Machine Learning Applications

Lecture 3: TensorFlow

Lecture Notes

M.Sc. in Financial Markets

ACADEMIC YEAR 2021-2022

Dr. Dominic O'Kane

Version: September 2021

Reminder: Categorical Cross Entropy

- We will later see that the cost function used in multiclass classification is called the **sparse categorical cross entropy**
- I want to explain why – recall that the multiclass cross entropy is given by

$$C = -\frac{1}{m}\sum_{j=1}^{m}\sum_{k=1}^{K} y_{ik}\ln(p_{ik})$$

- The labels y here are in the form of one-hot encodings
- This means that y is a matrix with m rows and K columns
- This can be slow to calculate if the number of classes is high
- But if the categories are mutually exclusive – this is not always the case – we can simplify use the **categorical cross entropy**

Sparse Categorical Cross Entropy

- If they are mutually exclusive, only one of the K labels will have a non-zero value, for example a sample vector will look like this

$$y_i = [0, 0, 0, 1, 0, ..., 0, 0]$$

- So instead we first calculate the value of k for each sample which is non-zero – we basically end up with a vector of integers

$$y_{ik*} = 1 \qquad k*(i) = \arg\max(y_i)$$

- This gives us the sparse categorical cross entropy

$$C = -\frac{1}{m}\sum_{i=1}^{m} y_{ik*(i)} \ln(p_{ik*(i)})$$

- All the terms with y_{ik}=0 vanish

Final Word on Sparse Categorical Cross Entropy

- Sparse categorical cross entropy is the obvious choice when the classes are mutually exclusive as it's faster and uses less memory
- If we already have the labels as a vector of integers k*(i)

y = [1, 8, 0, 4, 5, 1, … 8, 2, 5]

then this is the automatic choice

- If we have the labels as a one-hot encoding matrix

y = [[0,1,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,1,0],
[1,0,0,0,0,0,0,0,0], …., [0,0,0,0,0,1,0,0,0,0]]

then we need to convert them to a vector of integers

- There will be a command to do this

Training a Deep Learning Model

Going Beyond Pure Gradient Descent

- Gradient descent is the main approach for training DNN
- But it has a constant step size and training parameter
- Is there a way to make it faster ?
- Two ideas
  - introduce momentum
  - let the training rate change
- We train a model over a number of **epochs**
- An epoch means applying all of the training examples to the gradient descent algorithm once
- Typically the number of epochs required will be 10 to 100
- You just look at the corresponding validation score to decide

Initialisation of Weights

- For Backprop to work we need to break the weight symmetry which is usually done by randomly assigning values
- Initially practitioners used $w_{ij} \sim N(0,1)$ but it was shown that this contributed to the vanishing/exploding gradients problem
- It was noted that with a sigmoidal activation the variance of the output is greater than the input variance
- There are a few newer approaches that have been part of the reason for the success of ANNs

Initialisation Strategies

- Glorot and Bengio (2010) showed that if weights are generated from the distribution

$$w_{ij} \sim N(0,1)$$

where the F's are the number of incoming and outgoing neurons at a node, the gradient problems were both significantly reduced

$$w_{ij} \sim N\left(0, \frac{1}{\sqrt{0.5\left(F_{in} + F_{out}\right)}}\right)$$

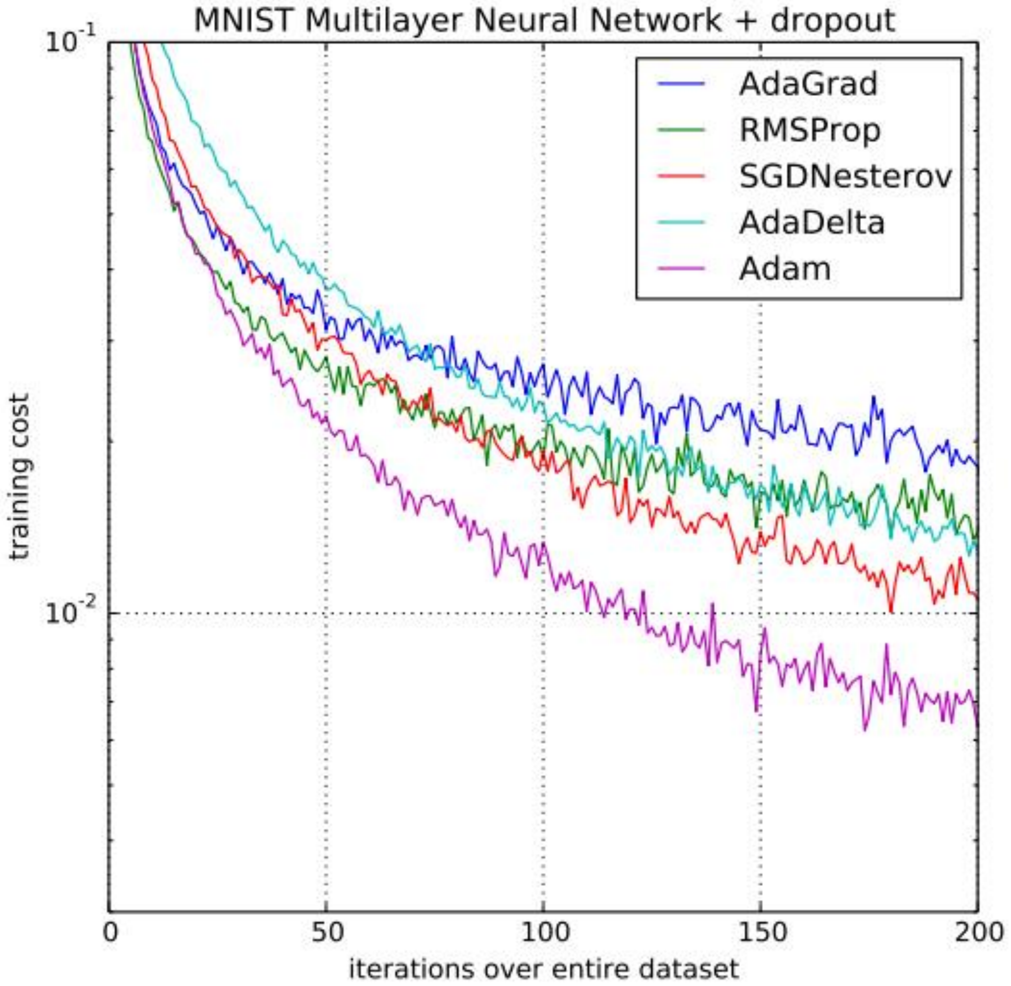
- There are other variations on this
  - LeCun initialisation
  - He initialisation

Adaptive Gradient Descent - ADAM Optimizer

- There are several gradient descent algorithms which combine some of these ideas
- An adaptive learning rate approach is used in AdaProp and RMSProp which already improve significantly on basic SGD
- SGD Nesterov uses momentum to adjust the next step – if two steps are in same direction the next step is bigger
- In 2014, Kingma and Ba published "ADAM: A method for stochastic optimisation"
- ADAM = Adaptive Moment Estimation
- ADAM uses the gradient and stores an exponentially decaying average of past squared gradients in its updating rule
- It has become the first choice for most deep learners

ADAM Beats the Other Adaptive Methods

- ADAM beat other algorithms on the MNIST digit recognition task



Batch Normalization

- Batch normalization removes extreme values from outputs
- It is an operation before or after the activation function of each layer and this operation
  - Centres the each input to the next layer around zero
  - It then normalises each input
- This is usually done in the training phase across samples
- There are two parameters required to do this
- Adding a BN layer at the first layer acts as a standard scaler

Dropout Layers

- Dropout is a trick invented by Hinton to improve NN performance
- Idea is that we randomly disable some neurons during training
- We set a probability p that a neuron will be removed but only during the training phase
- The idea is that by reducing dependency on individual neurons the other neurons are forced to learn more
- Typically, the probability p is 10-50%
- Can cause small but important improvement in performance

Neural Network Architecture

- Number of Hidden Layers
  - Start with a single layer see how it does
  - Add more layers – if performance improves, keep adding layers
- Number of Neurons per Layer
  - For input and output layer this is determined by the problem
  - For hidden layers, the same number per layer works fine
  - It also reduces the number of hyperparameters to tune !
  - You can experiment with different numbers
  - Maybe start with a high number and reduce it until the result starts getting worse

Neural Networks as Function Approximators

Why are Neural Networks Powerful

- It was shown that with one layer – the perceptron, we could only solve linearly separable problems
- We saw previously that feature engineering can be used to make a non-linearly separable problem linearly separable
- If the activation functions were linear then we would have a linear model and we would be dealing with a linear regression
- In neural networks the non-linear activation functions effectively create new non-linear features

Universal Approximation Theorem

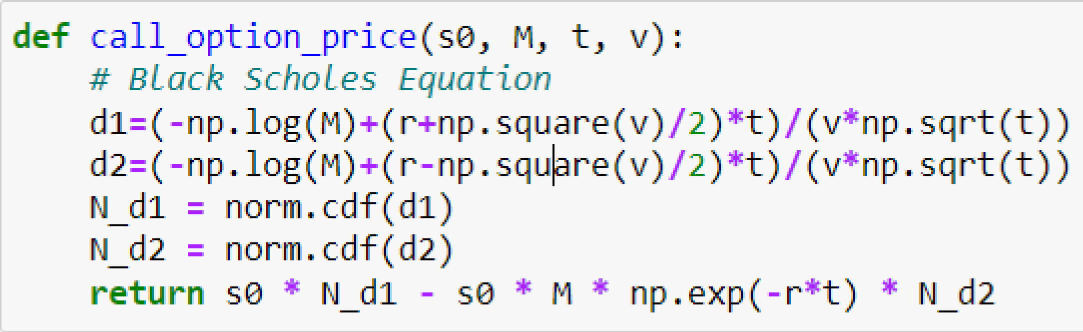
- Hornik showed in 1991 that it is not the specific choice of the activation function, but rather the multilayer feed-forward architecture itself which gives neural networks the potential of being universal approximators
- This relies on having sufficiently many hidden units
- Zu and Hanin proved mathematically that ANN can approximate any convex continuous function

Financial Case Study: Fitting to Black Scholes

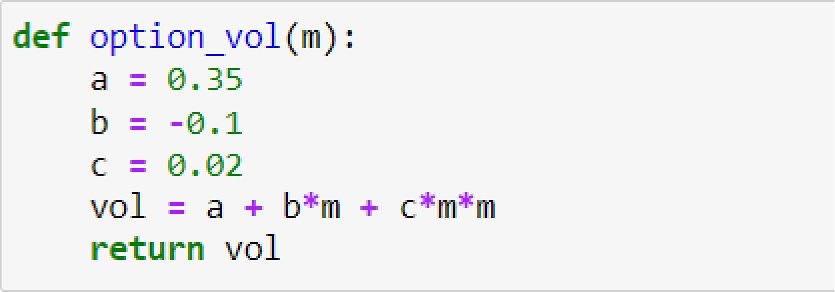
Fitting the Black Scholes Equation

- Let's do a financial case study with the Black-Scholes equation
- We're going to use our ANN as a function approximation tool
- We are going to define the Black Scholes equation with inputs
  - Moneyness K/S0
  - Time to Expiry
- We use moneyness as it removes the dependence on K and S0
- It reduces it to one variable
- We will set the volatility to be a function of moneyness
- The risk free rate is fixed and the dividend yield is zero
- We then want to see if it can correctly predict the price then for any moneyness K/S0 and Time To Expiry

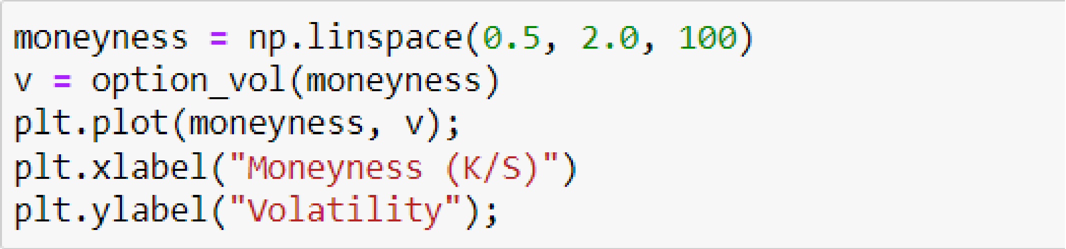
We rewrite Black Scholes in terms of moneyness M



