

Lab assignment 1: Image recognition using deep networks

By Ben Harvey

In these exercises, you will explore questions of visual image processing using the Keras library for RStudio. If you do not have experience with RStudio or R, we recommend that you first complete Chris Janssen's 'Crash Course' in R, which will familiarise you with the basics of R.

Keras is a high-level library for building artificial machine learning networks, specialising deep learning networks. RStudio is used to address Keras, while Keras primarily calls the open-source TensorFlow library, developed by the Google Brain team. TensorFlow in turn is written in Python and C++. As a result, Python must be installed too, which is typically done automatically when installing Keras.

Keras is therefore best viewed as an interface for TensorFlow, while TensorFlow is the underlying machine learning framework. Keras gives a more intuitive set of functions, making it easy to develop machine learning models, particularly in the context of deep learning networks. Keras, like TensorFlow and Python, has excellent cross-platform support, and importantly makes it very easy to run models on their CPUs, GPUs or clusters of either. Because artificial learning networks rely on many simple computations running in parallel, using GPUs is ideal for this application.

Because we will use relatively simple networks in this class, it is feasible to run these exercises on your own computers. However, modern laptops vary considerably in their CPUs and GPUs. Using a faster GPU will considerably reduce your waiting times, so if you have a choice of computers, choose the one with the better graphics card.

You should work in pairs on these exercises. We suggest doing these exercises on both of your computers simultaneously: this improves (student) learning and also makes it easier to find trivial mistakes.

In the following text, explanation is plain text, instructions are underlined, questions to answer are labelled, and RStudio console commands are written in different font. You are the first class doing this lab, so please note any mistakes and how you fixed them, as this will help our course development.

Installation:

If you do not already have R installed, download it here and install it:
<https://cran.rstudio.com/>. R is installed in many Linux distributions already.

If you do not already have RStudio installed, download it here and install it after installing R: <https://www.rstudio.com/products/rstudio/download/#download>

Open RStudio. In RStudio's console, install Keras using the following commands:

```
install.packages("keras")
install.packages("kerasR")
```

You may find that you this prompts you to install Python, go ahead.

Now load the Keras library. You may need to do this each time you restart RStudio.

```
library(keras)
```

Now install all the libraries that Keras relies on (like TensorFlow):

```
install_keras()
```

Exercise one: Identifying handwritten numbers

We will begin using a very simple image recognition example: classifying hand-written numbers. This is a very useful ability for computers as it allows mail carriers to read hand-written postal codes and house numbers, and thereby sort mail automatically. It also allows banks to read numbers from cheques.

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

We will use a database of labelled handwritten numbers, called MNIST.

First, download MNIST:

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

MNIST contains a training set of 60,000 grayscale images (each 28x28 pixels) of hand-written numbers (mnist\$train\$x) together with the numbers shown in each image, the labels (mnist\$train\$y). It also contains a similar test set of 10,000 images and labels (mnist\$test\$x and mnist\$test\$y).

Here we will use two types of artificial learning network. Before using a deep convolutional network, we will test a multilayer perceptron. This is a type of artificial neural network where all nodes in each layer are connected to all nodes in the next layer. Therefore, this is not a convolutional network because there is no spatially-restricted convolutional filter, and no use of spatial relationships.

Data preparation:

So we will start by flattening the two spatial dimensions to convert from a 60000x28x28 training set to a 60000x784 training set in which spatial relationships are removed.

Use the functions `array_reshape` and perhaps `nrow` to convert the flatten training and test set images from 28x28 pixels to a column of 784 pixels for each image (row). Save the results as new variables called `x_train` and `x_test` to avoid overwriting the original images. These should have dimensions 60000x784 and

10000x784 respectively (use the function `dim` to check this). You can use the help file for `array_reshape` to see how to call this function properly, using `help("array_reshape")`.

Now rescale `x_train` and `x_test` to values between zero and one by dividing each variable by 255.

The labels are currently specified as numbers between 1 and 10. For our network, these numbers must each be separate network units in the output layer. Use the function `to_categorical` to convert the train and testset labels to two new variables, called `y_train` and `y_test`. For each label, there should be 10 elements, 9 of them zeros and 1 of them a one.

Model definition:

You will want to keep a record of the code you write here so you can easily make modifications and run it again.

For our multi-layer perceptron (MLP), we will pass our 784 input units (flattened pixels) into a 256-unit fully connected hidden layer. This in turn feeds into our label (output) layer whose activation follows a softmax function to give the probability that each image is each digit. We initialise this model using the function `keras_model_sequential`. The fully-connected layers are defined by the Keras function `layer_dense` and linked using the pipe operator `%>%`.

So, to make the MLP model described above, enter the following code:

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

To check the resulting model is what you expect, use:

```
summary(model)
```

If this looks OK, compile the model with suitable loss functions, optimisation procedures, and performance measures, as follows:

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

Training and evaluation

Now we will fit the model to our training set, keeping a history of the performance at each stage. We will use 12 training epochs. We will set aside a random 20% of our data to check performance at each training epoch (which don't change between epochs). In each training epoch, we will use 128 image-label pairs per batch to

improve computational efficiency. We will use the verbose argument to tell us what is happening in each epoch.

To fit the model as described above, enter the following code:

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

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

Plot the training history and add it to your answers (Question 3)

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. (Question 4).

Evaluate the model performance on the test set using the following command:

```
score <- model %>% evaluate(  
  x_test, y_test,  
  verbose = 0  
)
```

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

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

Changing model parameters

In the previous model, we did not specify an activation function for our hidden layer, so it used the default linear activation. How does linear activation of units limit the possible computations this model can perform? (Question 7)

Now make a similar model with a rectified activation in the first hidden layer, by adding the extra argument:

activation = "relu"

to the model definition for this layer. Then compile, fit and evaluate the model.

