DR ADRIAN BEVAN

PRACTICAL MACHINE LEARNING

DEEP LEARNING WITH TENSORFLOW: MULTILAYER PERCEPTRONS (MLPS)

LECTURE PLAN

- What is deep learning?
- Deep MLPs
- Dropout
- Feature space normalisation
- Batch training
- Examples
- Summary

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

WHAT IS DEEP LEARNING?

- Deep learning is ill defined in that different people have different definitions.
- The working definition that we are going to use here is that a deep network has more than 2 hidden layers.
  - Deep networks with a few hidden layers are broad (hundreds of nodes per layer).
  - Deep networks with with many hidden layers generally are long and thin (i.e. not many nodes per layer)
- There are different types of algorithm that fall into this genre, we are going to focus on deep MLPs (this lecture) and Convolutional Neural Networks (later on in the course).

DEEP MLPS

- The structure of a deep MLP is define in the usual way:
- Input layer:
  - Input layer has the dimensionality of the input feature space.
- Hidden layers:
  - typically hundreds of nodes, leading to $O(10^3)$ - $O(10^6)$ HPs to determine.
- Output layer:
  - either a single node or for a multi classification problem N_{class} output nodes.

DEEP MLPS

- Consider the function approximation example - this starts off as a 1 hidden layer neural network and is extended to a 2 hidden layer variant.
- In general for a deep learning problem we want to be able to create a hidden layer with M inputs to N hidden nodes, each of which will provide an output.
- As the number of HPs requiring optimisation increases, the amount of computing resource required to determine those parameters also increases.
- Techniques for increasing convergence speed discussed earlier now become more important (e.g. dropout, normalisation of input parameters, de-correlation of input parameters) unless you have access to effectively infinite computing resource.

DROPOUT

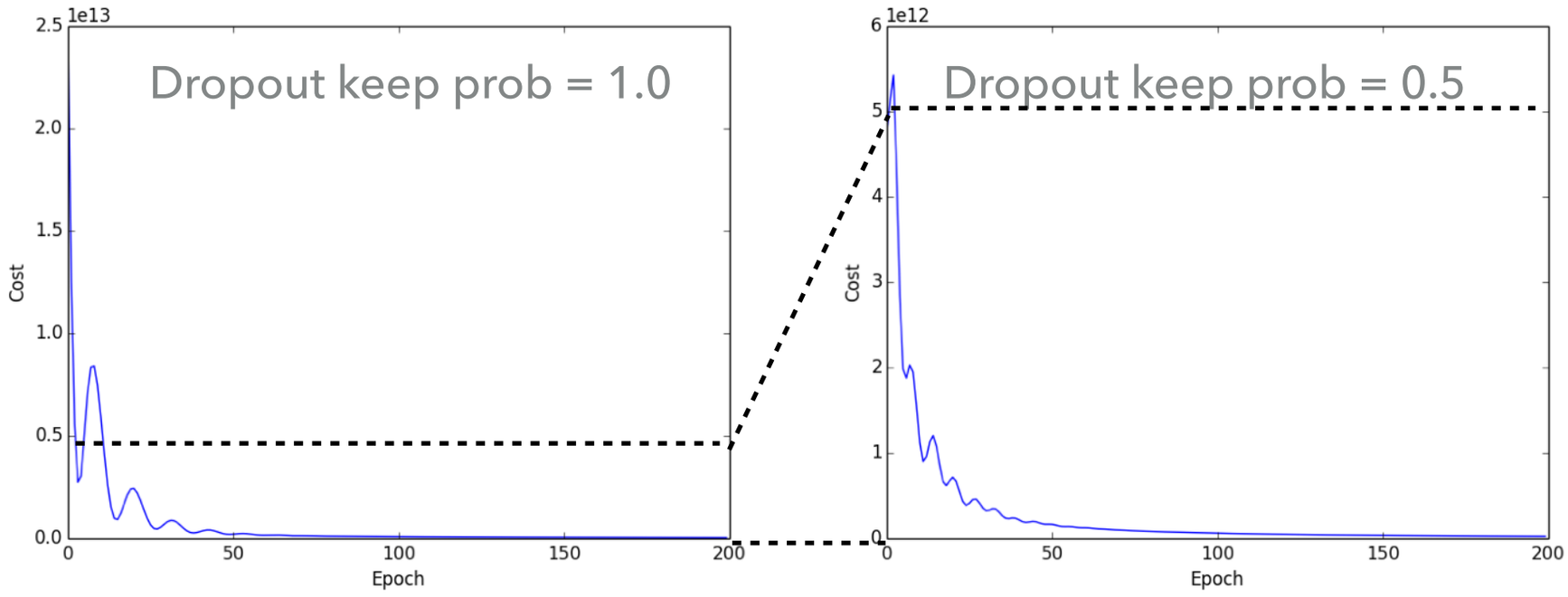
- Dropout is trivial to implement in the training cycle of a neural network.
  - Users just need to specify a keep probability for nodes.
    - 1.0 - keep all nodes
    - 0.5 - drop half of the nodes
    - 0.0 - drop everything (not viable).
      - In addition feeding the data and target values to the optimiser, the user can just specify a keep_prob value between 0.0 and 1.0 to use this technique.