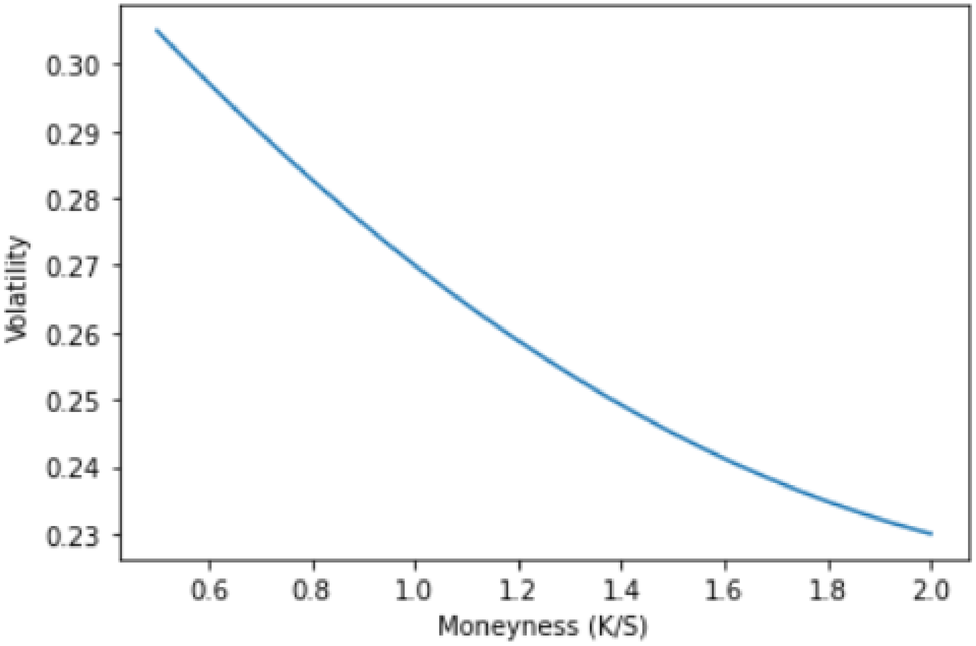
```
def call_option_price(s0, M, t, v):
    # Black Scholes Equation
    d1=(-np.log(M)+(r+np.square(v)/2)*t)/(v*np.sqrt(t))
    d2=(-np.log(M)+(r-np.square(v)/2)*t)/(v*np.sqrt(t))
    N_d1 = norm.cdf(d1)
    N_d2 = norm.cdf(d2)
    return s0 * N_d1 - s0 * M * np.exp(-r*t) * N_d2
```



```
def option_vol(m):
    a = 0.35
    b = -0.1
    c = 0.02
    vol = a + b*m + c*m*m
    return vol
```

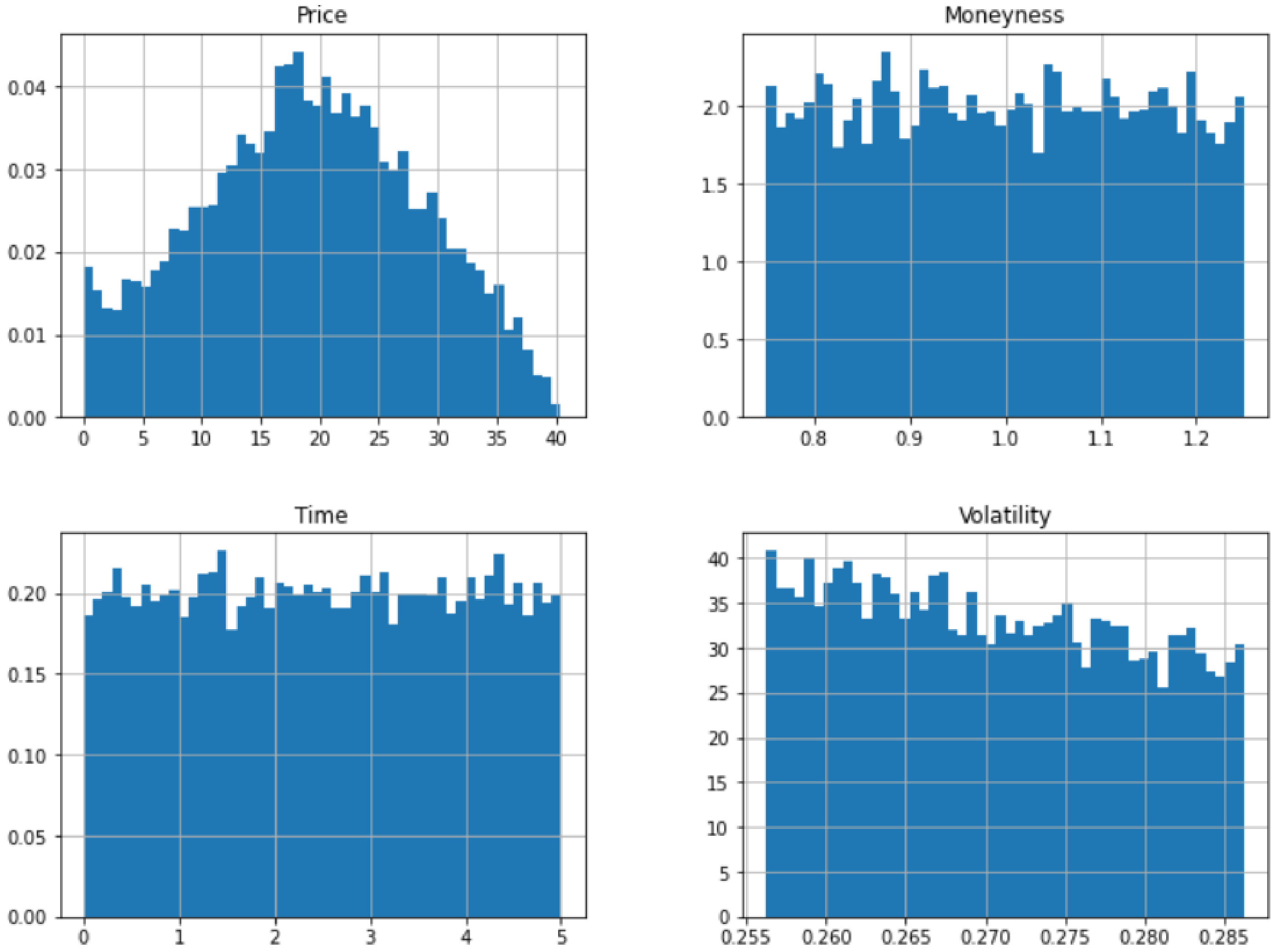


```
moneyness = np.linspace(0.5, 2.0, 100)
v = option_vol(moneyness)
plt.plot(moneyness, v);
plt.xlabel("Moneyness (K/S)")
plt.ylabel("Volatility");
```



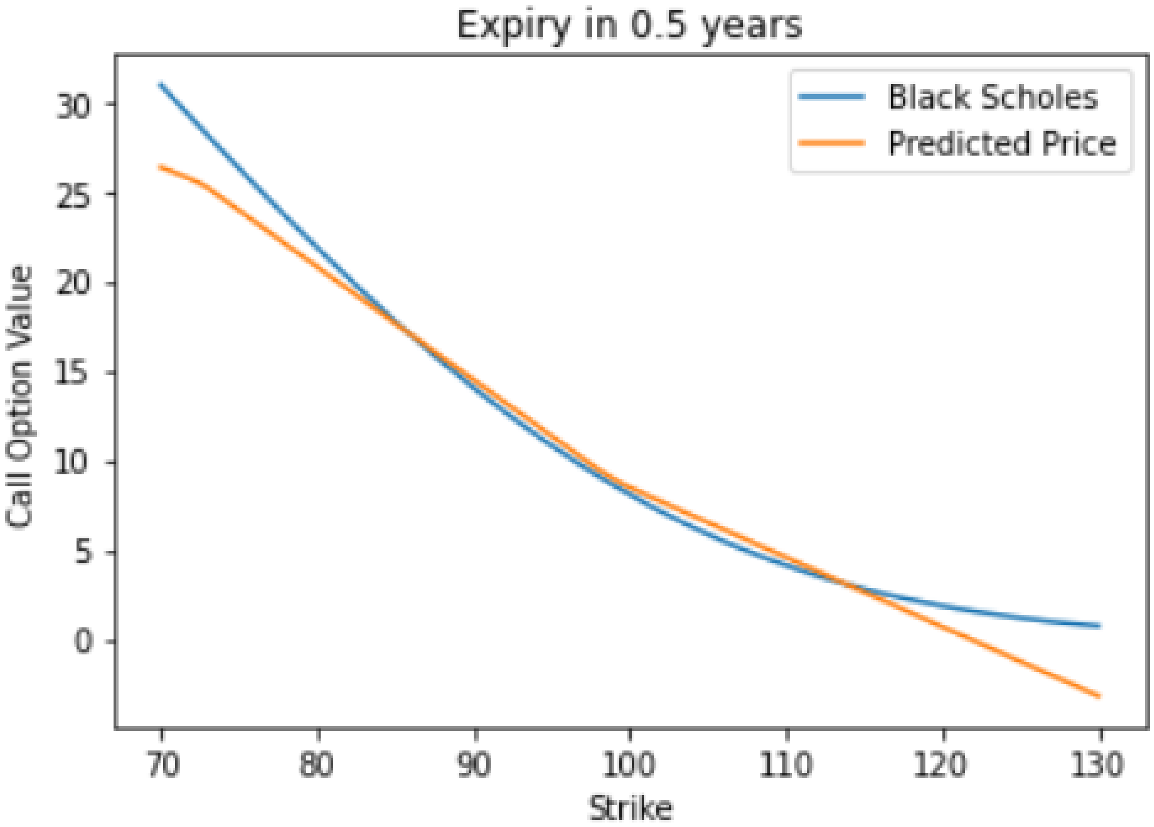
We then Draw Random Realisations of Black-Scholes

