DR ADRIAN BEVAN

PRACTICAL MACHINE LEARNING

MORE TENSORFLOW – CODING

LECTURE PLAN

- Introduction
- Model building
- Model optimisation
  - Optimisers
  - Feeding data
  - Regression
  - Classification
- Summary
- Suggested Reading

QMUL Summer School: https://www.qmul.ac.uk/summer-school/
Practical Machine Learning QMplus Page: https://qmplus.qmul.ac.uk/course/view.php?id=10006

INTRODUCTION

- The aim of these slides is to bridge the gap between the use of the tensor flow api and the level of understanding required to build and train a model.
- We've already seen some of the pieces of the problem via the linear regression example.
- At the end of this session you will be prepared for the next example: Function approximation using neural networks (see the next deck of slides)

INTRODUCTION

- We have seen the use of Python language constructs, plotting and NumPy arrays:
  - Python will be used as a framework to build and train models.
  - The plotting will be useful to visualise the evolution of the model optimisation process and the accuracy of models.
  - We will convert tensors of our data into NumPy arrays to feed data into our neural networks.
- TensorFlow ops will be used to build and evaluate our models.
  - Mainly matrix multiplications, but other operations will be useful for computing loss functions (for example).
- Optimisers and other functions will be used to train our models.

MODEL BUILDING

- `Example_FunctionApproximator.py`

- This example will take a single valued input and build a model to approximate the function that translates the input into a single valued output.
  - From this description we can see that we are constrained by the following:
    - 1 input feature
    - 1 output
    - This is a regression problem (continuous valued output required, rather than a classification decision).
- The form of the model is not defined, and the data scientist working on this problem can be creative.