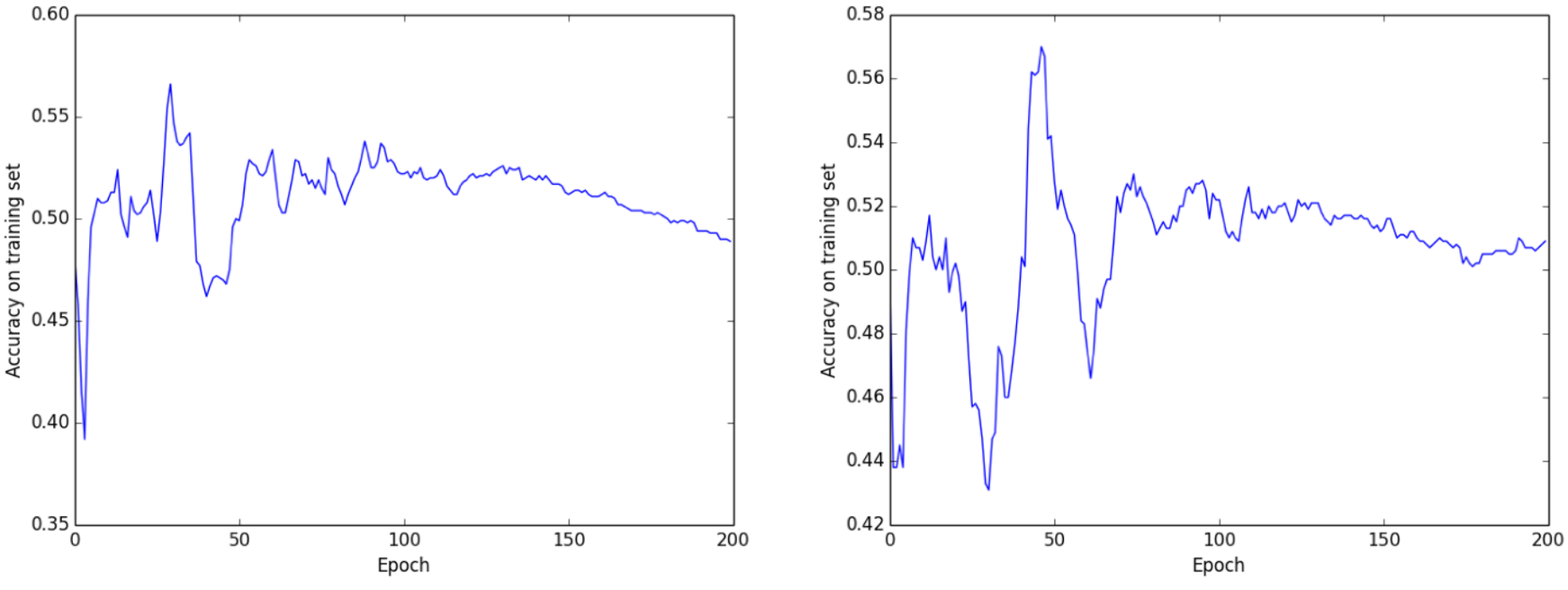
```
sess.run(optimizer, feed_dict={x_: traindata, y_: target_value, keep_prob: 0.8})
```

- Compare the performance of training convergence with and without dropout for one of your networks to help understand the benefits of this technique.

DROPOUT

- e.g. 5 layer DNN: [8:256:256:256:256:256:1]





Higgs Kaggle example with
200 epochs
α = 0.0005
1000 training events

This example illustrates the cost minimises more rapidly with a larger drop out fraction.

The network is clearly not a good choice as the accuracy is poor, and as this is a binary choice, it is little better than guessing on an example by example basis.

DROPOUT

- If I have a neural network, does it make sense to:
  - Use drop out on the input layer?
  - Use drop out on a hidden layer?
  - Use drop out on the output layer/node?

FEATURE SPACE NORMALISATION

- Feature normalisation functions exist in the `PracticalMachineLearning.py` file:
  - `NormaliseData`
  - `NormaliseDataNeg1To1`
  - `NormaliseColumn`

- While it is not necessary to normalise the feature space onto a standardised range, this is one of the suggestions for efficient back propagation.

