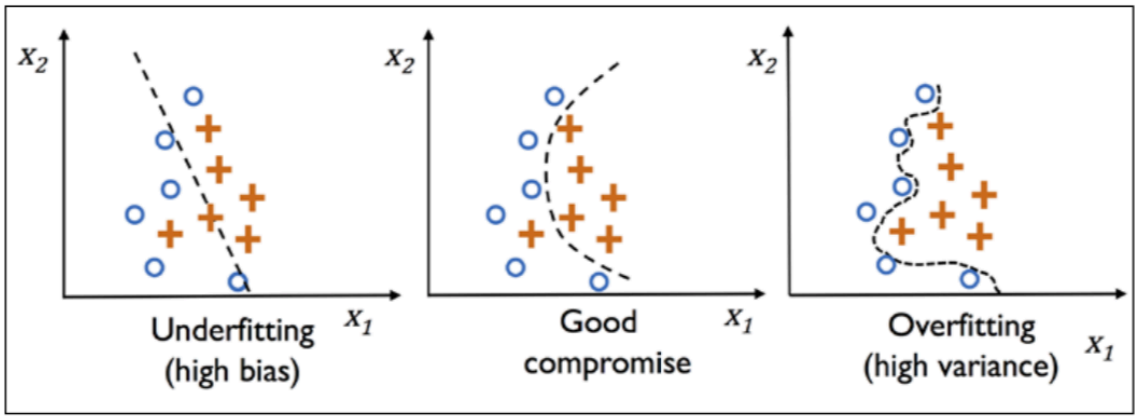
Irene Volpe, PML week 3 assignment

Summary
The assignment for this week is to explore the Kaggle H®t+t- data sets provided.
The aim was to apply my knowledge of machine learning to develop a deep network to solve this classification problem, and to strive to finding the minimum possible mis-classification error rate (best possible accuracy for the model predictions) to distinguish between signal and background.
I therefore explored the use of (mini) batch training, dropout and weight regularization as I trained my network configurations.

Data preprocessing (tackling overfitting):
- Weight regularization (bias added to cost function)
- Mini batch gradient descent (apply gradient on subsets)
- Learning rate (fast vs convergence issues)
- Dropout ()

Overfitting is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data). If a model suffers from overfitting, we also say that the model has a high variance, which can be caused by having too many parameters that lead to a model that is too complex given the underlying data.
Similarly, our model can also suffer from underfitting (high bias), which means that our model is not complex enough to capture the pattern in the training data well and therefore also suffers from low performance on unseen data.



Variance measures the variability of the model prediction for a particular sample instance if we were to retrain the model multiple times, for example, on different subsets of the training dataset. We can say that the model is sensitive to the randomness in the training data.
In contrast, bias measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets; bias is the measure of the systematic error that is not due to randomness.
One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization.

The concept behind regularization is to introduce additional information (bias) to penalize extreme parameter (weight) values. The most common form of regularization is so-called L2 regularization (sometimes also called L2 shrinkage or weight decay), which can be written as follows:

$$\frac{\lambda}{2}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

Here, λ is the so-called **weight regularization parameter**. The cost function for logistic regression can be regularized by adding a simple regularization term, which will shrink the weights during model training:
Via the regularization parameter λ, we can then control how well we fit the training data while keeping the weights small. By increasing the value of λ, we increase the regularization strength.

Mini batch gradient descent
To minimize a cost function we take a step in the opposite direction of a cost gradient that is calculated from the whole training set; this approach is sometimes also referred to as batch gradient descent.
Running batch gradient descent can be computationally quite costly when we have a very large dataset with millions of data points, since we need to reevaluate the whole training dataset each time we take one step towards the global minimum.
An alternative to batch gradient descent (and stochastic gradient descent) is so-called mini-batch learning. Mini-batch learning can be understood as applying batch gradient descent to smaller subsets of the training data.
The advantage over batch gradient descent is that convergence is reached faster via mini-batches because of the more frequent weight updates.