- We need data to train the model – we generate 10K prices



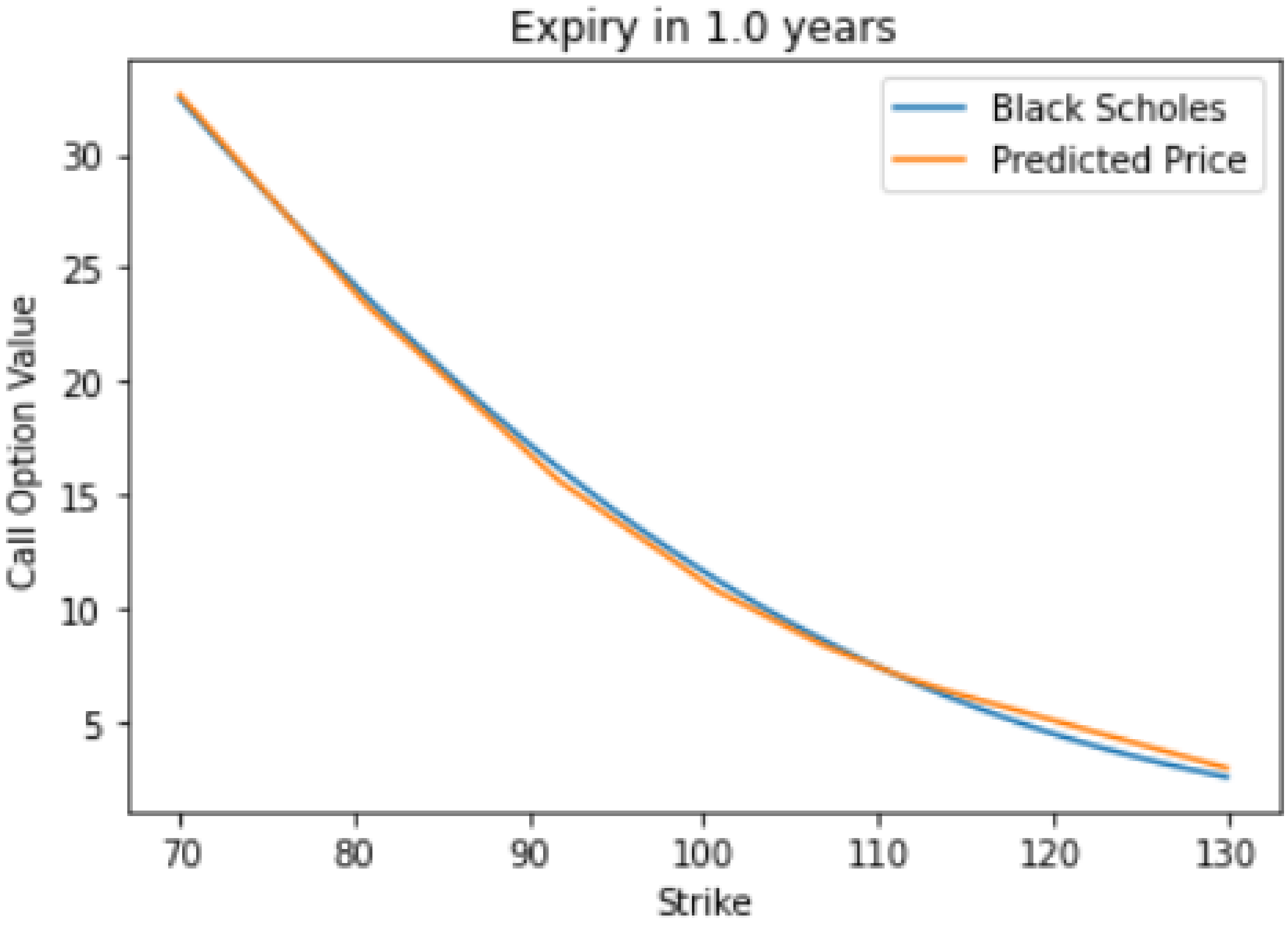
Training on moneyness from 0.75 to 1.25

- We train the MLP to learn the mapping from moneyness and time to expiry to option price
- The deep ITM and deep OTM fits are not great



Training on moneyness from 0.5 to 1.5

- With a broader range of moneyness the fit improves



Notebook_L3_CS_SimpleBlackScholesApproximation

Add your notes here

Scikit is Excellent but Limited for Deep Learning

- Scikit has been an amazingly powerful library
- However, it is not designed to do deep learning
- Deep learning is NN when we have more than 2 hidden layers
- We need a library that can perform fast back propagation
- We need a library with the flexibility to implement a lot of different neural network architectures
- We need to switch to Tensor Flow or PyTorch
- We have chosen TensorFlow – version 2.0

Tensor Flow 2.0

- TensorFlow is a low-level maths API like numpy but tailored for deep learning, and specifically for building multilayer NNs
- Designed by the Google Brain team
- If specialises in differential programming (needed to implement back propagation) and manipulating N-dim matrices (tensors)
- You define complex compute "graphs" of calculations and dependencies are converted to compiled C++ / **CUDA** code
- **CUDA** is an API layer for calling into GPUs – Graphical Processing Units are designed to do basic calculations in parallel – very fast
- TensorFlow1 was low-level and not so easy to learn!
- TensorFlow2 has changed the library in several important ways and has integrated a high-level interface called Keras

Keras

- Created by Francois Chollet, a google engineer, in 2015
- It used to sit on top of TensorFlow, Theano and CNTK
- Since 2019 Keras is **fully integrated** into TensorFlow 2.0
- It comes as part of the TensorFlow installation
- Supports a broad range of neural network architectures
- Allows use of distributed training on GPUs
- There are two flavours of Keras
    - Standalone Keras which you install and call separately
    - tf.keras which is the Keras API integrated into TensorFlow2
- We will be using tf.Keras which is part of TensorFlow2

Accessing Keras

- To install Keras install the TensorFlow library from your notebook as follows

```
! pip install tensorflow
```

- To check the version

```
import tensorflow
print(tensorflow.__version__)
```

- To access Keras

```
# example of tf.keras python idiom
import tensorflow as tf
# use keras API
model = tf.keras.Sequential()
```

Defining a Neural Network Model

- Keras provides several Neural Network model types
- **Sequential** is the basic feedforward neural network model
- A sequential model is a stack of layers where each layer has one input tensor and one output tensor
- To begin, we need to pull in the Sequential model type and the layer type which here is Dense

```
# example of a model defined with the sequential api
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
```

- Dense layers means that every node in a layer is connected to every node in the next layer

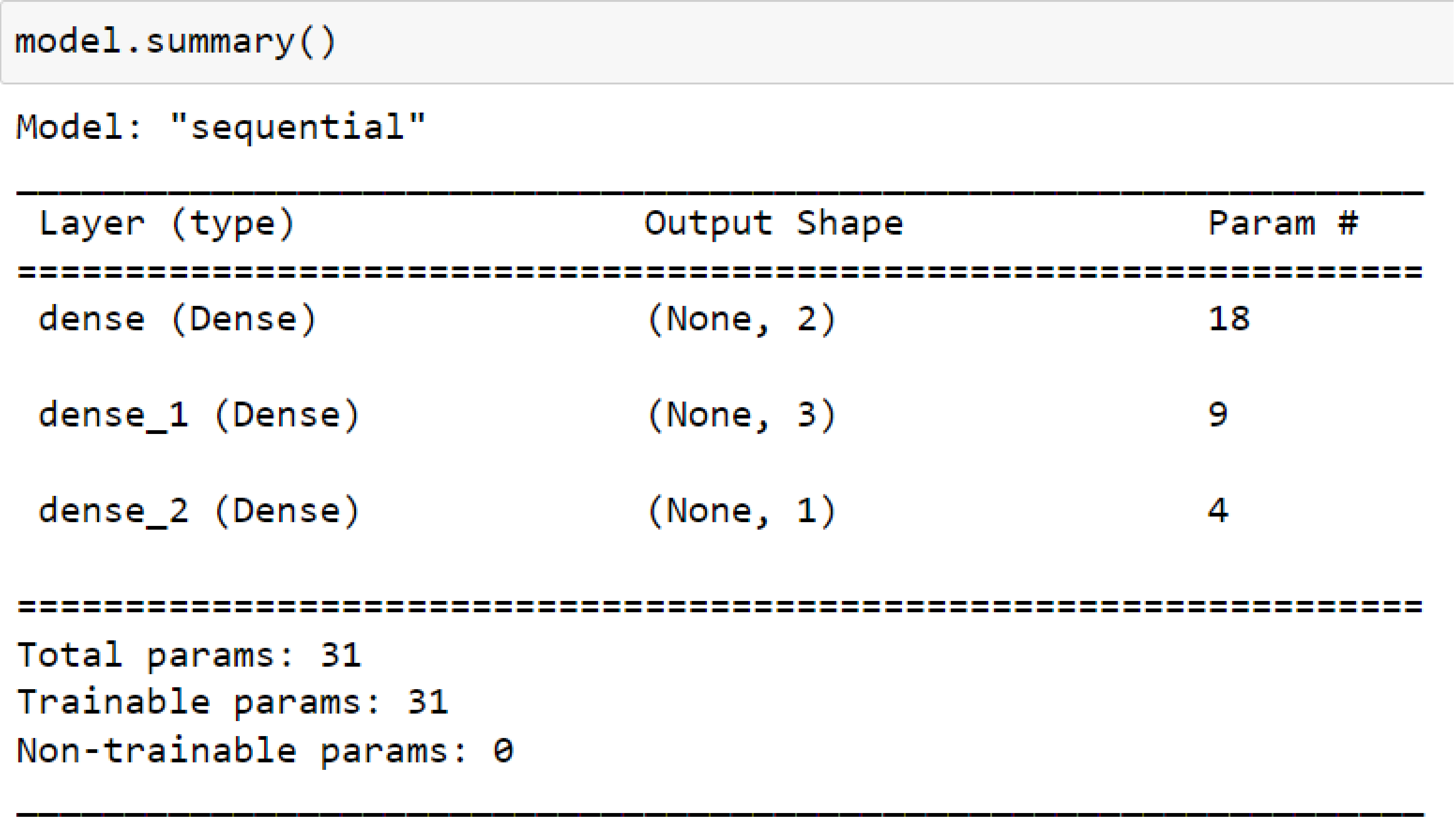
Creating a Sequential Model

- This is the simplest way to build a feedforward neural network

```
model = Sequential()
model.add(Dense(2, input_shape=(8,), activation="relu"))
model.add(Dense(3, activation="relu"))
model.add(Dense(1))
```

- Each layer is a set of neurons with defined connectivity (Dense) and a defined activation function (ReLu)
- This architecture defines 3 layers –a hidden layer with 2 neurons, a second hidden layer with 3 neurons and an output layer with 1 neuron – you can define the size of the input shape explicitly
- The input size should correspond to the size of your training data inputs – here we expect a vector of 2 numbers
- We also have a single output of one value

We can see the parameters in the Model Summary



```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 2)                 18

 dense_1 (Dense)             (None, 3)                 9

 dense_2 (Dense)             (None, 1)                 4

=================================================================
Total params: 31
Trainable params: 31
Non-trainable params: 0
_________________________________________________________________
```

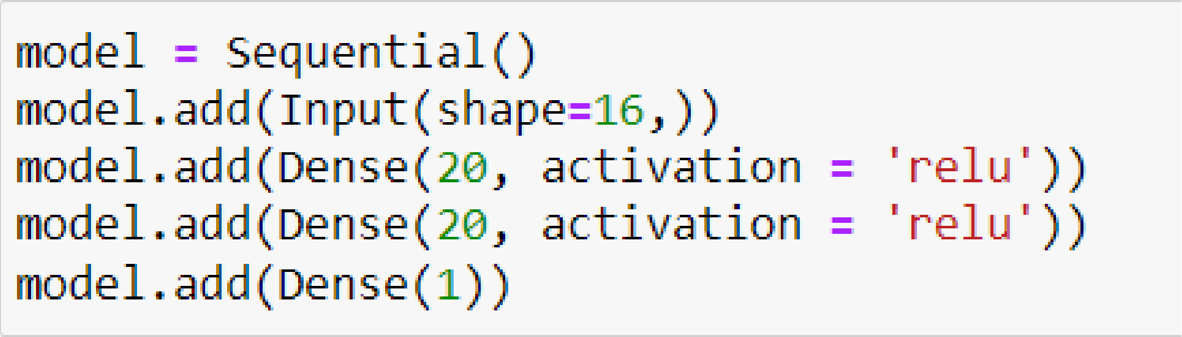
- Layer 1: 8 inputs plus 1 bias to two layers = (8+1) x 2 = 18 params
- Layer 2: 2->3 neurons to 1 output = (2+1) x 3 = 9 params
- Layer 3: 3 neurons to 1 output = (3+1) = 4 params

The Different Layer Types

- Flatten
  - Converts a multidimensional array to a 1D array
- Dense
  - Connects all the neurons in the previous layer to every neuron in the current layer
- Dropout
  - During training, the dropout layer randomly turns off neurons in the previous layer with a given probability
- Input Layer
  - Not required but recommended as you can just give the input tensor and it then knows the shape of the data
- There are others … we will encounter them later

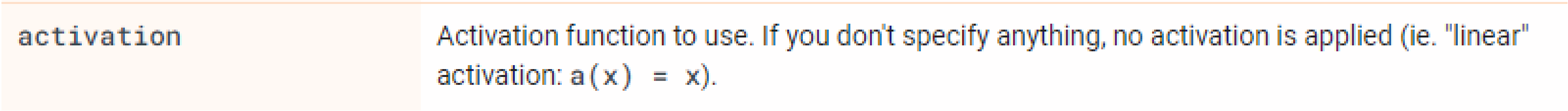
Creating a Deeper ANN

- An ANN with 2 fully connected layers of 20 neurons and 1 output



```
model = Sequential()
model.add(Input(shape=16,))
model.add(Dense(20, activation = 'relu'))
model.add(Dense(20, activation = 'relu'))
model.add(Dense(1))
```

