Machine Learning Practical (MLP) Coursework 3 Georgios Pligoropoulos - s1687568

Million Song Dataset Classification Problem

We are given a subset of the MSD dataset which contains 40000 instances in the training set and 10000 instances in the validation dataset.

The songs have been preprocessed to extract timbre, chroma and loudness from various segments within the song. Each segment is represented by 25 features in total. In order to have a fixed input length we are keeping the centra 120 segments. This gives a total input of dimensionality 3000.

MSD-10 Classification Problem

Our goal is to classify which of the ten genres correspond to which song, as accurately as possible using a multilayer perceptron (MLP) deep neural network.

MSD-25 Classification Problem

Our goal is to classify which of the 25 genres correspond to which song, as accurately as possible using a multilayer perceptron (MLP) deep neural network.

Tools

For all experiments we are using Python Programming Language and Tensorflow Framework.

Baseline Architecture

Before research the optimal architecture we pick a very simple architecture to serve as a basis for our further exploration.

We will be using **affine transformations** which are interleaved with nonlinearities, and at the end we always have a **softmax output layer**.

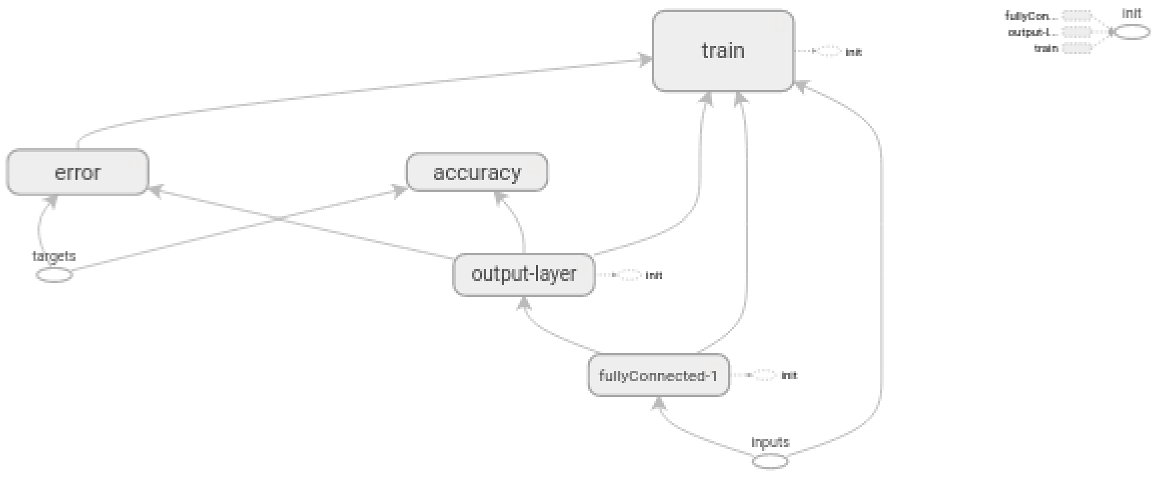
We are using a Shallow Neural Network with one hidden layer of 200 dimensionality.
Activation function is ReLU.
Learning Rate is AdamOptimizer with default learning rate of 1e-3.
There are no other hyperparameters to tune.

Note that we are introducing a small positive bias of 0.1 in ReLU layers, instead of zero, in order to avoid dead-neurons effects.

We will be using a **batch size of 50** for training.

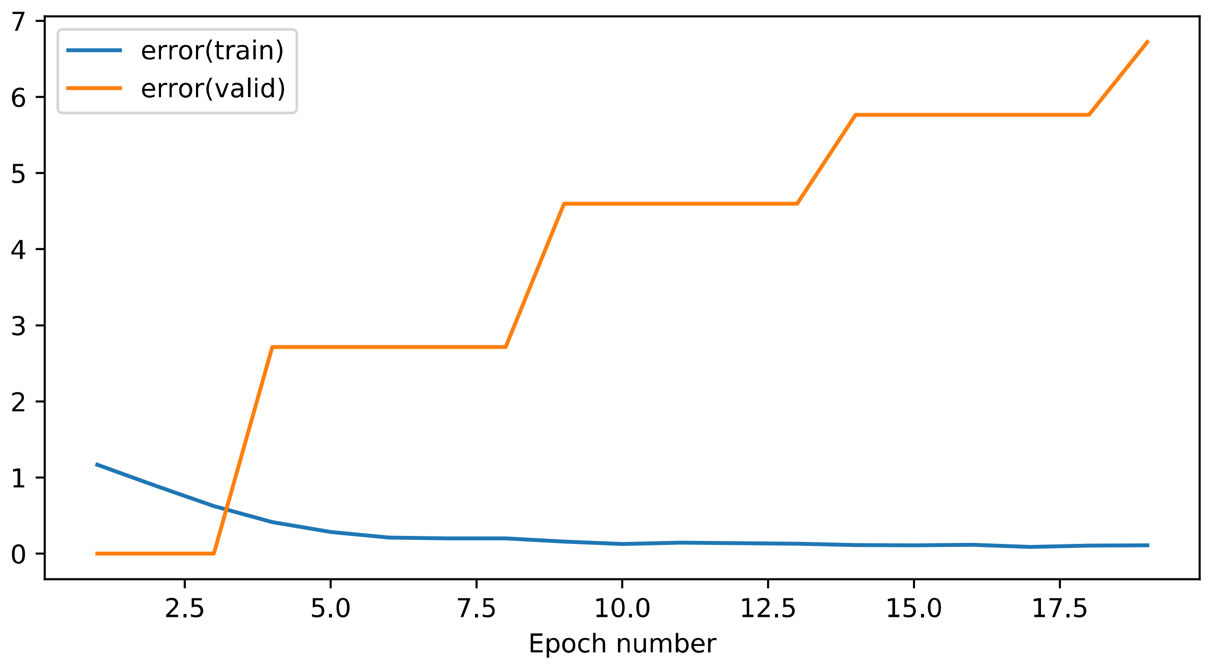
Note: When in our explanations refer to **accuracy** we mean the final validation accuracy and when we refer to **performance** we mean how fast we reached the optimal accuracy for the current experiment.

Note that we will not be reporting final errors and accuracies after each experiment because in most cases the final error is not the minimum error and the final accuracy is not the maximum accuracy. We are most interested in the approximate results since we are far away from 100% accuracy to care for the decimal part of the metrics.

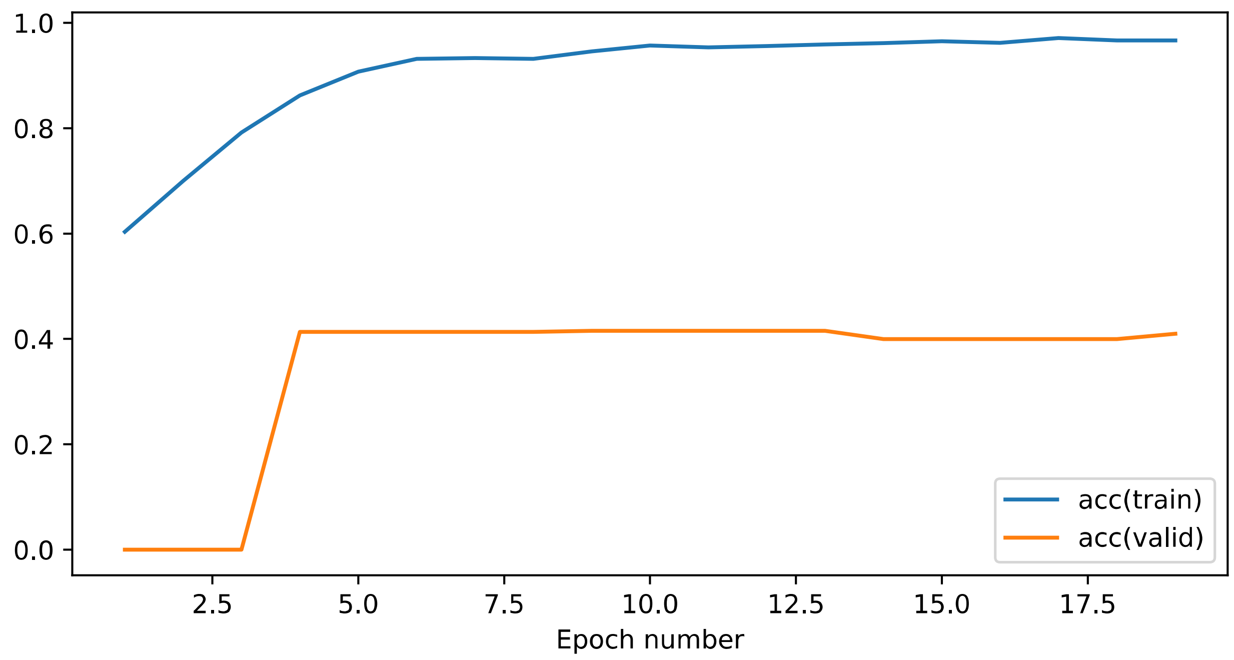


Graph 1: Shallow Neural Net – single hidden layer of 400 dimensionality – ReLU

Results



Plot 1: Training & Validation Error - Shallow neural net - 200 dimensionality of single hidden layer - ReLU nonlinearity



Plot 2: Training & Validation Accuracy - Shallow neural net - 200 dimensionality of single hidden layer - ReLU nonlinearity

Conclusion

This shallow unregularized and non-optimized neural network starts with a validation accuracy of 40%. Our efforts will be focused on increasing the validation accuracy.

Research Question: Pretraining Helps shallow neural network?

One thing we can consider is that we are starting off at a very “bad” place by random initialization and perhaps the model could perform better if we initialize the variables more properly

Method of Pretraining: Stacked Autoencoder

Stacked Autoencoders is a way to pretrain a neural network by making each layer be the autoencoded representation of the previous layer. In other ways each layer is like a distorted version of the input which is actually better than having the weights be randomly initialized.

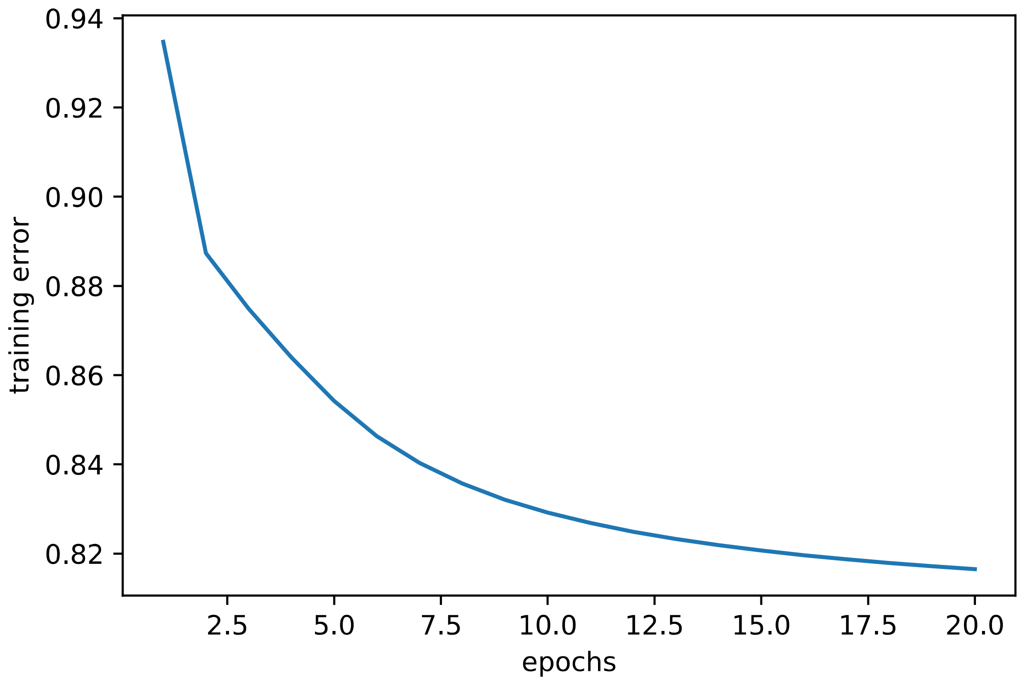
The process is as follows:

1. We forward propagate the neural network up to the already pretrained layer, unless we have not pretrained any layers at all yet
2. We then use the result of the forward propagation for both input and output and we introduce a hidden layer between them.
3. We use as error the Mean Sum of Squares
4. We train for several epochs until the rate of which the error is decreasing becomes too low
5. We keep the weights and biases for the layer from input to hidden layer. We discard the weights and biases from hidden layer to output
6. We repeat from step 1 until all layers are pretrained.

We will experiment with only **Non-Linear Stacked Autoencoders** which use **Sigmoid** non-linearities.

Here we used non-linear autoencoder with sigmoid non-linearities of hidden dimensionality of 200 and then 400. We use the latter one to train our classifier.

Results



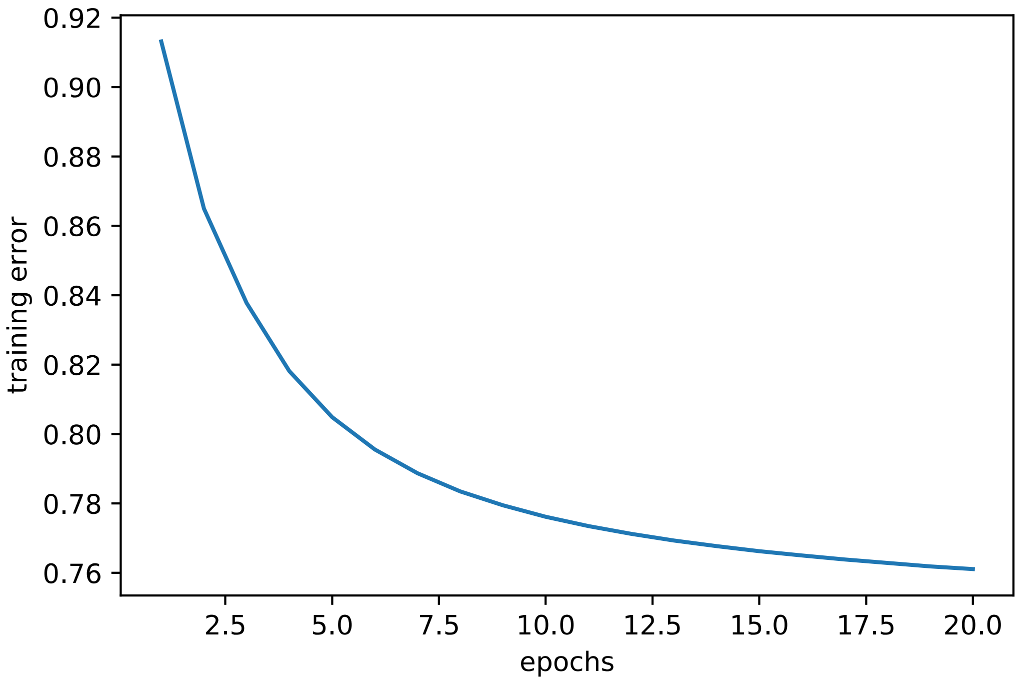
Plot 3: Autoencoder Training Error - 200 dimensionality of hidden layer - Sigmoid nonlinearity

Conclusion

The result from the experiment above is that the hidden dimensionality of 200 seems too little to autoencode the data. The error of 0.82 where the autoencoder converged before than 20 epochs means that we could perhaps do better with a larger dimensionality.

