Introduction and Set-up

This document provides a brief, practical, guide to developing neural network models in TensorFlow using the Keras high-level API in python.

On the machines in the KCL computer labs an appropriate python environment for building neural networks should have already been set up. You need to activate this environment, which can be done from the command line with the command:

```
conda activate NN
```

If this environment has not been created on your account (i.e. if after entering the above command you receive the error message “Could not find conda environment: NN”), or if you are not using a lab computer, then you can create the NN environment using the instructions below. These instructions assume that anaconda, an open-source python distribution (https://www.anaconda.com/), has been installed. Anaconda is installed on all machines in the KCL computer labs.

Anaconda can be used to set up separate environments (isolated copies of python, tools and libraries) for different projects, allowing you to work on multiple projects without risking issues with conflicting dependencies. A new environment in anaconda can be set up using the graphical user interface (anaconda-navigator) or using the the command line interface (with the conda command). To set up the NN environment for building neural networks enter the following at the command line:

```
conda create --name NN python
conda activate NN
```

Tools and libraries can be added to the new environment using anaconda-navigator or the command line. To add Keras, TensorFlow and other packages useful for developing neural network models to the NN environment enter the following at the command line:

```
conda install pandas
conda install scikit-learn
conda install tensorflow
conda install keras
conda install matplotlib
```

Loading Data

A dataset is required in order to train and test a neural network. Some standard datasets are provided as part of Keras (https://keras.io/datasets/) and can be easily loaded. For example, to load the MNIST dataset use the following python code:

```
from keras.datasets import mnist
(x_train, labels_train), (x_test, labels_test) = mnist.load_data()
```

The MNIST images are stored in the form of integers with values in the range [0,255]. To convert to floating-point numbers in the range [0,1] use the following python code:

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

The category labels are in the form of integers 0 to 9. To define the output that the network should produce in response to each sample (a one hot encoding) use the following python code:

```
from keras.utils import to_categorical
y_train = to_categorical(labels_train, 10)
y_test = to_categorical(labels_test, 10)
```

If the data is to be used as input to a dense layer, then it should be reshaped into a matrix where each row is a sample, using the following python code:

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
```

If the data is to be used as input to a convolutional layer, then it should be reshaped into a four-dimensional matrix where the first dimension corresponds to the number of exemplars, the second and third dimensions correspond to the width and height of each image, and the fourth dimension corresponds to the number of colour channels in each image:

```
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
```

Defining a Neural Network

The first step in developing a neural network is to define its architecture (the number and types of layers, and their interconnectivity). Keras allows neural networks to be defined in several different ways, two are described below.

Method 1 Sequential model

The first method allows you to define a sequence of layers, it is assumed that the output of one layer provides the input to the next. For example, to build a three layer MLP network called "net" use the following python code:

```
from keras.models import Sequential
from keras.layers import Dense
net = Sequential()
net.add(Dense(800, activation='relu', input_shape=(784,)))
net.add(Dense(400, activation='relu'))
net.add(Dense(10, activation='softmax'))
```

Note that when adding a layer to the network, we can define parameters, such as: the number of neurons (in the example above the 1st layer has 800 neurons, the second has 400 neurons, and the 3rd layer has 10 neurons), and the activation function (in the example above the 1st and 2nd layers use RELU, and the 3rd layer uses softmax). Other activation functions are also available (see https://keras.io/activations/).

There are many different types of layers that can be used. (see: https://keras.io/layers/core/). For example, to build a simple CNN using convolutional, maxpooling, as well as dense layers (and with batch normalisation and dropout for some layers):

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout,
BatchNormalization
net = Sequential()
net.add(Conv2D(filters=32, kernel_size=(5,5), activation='relu',
input_shape=(28,28,1)))
net.add(BatchNormalization())
net.add(Conv2D(64, (3, 3), activation='relu'))
net.add(MaxPool2D(pool_size=(2, 2)))
net.add(Flatten())
net.add(Dense(256, activation='relu'))
net.add(Dropout(rate=0.5))
net.add(Dense(10, activation='softmax'))
```