- *Aside*: with some other algorithms (Support Vector Machines) there can be significant performance degradation for some problems if the user does not normalise the input feature space to a standardised range.

- This is used for the `Example_KaggleHiggs.py` code (see later).

FEATURE SPACE NORMALISATION

- Convert a tensor `newData` into a NumPy array `normalisedData`, normalising to [0, 1] by default.

```
normalisedData       = NormaliseData(newData)
normalisedDataTensor = tf.Variable(normalisedData)
```

- Alternatively one can specify the normalisation to some range [MinRange, MaxRange] using:

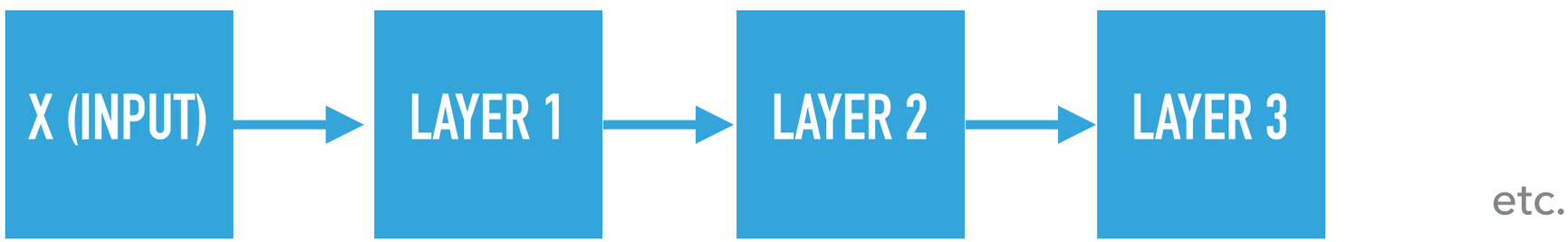
```
normalisedData       = NormaliseData(newData, MinRange,
                                                MaxRange)
normalisedDataTensor = tf.Variable(normalisedData)
```

BATCH TRAINING

- We can restrict the data sample and target label inputs to the optimiser to loop over a number of sub-sets of the full data set.
- This means each optimisation step occurs on a sample that is assumed to provide on average, a reliable gradient estimation of that obtained on the full sample.
- Promotes faster optimisation convergence.
- To implement this in TensorFlow we need to split our training data and labels into subsets, and feed those subsets to the `run.sess(optimiser, …)` line of the code.

EXAMPLES

- One can implement an MLP with multiple layers (5 here) by repeating the pattern of the first layer and considering the flow of outputs from one hidden layer as the inputs to the next hidden layer.
- e.g. for 3 hidden layers one has



etc.

EXAMPLES

- One can implement an MLP with multiple layers (5 here) by repeating the pattern of the first layer and considering the flow of outputs from one hidden layer as the inputs to the next hidden layer.
- e.g. for 3 hidden layers one can write

```
print("Creating a hidden layer with ", n_hidden_1, " nodes")
w_layer_1    = tf.Variable(tf.random_normal([n_input, n_hidden_1]), name="WeightsForLayer1")
bias_layer_1 = tf.Variable(tf.random_normal([n_hidden_1]), name="BiasForLayer1")
layer_1      = tf.nn.relu(tf.add(tf.matmul(x_,w_layer_1),bias_layer_1))
dlayer_1 = tf.nn.dropout(layer_1, keep_prob)

print("Creating a hidden layer with ", n_hidden_2, " nodes")
w_layer_2    = tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2]), name="WeightsForLayer2")
bias_layer_2 = tf.Variable(tf.random_normal([n_hidden_2]), name="BiasForLayer2")
layer_2      = tf.nn.relu(tf.add(tf.matmul(dlayer_1,w_layer_2),bias_layer_2))
dlayer_2 = tf.nn.dropout(layer_2, keep_prob)

print("Creating a hidden layer with ", n_hidden_3, " nodes")
w_layer_3    = tf.Variable(tf.random_normal([n_hidden_2, n_hidden_3]), name="WeightsForLayer3")
bias_layer_3 = tf.Variable(tf.random_normal([n_hidden_3]), name="BiasForLayer3")
layer_3      = tf.nn.relu(tf.add(tf.matmul(dlayer_2,w_layer_3),bias_layer_3))
dlayer_3 = tf.nn.dropout(layer_3, keep_prob)
```

SUMMARY

- We have discussed how to extend a single layer perceptron into a multilayer perceptron that is potentially a deep network.
- The use of:
  - dropout
  - feature space normalisation
  - batch sample training
- have also been illustrated using code.
- All that remains is for you to explore the use of these techniques to consolidate your understanding of how they can benefit optimisation performance for deep networks.

SUGGESTED READING

- Discussion of event classification in text books
  - Goodfellow: *Deep Learning*
    - Section: II