We are going to repeat the experiment by doubling the dimensionality from 200 to 400.
So the layer is 3000 ↔ 400 ↔ 3000

Results

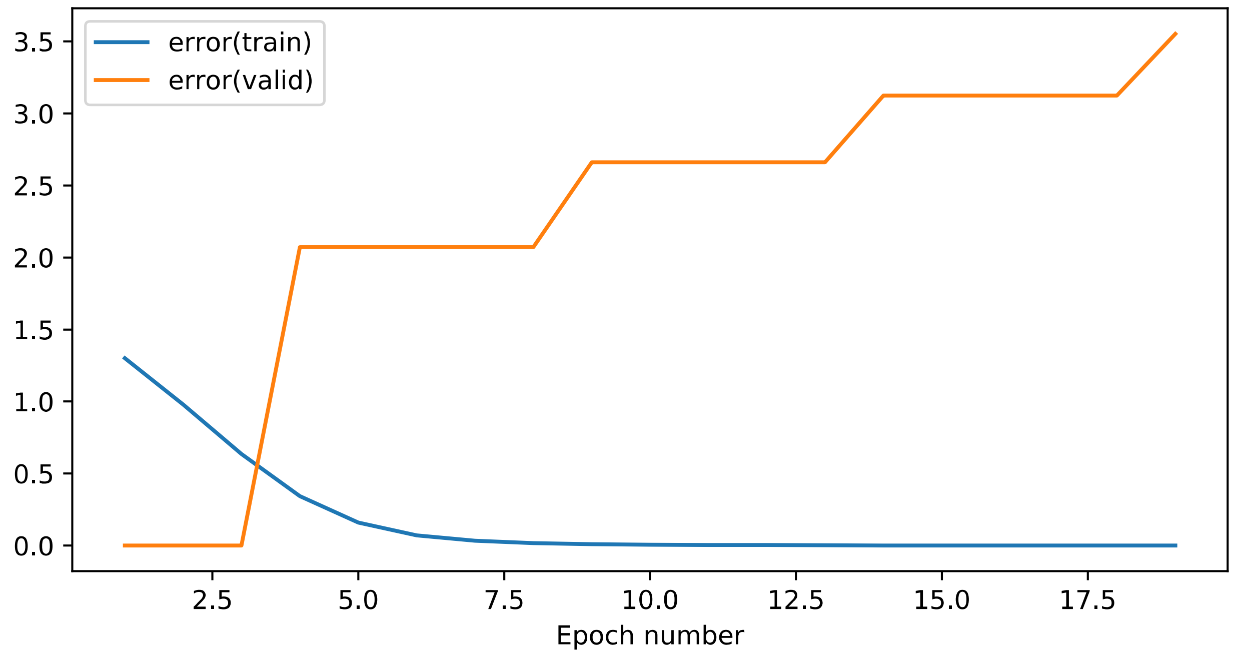


Plot 4: Autoencoder Training Error - 400 dimensionality of hidden layer - Sigmoid nonlinearity

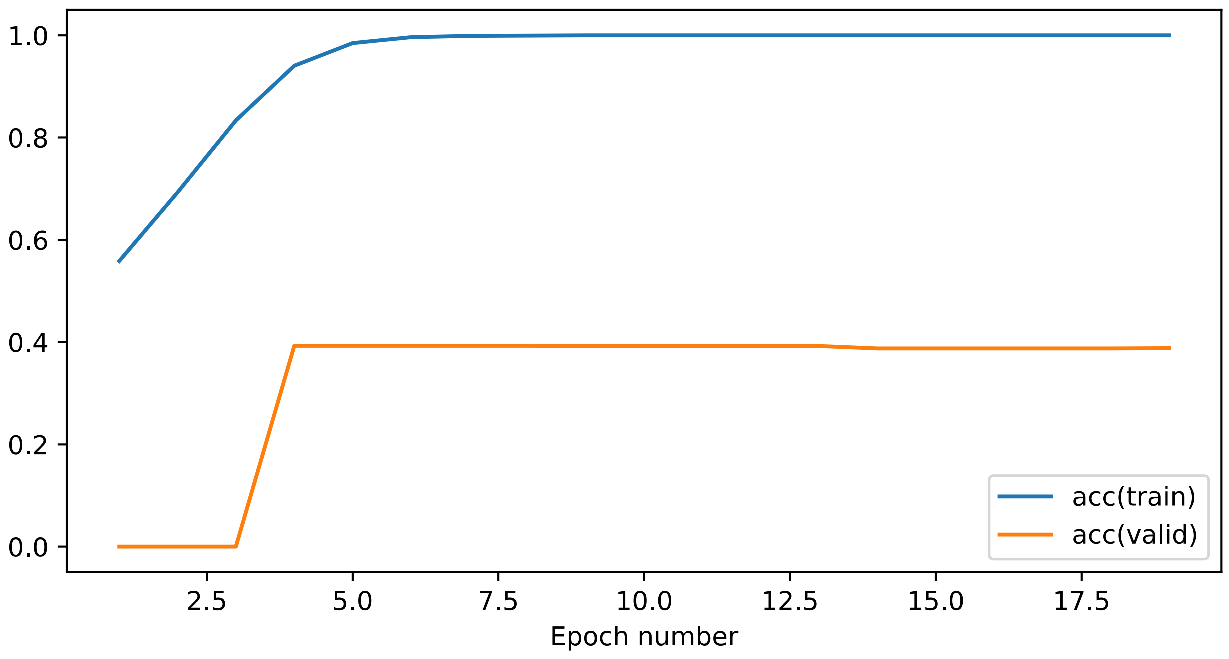
Conclusion

Here we see that the performance has increased in terms of better autoencoding.
Next step is to train our classifier by initializing the weights and biases based on the pretrained values of the stacked autoencoder

Results



Plot 5: Training & Validation Error - Pretrained with Stacked Autoencoder - ReLU nonlinearity - 400 dimensionality of hidden layer



Plot 6: Training & Validation Accuracy - Pretrained with Stacked Autoencoder - ReLU nonlinearity - 400 dimensionality of hidden layer

Conclusions

The answer to the original research question is no. The pretraining, at least for this shallow neural network with one hidden layer of 400, is not able to help our classification problem at all.

We will try at a later step for pretraining when we build a deeper neural network.

From all the above plots one major apparent issue are the overfitting effects which we need to address.

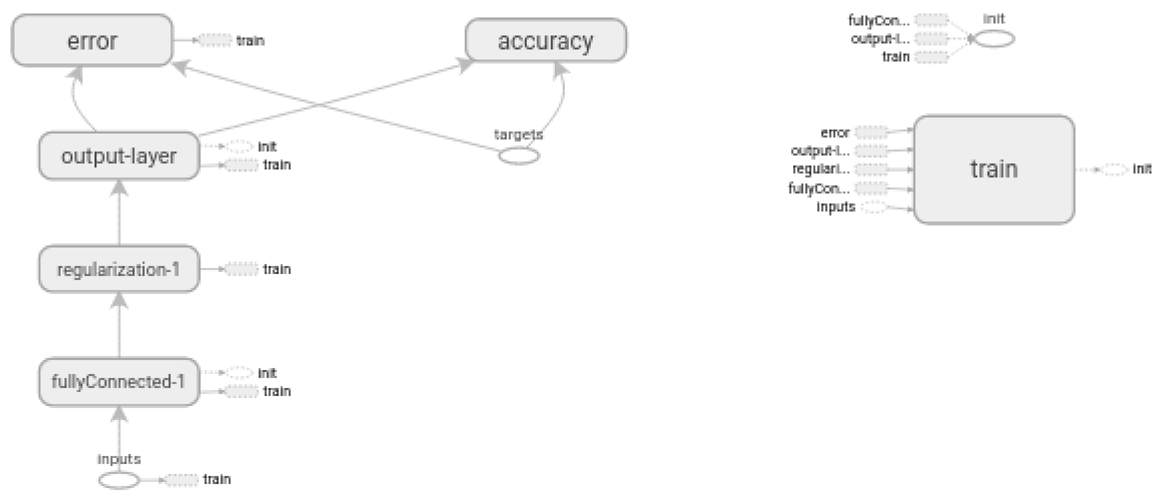
Research Question: Dropout before readout layer helps mediating overfitting effects of the shallow neural network?

Commonly there are one or more intermediate dropout layers used to tackle overfitting effects.

Method of Regularization: Dropout before readout layer

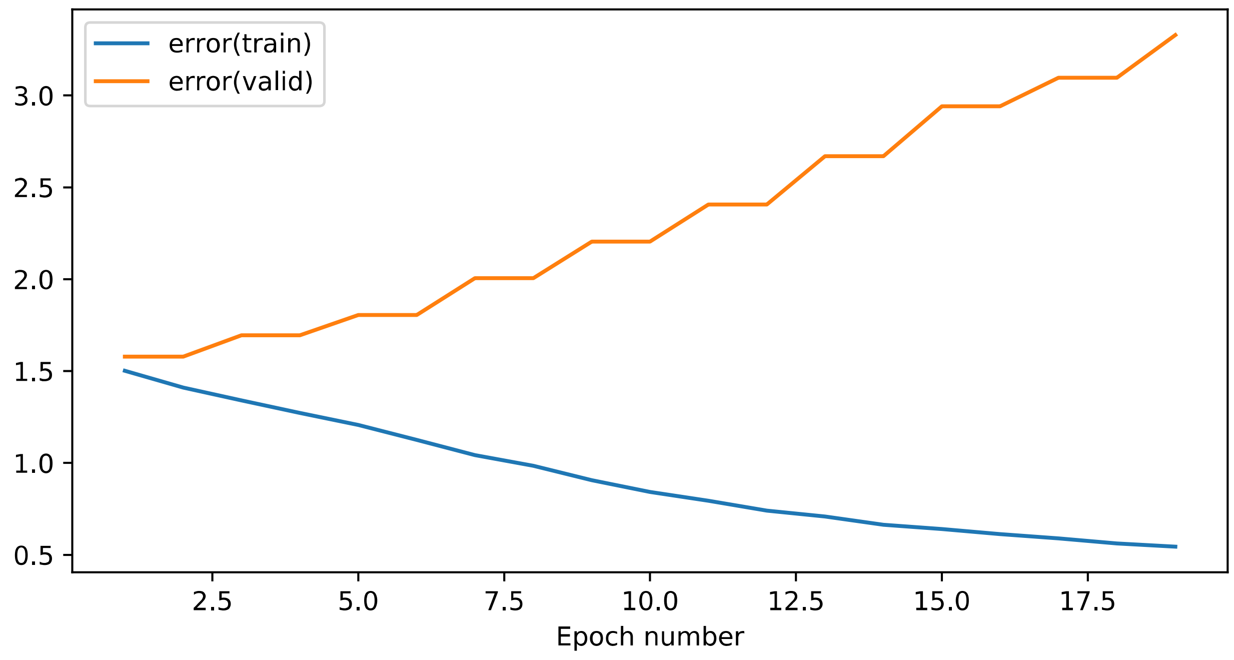
Dropout as a layer keeps randomly only a part of the full dimensionality of the layer and zeros everything else. This has the effect of simulating approximately an exponential number of neural networks.
Here we are using one layer of Dropout before the final readout layer.
The keep probability is equal to 50%.

The tensorflow graph is displayed below.

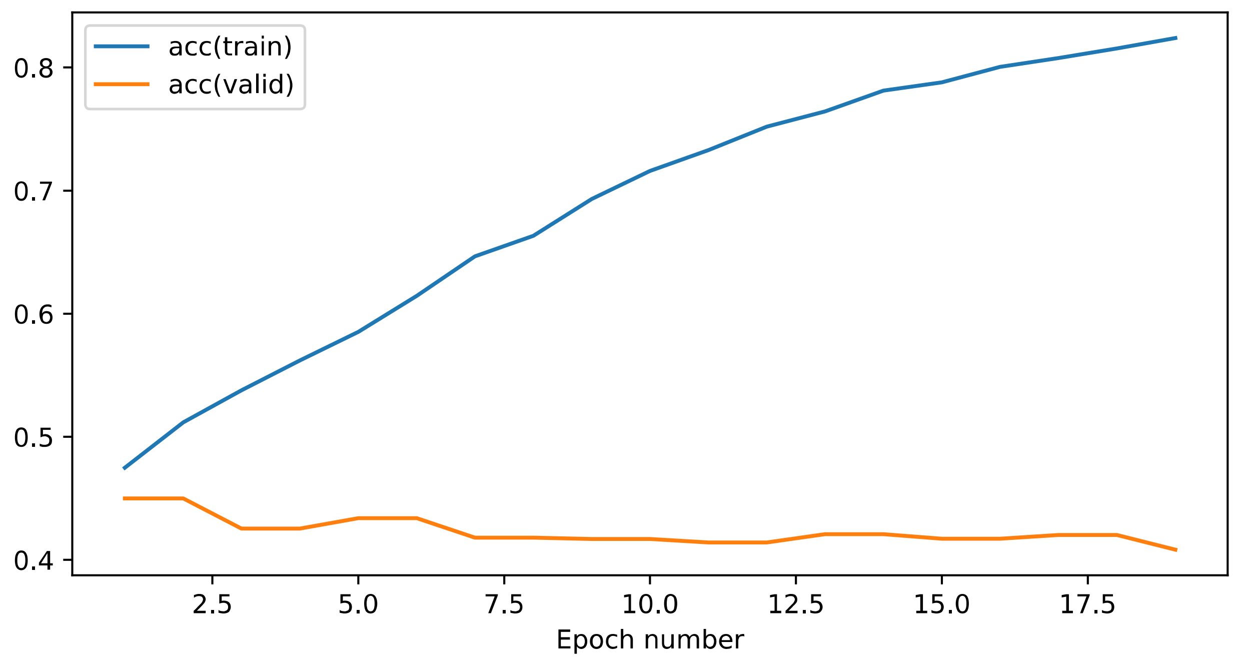


Graph 2: Shallow Neural Net - single hidden layer with 400 dimensionality - Dropout at readout with keep probability as placeholder – ReLU nonlinearity

Results



Plot 7: Training & Validation Error - Shallow Neural Net - single hidden layer with 400 dimensionality - Dropout at readout with 50% keep prob – ReLU nonlinearity



Plot 8: Training & Validation Accuracy - Shallow Neural Net - single hidden layer with 400 dimensionality - Dropout at readout with 50% keep prob – ReLU nonlinearity

Conclusions

The dropout of probability 50% before the readout layer has helped to increase the validation accuracy from 40 to 45%, as it is visible of the first two epochs.