Learning rate
Learning rate has a major impact on a network's performance. A learning rate that is too small doesn't learn quickly enough, but a learning rate that is too large may have difficulty converging as we approach a local minima or region that is ill-conditioned.
One of the major breakthroughs in modern deep network optimization was the advent of learning rate adaption, so the optimal learning rate is appropriately modified over the span of learning to achieve good convergence properties. We used Adam, a popular adaptive learning rate algorithms.

Dropout
While training, dropout is implemented by only keeping a neuron active with some probability *p* (a hyperparameter), or setting it to zero otherwise. Intuitively, this forces the network to be accurate even in the absence of certain information. It prevents the network from becoming too dependent on any one (or any small combination) of neurons.
Expressed more mathematically, it prevents overfitting by providing a way of approximately combining exponentially many different neural network architectures efficiently.

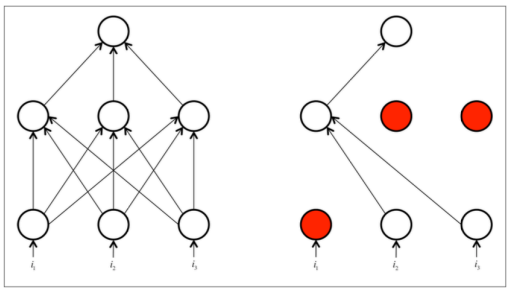


Figure on the left: dropout sets each neuron in the network as inactive with some random probability during each mini-batch of training.

For the training we extracted 85668 examples from the signal and 164334 examples from the background dataset, while for the testing we extracted 34026 examples from the signal and 65976 examples from the background dataset. 8 features out of 30 were selected for all the 4 datasets: 'DER_mass_MMC', 'DER_mass_transverse_met_lep', 'DER_mass_vis', 'DER_pt_tot', 'DER_sum_pt', 'PRI_tau_pt', 'PRI_lep_pt', 'PRI_met'. We then created a merged data sample with 5000 examples from signal and background, 1500 each, for the training and 1500 for the testing, 750 each (I chose a 30-70 split between test and train data).

```
print ("  Extracting signal data ......................")
sFeatureSpace, sFeatures, sLabels, sWeights = BuildFeatureSpace("data/train_sig.csv") #85.668
sFeatureSpaceTest, sFeaturesTest, sLabelsTest, sWeightsTest = BuildFeatureSpace("data/test_sig.csv") #34.026
print ("  Extracting background data ..................")
bFeatureSpace, bFeatures, bLabels, bWeights = BuildFeatureSpace("data/train_bg.csv") #164.334
bFeatureSpaceTest, bFeaturesTest, bLabelsTest, bWeightsTest = BuildFeatureSpace("data/test_bg.csv") #65.976
```

We normalized the feature space (Input features are arbitrary; whereas activation functions have a standardized input domain of [-1, 1] or [0, 1].

```
FeatureSpace, Labels, Weights = MergeFeatureSets(sFeatureSpace, bFeatureSpace, sLabels, bLabels, sWeights,
    bWeights, NTrainingData, "[-1,1]")
FeatureSpaceTest, LabelsTest, WeightsTest = MergeFeatureSets(sFeatureSpaceTest, bFeatureSpaceTest, sLabelsTest,
    bLabelsTest, sWeightsTest, bWeightsTest, NTestData, "[-1,1]")
```

I configured the network architecture to have different randomly generated values for the biases at each layer, to penalize extreme weight values, and a dropout to combine many different neural network architectures at each epoch by dropping a random number of nodes at each layer.

```
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)
```