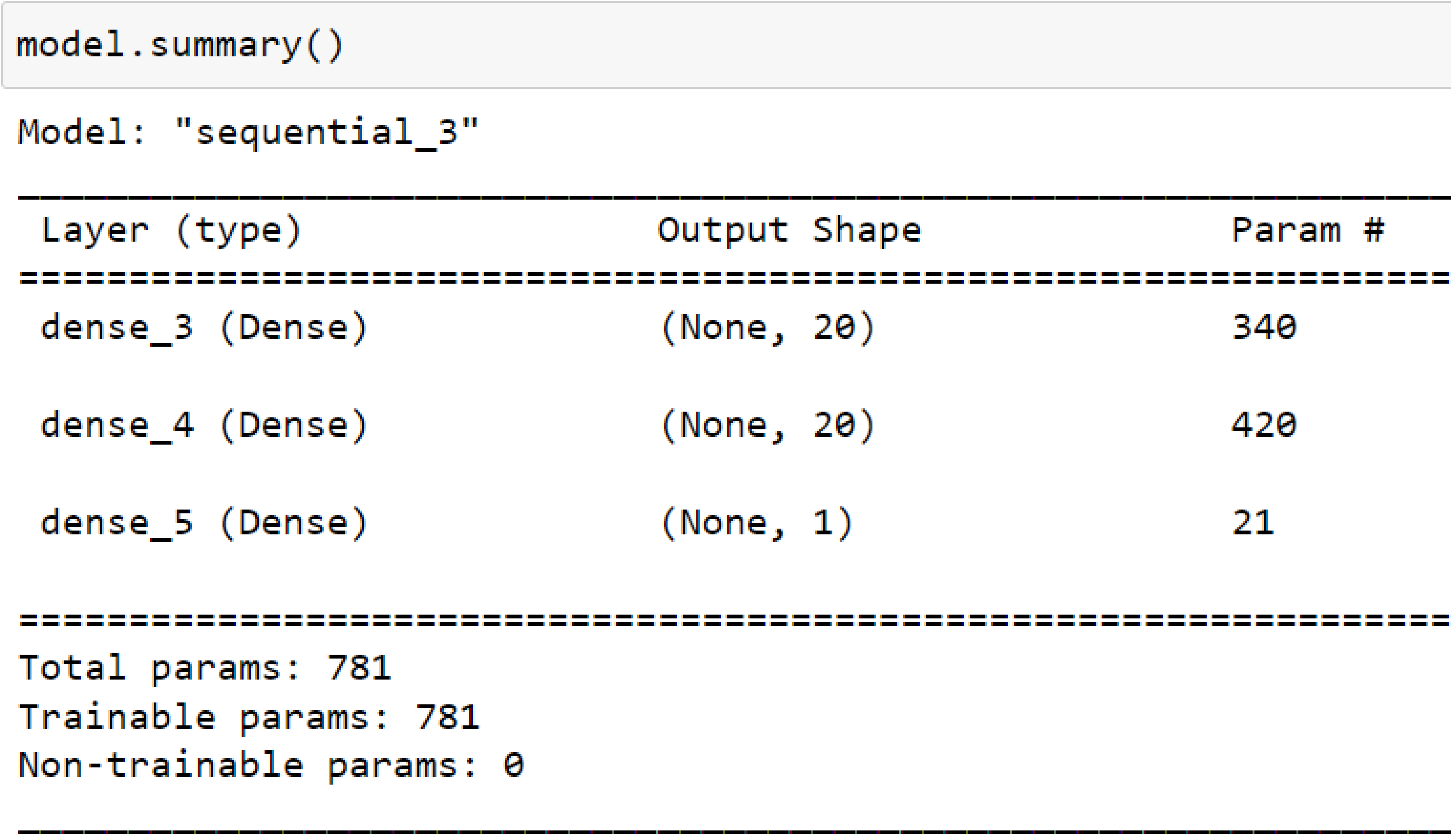
- What activation is used for the output layer ?
- www.tensorflow.org/api_docs/python/tf/keras/layers/Dense



`activation`	Activation function to use. If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).

- If you don't specify an activation, a linear function is assumed
- This ANN would be used for regression

Look at the Model Summary



```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 20)                340

 dense_4 (Dense)             (None, 20)                420

 dense_5 (Dense)             (None, 1)                 21

=================================================================
Total params: 781
Trainable params: 781
Non-trainable params: 0
_________________________________________________________________
```

- Consider input layer of 16 to 20 neurons
- That’s (16 weights + 1 bias) x 20 = 340 parameters
- Note – in GANs we will not allow some params to be trainable

Flattening the Inputs

- When inputs are not in vector format, we need to flatten them
- This is what we often do with image data
- We need to add in a Flatten layer to do this
- **Here we build the ANN by passing in a list of layers**
- You will see this used widely too – I use both but I prefer the add layer approach – fewer brackets to worry about

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, kernel_initializer="he_normal"),
    keras.layers.Dense(100, kernel_initializer="he_normal"),
    keras.layers.Dense(10, activation="softmax")
])
```

- What is this network for ?

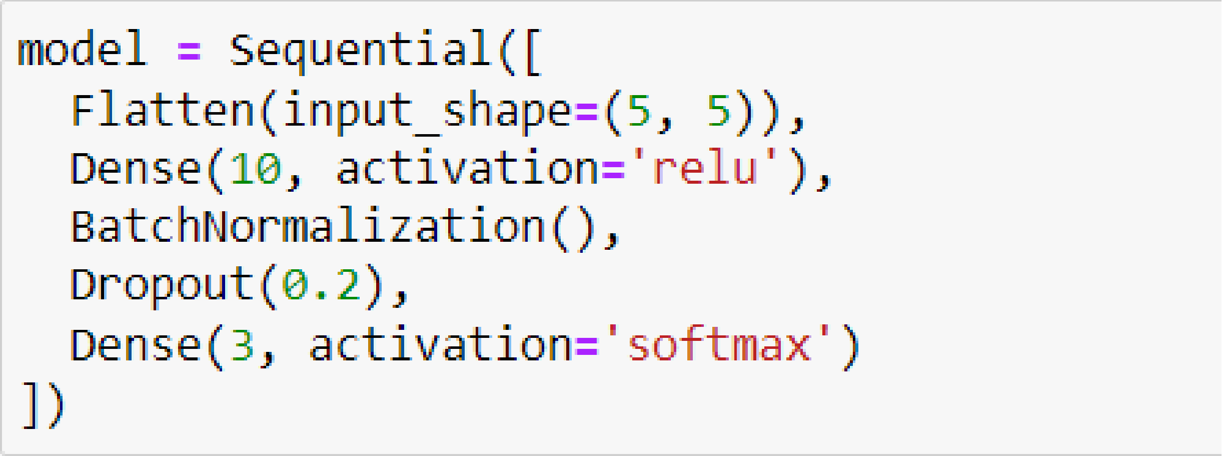
Passing In Data

- TensorFlow wants inputs of type float32, so we convert from float64 before we do the fit and predict on the test examples

```
X_train = tf.cast(X_train, tf.float32)
y_train = tf.cast(y_train, tf.float32)
mlp.fit(x=X_train, y=y_train, epochs = 250, verbose = 0);
y_pred_test = mlp.predict(X_test)
```

Batch Normalization as a Layer

- Batch normalisation accelerates training by fixing the mean and variances of layer inputs – making the layers less scale dependent
- We simply calculate the mean and variance of the inputs and then rescale them to some chosen mean and variance
- In TensorFlow it is implemented as a layer

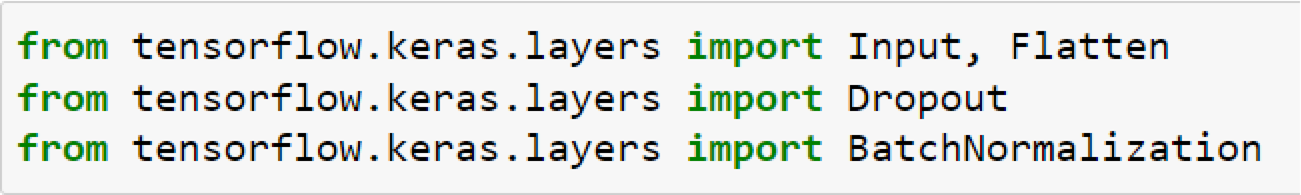


```
model = Sequential([
  Flatten(input_shape=(5, 5)),
  Dense(10, activation='relu'),
  BatchNormalization(),
  Dropout(0.2),
  Dense(3, activation='softmax')
])
```

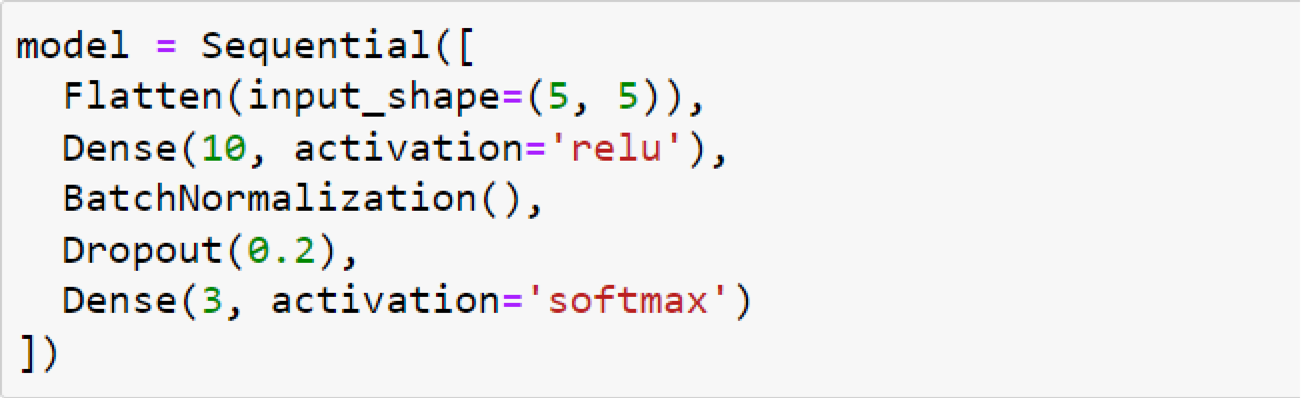
- Here we have Batch Normalisation between the previous and next layer

A Bigger Model

- We can add on more layer types including Dropout and Batch Normalization



```python
from tensorflow.keras.layers import Input, Flatten
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import BatchNormalization
```

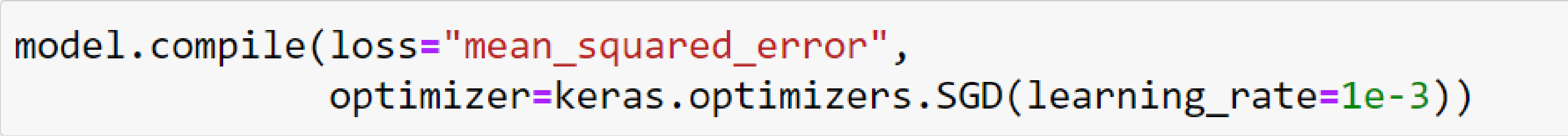


```python
model = Sequential([
  Flatten(input_shape=(5, 5)),
  Dense(10, activation='relu'),
  BatchNormalization(),
  Dropout(0.2),
  Dense(3, activation='softmax')
])
```

- Here we have 3 outputs and a Softmax
- This is a multiclass classification problem

Compiling the Model – Regression Model

- The last thing to do is to compile the model

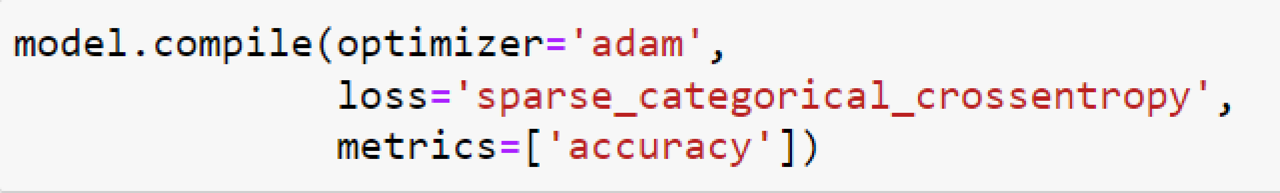