The issue is that we are still experiencing severe overfitting by observing the monotonically increase of the validation error which reduces the validation accuracy.

Next step is to introduce more regularization via dropout.

Research Question: Dropout on both layers helps mediating overfitting effects of the shallow neural network?

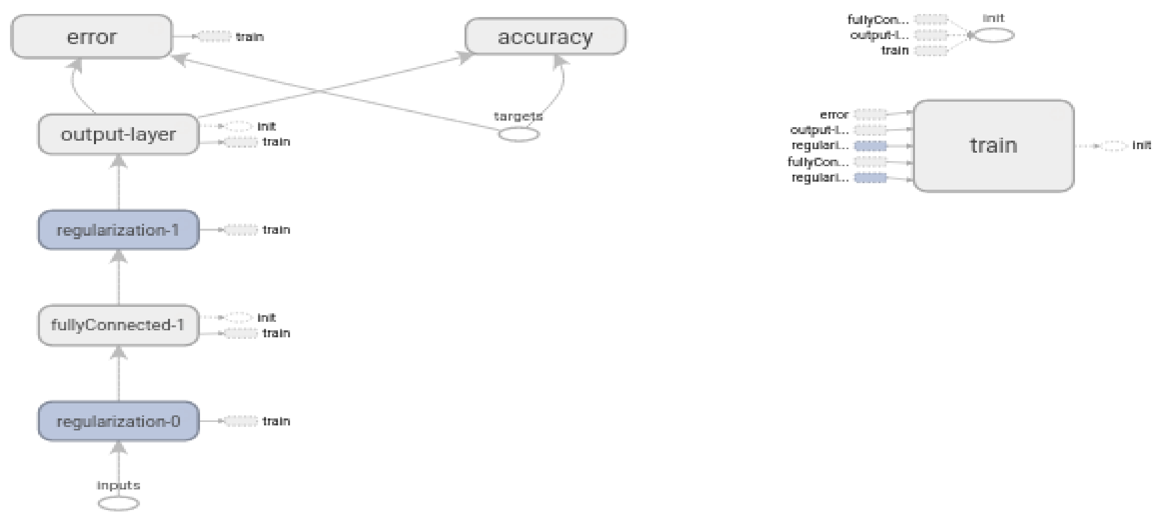
The difference from the previous experiment is that we are introducing a dropout layer at the inputs.

Method of Regularization: Dropout on both layers, .9 input probability, .5 hidden probability

Here we are using one layer of Dropout before the final readout layer and one layer of Dropout at the input layer. Typically dropout probability at the input layer should be relatively high so that the neural network has a good chance to see a noisy representation of the input. If the probability at the input is too low then the neural network will be trained against a severly distorted input and it will fail to generalize properly.

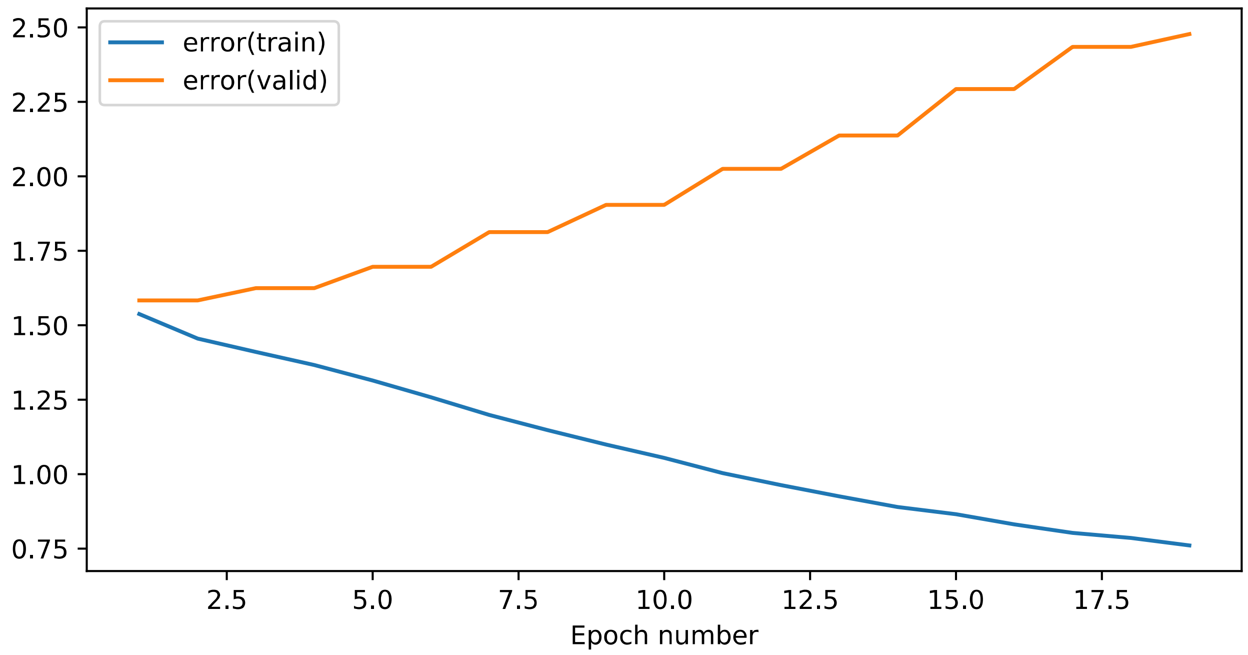
The input dropout probability is equal to 90%
The readout layer dropout probability is equal to 50%

The tensorflow graph is displayed below.

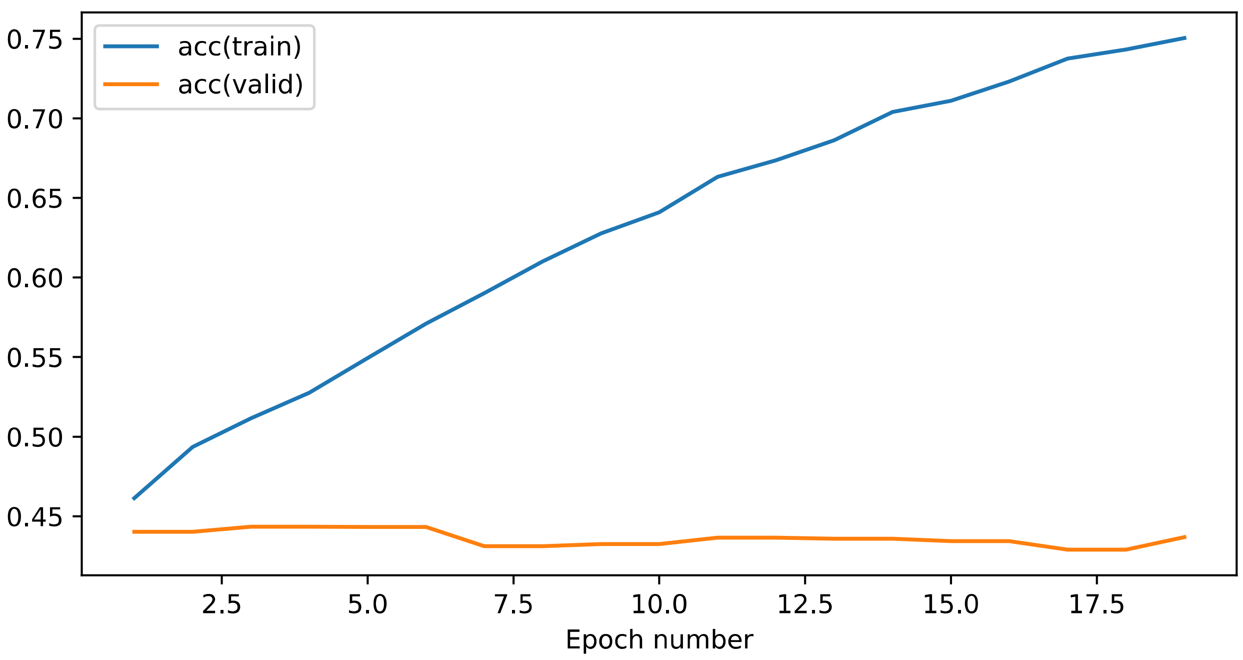


Graph 3: Shallow Neural Net - single hidden layer with 400 dimensionality - Dropout at input and readout with keep probabilities as placeholders – ReLU nonlinearity

Results



Plot 9: Training & Validation Error - Shallow Neural Net - single hidden layer with 400 dimensionality - Dropout at input with 90% keep prob and at readout with 50% keep prob – ReLU nonlinearity



Plot 10: Training & Validation Accuracy - Shallow Neural Net - single hidden layer with 400 dimensionality - Dropout at input with 90% keep prob and at readout with 50% keep prob – ReLU nonlinearity

Conclusions

By introducing a dropout at the input layer means that we did not let the algorithm train itself within 20 epochs. The final training error is smaller than the previous experiment.

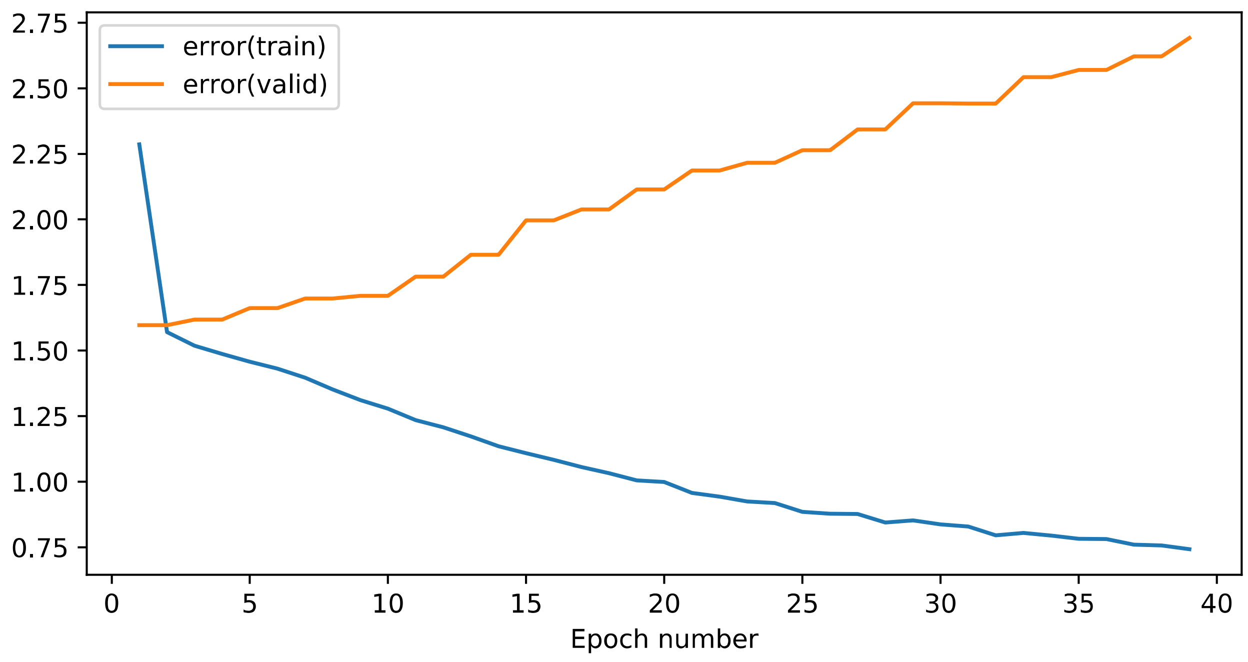
The good outcome however is that we managed to maintain a higher validation probability which is a good sign that regularization with dropout helps but it is not optimally tuned yet.

We are going to lower a little both drop probabilities and let it run for more epochs.

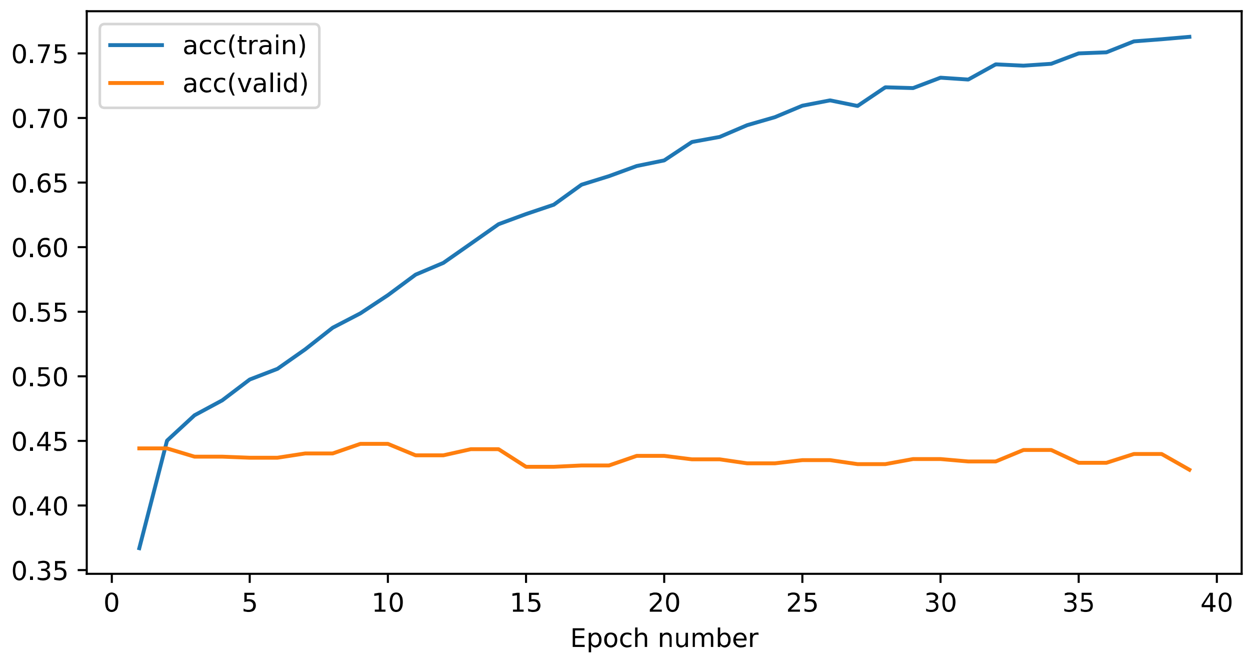
Method of Regularization: Dropout, 85% input probability, 45% hidden probability

We are going to let the neural network train for twice as more epochs and we are lowering the probabilities to try and achieve a stable regularization.
The input dropout probability is equal to 85%
The readout layer dropout probability is equal to 45%

Results



Plot 11: Training & Validation Error - Shallow Neural Net - single hidden layer with 400 dimensionality - Dropout at input with 85% keep prob and at readout with 45% keep prob – ReLU nonlinearity



Plot 12: Training & Validation Accuracy - Shallow Neural Net - single hidden layer with 400 dimensionality - Dropout at input with 85% keep prob and at readout with 45% keep prob – ReLU nonlinearity

Conclusions

Here we see that despite the model is gradually getting better for the training dataset it does not improve its classification performance for the validation dataset within more epochs.