I used a sigmoid activation function for output nodes, and a l2 cost function to compare the model with. A regularization parameter, here called lambda, was added for the L2 cost function, to shrink the weights during model training, such that the cost function is equal to

$$\| \text{output\_layer} - y \|_2 + \lambda \cdot \sum_i w_i^2$$

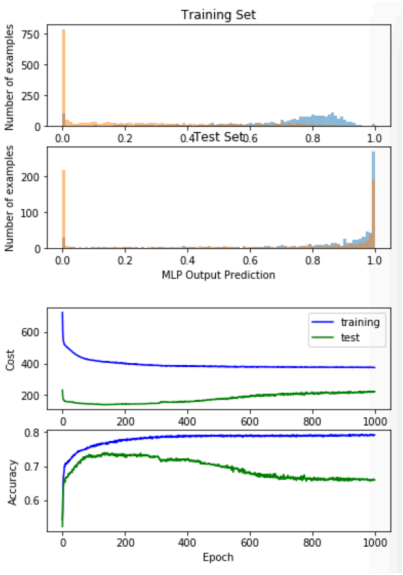
```
output_layer = tf.nn.sigmoid(tf.matmul(dlayer_1, output) + bias_output)
cost = tf.nn.l2_loss(output_layer - y) + lambda_ * (tf.reduce_sum(output)+tf.reduce_sum(w_layer_1))
```

Adam optimizer was selected to optimize the l2 cost function at a set of different learning rates provided at each analysis run.

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

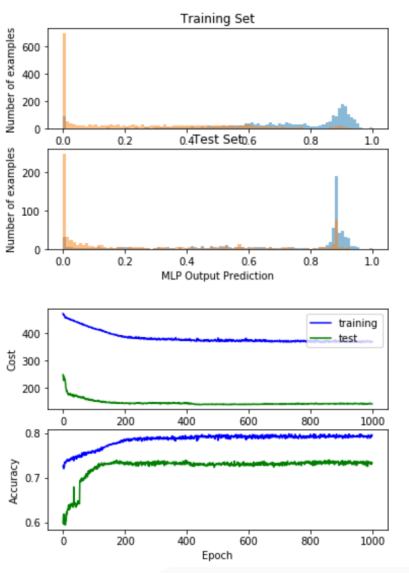
I then proceeded in splitting the data up into n_batches different batches to apply batch gradient descent to smaller subsets of the training data to ensure that convergence is reached faster because of the more frequent weight updates.

```
batchSize = int(len(traindata)/n_batches)
for i in range(n_batches):
    # data
    batch.append(traindata[i*batchSize:(i+1)*batchSize])
    # labels
    lbatch.append(target_value[i*batchSize:(i+1)*batchSize])
```



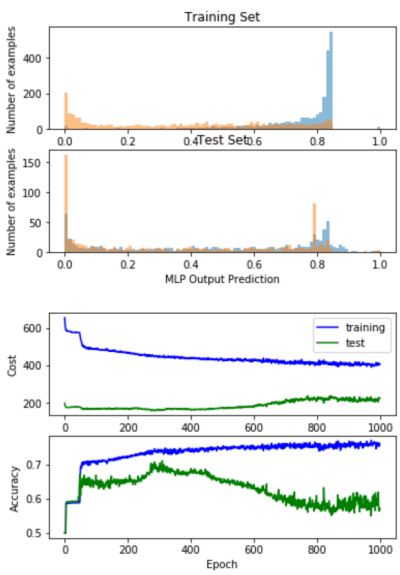
Model 1
learning_rate = 0.001
training_epochs 1000
DropOutKeepProb = 0.9
1 hidden, 10 nodes
n_batches = 200
lambda_ = 0

training accuracy = 0.792
test accuracy = 0.660



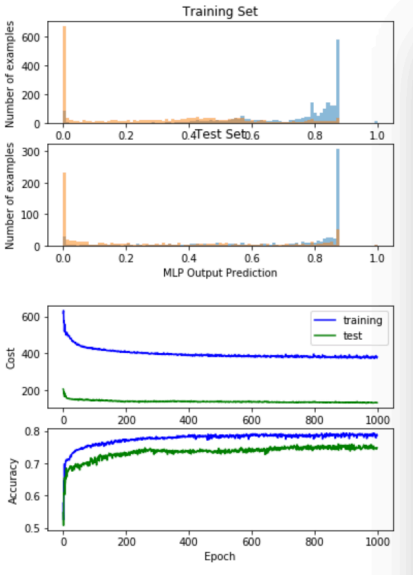
Model 2
learning_rate = 0.001
training_epochs 1000
DropOutKeepProb = 0.9
2 hiddens, 10 nodes each
n_batches = 200
lambda_ = 0

