"DeepLearning.AI TensorFlow Developer" Specialization

Andrii X

1 Introduction to TensorFlow for AI, ML, and DL

1.1 Week 1

- **Simple example aka "Hello, World!":**
  Define NN (1 layer with 1 neuron):

```
# Build a simple Sequential model
model = tf.keras.Sequential(
[keras.layers.Dense(units=1, input_shape=[1])]
)
```

Compile the model:

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

Provide the data:

```
# Declare model inputs and outputs for training
xs = np.array([-1.0,  0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)
```

Train the NN:

```
# Train the model
model.fit(xs, ys, epochs=500)
```

Use trained NN for new data:

```
# Make a prediction
print(model.predict([10.0]))
```

1.2 Week 2

- **A Computer Vision Example: Fashion MNIST dataset**
  The Fashion MNIST dataset is a collection of grayscale 28x28 pixel clothing images.
  1) Load the Fashion MNIST dataset:

```
fmnist = tf.keras.datasets.fashion_mnist
```

  2) Load the training and test split of the Fashion MNIST dataset:

```
(training_images, training_labels),
(test_images, test_labels) = fmnist.load_data()
```

  3) Normalize the pixel values of the train and test images:

```
training_images  = training_images / 255.0
test_images = test_images / 255.0
```

  4) Build the classification model:

```
model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(128, activation=tf.nn.relu),
tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

  Sequential - defines a sequence of layers in the neural network.
  Flatten - converts a 28x28 matrix into a 1-D array.
  Dense - adds a layer of neurons. Activation function **relu** passes values greater than 0 to the next layer.
  Softmax takes a list of values and scales these so the sum of all elements will be equal to 1. When applied to model outputs, you can think of the scaled values as the probability for that class.
  5) Compile and train the model:

```
model.compile(optimizer = tf.optimizers.Adam(),
loss = 'sparse_categorical_crossentropy',
metrics=['accuracy'])

model.fit(training_images, training_labels, epochs=5)
```

  6) Evaluate the model on unseen data

```
model.evaluate(test_images, test_labels)
```

  Exploration Exercises
  ex1: the below code c**reates a set of classifications for each of the test images**, and then prints the first entry in the classifications.

```
classifications = model.predict(test_images)
print(classifications[0])
# The output of the model is a list of 10 numbers.
# These numbers are a probability that the value
# being classified is the corresponding value
```

ex2: **adding more Neurons** we have to do more calculations, slowing down the process, but in this case they have a good impact – **we do get more accurate**. That doesn't mean it's always a case of 'more is better', **you can hit the law of diminishing returns very quickly**!
ex3: it may seem vague right now, but it reinforces **the rule of thumb that the first layer in your network should be the same shape as your data**. Right now our data is 28x28 images, and 28 layers of 28 neurons would be infeasible, so it makes more sense to 'flatten' that 28,28 into a 784x1.
ex4: **another rule of thumb** – the number of neurons in the last layer should match the number of classes you are classifying for.
ex5: consider the effects of additional layers in the network. There isn't a significant impact – because this is relatively simple data. **For far more complex data** (including color images to be classified as flowers that you'll see in the next lesson), **extra layers are often necessary**.
ex6: consider the **impact of training for more or less epochs**. Try 15 epochs – you'll probably get a model with a much better loss than the one with 5. Try 30 epochs – you might see the loss value decrease more slowly, and sometimes increases. This is a side effect of something called **'overfitting'**.
ex7: If you try to train the model without normalizing the data, you might find that the model takes longer to train, or that it's unable to learn effectively from the training data, leading to poorer performance on the test data. The reason you get different results with and without normalization is because the scale of the inputs can significantly impact the gradient of the loss function, and hence the updates to the weights during training. **Normalization ensures that the scale of the inputs is consistent**, which can make the training process more stable and efficient.
ex8: 'wouldn't it be nice if I could stop the training when I reach a desired value?' – i.e. 85% accuracy might be enough for you, and if you reach that after 3 epochs, why sit around waiting for it to finish a lot more epochs.... you have callbacks!