In fact the validation error is gradually increasing meaning that we have not managed to prevent overfitting.

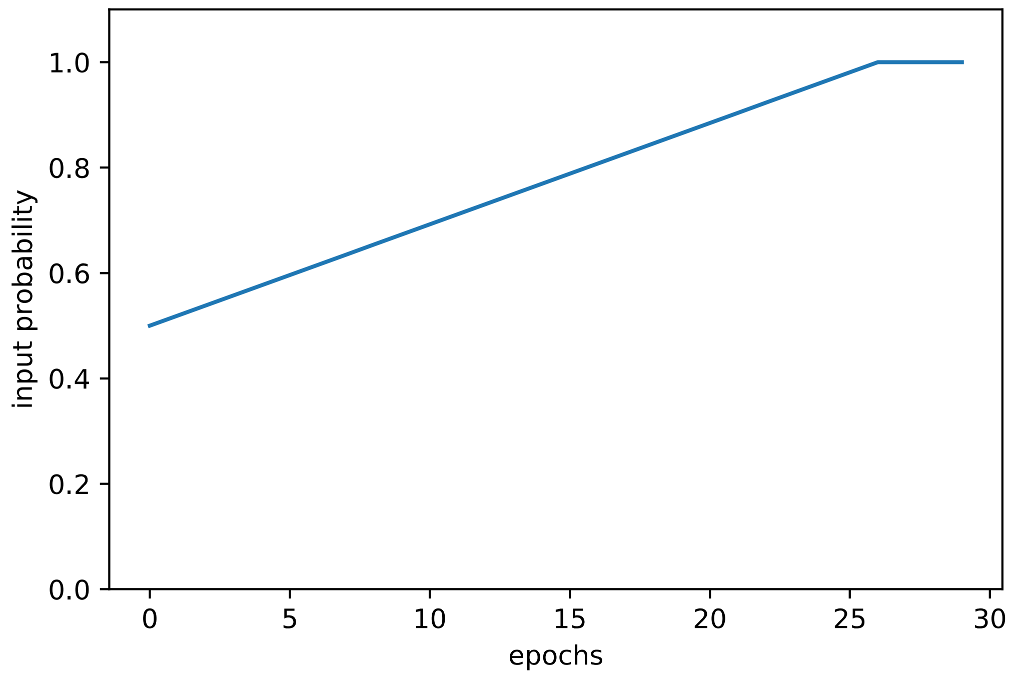
Since this manual tweaking of the dropout probabilities has not converged to any solid results or conclusions we will try and use annealing dropout and see if we could get an overall better performance and result.

Research Question: Linear Annealing Dropout helps mediating overfitting effects of the shallow neural network better than simple dropout?

With annealing dropout we start from a lower probability where we have only a few neurons, randomly chosen, play the role of small models which are expected to fail but they are going to be highly pretrained ancestors of the new more complex models when we start increasing the probability. Unnecessary co-adaptation between neurons that we might have if we start training the entire network altogether is now avoided.

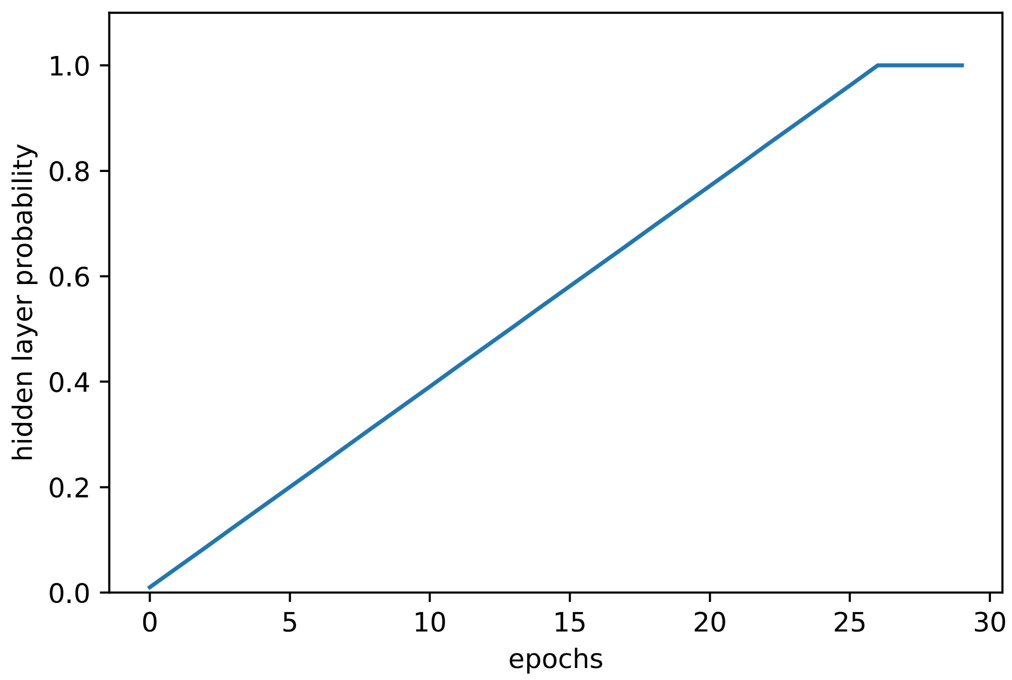
Method of Regularization: Linear Annealing Dropout

The initial probability of the input layer is 50% and is linearly increamented for 90% of the epochs and then stays at the top probability of 100% for the rest 10% of the epochs.



Plot 13: linear increment of dropout keep probability at the inputs along time steps (epochs)

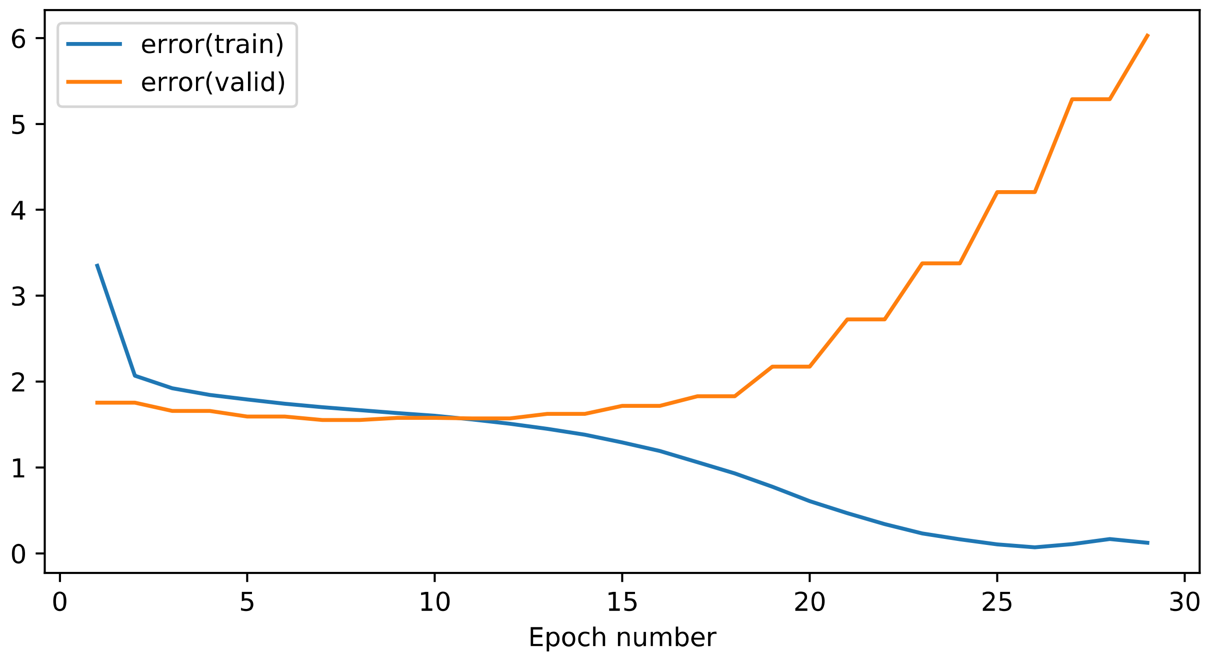
The initial probability of the hidden layer is 1% and is linearly increamented for 90% of the epochs and then stays at the top probability of 100% for the rest 10% of the epochs.



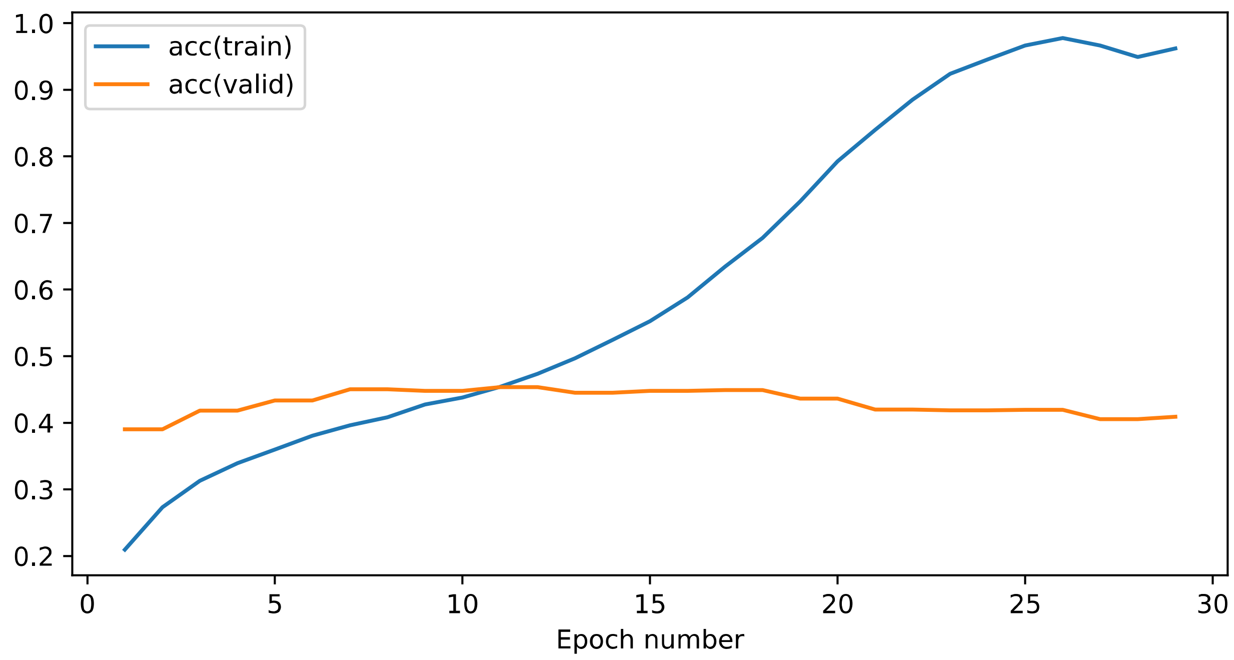
Plot 14: linear increment of dropout keep probability at the hidden layer along time steps (epochs)

Results

After running the experiment for 30 epochs we get the following results:



Plot 15: Training & Validation Error - Shallow Neural Net - single hidden layer with 400 dimensionality – Linear Annealing Dropout – ReLU nonlinearity



Plot 16: Training & Validation Accuracy - Shallow Neural Net - single hidden layer with 400 dimensionality – Linear Annealing Dropout – ReLU nonlinearity

Conclusions

The result from this experiment is that while at the beginning the regularization was enough to keep the validation error at low levels the linear increment of the drop probabilities results in the drop of the validation accuracy.

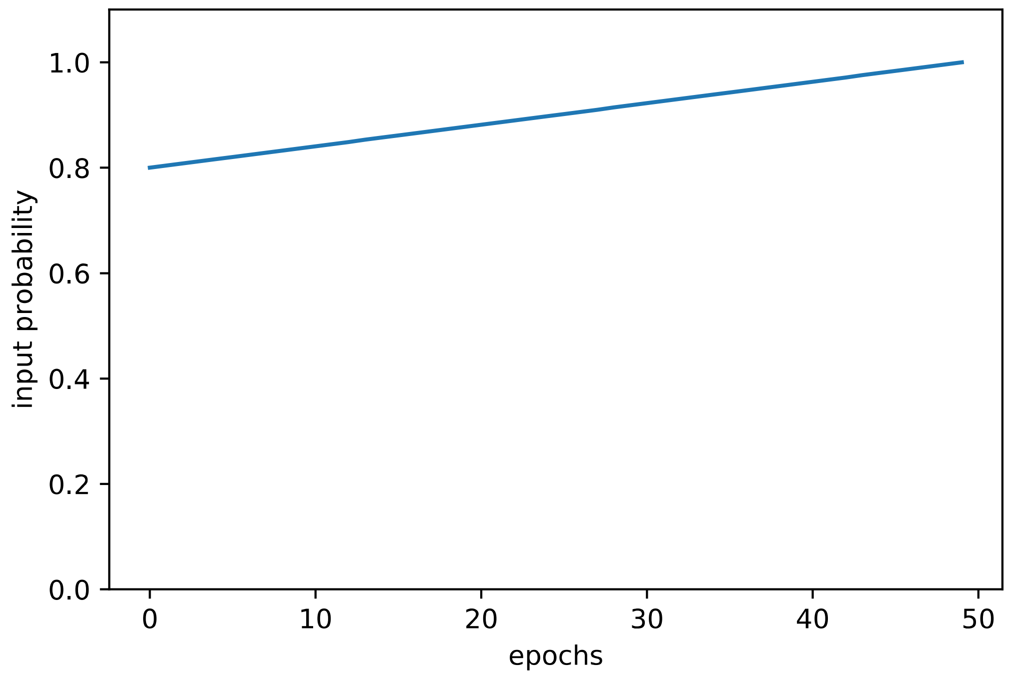
As a next step we will consider annealing dropout which is not linear but it is slowly increasing for most of the epochs with a more fast increment at the end.