```
model.compile(loss="mean_squared_error",
              optimizer=keras.optimizers.SGD(learning_rate=1e-3))
```

- This tells the model
  - The loss function to minimise
  - The optimizer
  - The name of any metrics you want to report
- At this point TF knows all it needs to know about how the backprop algorithm will run
- To apply the model to a training set, you need to call the fit method

Compiling a Multi-Class Classification Network

- The last thing to do is to compile the model



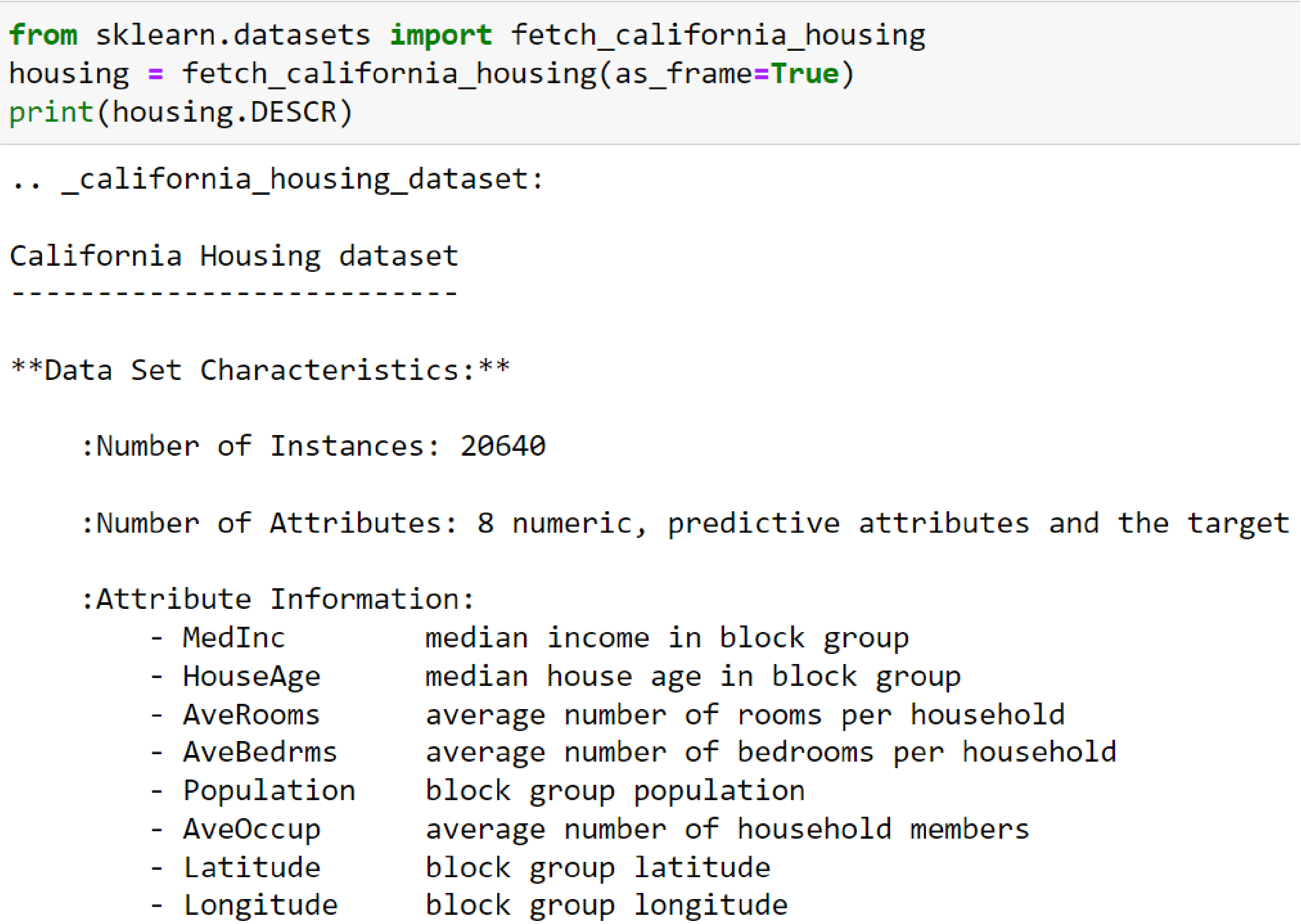
```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

- This tells the model
  - The loss function to minimise
  - The optimizer ADAM
  - The name of any metrics you want to report
- At this point TF knows all it needs to know about how the backprop algorithm will run
- To apply the model to a training set, you need to call the fit method

Case Study:
Tensor Flow Regression
California Housing

Simple Introduction to TensorFlow Regression

- The California Housing Price dataset is in the Scikit Learn library

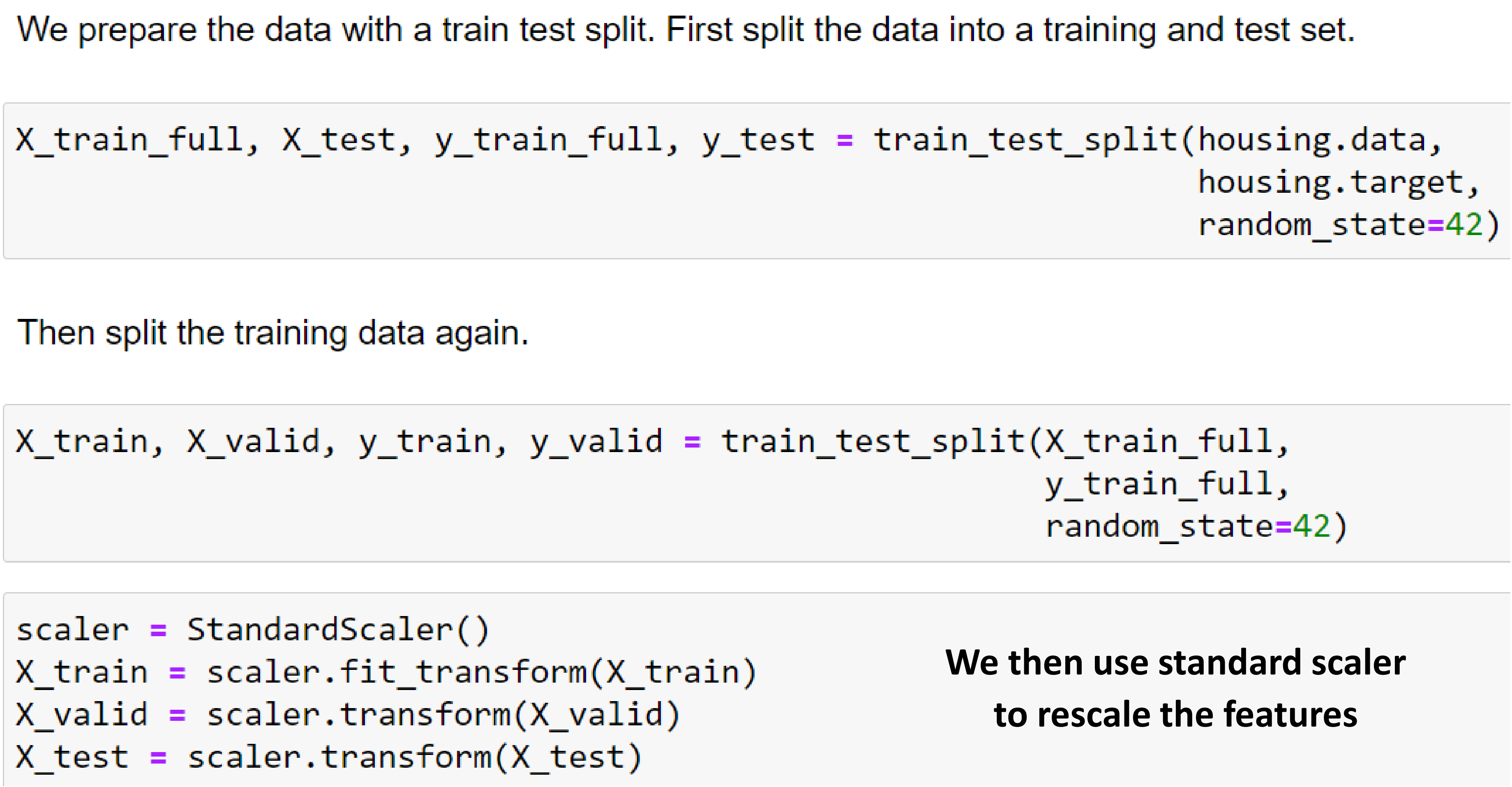


```python
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing(as_frame=True)
print(housing.DESCR)
```

```
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

    :Number of Instances: 20640

    :Number of Attributes: 8 numeric, predictive attributes and the target

    :Attribute Information:
        - MedInc        median income in block group
        - HouseAge      median house age in block group
        - AveRooms      average number of rooms per household
        - AveBedrms     average number of bedrooms per household
        - Population    block group population
        - AveOccup      average number of household members
        - Latitude      block group latitude
        - Longitude     block group longitude
```

Preparing the Dataset

- We split the data into a training and test set
- We then split the training set into a training and validation set



We prepare the data with a train test split. First split the data into a training and test set.

```
X_train_full, X_test, y_train_full, y_test = train_test_split(housing.data,
                                                              housing.target,
                                                              random_state=42)
```

Then split the training data again.

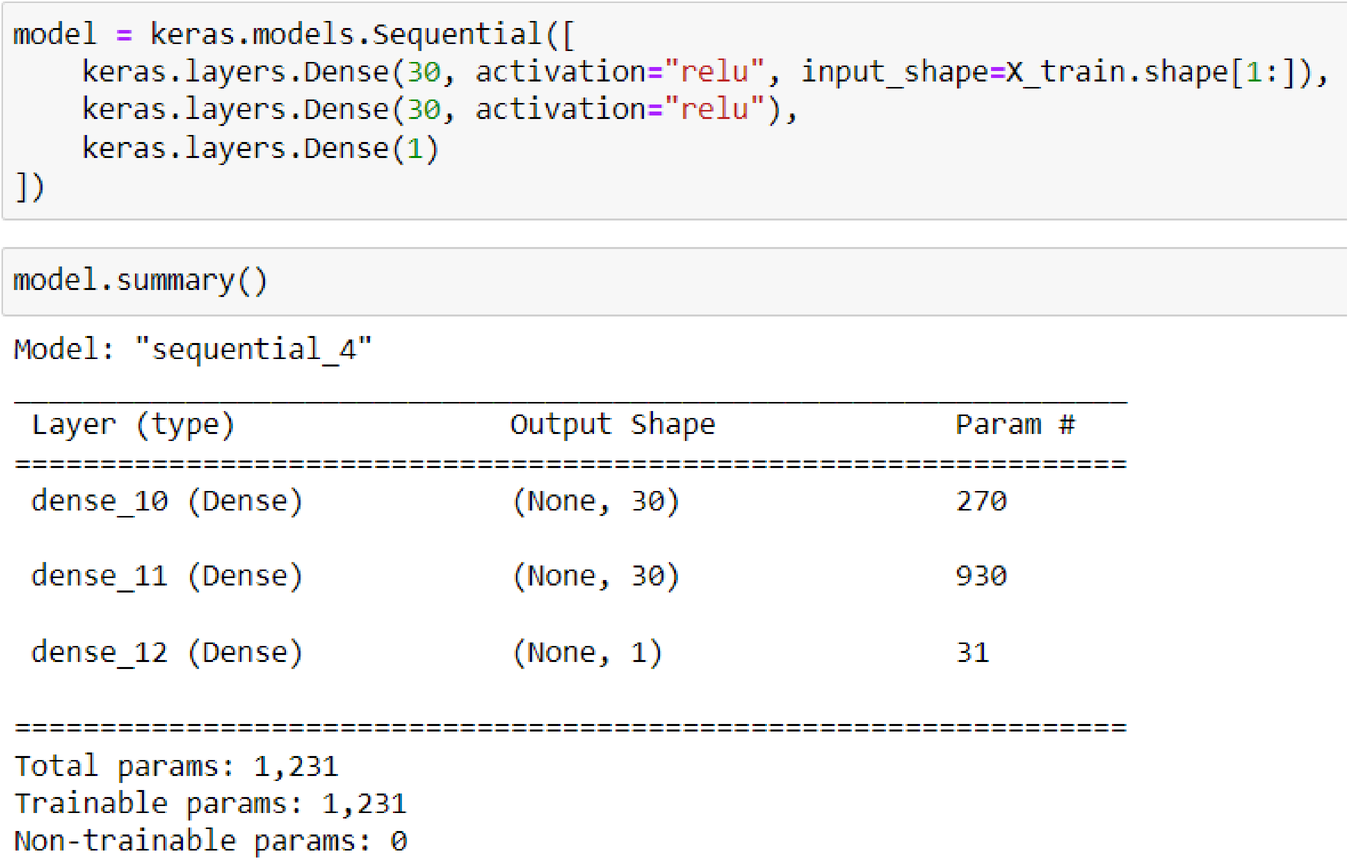
```
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full,
                                                      y_train_full,
                                                      random_state=42)
```