MODEL BUILDING

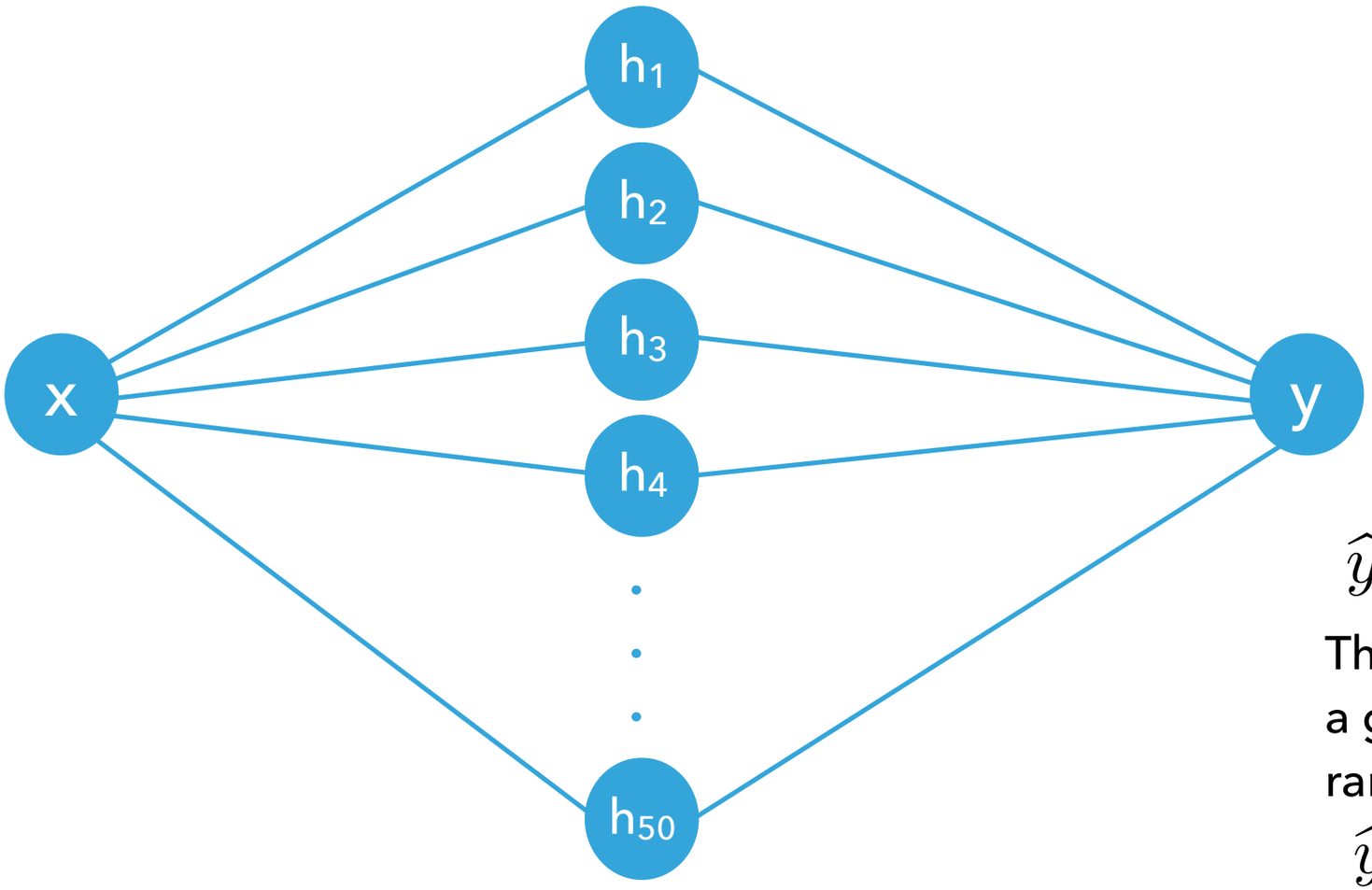
- `Example_FunctionApproximator.py`

- This example will take a single valued input and build a model to approximate the function that translates the input into a single valued output.
  - From this description we can see that we are constrained by the following:
    - 1 input feature
    - 1 output
    - This is a regression problem (continuous valued output required, rather than a classification decision).
- The form of the model is not defined, and the data scientist working on this problem can be creative.

MODEL BUILDING

- `Example_FunctionApproximator.py`

- We can now choose the form of our model to fit these constraints:
  - This example implements a single layer perceptron.



Function to approximate is

$$f = f(x) = x^2$$

Function used to approximate this is

$$\widehat{y} = v^T h, \text{ where } h = w^T x$$

The optimisation target is to obtain a good approximation for some range of x; i.e.

$$\widehat{y} \approx y \text{ for } x \in [x_{min}, x_{max}]$$

MODEL BUILDING

- `Example_FunctionApproximator.py`

- placeholders are used for the input values and true values (also called ground truth). These variables have the names `x_` and `y_`, respectively.

```
# tf Graph input:
#  x_: is the tensor for the input data (the placeholder entry None is used for that;
#     and the number of features input (n_input = 1).
#
#  y_: is the tensor for the output value of the function that is being approximated by
#     the MLP.
#
x_ = tf.placeholder(tf.float32, [None, n_input], name="x_")
y_ = tf.placeholder(tf.float32, [None, n_classes], name="y_")
```

MODEL BUILDING

- `Example_FunctionApproximator.py`

- The hidden layer takes an input (`x_`) and multiplies that by the weight tensor (`w_layer_1`) and adds a bias (`bias_layer_1`).
- A relu activation function is used, taking the argument

$$\arg = w^T x + b$$

```
# We construct layer 1 from a weight set, a bias set and the activiation function used
# to process the impulse set of features for a given example in order to produce a
# predictive output for that example.
#
#  w_layer_1:    the weights for layer 1.  The first index is the input feature (pixel)
#                and the second index is the node index for the perceptron in the first
#                layer.
#  bias_layer_1: the biases for layer 1.  There is a single bias for each node in the
#                layer.
#  layer_1:      the activation functions for layer 1
#
print("Creating a hidden layer with ", n_hidden_1, " nodes")
w_layer_1    = tf.Variable(tf.random_normal([n_input, n_hidden_1]))
bias_layer_1 = tf.Variable(tf.random_normal([n_hidden_1]))
layer_1      = tf.nn.relu(tf.add(tf.matmul(x_,w_layer_1),bias_layer_1))
```

MODEL BUILDING

- `Example_FunctionApproximator.py`

- The final step is to compute the model output (ŷ)
- Similarly to the hidden layer, take the outputs of the hidden layer nodes, and combine those with weights (`output`) and bias (`bias_output`) to compute ŷ = `output_layer`.

```
# Similarly we now construct the output of the network, where the output layer
# combines the information down into a space of evidences for the possible
# classes in the problem (n_classes=1 for this regression problem).
print("Creating the output layer ", n_classes, " output values")
output       = tf.Variable(tf.random_normal([n_hidden_1, n_classes]))
bias_output  = tf.Variable(tf.random_normal([n_classes]))
# define operation for computing the regression output - this is our model prediction
probabilities = tf.matmul(layer_1, output) + bias_output
```

- In this case the matrix multiplication itself is used (with a trivial activation function) to compute the model output.