training accuracy = 0.796
test accuracy = 0.733



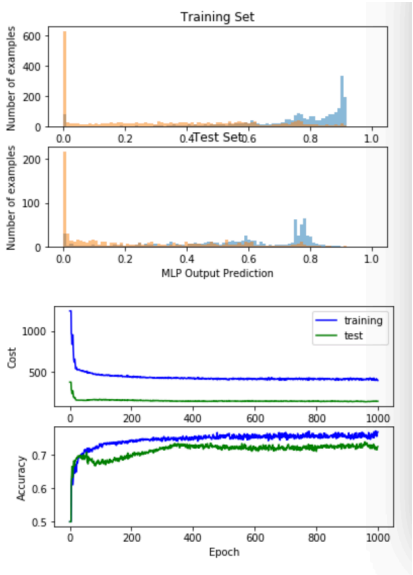
Model 3
learning_rate = 0.001
training_epochs 1000
DropOutKeepProb = 0.9
3 hiddens, 10 nodes each
n_batches = 200
lambda_ = 0

training accuracy = 0.759
test accuracy = 0.569



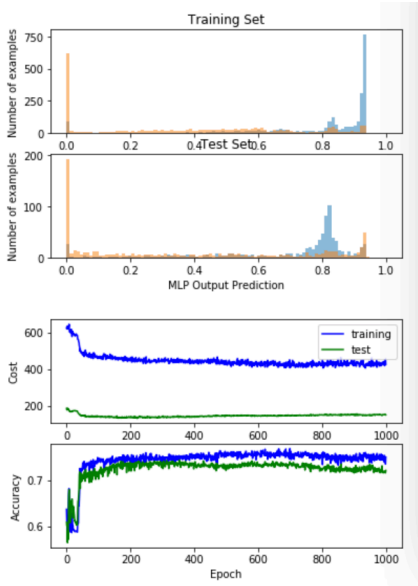
Model 4
learning_rate = 0.001
training_epochs 1000
DropOutKeepProb = 0.9
4 hiddens, 10 nodes each
n_batches = 200
lambda_ = 0

training accuracy = 0.787
test accuracy = 0.748



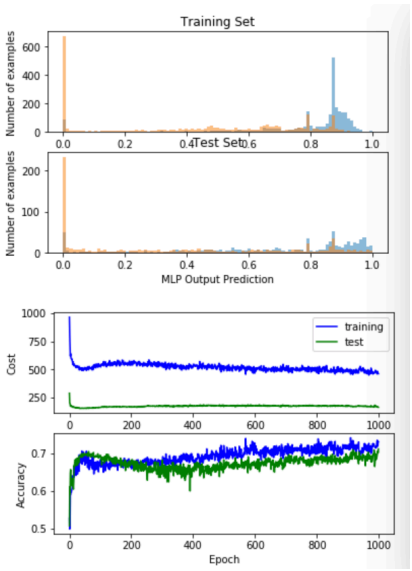
Model 5
learning_rate = 0.001
training_epochs 1000
DropOutKeepProb = 0.9
5 hiddens, 10 nodes each
n_batches = 200
lambda_ = 0