Research Question: Exponential Annealing Dropout helps mediating overfitting effects of the shallow neural network better than simple dropout?

Method of Regularization: Exponential Annealing Dropout

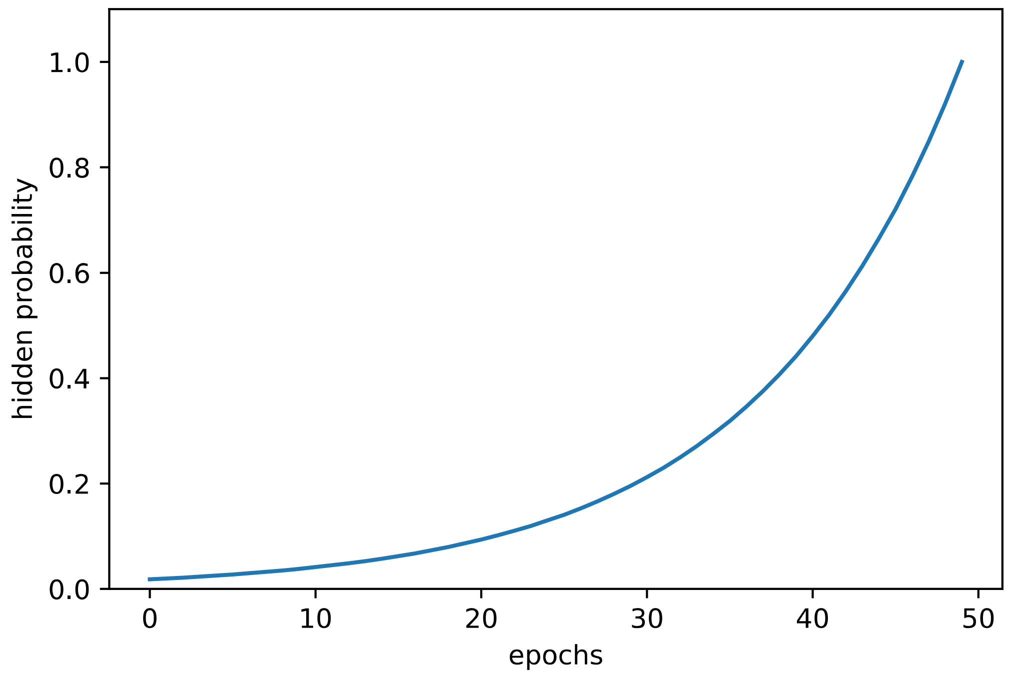
Here we have the same experiment as previously but the probability of the hidden layer has an exponential growth, starting slowly and increasing faster towards the end.

The initial probability of the input layer is 80% and is again still linearly increased for 50 epochs until 100%.



Plot 17: linear increment of dropout keep probability at the inputs along time steps (epochs)

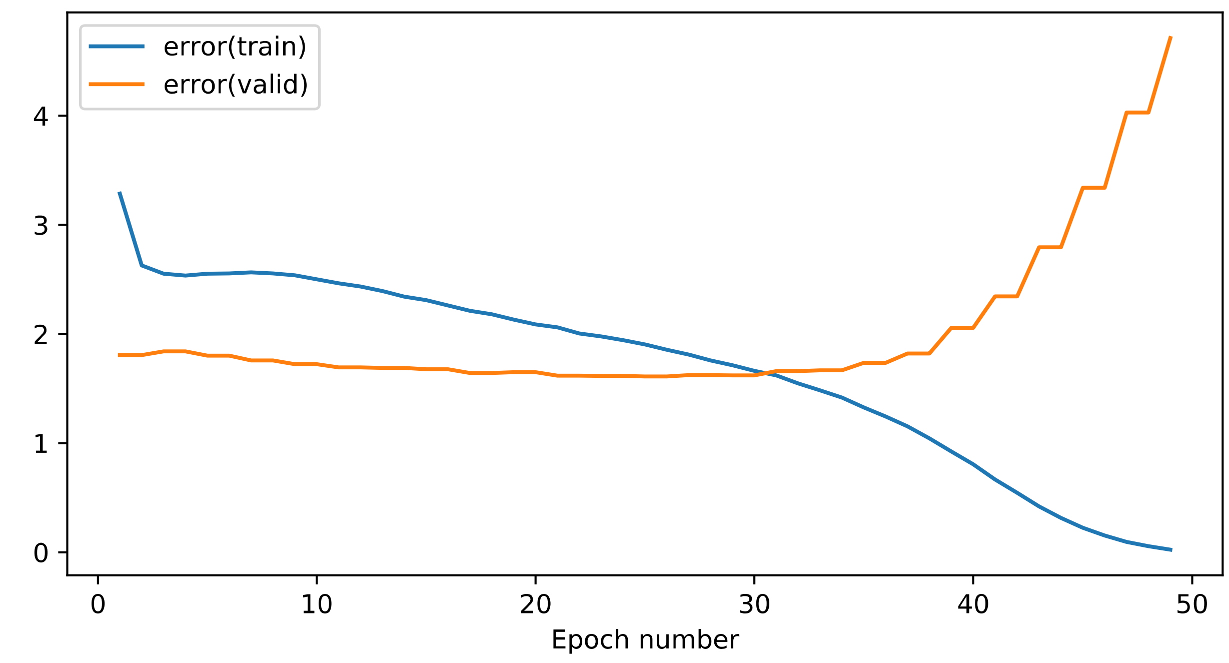
The initial probability of the hidden layer is near zero and then is exponentially increased for 50 epochs until 100%.



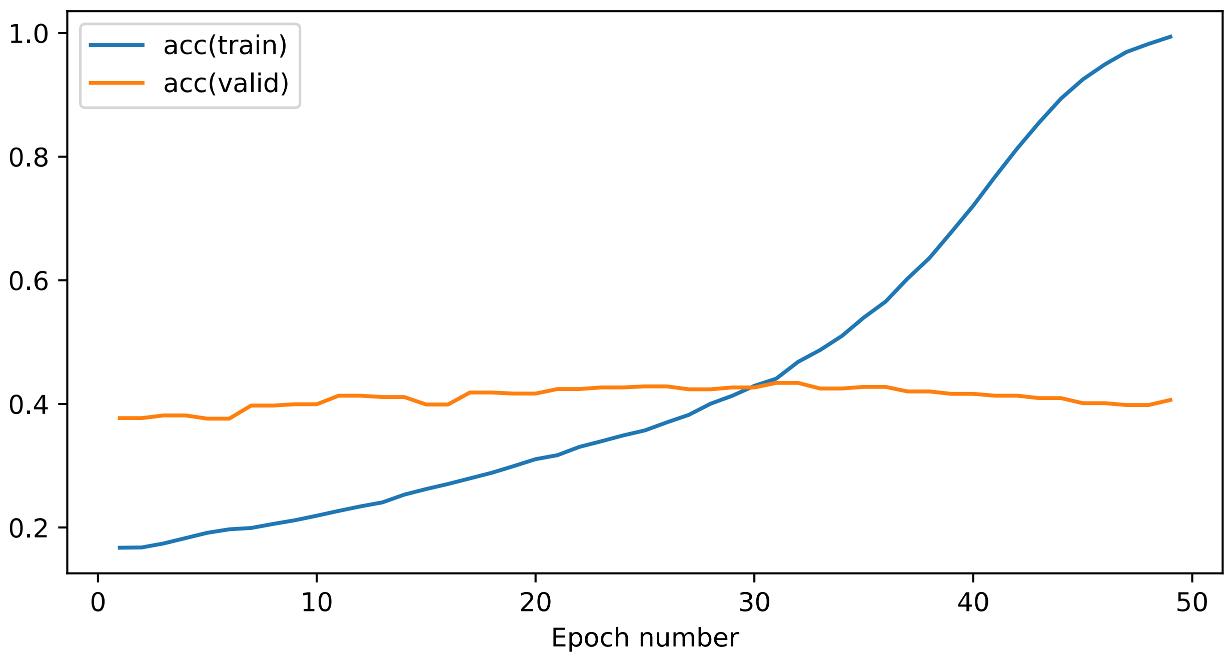
Plot 18: exponential increment of dropout keep probability at the hidden layer along time steps (epochs)

Results

After running the experiment for 50 epochs we get the following plots:



Plot 19: Training & Validation Error - Shallow Neural Net - single hidden layer with 400 dimensionality – Exponential Annealing Dropout – ReLU nonlinearity



Plot 20: Training & Validation Accuracy - Shallow Neural Net - single hidden layer with 400 dimensionality – Exponential Annealing Dropout – ReLU nonlinearity

Conclusions

We notice that in the Linear Annealing Dropout we have an increase in validation error after 10 epochs which is when the hidden layer probability is at ~40%.
While at the Exponential Annealing Dropout we have an increase in validation error after 30 epochs which is when the hidden layer probability is at ~20%.

Research Question: Configuring Dropout probability dynamically based on the validation error helps mediating overfitting effects of the shallow neural network better than simple dropout?

We have failed to choose manually a proper number of parameters so we will try an iterative algorithm where the probability dropout will change dynamically based on the validation accuracy (or error)

Method of Regularization: Dynamic Dropout

Here we are using the most simple conceivable algorithm in an attempt to automatically search for the best dropout probabilities of the input and the hidden layers.

We are keeping the history of the validation error in a variable called *prevValidError* and given that the current validation error is provided by the *valid_error* variable this mathematical formula holds:

$$prevValidError = gamma * prevValidError + valid_{error} * (1 - gamma)$$

Where *gamma* is a hyperparameter that we need to configure in order for enough of the validation error history to be “remembered”.

So the dropout probabilities are being increased and decreased according to the comparison between the current validation error and the history of the previous validation errors.
If the current validation error is smaller that previous validation errors then this means that the system converges towards a better classifier and we can reduce the regularization by increasing the input and hidden dropout probabilities.
On the other hand if the current validation error becomes larger than the history of the previous errors then we are increasing the regularization by reducing the input and hidden dropout probabilities.

In order to avoid getting invalid or unwanted probabilities we will be thresholding them by setting a minimum value different for the dropout input probability and the hidden dropout probability.

So here minimum input dropout probability is 50% and minimum hidden dropout probability is 10%. Maximum probability is allowed to always be 100%.

Note that for this experiment the gamma hyperparameter is set to: 0.7

The rate at which we increase or decrease the dropout probabilities is handled by another hyperparameter which we call *resolution*. Resolution is a fraction of the distance between the maximum and minimum dropout probability.

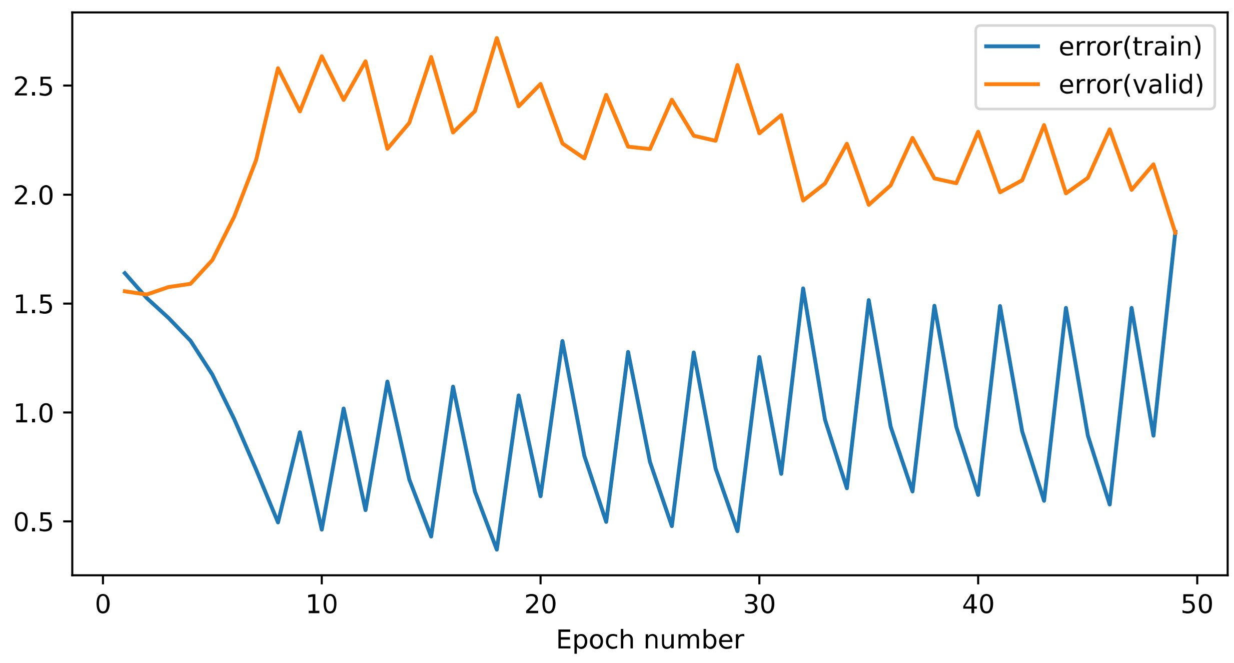
There two formulas hold:

$$step\ for\ input\ dropout\ probability = (1 - minimum\ input\ dropout\ probability) / resolution$$

