity. Categorization performance was measured on Support Vector Machines Linear Classifier and cross validation testing was done on them. To assess the neural predictivity a linear regression for each target ITC neuron site was used which was mapped to identifying a synthetic neuron built on linear weighting of the model outputs that would resemble or match that space on fixed sample images and then tested response predictions against actual neural site outputs on novel images. Models were built from large parameter space of CNN's which approximated the general retinotopic structure of the ventral system through spatial complexity through computations in any particular region of vision identical to other places. The CNN layers were stacked hierarchically to create deep neural networks.

**Results:**

The steps followed show that there is optimization involved to directly guide neural mechanisms. A model with perfect neural predictivity in Inferior Temporal Cortex will exhibit high performance because the ITC itself does. The converse being true is demonstrated within a biologically plausible model class which is made by combining high throughput computational and electrophysiology techniques to explore biologically plausible hierarchical neural network models and then measure them against V4 and ITC. This is also used to show that there is a strong correlation between a model's performance on high variation object recognition and translating it to predict individual ITC neuron firings. It is also proved that top down performance thresholds directly shape the intermediary visual representations.

## Question 23: Play around with these settings and see how they affect your ability to learn classification of different data sets.

Write down what you found and how you interpret the effects of these settings. Depending on your inclination and how long the other questions took you, this may be 10 minutes work or an hour.

**Findings:**

We need to classify a bunch of points based on their location in a 2d image. The data set consists of different coordinates classified as blue or orange. Our objective here is to create a neural network that, given no prior knowledge, can figure out if a given point should be blue or orange and predicts successfully which classification it should be. We know ahead of time the correct classification for each of the points using which we will train our neural network. Starting with a dataset that we want to play with. The inputs are the x and y coordinates of each data point. So, for classification our neural network only works with these two values

and they start off as equally weighted. So, each of the inputs are connected to neurons in the hidden layer by the factor of a weight, which can be adjusted/manipulated to create the learning that we want. These in turn are fed into more hidden layers or the output neurons, which will ultimately decide which classification will be predicted. Keeping in mind, this is a binary classification problem, i.e. either blue or orange. Thus, we only need a single signal in actuality which comes into the output. The thickness of the connections signify their weights.

## Settings:

. Learning rate:

- With what rate are the weights adapted? How large is the effect of the error on each weight. You don't want the learning rate to be to high; every mistake will bring about large changes in the network. If the learning rate is very low however, it takes the network longer to adapt to the feedback.

. Activation:

- The activation function. Defines relation between input of a neuron and the output. A linear relation leads to an increase in output equal to increase in input. ReLu, Sigmoid, or Tanh, impose certain thresholds. The input needs to exceed this threshold to lead to activation of the neuron. Sigmoid and tanh have a upper limit for the input strength to lead to activation of the neuron, in addition to the lower threshold.

. Regularization:

- Kind of smooths the model prediction. Reduces the variability of the model and consequently can prevent overfitting.

. Regularization rate:

- How much the model is regularised.

. Problem type:

- Classification/regression

. Ratio of training to test data:

- How much of the data should be used for training the model and how much should be used to test the model?

. Noise:

- How large should the irreducible error of the data be?

. Batch size:

- The number of datapoints that are used per iteration to train the network. The smaller, the more accurate, yet it takes longer.

. Input features:

- What features do we use to categorise the x and y coordinates?

. Hidden layers:

- How many hiddenlayers do we use.

. n neurons:

- How many neurons does each layer have?

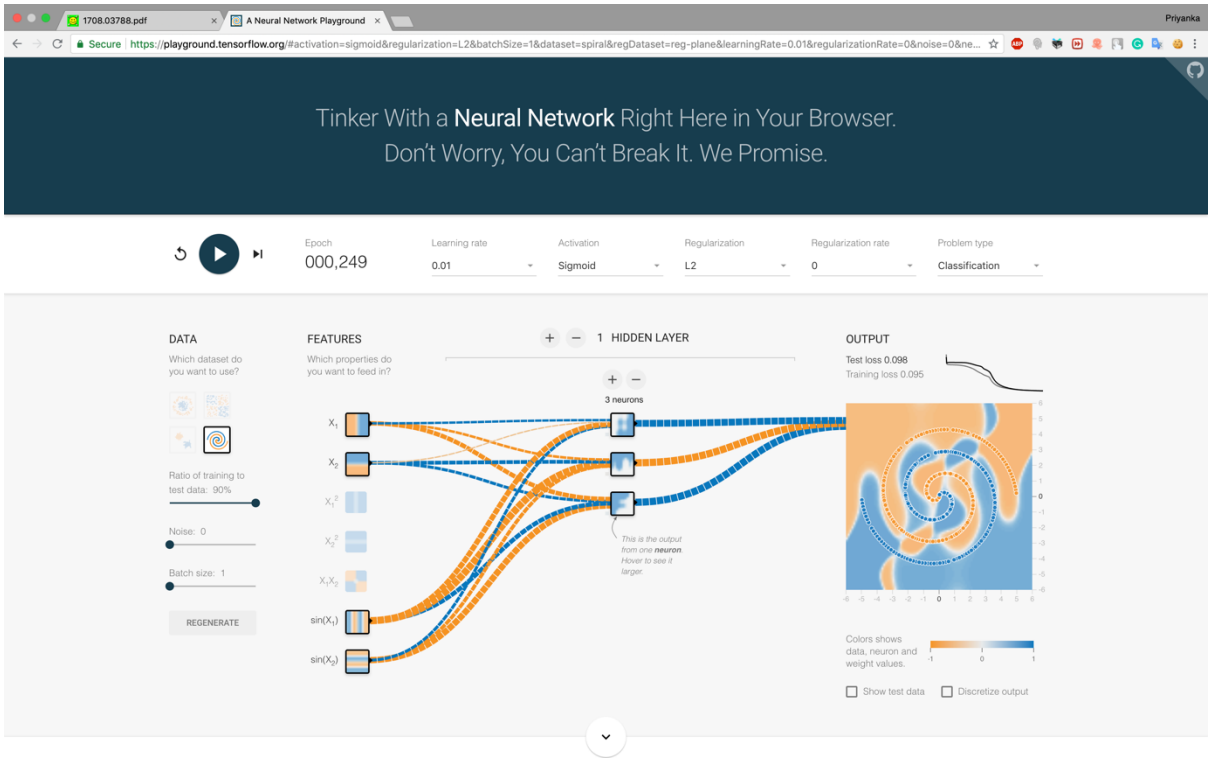


Figure 5: Spiral recognition model

## Question 24: What is the minimum you need in the network to classify the spiral shape with a test set loss of below 0.1?

See Figure 5.

. Learning rate: 0.01

. Activation: Sigmoid

. Regularization: L2

. Regularization rate: 0 (default)

. Problem type: Classification

. Ratio of training to test data: 90:10

. Noise: 0 (default)

. Batch size: 1

- 1 [Mini batch; gives more improved accuracy but is computationally more expensive] Most of the parameters have been left as default as the question asks for the "minimum" needed in the network to classify the data. Before the 230th epoch, we have already achieved Test loss of below 0.1 as asked in the question.

. Input features: X1, X2, Sin(X1) and Sin(X2)

. Hidden layers: 1

. n neurons: 3 neurons