training accuracy = 0.769
test accuracy = 0.725



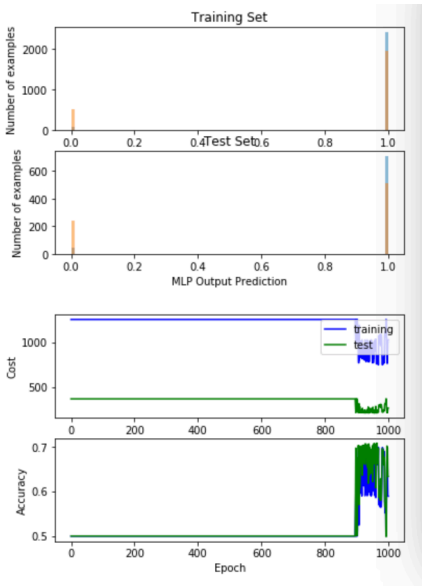
Model 6
learning_rate = 0.001
training_epochs 1000
DropOutKeepProb = 0.9
5 hiddens, 10 nodes each
n_batches = 200
lambda_ = 10^(-9)

training accuracy = 0.749
test accuracy = 0.720



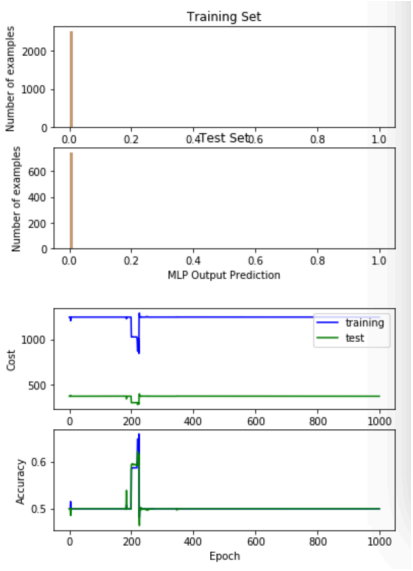
Model 7
learning_rate = 0.001
training_epochs 1000
DropOutKeepProb = 0.9
5 hiddens, 10 nodes each
n_batches = 200
lambda_ = 10^(-15)

training accuracy = 0.730
test accuracy = 0.711



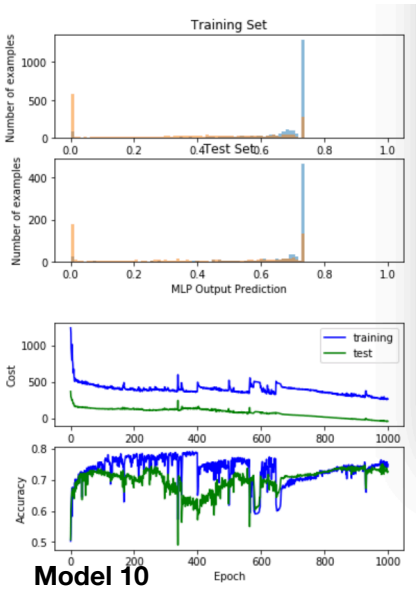
Model 8
learning_rate = 0.001
training_epochs 1000
DropOutKeepProb = 0.9
5 hiddens, **20 nodes** each
n_batches = 200
lambda_ = 0

training accuracy = 0.590
test accuracy = 0.634



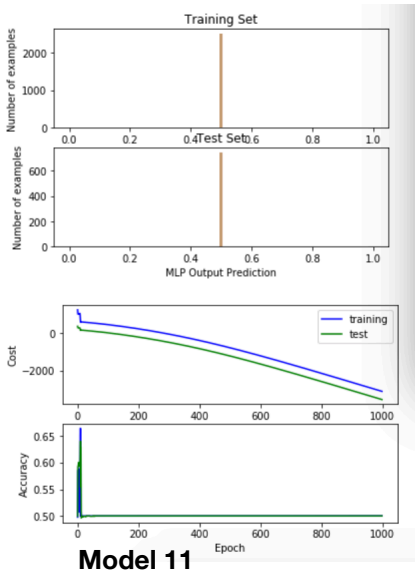
Model 9
learning_rate = 0.001
training_epochs 1000
DropOutKeepProb = 0.9
6 hiddens, 20 nodes each
n_batches = 200
lambda_ = 0