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

We then use standard scaler to rescale the features

Defining the NN Architecture

- The number of input cells is the number of features = 8
- There is then a Dense layer of 30 neurons and this is repeated
- The final layer has 1 neuron with no (linear) activation



```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(30, activation="relu"),
    keras.layers.Dense(1)
])
```

```
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_10 (Dense)            (None, 30)                270

 dense_11 (Dense)            (None, 30)                930

 dense_12 (Dense)            (None, 1)                 31

=================================================================
Total params: 1,231
Trainable params: 1,231
Non-trainable params: 0
```

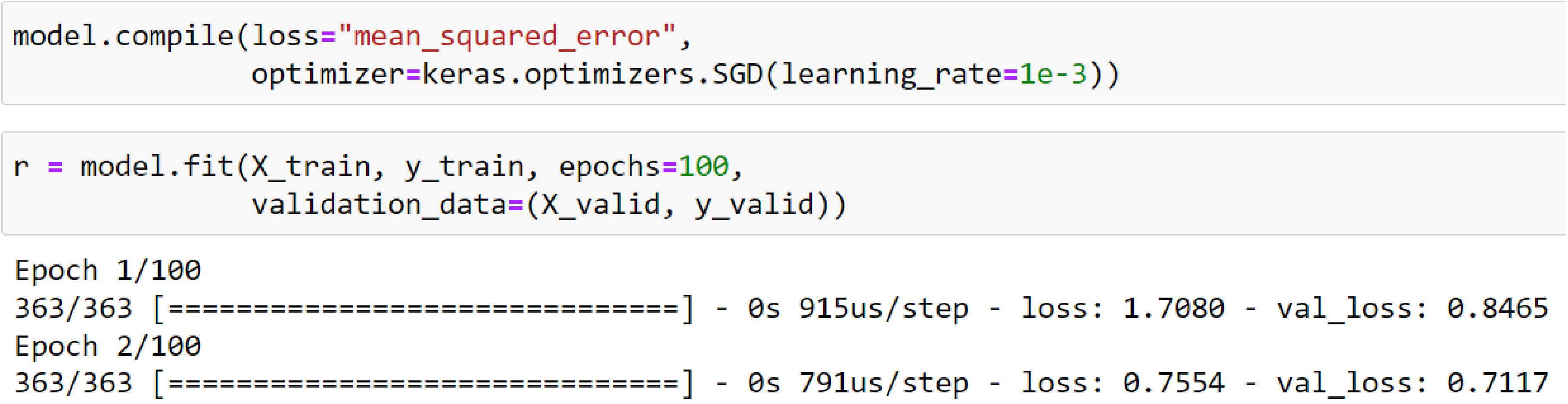
270 = (8 wts + 1 bias) x 30

930 = (30 wts + 1 bias) x 30

31 = (30 wts + 1 bias)

Training the Model

- The loss is the mean squared error
- We use stochastic gradient descent
- Fitting is over 100 epochs (cycles of the training set)
- Each epoch has 20640 samples !



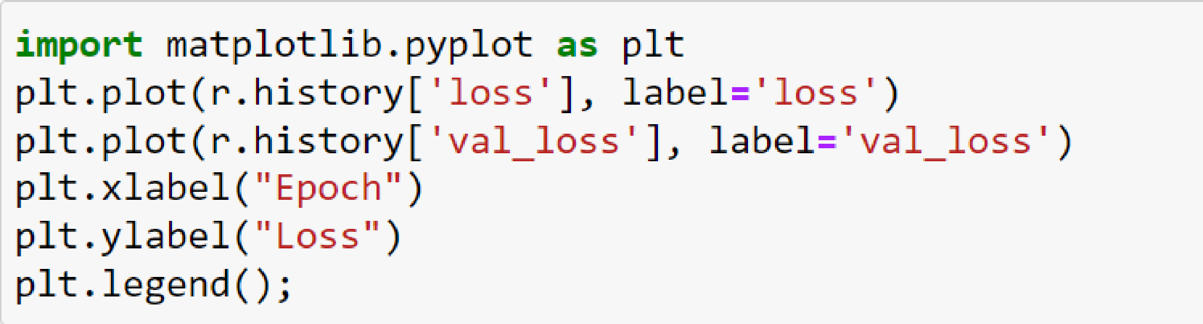
```
model.compile(loss="mean_squared_error",
              optimizer=keras.optimizers.SGD(learning_rate=1e-3))
```

```
r = model.fit(X_train, y_train, epochs=100,
              validation_data=(X_valid, y_valid))
```

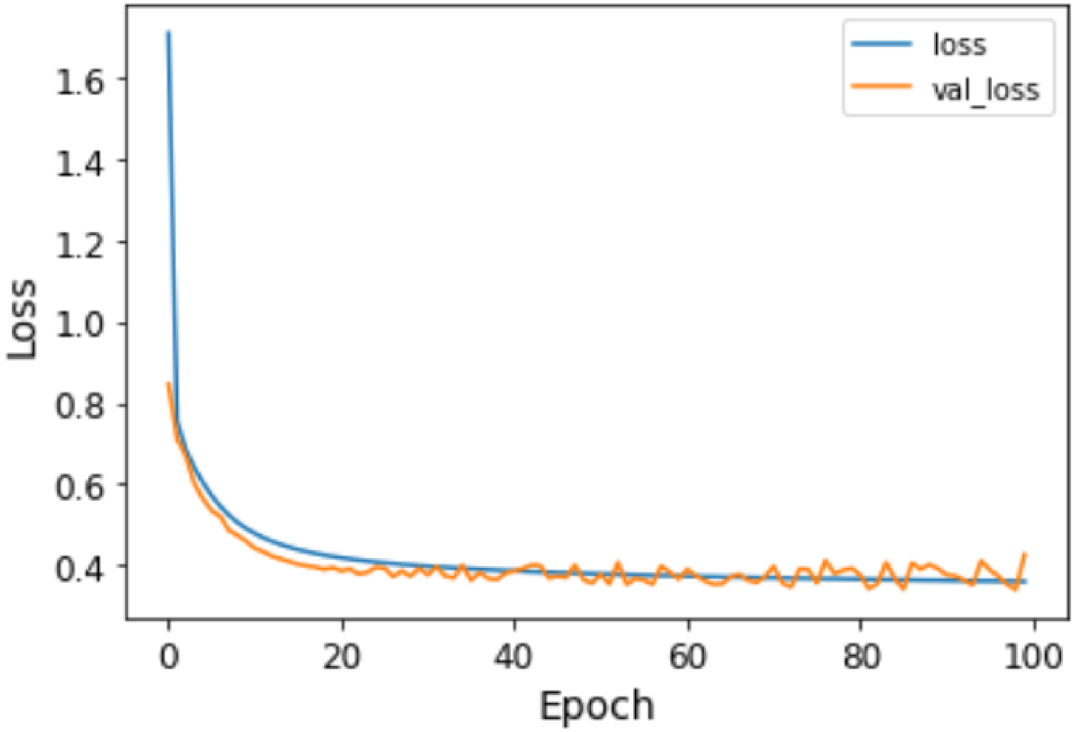
```
Epoch 1/100
363/363 [==============================] - 0s 915us/step - loss: 1.7080 - val_loss: 0.8465
Epoch 2/100
363/363 [==============================] - 0s 791us/step - loss: 0.7554 - val_loss: 0.7117
```

Linear Regression Results

- The Loss falls and seems to plateau after about 50 epochs

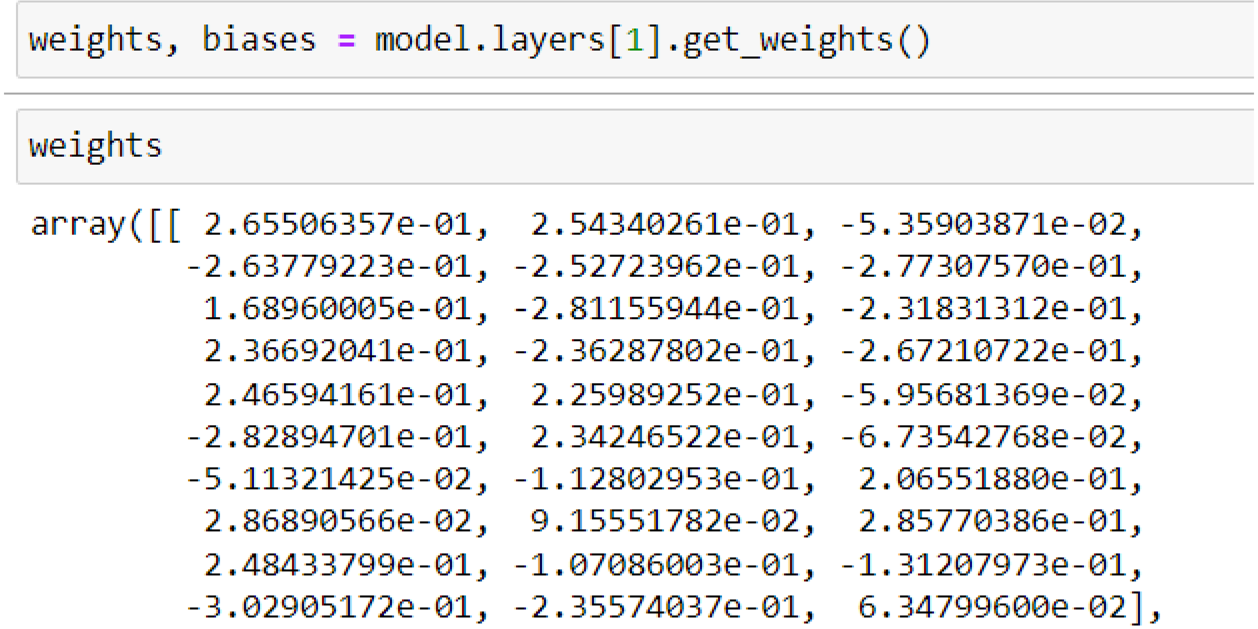


```
import matplotlib.pyplot as plt
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend();
```



Inspecting the Weights and Biases

- If you want to, you can see the weights and the biases



```
weights, biases = model.layers[1].get_weights()
```