Method 2 functional API

The second method lets you give names to the output of each layer, and to use these name to define where each layer receives input from. For example, to build a three layer MLP network, equivalent to that built using method 1 use the following python code :

```
from keras.models import Model
from keras.layers import Input, Dense
input_img = Input(shape=(784,)) # define a placeholder for the input data
x = Dense(800, activation='relu')(input_img)
y = Dense(400, activation='relu')(x)
z= Dense(10, activation='softmax')(y)
net = Model(input_img, z)
```

This method allows the construction of networks in which one layer supplies input to multiple other layers, and/or one layer receives input from multiple other layers, e.g.:

```
from keras.models import Model
from keras.layers import Input, Dense
from keras.layers.merge import concatenate
input_img = Input(shape=(784,)) # define a placeholder for the input data
x = Dense(800, activation='relu')(input_img)
y1= Dense(100, activation='tanh')(x)
y2= Dense(200, activation='relu')(x)
y = concatenate([y1, y2])
z= Dense(10, activation='softmax')(y)
net = Model(input_img, z)
```

Note, the variable name used for one layer can be reused (overwritten) by another layer. For example, a simple CNN equivalent to that built with method 1, can be constructed like so:

```
from keras.models import Model
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout,
Input, BatchNormalization
inputs = Input(shape=x_train.shape[1:])
x = Conv2D(filters=32, kernel_size=(5,5), activation='relu')(inputs)
x=  BatchNormalization(x)
x = Conv2D(filters=64, kernel_size=(3,3), activation='relu')(x)
x = MaxPool2D(pool_size=(2, 2))(x)
x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dropout(rate=0.5)(x)
outputs = Dense(10, activation='softmax')(x)
net = Model(inputs=inputs, outputs=outputs)
```

Whichever method is used to define a network, a textual description of its structure can be obtained using:

```
net.summary()
```

or an image can be obtained using:

```
from keras.utils import plot_model
plot_model(net, to_file='network_structure.png', show_shapes=True)
```

Training a Neural Network

This is achieved by first compiling the network, at this stage the cost function (https://keras.io/losses/) and the optimizer (https://keras.io/optimizers/) that is to be used for training is defined. Next the training is performed using the function fit. This function needs to be supplied with the training data, and optionally, validation data, and other parameters such as the number of epochs and the batch size to be used, e.g.:

```
net.compile(loss='categorical_crossentropy', optimizer='adam')
history = net.fit(x_train, y_train,
                  validation_data=(x_test, y_test),
                  epochs=20,
                  batch_size=256)
```

The history variable returned by the fit function can be used to produce a plot showing the change in the cost function during training using the following python code :

```
import matplotlib.pyplot as plt
plt.figure()
plt.plot(history.history['loss'], label='training loss')
plt.plot(history.history['val_loss'], label='validation loss')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend()
```

A trained network can be saved to disk using:

```
net.save("network_for_mnist.h5")
```

A saved model can be reloaded using:

```
from keras.models import load_model
net=load_model("network_for_mnist.h5")
```

Testing a Neural Network

The performance of the trained network can be tested, for example, by finding the outputs that it produces to the testing data, and comparing these to the true category labels using the following python code :

```
outputs=net.predict(x_test)
labels_predicted=np.argmax(outputs, axis=1)
misclassified=sum(labels_predicted!=labels_test)
print('Percentage misclassified = ',100*misclassified/labels_test.size)
```

To show the outputs a network produces for a few specific exemplars use the following python code:

```
plt.figure(figsize=(8, 2))
for i in range(0,8):
    ax=plt.subplot(2,8,i+1)
    plt.imshow(x_test[i,:].reshape(28,28), cmap=plt.get_cmap('gray_r'))
    plt.title(labels_test[i])
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

for i in range(0,8):
    output = net.predict(x_test[i,:].reshape(1, 784)) #if MLP
    #output = net.predict(x_test[i,:].reshape(1, 28,28,1)) #if CNN
    output=output[0,0:]
    plt.subplot(2,8,8+i+1)
    plt.bar(np.arange(10.),output)
    plt.title(np.argmax(output))
```