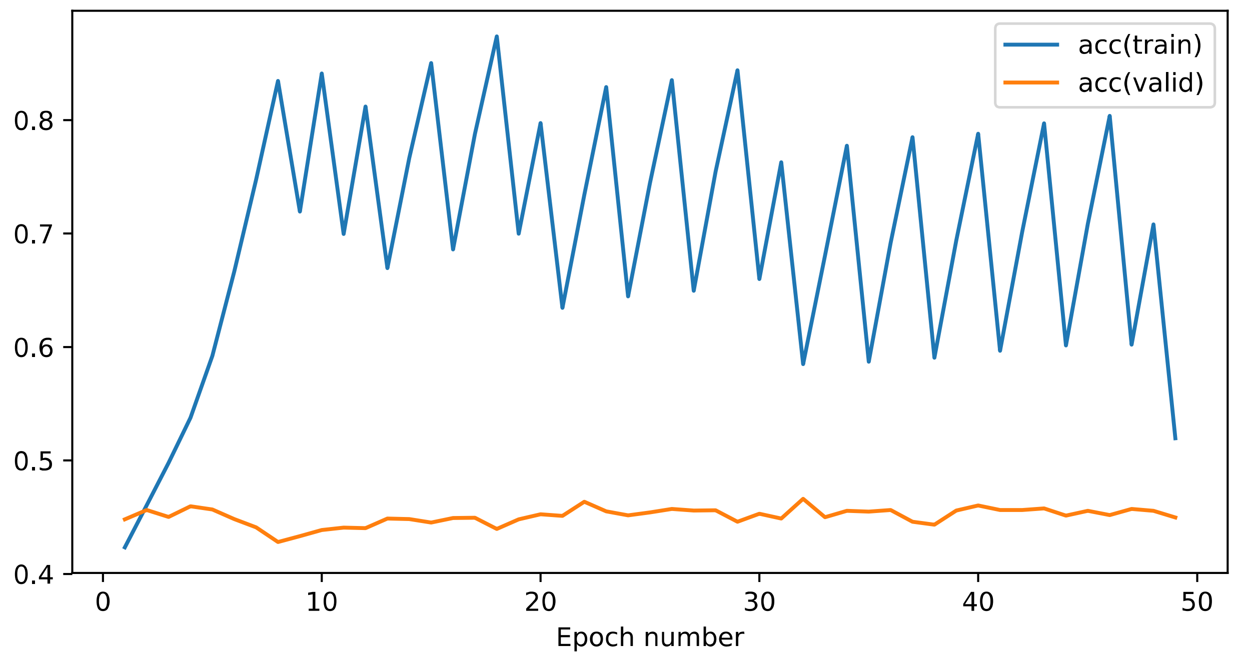
$$step\ for\ hidden\ dropout\ probability = (1 - minimum\ hidden\ dropout\ probability) / resolution$$

Results

After running the experiment for 50 epochs we get the following plots:



Plot 21: Training & Validation Error - Shallow Neural Net - single hidden layer with 400 dimensionality – Dynamic Dropout – ReLU nonlinearity



Plot 22: Training & Validation Accuracy - Shallow Neural Net - single hidden layer with 400 dimensionality – Dynamic Dropout – ReLU nonlinearity

Conclusions

From the above experiment we managed to maintain the best level of validation accuracy around 45-47% for 50 epochs which is better than all the previous experiments.

On the other hand our current approach includes a lot of oscilation which means that it takes longer to converge.

The main outcome however is that we let the algorithm try different combinations of dropout probabilities instead of trying to figure out the best values manually.

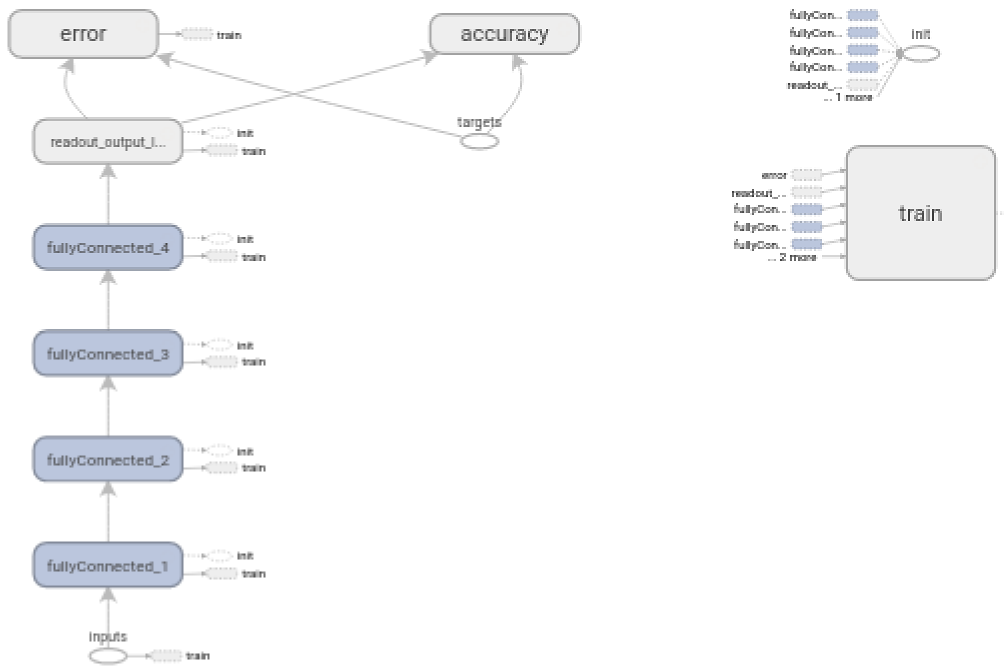
Moreover the validation error is maintained within some boundary and it does not have an increasing trend.

Research Question: Larger dimensionality helps in producing a better classifier?

Here we have an input dimensionality of 3000 and a difficult classification problem so it is reasonable a single hidden layer of dimensionality 400 to not be enough to produce a good model for our classification problem. We will try a deeper architecture.

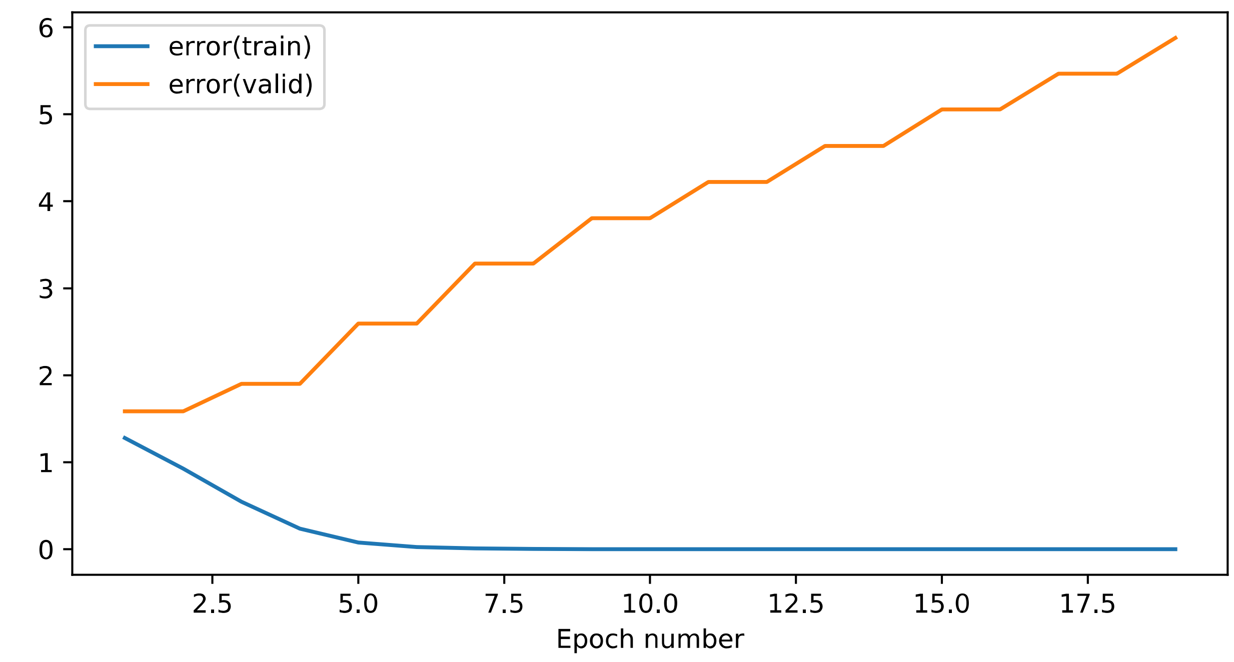
Method: Deep architecture with four levels of 500 dimensionality each

Here we start without any regularization trying to see the initial behavior of the deeper architecture but we set the learning rate of **Adam Learning Rule at 1e-4**. The tensorflow graph is displayed below:

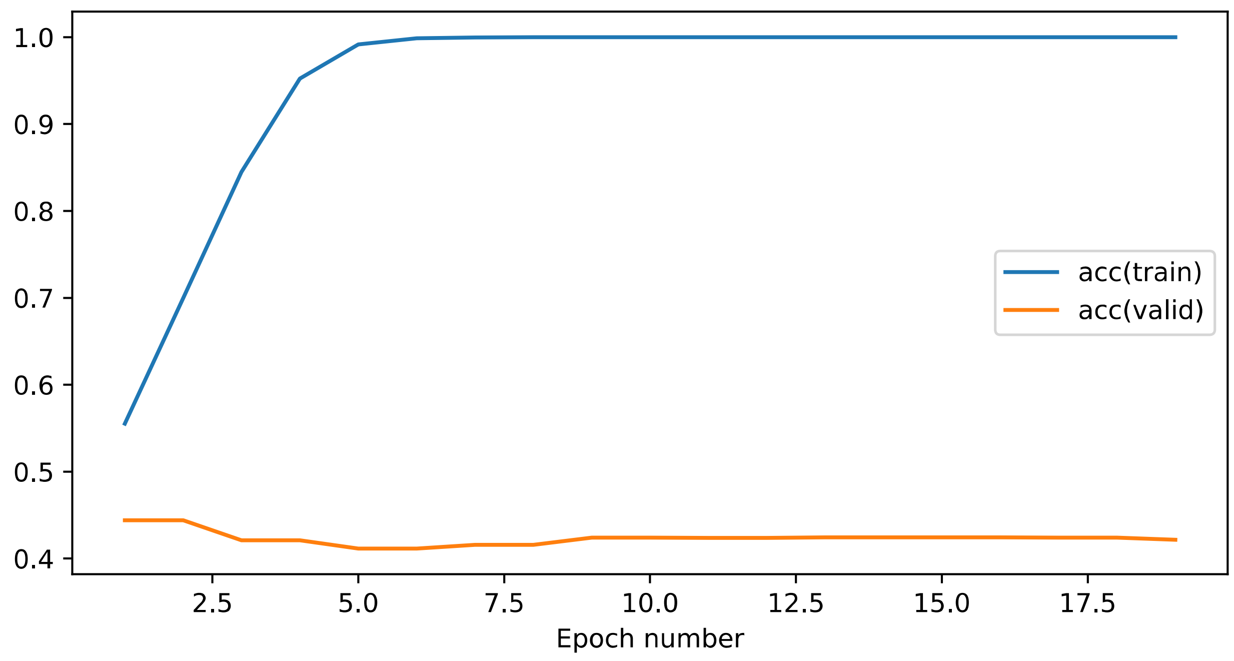


Graph 4: Deep Neural Network - 4 hidden layers of 500 dimesionality each - Adam Learning Rule 1e-4 learning rate - ReLU nonlinearity

Results & Conclusions



Plot 23: Training & Validation Error - Deep Neural Net - 500 dimensionality of all hidden layers - Learning Rate 1e-4 - ReLU non-linearity



Plot 24: Training & Validation Accuracy - Deep Neural Net - 500 dimensionality of all hidden layers - Learning Rate 1e-4 - ReLU non-linearity

We started at 45% without regularization which is a good indication that this model performs better but because of overfitting we cannot derive any concrete conclusions.

Even though we have introduced a smaller learning rate we still get high overfitting effects, so no real conclusion can be derived until we amend the overfitting effect

Research Question: Can we use Dropout to regularize the deep neural network?

It makes sense to use Dropout for regularization purposes as it already worked for the shallow neural network.

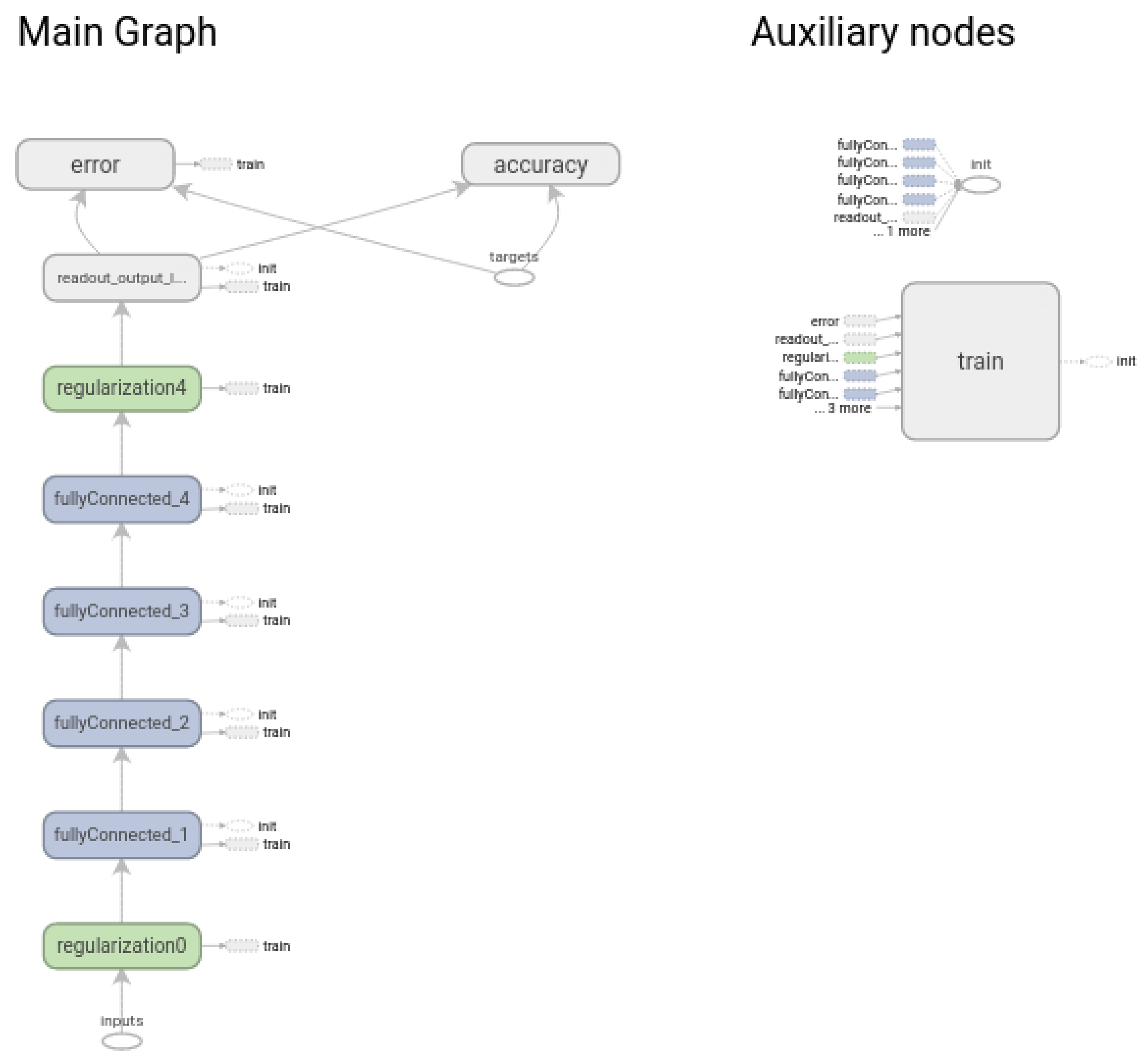
Method: 4 hidden layers of dimensionality 500 regularized with Dropout at inputs and readout layer

First we will try and introduce the dropout layer only at the input and the readout layer and see the effect.