Plot the training history and add it to your answers (Question 8)

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? (Question 9)

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

Deep convolutional networks

Our first two models used fully-connected networks for number recognition. They ran quickly, largely because of their very simple structure. They learned the relationships between our pixels and the numbers they represent fairly well. However, they had a limited ability to generalise to new data that they were not trained on.

In deep convolutional networks, each convolutional filter samples from the limited space in the previous layer's feature map. To use the whole image to determine outputs, they need more layers to allow more spatial integration. They also need multiple feature maps at each layer to capture the multiple meaningful spatial relationships that are possible. All of this greatly increases computational load.

First, we need to prepare our data differently. Convolutional layers don't flatten x and y spatial dimensions, and need an extra dimension for colour channels (in the input image) or multiple feature maps (from previous convolution steps). Reshape `mnist$train$x` to a new variable (`x_train`) of size 60000, 28, 28, 1. Reshape `mnist$test$x` to a new variable (`x_test`) of size 10000, 28, 28, 1. Rescale both results to values between zero and one as before. `y_train` and `y_test` are categorical units, as before.

Now we will define a convolutional learning model with 2 convolutional layers that result from 32 convolutional filters into the first layer and 64 filters into the second. We will use 3x3 pixel filters to sample from the image to the first layer, and the same to sample from the first layer to the second. We will use rectified activation functions for both convolutional layers. We will use pooling to downsample the second convolutional layer to half its size in both spatial dimensions (so one quarter of the pixels). We will flatten the resulting feature map to one dimension, then use one fully-connected layer to link our network to the labels.

To make the (slightly deep) convolutional network model described above, enter the following code:

```
model <- 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')
```

Now summarise the model to check it is what you want.

When compiling the model, we will use a slightly different backpropagation of error procedure, so change the 'optimizer' part of the compile command to:
`optimizer = optimizer_adadelta(),`

Fit the model as before, but using only 6 epochs. Expect that model fitting will take far longer.

Plot the training history and add it to your answers (Question 11)

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? (Question 12)

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

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

'Dropout' is a method used in deep network training to prevent overfitting of training data and focus on aspects of the learning model that will generalise to new data. Describe the principles of overfitting and how dropout can reduce this (Question 15)

Add dropout layers (`layer_dropout()`) after the max pooling stage (rate = 0.25) and after the fully-connected (dense) layer (rate = 0.5). Compile and train the resulting model as before. 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? (Question 16)

What does this tell us about the generalisation of the two models? (Question 17)

Exercise two: Identifying objects from images

Identifying objects from images with a high variation in object position, size and viewing angle is a particularly difficult problem, and a major application of artificial deep learning networks. Here we will build a network to do this. This network has more layers and is more computationally-intensive than previous exercises, even using the low-resolution images we will use here.

The CIFAR-10 dataset contain colour images of objects, each 32x32x3 pixels (for the three colour channels). These have 10 categories (or classes) of object (airplane, automobile, bird, cat, deer, dog, frog, horse, ship & truck) with 5,000 images in each, making a total of 50,000 images in the training set (`cifar10$train$x`), randomly ordered with numerical labels for each (1=airplane, 2=automobile etc.). The test set (`cifar10$test`) contains 10,000 images ordered by their label.

First, download and load the data set using:
`cifar10 <- dataset_cifar10()`

Prepare the test and training images by dividing their values by 255, storing the result in variables `x_train` and `x_test`. The shapes of the image matrices are already correct for input into Keras. Convert the training and test labels to categorical

variables, as before with the handwritten digits, storing the result in variables `y_train` and `y_test`.

Define the model using the convolutional network with dropout (from Questions 11 and 16) as a template.

-For the first convolutional layer, add the arguments:

`input_shape = c(32, 32, 3)`, `padding = "same"`
to the `layer_conv_2d` call.

- In the second convolutional layer, use 32 filters instead of 64 to reduce computational load.

-After max pooling and dropout layers, repeat these layers again (add `conv`, `conv`, `pool`, `dropout`, after the existing `conv`, `conv`, `pool`, `dropout`). There is no need to define `input_shape` here.

-Flatten the result and link it to a larger fully-connected layer than before, using 512 units instead of 128, with dropout as before.

-Link this to a 10-unit output layer as before.

-What code did you use to define the model described here? (Question 18)

In compiling the model, we will use a specialised optimizer module:

`optimizer = optimizer_rmsprop(lr = 0.0001, decay = 1e-6)`

Now we will fit the model, which **will take a couple of hours at least**. During fitting, there are other exercises to do (Question 22 and Exercise three, below), so you don't need to wait. You may choose to run this overnight, particularly if each training epoch lasts more than 10 minutes.

In the model fit command, set the batch size to 32 and the number of epochs to 20. Rather than splitting the training data to give a validation set, we will pass in the test set as follows:

`validation_data = list(x_test, y_test),`

Finally, set:

`shuffle = TRUE`

to run through the images in a different order of batches each epoch.

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

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

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 (Question 21)

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. (Question 22)

Exercise three: Play time

It should now be clear that even a relatively simple deep convolutional learning network is quite computationally intensive to run on a personal computer. So we will now move to a web-based interface for deep learning, at:

<http://playground.tensorflow.org/>

Here, you can classify the object positions in different data sets (left panel) using deep convolutional network of differing complexity, different numbers of feature maps (number of 'neurons' in each hidden layer), different numbers of layers and different inputs ('features' column). At the top, you can also change the activation function and add normalisation (regularization). In the left column, you can change the ratio of training and test data, and add noise to the network to improve generalization to simulate imperfect inputs.

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 (Question 23)

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