MODEL OPTIMISATION: OPTIMISERS ▸ `Example_FunctionApproximator.py`

- Optimisation (as we will see later) requires a few parameters to be defined:

```
learning_rate    = 0.01
training_epochs  = 100
```

  - `learning_rate`: This is a parameter that tunes the step size taken in the optimisation process.
    - Small values of the learning rate will allow the optimisation algorithm to converge on a good approximation to the optimal solution - but this may take a long time.
    - Large values of this parameter can allow for faster convergence to the region where the optimal solution lies; but there is a limit to how close one can converge to that solution.
    - Very large values of this parameter can result in failure to converge.
    - We will discuss a hybrid solution that has an adaptive learning rate later in the course (the ADAM optimiser)
  - `training_epochs`: This is simply the number of iterations in the optimisation process that is used. A larger number of iterations will be required for smaller learning rates.

MODEL OPTIMISATION: OPTIMISERS

`Example_FunctionApproximator.py`

- We need a figure of merit to optimise.
  - For the linear regression example encountered earlier we use the X^2 as a figure of merit.
  - In Machine Learning we call quantities like this the loss or cost function. It is this that we need to minimise when we optimise the model parameters.
- Just as with the linear regression example, we take the model prediction and compare that against some target data to determine how good the prediction is.
  - Unlike the linear regression example we don't have to worry about the uncertainty on the target data example because we use known training examples (this is supervised learning).

MODEL OPTIMISATION: OPTIMISERS

`Example_FunctionApproximator.py`

- L2(-norm) loss function is given by:

$$L_2 = \sum_{i=1}^{N} (t_i - y_i)^2$$

  - Here the sum is over training examples, of which there are N in total.
  - t_i is the true target value of the i^{th} training example.
  - y_i is the model output prediction of for the i^{th} training example.

  - Note: the ordering of the t_i and y_i in the parentheses is irrelevant as the difference is squared in this loss function. Sign conventions can differ for loss functions in general.

This is just like the X^2 function, but without the normalisation by error on each data point.

MODEL OPTIMISATION: OPTIMISERS

`Example_FunctionApproximator.py`

- Supervised learning:
  - The use of training examples of known types allows us to compute the L2 loss function on an example by example basis.

  - This procedure is known as supervised learning, as the algorithm is trained to identify particular patterns from known training examples.

- A practical hint for a classification problem is to present different types of training example to the optimisation algorithm as this generally leads to a faster convergence of the training. For the example given this is not an issue as we are using the model to approximate the function $y = x^2$.

MODEL OPTIMISATION: OPTIMISERS

`Example_FunctionApproximator.py`

- We can implement the L2 loss function in several different ways in TensorFlow.
  - The least squares example uses:

```
loss = tf.reduce_sum((y - y_) * (y - y_))
```

- The function approximator example uses:

```
loss = tf.nn.l2_loss(y_ - probabilities)
```

True label for the example data

Model prediction

MODEL OPTIMISATION: OPTIMISERS

`Example_FunctionApproximator.py`

- With the loss defined, we can move on to consider the optimiser choice.

- We will discuss the ADAM optimiser in a few sessions time, but for now we just take it on faith that the algorithm works and will select a good set of model parameters for us.

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)
```

- See https://www.tensorflow.org/api_docs/python/tf/train for different optimisers in the tf.train class.

MODEL OPTIMISATION: FEEDING DATA

`Example_FunctionApproximator.py`

- The training is performed by evaluating the optimiser, using a feed_dict to provide input data (`traindata`) and corresponding true values of the function (`target_value`).

```
sess.run(optimizer, feed_dict={x_: traindata, y_: target_value})
the_loss = sess.run(loss, feed_dict={x_: traindata, y_: target_value})
```

- The loss function for that optimisation step can be computed similarly.
- For large data samples training can be done by feeding sub-sets of the data to the optimiser for each epoch.

MODEL OPTIMISATION: REGRESSION

`Example_FunctionApproximator.py`

- The model output is the value that we need to evaluate for a regression problem.
- For this example we can evaluate the model output while feeding data to the model.
- Here the model output is assigned to a variable called `probabilities`.
- The model is evaluated by feeding an x value to it:

```
pred = probabilities.eval(feed_dict={x: [[thisx]]}, session=sess)
```

MODEL OPTIMISATION: CLASSIFICATION

- Classification can be performed in one of several ways.
- We can compare the model output regression value to some threshold in order to determine if the model prefers one type over another.
- We will encounter this when looking at the Higgs Kaggle problem, where we use the tf.cast function:

```
predictions = tf.cast(probabilities > 0.5, tf.float32)
```

SUMMARY

- We have seen how to build and optimise both a classification and a regression model.
  - There is an example regression model script provided - we will encounter a classification problem when we discuss CNNs.
- The example model will be used to help you explore some of the practicalities of machine learning.