Dropout probability for the input is 90%

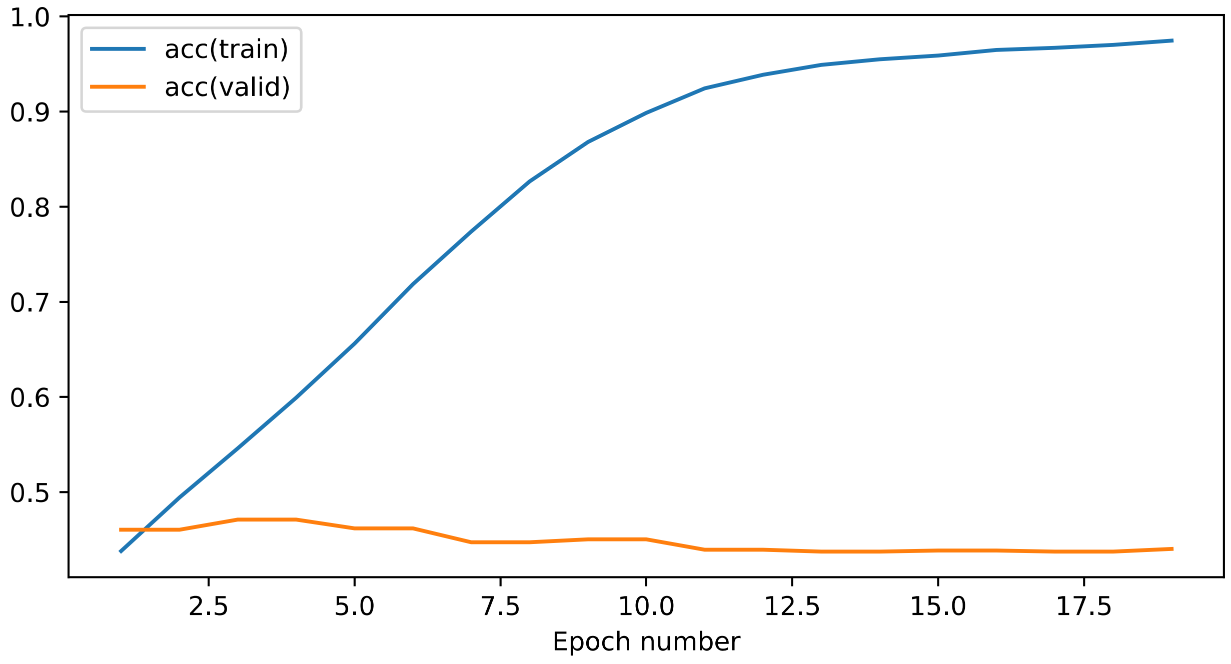
Dropout probability for the readout layer is 50%

The new tensorflow graph is as follows:

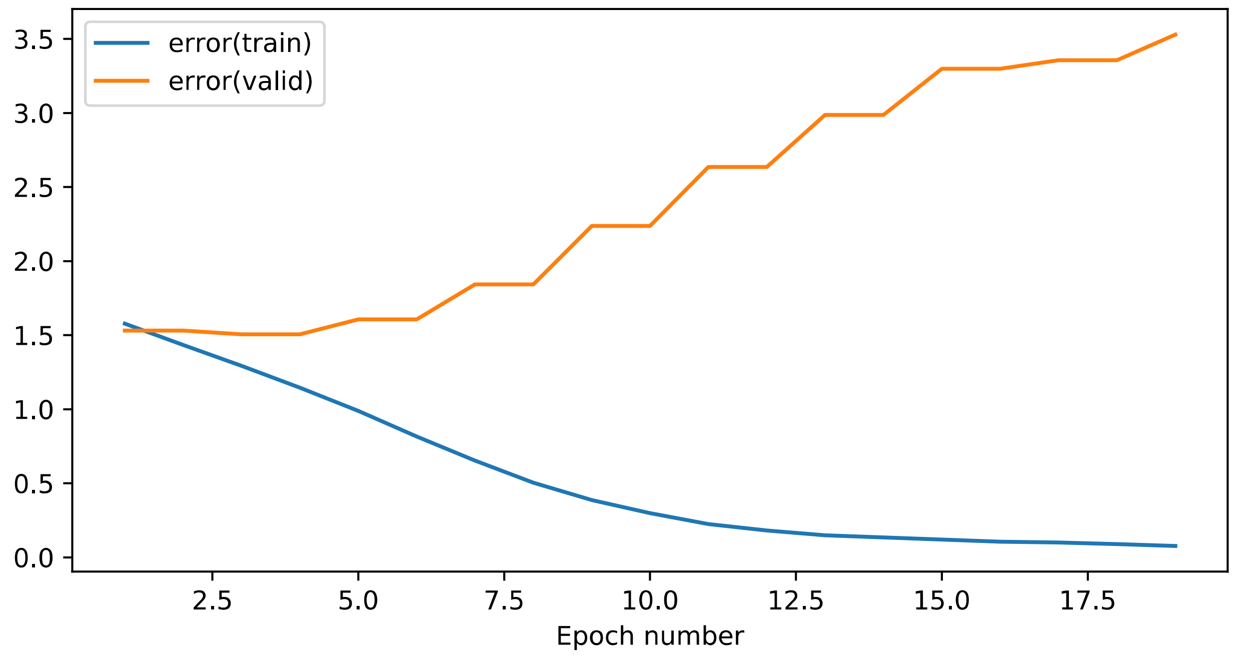


Graph 5: Deep Neural Network - 500 dimensionality of all hidden layers - Learning Rate 1e-4 - ReLU non-linearity - Dropout at Input and Readout layers with keep probabilities as placeholders

Results



Plot 26: Training & Validation Accuracy - Deep Neural Net - 500 dimensionality of all hidden layers - Learning Rate 1e-4 - ReLU non-linearity - Dropout Keep Probability at Input layer 90% - Dropout Keep Probability at Hidden Layer 50%



Plot 25: Training & Validation Error - Deep Neural Net - 500 dimensionality of all hidden layers - Learning Rate 1e-4 - ReLU non-linearity - Dropout Keep Probability at Input layer 90% - Dropout Keep Probability at Hidden Layer 50%

Conclusions

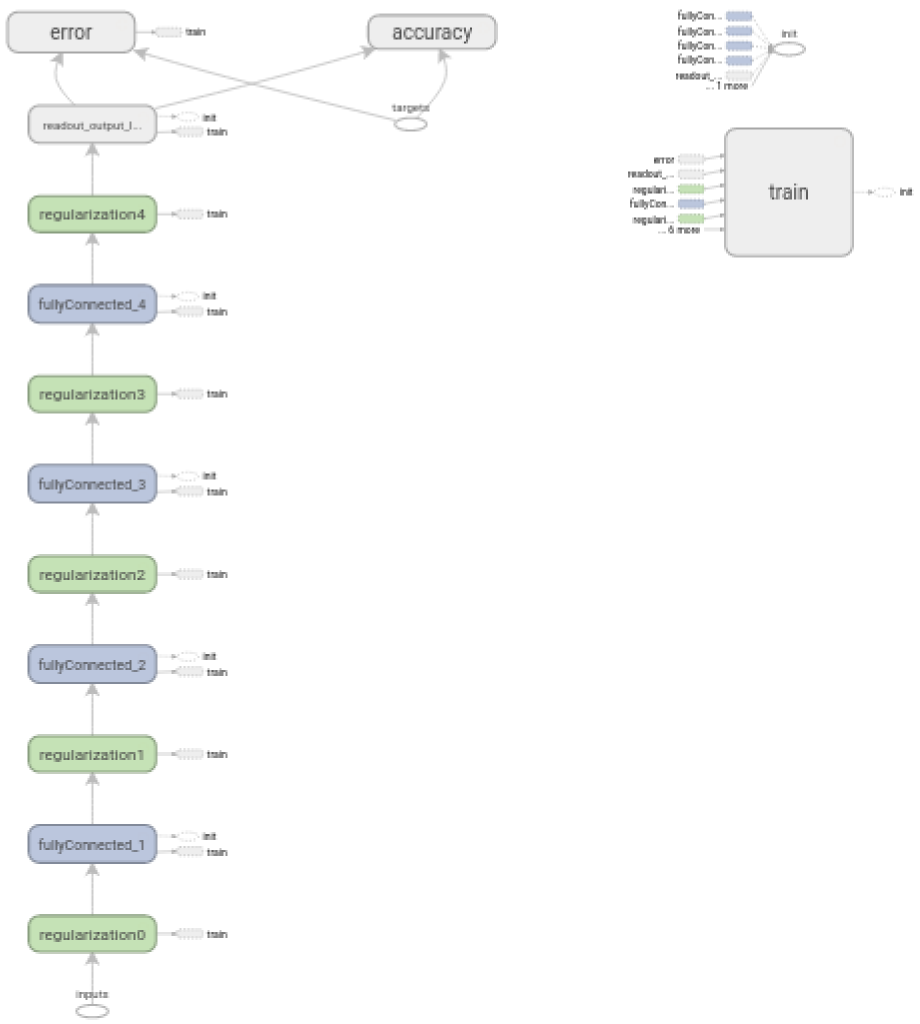
We are seeing some regularization occuring since the final validation error is half than it was before but the overfitting effects are still severe.

Research Question: Will dropout on all layers of the deep neural network will work better than before?

More regularization is needed so introducing one dropout layer before each layer of the deep neural network is expected to bring a better regularization.

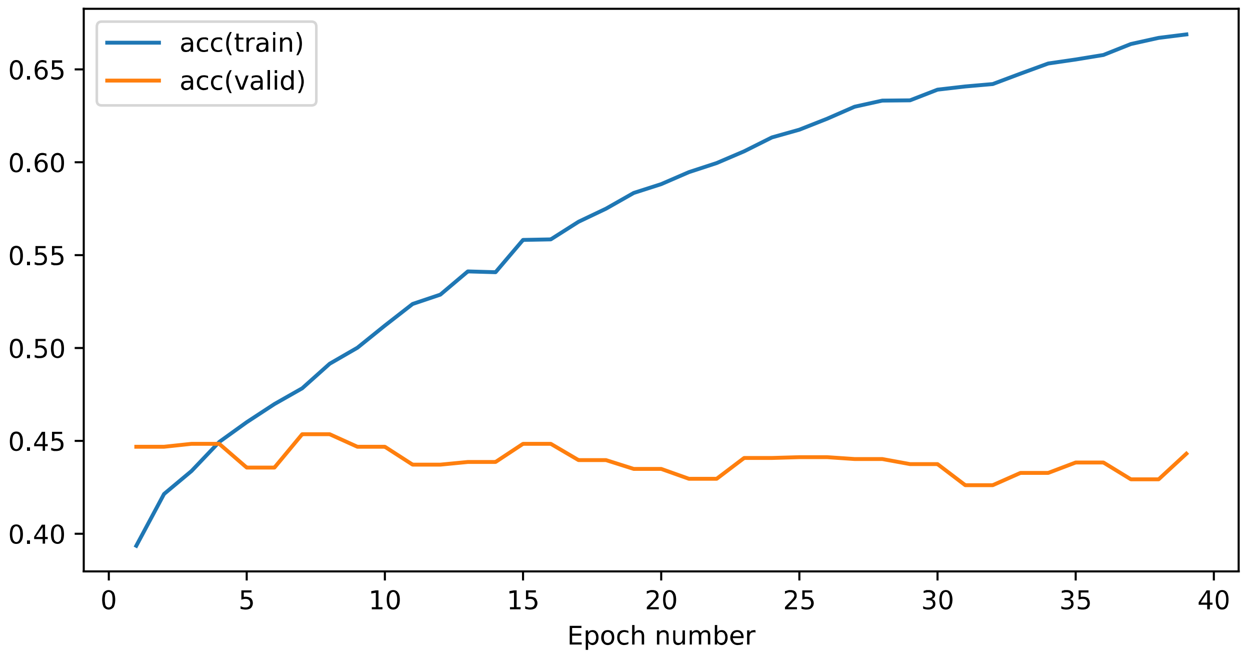
Method: 4 hidden layers of dimensionality 500 regularized with Dropout at all layers

We are using a dropout layer on all layers of our deep neural network.
Dropout probability for the **input is 90%** and
dropout probability for **all the hidden layers is 50%**.
The **Learning Rate** of **Adam Learning Rule** is set at the default of **1e-3**
The new tensorflow graph is as follows:

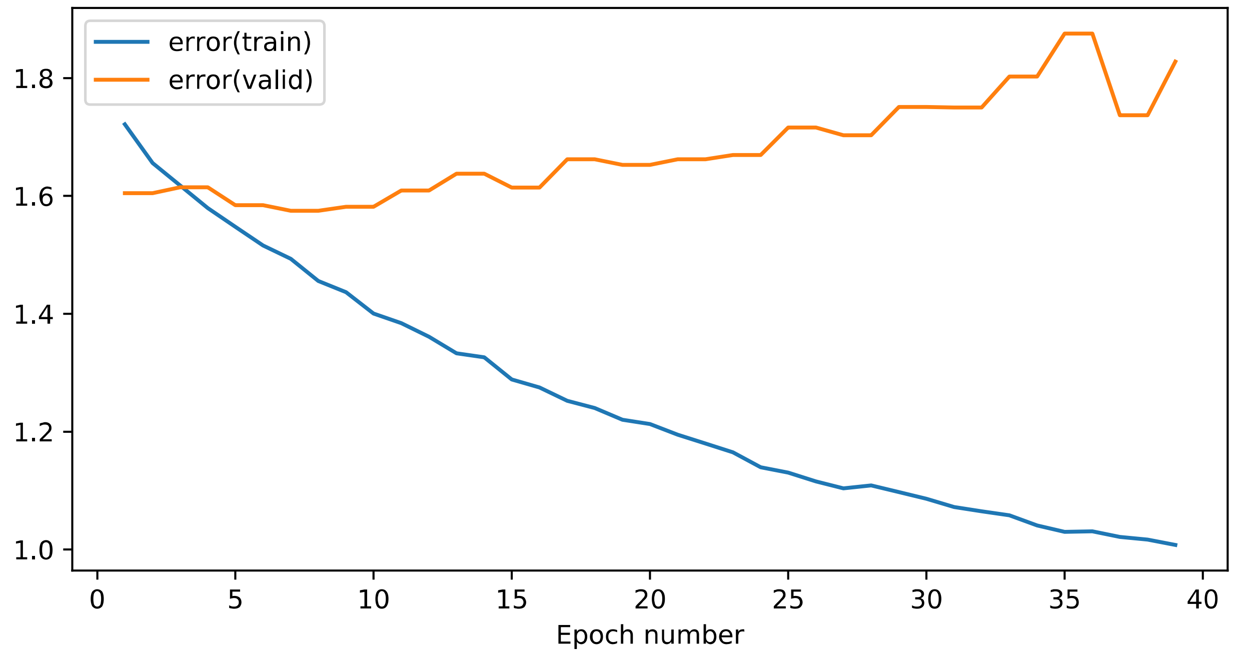


Graph 6: Deep Neural Network - 500 dimensionality at all hidden layers - ReLU non-linearity – Dropout at all layers with keep probabilities as placeholders

Results



Plot 28: Training & Validation Accuracy – Deep Neural Net – 500 dimensionality at all hidden layers – ReLU nonlinearity - Dropout at Input with keep prob 90% - Dropout at all hidden layers with keep prob 50%



Plot 27: Training & Validation Error – Deep Neural Net – 500 dimensionality at all hidden layers – ReLU nonlinearity - Dropout at Input with keep prob 90% - Dropout at all hidden layers with keep prob 50%

Conclusions

We see that by using dropout on all layers and by setting dropout probabilities low enough we have not achieved a very good regularization and the training has become quite slow.