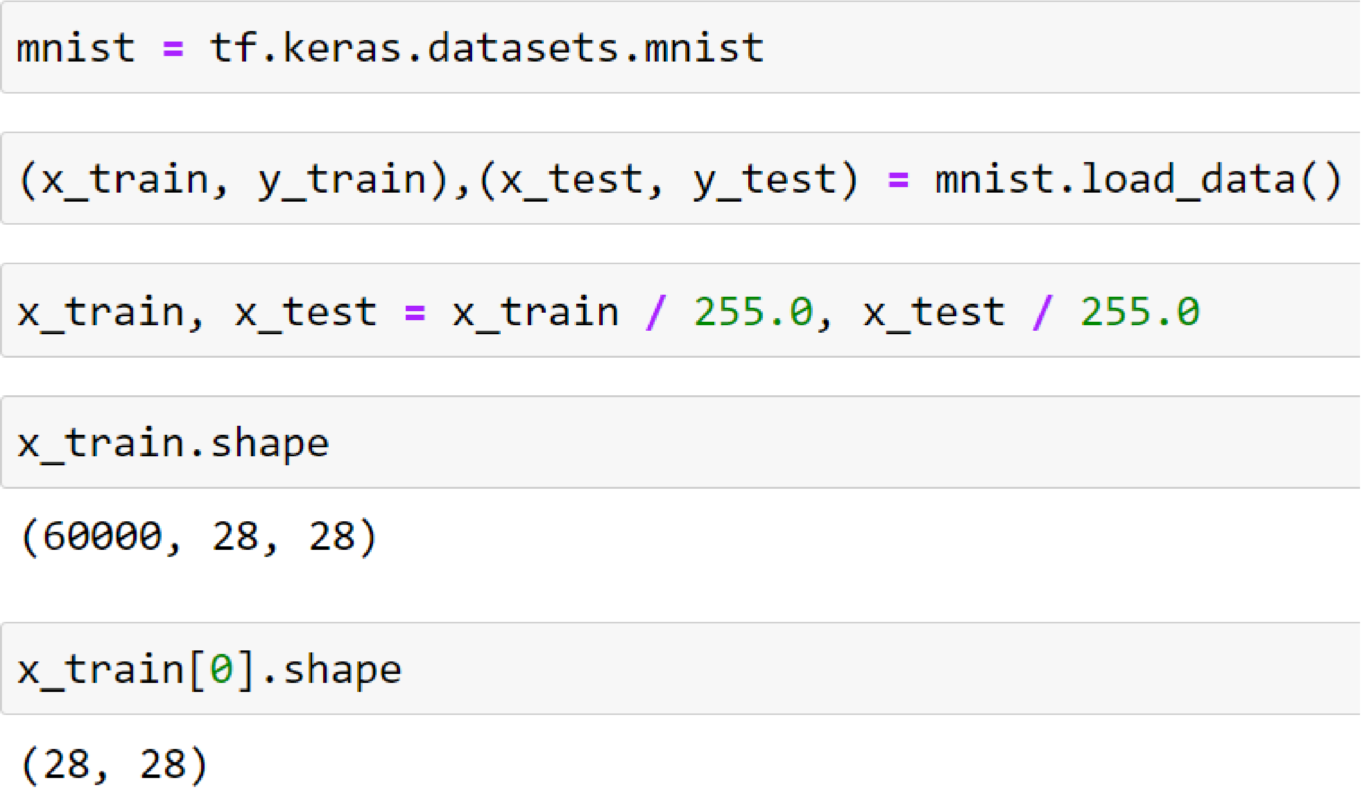
```
weights
```

```
array([[ 2.65506357e-01,  2.54340261e-01, -5.35903871e-02,
        -2.63779223e-01, -2.52723962e-01, -2.77307570e-01,
         1.68960005e-01, -2.81155944e-01, -2.31831312e-01,
         2.36692041e-01, -2.36287802e-01, -2.67210722e-01,
         2.46594161e-01,  2.25989252e-01, -5.95681369e-02,
        -2.82894701e-01,  2.34246522e-01, -6.73542768e-02,
        -5.11321425e-02, -1.12802953e-01,  2.06551880e-01,
         2.86890566e-02,  9.15551782e-02,  2.85770386e-01,
         2.48433799e-01, -1.07086003e-01, -1.31207973e-01,
        -3.02905172e-01, -2.35574037e-01,  6.34799600e-02],
```

Case Study:
Tensor Flow Classification
MNIST Digits

Simple Introduction to TensorFlow Classification

- We do the MNIST digit recognition - load and standardise data



```
mnist = tf.keras.datasets.mnist
```

```
(x_train, y_train),(x_test, y_test) = mnist.load_data()
```

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

```
x_train.shape
```

```
(60000, 28, 28)
```

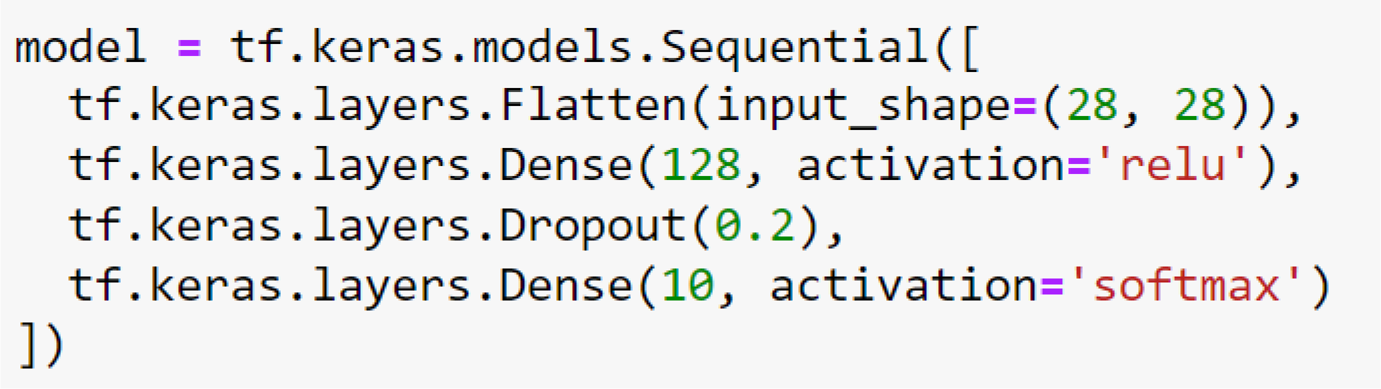
```
x_train[0].shape
```

```
(28, 28)
```

- We have 60,000 training images, each 28 x 28 pixels
- We also have 10,000 testing images, each 28 x 28 pixels

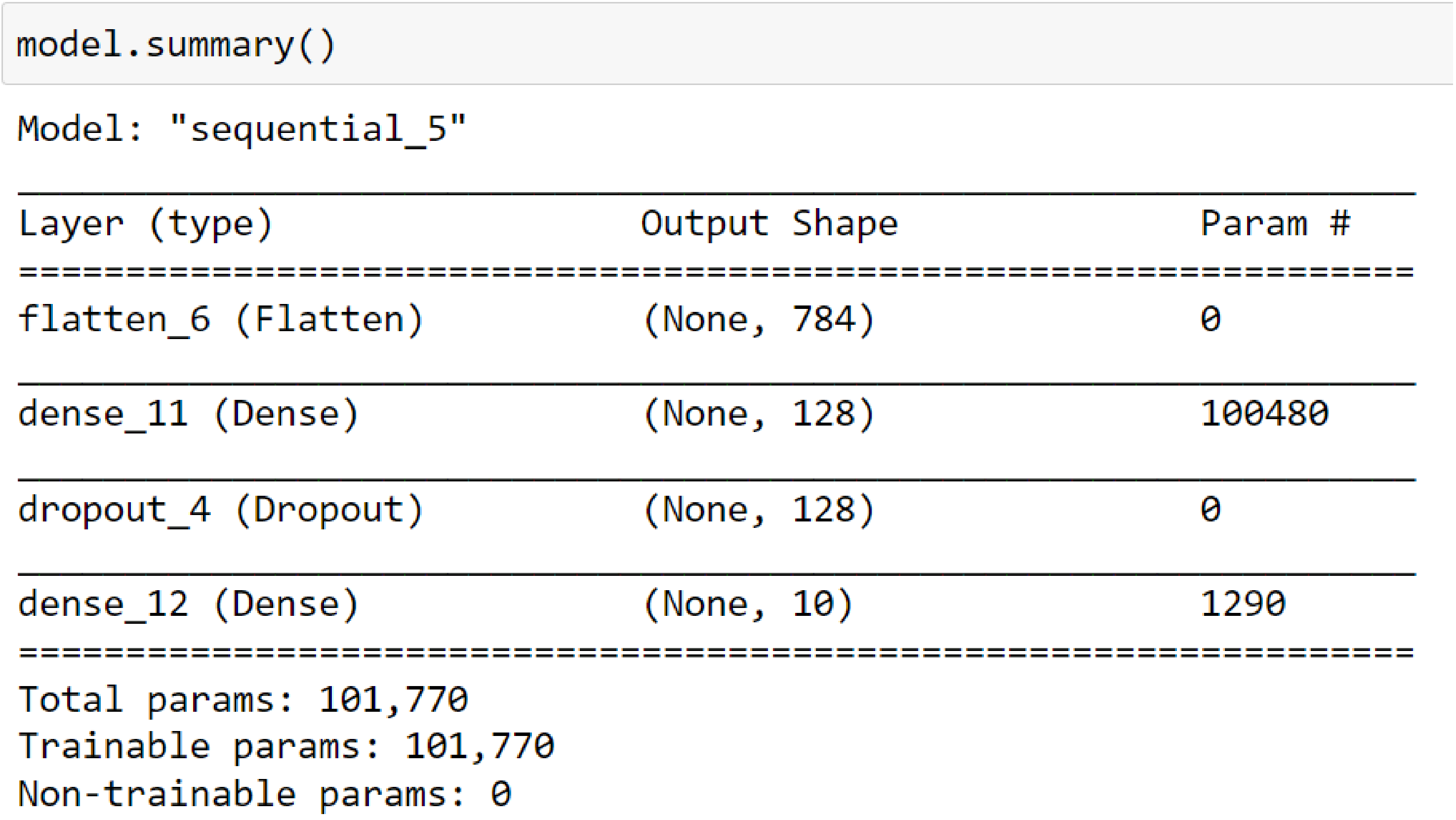
Building the Neural Network in TensorFlow

- We want a network with the following attributes:
  - It expects an array of 28 x 28 inputs.
  - Flatten layer to convert the input to a 1D vector
  - Dense layer with 128 hidden units with RELU activation
  - Dropout layer with dropout probability of 20%
  - Dense layer with 10 neurons (one for each digit) in the output layer with a SoftMax activation



```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Model Summary gives us the Trainable Parameters



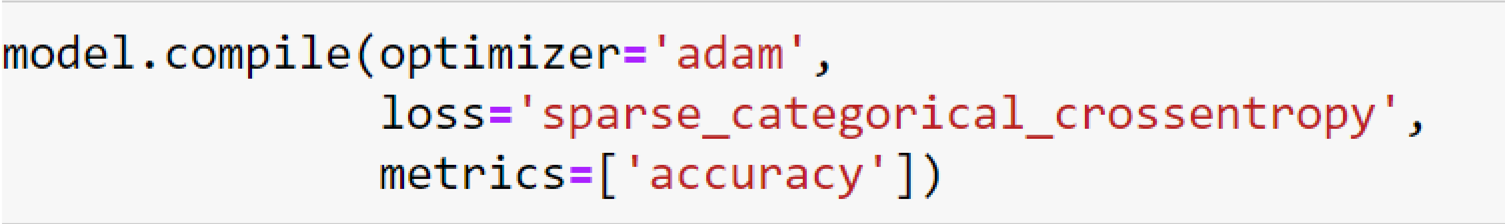
```
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten_6 (Flatten)          (None, 784)               0         
_________________________________________________________________
dense_11 (Dense)             (None, 128)               100480    
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_12 (Dense)             (None, 10)                1290      
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
```

- 100480 = 784 x 128 (weights) + 128 (bias terms)
- 1290 = 128 x 10 + 10 (bias terms)
- Dropout layers don’t have parameters

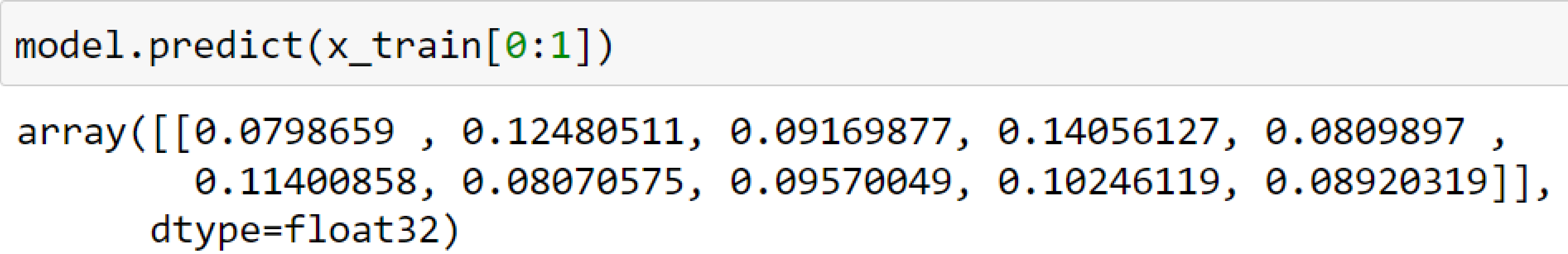
Compiling the Model

- We set the optimizer to Adam
- We set the loss to sparse categorical cross entropy
- As it's a classification problem we track the entropy



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

- Let's look at the output (before the model is trained)



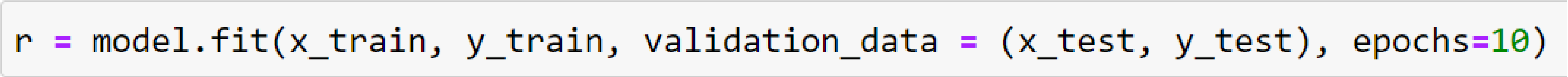
```
model.predict(x_train[0:1])
```