training accuracy = 0.500
test accuracy = 0.500



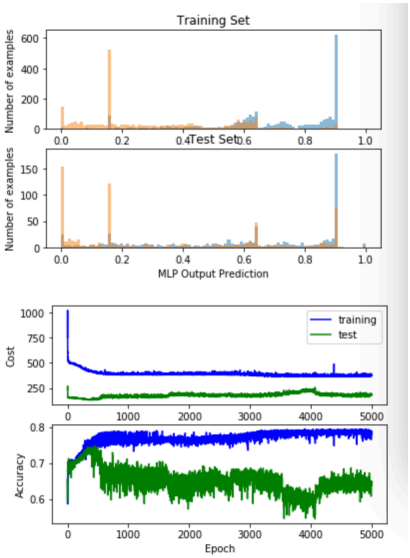
Model 10
learning_rate = 0.01
training_epochs 1000
DropOutKeepProb = 0.9
5 hiddens, 10 nodes each
n_batches = 200
lambda_ = 0.01

training accuracy = 0.750
test accuracy = 0.727



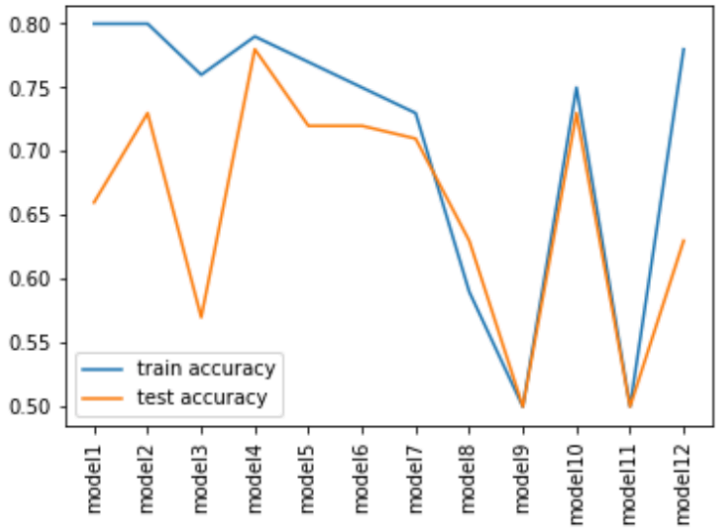
Model 11
learning_rate = 0.01
training_epochs 1000
DropOutKeepProb = 0.9
5 hiddens, 10 nodes each
n_batches = 200
lambda_ = 0.1

training accuracy = 0.5
test accuracy = 0.5



Model 12
learning_rate = 0.01
training_epochs 5000
DropOutKeepProb = 0.9
5 hiddens, 10 nodes each
n_batches = 200
lambda_ = 0

training accuracy = 0.782
test accuracy = 0.638



As we can see from the above plot, models 1,2 and 3's accuracy differ more significantly between the train and the test. Those models correspond to the simpler models (ie models with less number of parameters to tune): fewer hidden layers and nodes, no lambda. They were overfitting the train set, which leads to poor generalization properties.

Adding a forth layer we still have a quite simple model, and not only we get closer accuracy results between the train and test, we also get the best model overall for predicting the test set.

The worst models are number 9 and 11: the former only achieved good performances at around epoch 200, the latter only at the very first epochs.
Model 9 implemented a 6th layer, which I actually would have guess to be overfitting the training and very likely get bad test accuracy results; model 11 might have been affected by too big values for both the step size and the weight regulator lambda.

Models trained with 5 hiddens (ie models 5,6,7 and 8) performed well when only having 10 nodes instead of 20 (model 8).

Model 4,6,7 and 10 scored almost the when trained on the train and test sets. Also, results are similar and good: model 4 was a simple 4 layer 10 node model, 6 and 7 were very similar to 4 but added a layer and a small lambda (10exp-9 and 10exp-15 accordingly) and 10 still had 5 layers, but had a bigger lambda and learning rate.

Models trained with a lambda (ie models 6,7,10,11) had a less constant performance. Models with the peaks of performances (ie models 8,9,11) had too many nodes, too many hidden layers and a too big value for lambda respectively.