TensorFlow fundamentals

This is a basic document to refer back to for remembering/refreshing the fundamentals of TensorFlow. This content has been primarily compiled from the following sources:

1. Introduction to TensorFlow for Artificial Intelligence, Machine Learning, and Deep Learning on coursera.org

The basic neural network model is a **Sequential model** and the basic neural network layer is the **Dense layer**. To make a basic Sequential neural network, the following code can be used:

```
model = keras.Sequential([keras.layers.Dense(units = 1, input_shape = [1])])
```

The above code creates a Sequential model with a single layer having a single neuron. Subsequent layers can be added as subsequent elements in the list and the order in which they are written from left to right is the order in which the input data travels through the model.

Once a model is defined, it needs to be compiled. Models are compiled as follows:

```
model.compile(optimizer = sgd, loss = 'mean_squared_loss')
```

The parameter **optimizer** is used to define the method for optimization wherer **sgd** stands for [[StochasticGradientDescent]] and **loss** stands for the loss measure to be considered in the minimization for optimization of the model.

Once the model has been defined, it can be trained using the command:

```
model.fit(xdata, ydata, epochs = 500)
```

where **epochs** stands for number of iterations to carry out.

In order to then predict after the model has been trained, the following command can be used:

```
model.predict([...])
```

In order to define a flat layer, the following can be used:

```
keras.layers.Flatten(inputs_shape = (...))
```

where **input_shape** takes a tuple of the shape of the input.

For creating a classifier network that should have **n** classes, make the last layer of the network a Dense layer with n units.

Usually, it is required that the **input_shape** be explicitly specified in the first layer of a network. However, it is not necessary to specify this for the subsequent layes as it is calculated and set automatically.

Most layers have an attribute known as **activation** which needs to be specified. For a classifier, the activation for the last layer of a classifier should be put as

tf.nn.softmax. A common popular activation function is the RelU function which can be set by specifying the activation as **tf.nn.relu**.

A basic measure of the performance after training can be done as follows:

```
model.evaluate(testx, testy)
```

Callbacks can be used to stop training to completion if some condition is met. This done with a particular class creation and method definition as follows:

```
class CallbackName(tf.keras.callbacks.Callback):
    # This is the exact method that is called that is called at the end
    # of ever epoch so the name must be so
    def on_epoch_end(self, epoch, logs={}):
        # This is just an example
        if( logs.get('loss')<0.4):
            do something
            self.model.stop_training = True
```

When using callbacks, an instance of the callback must be defined and then passed in a list to the **fit()** method like so:

```
myCallback1 = someCallback()
myCallback2 = someOtherCallback()
.
.
.
model.fit(trainx, trainy, epochs = 100, callbacks = [myCallback1 myCallback2, ...])
```

The python script for the basic model run on the Fashion MNIST dataset can be found here.

The script provides a basic usage overview of: - The Sequential model - The Dense and Flatten layers - The RelU and softmax activation functions - Callbacks - Defining a model, compiling it, fitting it to basic data and evaluating its performance on a test set

To create a basic convolution layer, the following can be used:

```
keras.layers.Con2D(filternum, kernel_size, padding, input_shape, strides, ...)
```

where **filternum** is the number of filters to use for the layer, **kernel_size** is the shape of the kernel to be used passed as a tuple, **padding** is the type of padding such as **valid** and **same**, **input_shape** is as with other layers and **strides** is the number of pixels to skip before doing another convolution.

To create a max pooling layer, the code is:

```
keras.layers.MaxPooling2D(poolingshape,...)
```

The other types of pooling layers follow a similar syntax.

A good way to get an overview of a model is to use the following command:

```
model.summary()
```

In order to intelligently load images from directories, an `ImageDataGenerator` may be used

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(rescale = 1./255)
generator = datagen.flow_from_directory(
    trainingDir,
    target_size = (...),
    batch_size = ...,
    class_mode = ...)
```

where: * **rescale** is used to normalize the data. * **flow_from_directory()** is used to create the data generator * **trainingDir** points to the **root** training directory that contain subdirectories containing the different images, each subdirectory belonging to a particular class. * **target_size** specifies the size of the image as a tuple. All images are reshaped to this size. * **batch_size** specifies the batches. * **class_mode** specifies whether it is binary or a multiple classification scenario.

The same method is to be used to create a data generator for the validation images.

When using a data generator, one invokes the `model.fit_generator()` method instead of the standard `model.fit()`. The basic syntax is as follows

```
model.fit_generator(
    train_generator,
    steps_per_epoch = ...,
    epochs = ...
    validation_data = validation_generator,
    validation_steps = ...
    verbose = ...)
```