```
array([[0.0798659 , 0.12480511, 0.09169877, 0.14056127, 0.0809897 ,
        0.11400858, 0.08070575, 0.09570049, 0.10246119, 0.08920319]],
      dtype=float32)
```

- We have 10 "probabilities" in the output layer

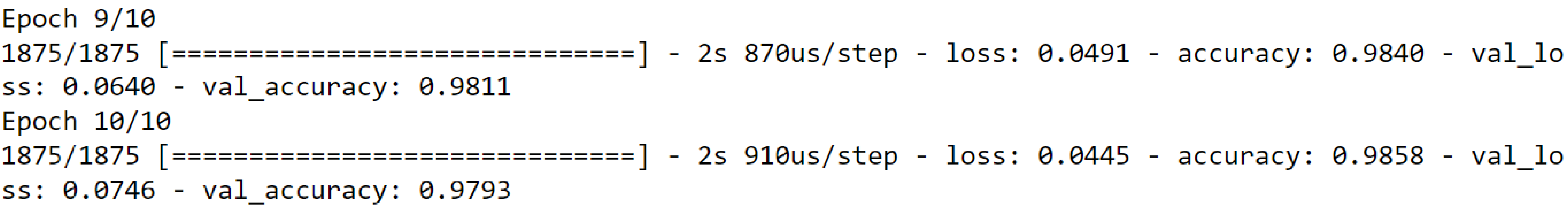
Training the Model

- We start the training using both training and test data
- We run for 10 epochs (10 times through the training dataset)



```
r = model.fit(x_train, y_train, validation_data = (x_test, y_test), epochs=10)
```

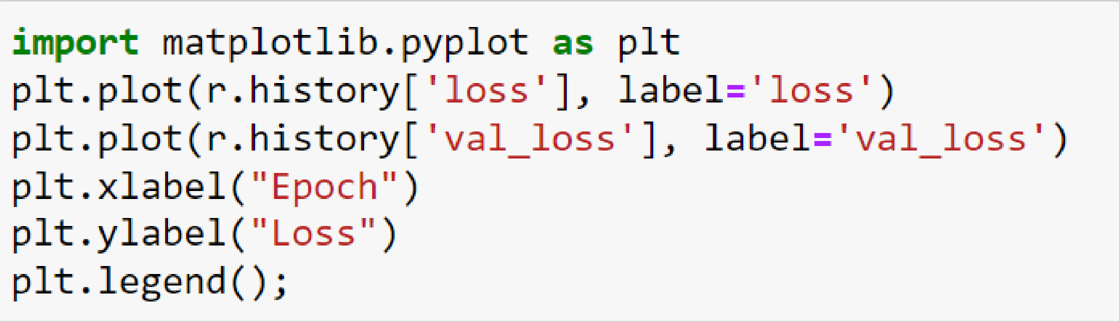
- The training goes well – we see the loss fall and the accuracy approach 98%



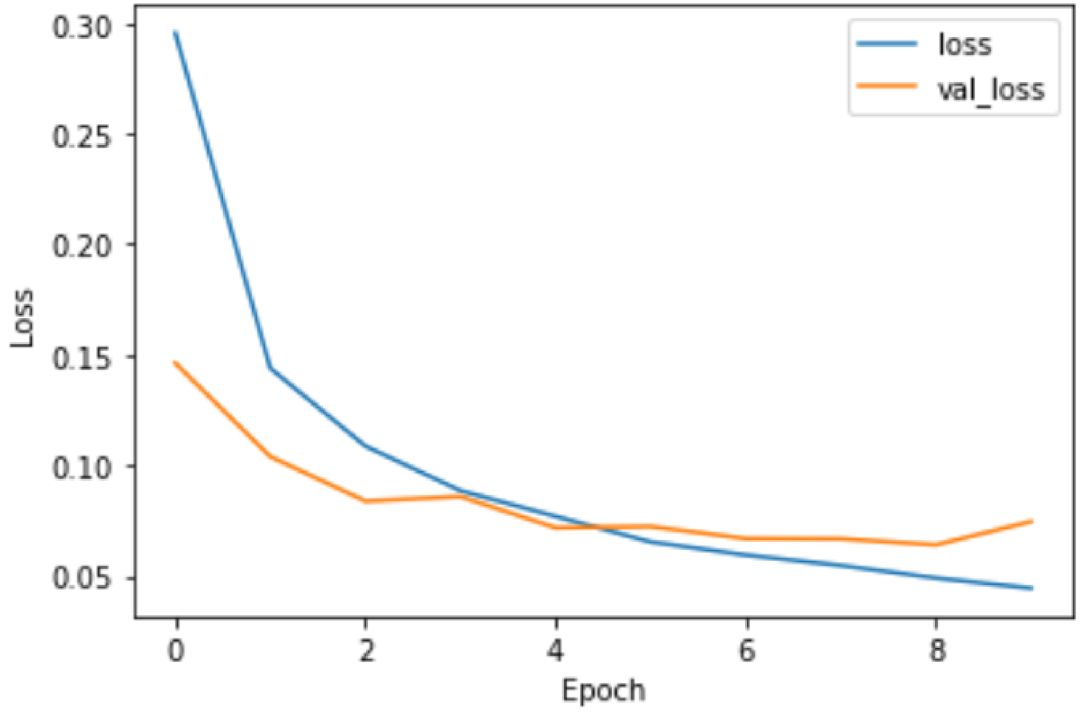
```
Epoch 9/10
1875/1875 [==============================] - 2s 870us/step - loss: 0.0491 - accuracy: 0.9840 - val_lo
ss: 0.0640 - val_accuracy: 0.9811
Epoch 10/10
1875/1875 [==============================] - 2s 910us/step - loss: 0.0445 - accuracy: 0.9858 - val_lo
ss: 0.0746 - val_accuracy: 0.9793
```

- We can plot some results

Training Loss Evolution

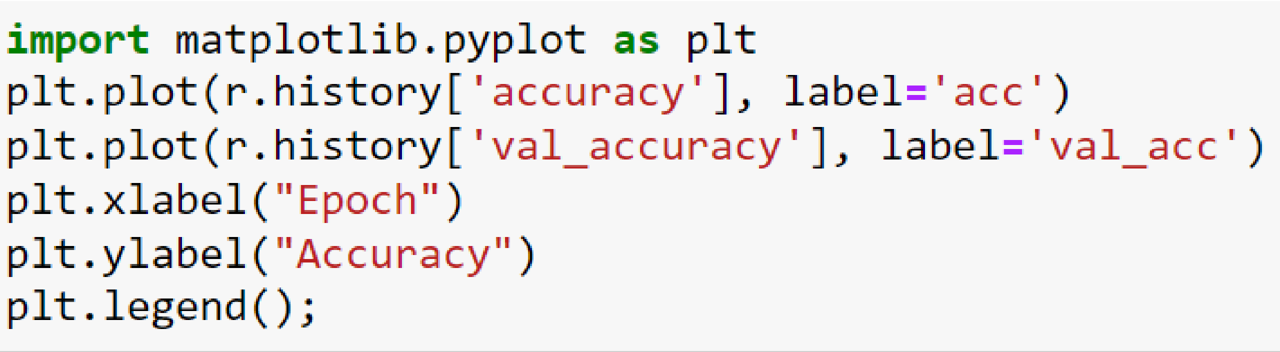


```
import matplotlib.pyplot as plt
plt.plot(r.history['loss'], label='loss')
plt.plot(r.history['val_loss'], label='val_loss')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend();
```

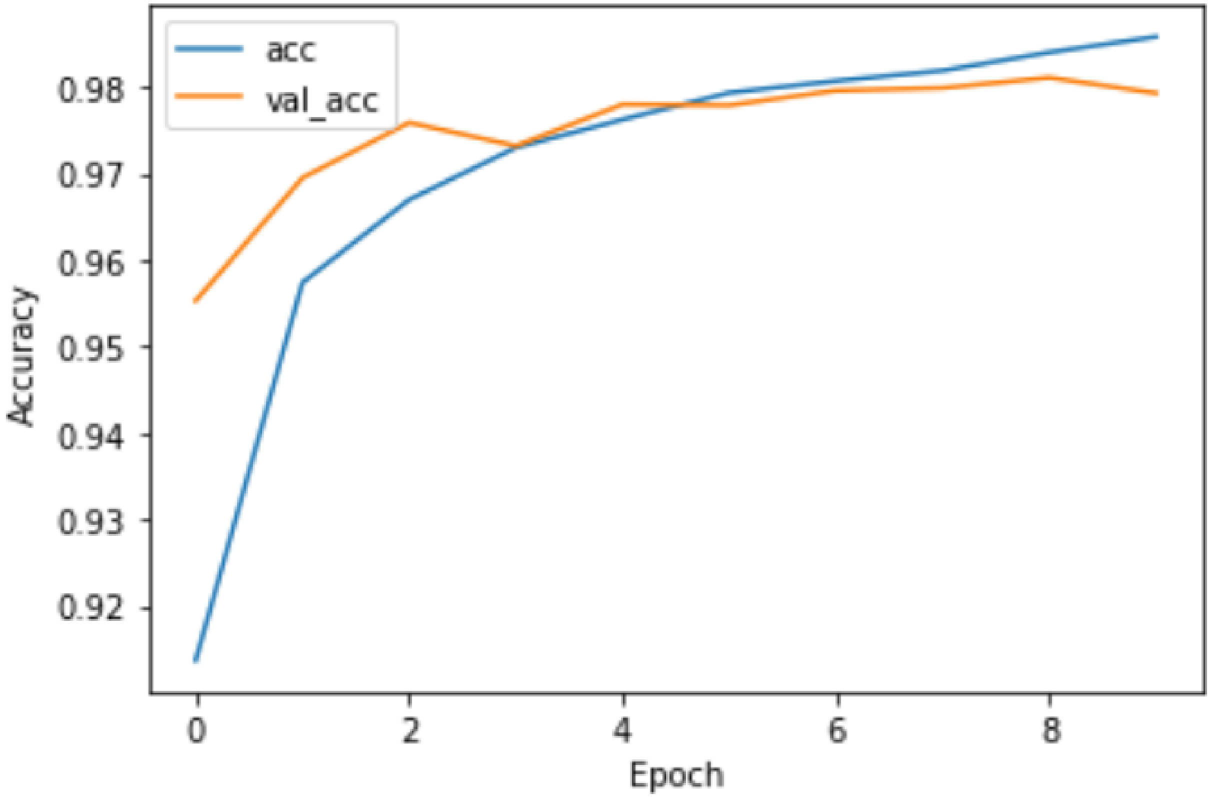


- We see the validation loss increase after 8 epochs - overtraining

Same for Accuracy



```
import matplotlib.pyplot as plt
plt.plot(r.history['accuracy'], label='acc')
plt.plot(r.history['val_accuracy'], label='val_acc')
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend();
```



Exercise

- Convert the Black Scholes function approximator in Scikit learn to TensorFlow
- Is it faster ?