```
class myCallback(tf.keras.callbacks.Callback):
def on_epoch_end(self, epoch, logs={}):
if logs.get('accuracy') is not None
and logs.get('accuracy') > 0.60:
print("\nReached 60% accuracy so cancelling training!")
self.model.stop_training = True

callbacks = myCallback()
```

```
fmnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels) ,
(test_images, test_labels) = fmnist.load_data()

training_images=training_images/255.0
test_images=test_images/255.0
model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(128, activation=tf.nn.relu),
tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# model fitting with callback
model.fit(training_images, training_labels,
epochs=5, callbacks=[callbacks])
```

1.3 Week 3: Convolutional NN

- **Improving Computer Vision Accuracy using Convolutions**
  A neural network containing three layers – the input layer (in the shape of the data), the output layer (in the shape of the desired output) and only one hidden layer gives accuracy about 89% on training and 87% on validation. How does one make that even better? One way is to use something called **convolutions**. The ultimate **concept** is that they **narrow down the content of the image to focus on specific parts** and this will likely improve the model accuracy. This is perfect for computer vision because **it often highlights features that distinguish one item from another**. Moreover, the **amount of information needed is then much less** because one will just **train on the highlighted features**. That's the concept of **Convolutional Neural Networks**.

```
# Define the model
model = tf.keras.models.Sequential([

# Add convolutions and max pooling
tf.keras.layers.Conv2D(32, (3,3), activation='relu',
input_shape=(28, 28, 1)),
tf.keras.layers.MaxPooling2D(2, 2),
tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),

# Add the same layers as before
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(128, activation='relu'),
```

```
tf.keras.layers.Dense(10, activation='softmax')
])

# Print the model summary
model.summary()

# Use same settings
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Train the model
print(f'\nMODEL TRAINING:')
model.fit(training_images, training_labels, epochs=5)

# Evaluate on the test set
print(f'\nMODEL EVALUATION:')
test_loss = model.evaluate(test_images, test_labels)
```

- **Lab**:

```
import os
import numpy as np
import tensorflow as tf
from tensorflow import keras

# Load the data
# Get current working directory
current_dir = os.getcwd()
# Append data/mnist.npz to the previous path to get
# the full path
data_path = os.path.join(current_dir, "data/mnist.npz")
# Get only training set
(training_images, training_labels), _ =
tf.keras.datasets.mnist.load_data(path=data_path)
```

Pre-processing the data:
– Reshape the data so that it has an extra dimension. The reason for this is that commonly you will use 3-dimensional arrays (without counting the batch dimension) to represent image data. The third dimension represents the color using RGB values. This data might be in black and white format so the third dimension doesn't really add any additional information for the classification process but it is a good practice regardless.
– Normalize the pixel values so that these are values between 0 and 1. You can achieve this by dividing every value in the array by the maximum.

```
def reshape_and_normalize(images):
### START CODE HERE
```

```
# Reshape the images to add an extra dimension
images = images.reshape(images.shape[0],
images.shape[1],
images.shape[2],
1)
# Normalize pixel values
images = images / 255.0
### END CODE HERE
return images

# rest of the code like previously
```

1.4 Week 4: Real-life example

- **Lab: Training with ImageDataGenerator:**

```
import tensorflow as tf

model = tf.keras.models.Sequential([
# Note the input shape is the desired size of the
#image 300x300 with 3 bytes color
# This is the first convolution
tf.keras.layers.Conv2D(16, (3,3), activation='relu',
input\_shape=(300, 300, 3)),
tf.keras.layers.MaxPooling2D(2, 2),
# The second convolution
tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),
# The third convolution
tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),
# The fourth convolution
tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),
# The fifth convolution
tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
tf.keras.layers.MaxPooling2D(2,2),
# Flatten the results to feed into a DNN
tf.keras.layers.Flatten(),
# 512 neuron hidden layer
tf.keras.layers.Dense(512, activation='relu'),
# Only 1 output neuron. It will contain a value
# from 0-1 where 0 for 1 class ('horses') and 1 for
# the other ('humans')
tf.keras.layers.Dense(1, activation='sigmoid')
])
```

The **increasing number of filters** in deeper layers allows the network to **capture a hierarchy of features**, from simple to complex, which is essential for the network to understand and classify intricate patterns in images.

```
from tensorflow.keras.optimizers import RMSprop

model.compile(loss='binary_crossentropy',
optimizer=RMSprop(learning_rate=0.001),
metrics=['accuracy'])
```

In this case, using the **RMSprop optimization algorithm** is preferable to stochastic gradient descent (SGD), because **RMSprop automates learning-rate tuning** for us. (Other optimizers, such as Adam and Adagrad, also automatically adapt the learning rate during training, and would work equally well here.)