Research Question: Using Tanh Activation Function yields better results?

One alternative to ReLU which has yielded good results in other classification problems is the tanh non-linearity to be used as activation function of the output of the affine layers.

"*For the learning time to be minimized, the use of non-zero mean inputs should be avoided. Now, insofar as the signal vector xx applied to a neuron in the first hidden layer of a multilayer perceptron is concerned, it is easy to remove the mean from each element of xx before its application to the network. But what about the signals applied to the neurons in the remaining hidden and output layers of the network? The answer to this question lies in the type of activation function used in the network. If the activation function is non-symmetric, as in the case of the sigmoid function, the output of of each neuron is restricted to the interval [0,1][0,1]. Such a choice introduces a source of {\em systematic bias} for those neurons located beyond the first layer of the network. To overcome this problem we need to use an antisymmetric activation function such as the hyperbolic tangent function. With this latter choice, the output of each neuron is permitted to assume both positive and negative values in the interval [−1,1][−1,1], in which case it is likely for its mean to be zero. If the network connectivity is large, back-propagation learning with antisymmetric activation functions can yield faster convergence than a similar process with non-symmetric activation functions, for which there is also empirical evidence (LeCun et al. 1991).*"

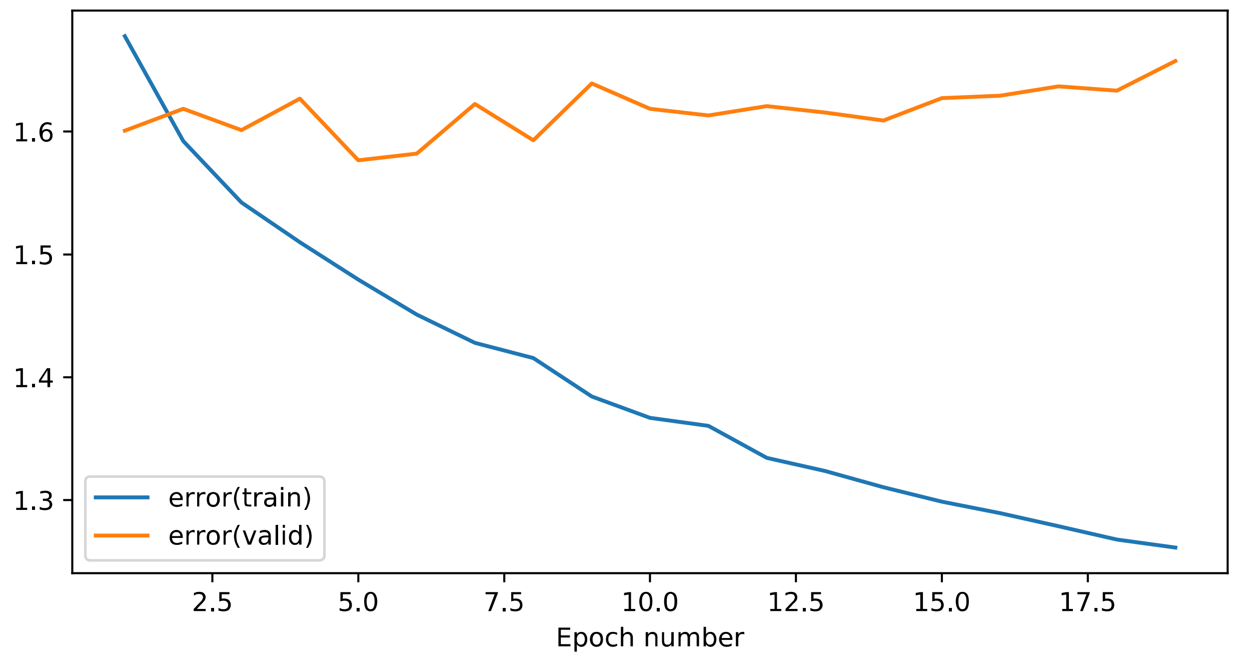
The cited reference is:

Y. LeCun, I. Kanter, and S.A.Solla: "Second-order properties of error surfaces: learning time and generalization", Advances in Neural Information Processing Systems, vol. 3, pp. 918-924, 1991.

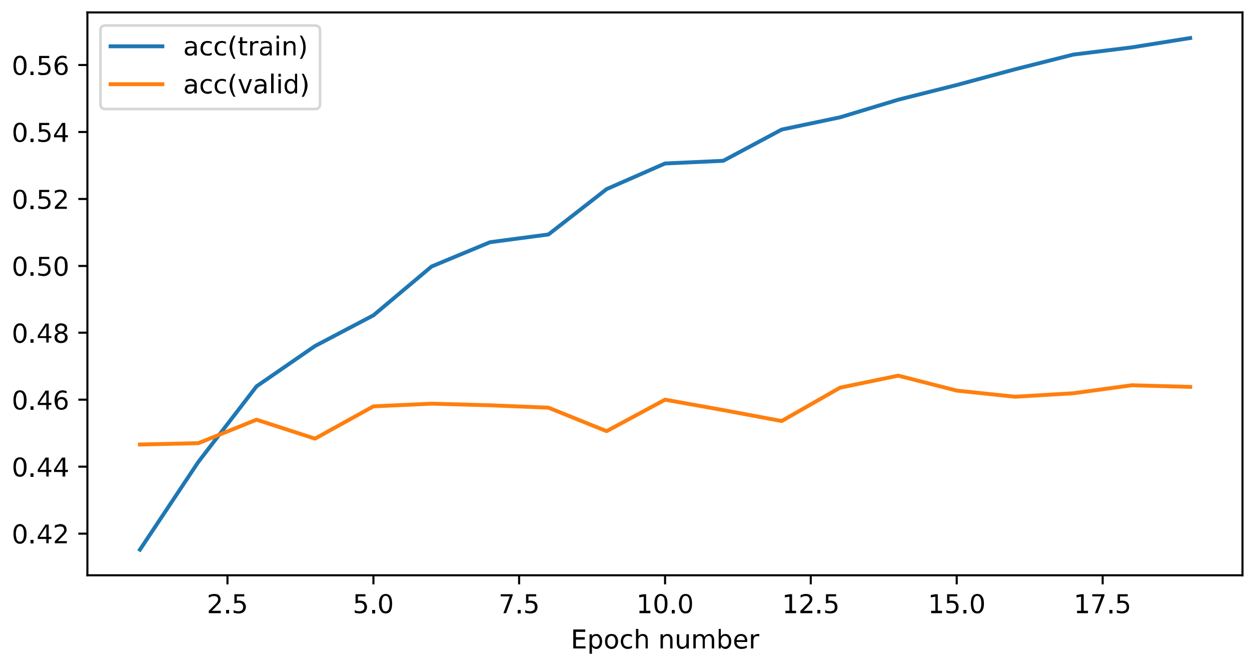
Method: 4 hidden layers of dimensionality 500 regularized with Dropout at all layers and using Tanh as the activation function

We will be using exactly the same architecture as above but all of ReLUs non-linearities are substituted with Tanh activation functions

Results



Plot 29: Training & Validation Error – Deep Neural Net – 500 dimensionality at all hidden layers – Tanh nonlinearity - Dropout at Input with keep prob 90% - Dropout at all hidden layers with keep prob 50%



Plot 30: Training & Validation Accuracy – Deep Neural Net – 500 dimensionality at all hidden layers – Tanh nonlinearity - Dropout at Input with keep prob 90% - Dropout at all hidden layers with keep prob 50%

Conclusions

The results are better. The system seems to behave in a more stable way than before but the learning rate is still slower than expected.
The curve of the validation error is more stable.
The curve of the validation accuracy has a slightly increasing trend.
It seems that overall Tanh is more suited for this classification task. We will use Tanh is all of our experiments below unless noted otherwise.

Research Question: How many hidden layers and how large hidden dimensionalities cause a plateau on the training error graph that is above zero ?

Since the attempt to find the best regularization for a certain set of high dimensionality will take lots of resources timewise, we should instead go back and experiment what would be the maximum number of layers and dimensionalities that would make sense for our classification problem.

Since we do not have a solid metric to check how many layers are too many and how many are just enough we are going to base it on the training error rate. (We will not care about validation at this experiment).

Based on the training error we can consider two things:

- If the error is minimized and the training accuracy is close to 100% within only a few epochs then this means that we could introduce more complexity in the model with larger hidden dimensionalities and/or more hidden layers.

- On the other hand if we see that for a few epochs the error reaches at a plateau then we could consider that this architecture is too large and we should try and reduce the hidden dimensionalities and/or hidden layers.

Method: Multiple layers of fixed dimensionality

We are not using any regularization here.

All of the eight hidden layers have dimensionality of 2000.

To simplify the problem even further we are using simple **Gradient Descent** with **Learning Rate of 1e-1** as our optimizer.

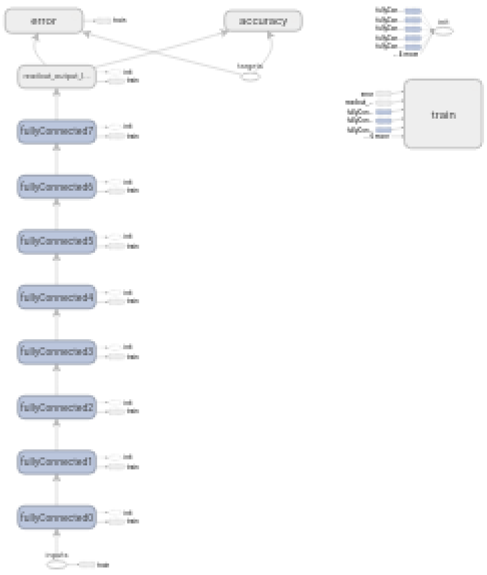
We are trying various different combinations.

Minimum layers are 4 and maximum layers are 8.

Minimum dimensionality of each layer is 500 and maximum is 2000.

We already know that 4 layers with dimensionality of 500 each do not cause a plateau so we are not expecting that smaller architectures will have this effect.

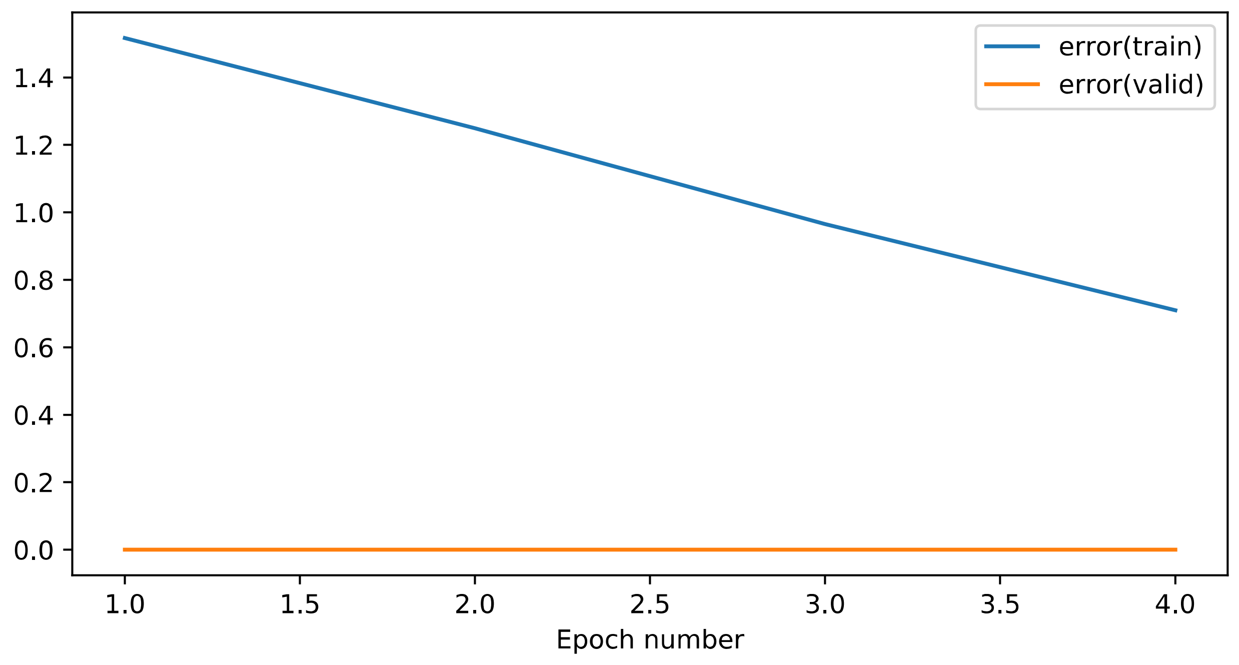
The maximum layers of 8 and the maximum dimensionality for each layer of 2000 are chosen because of restriction on the computational resources



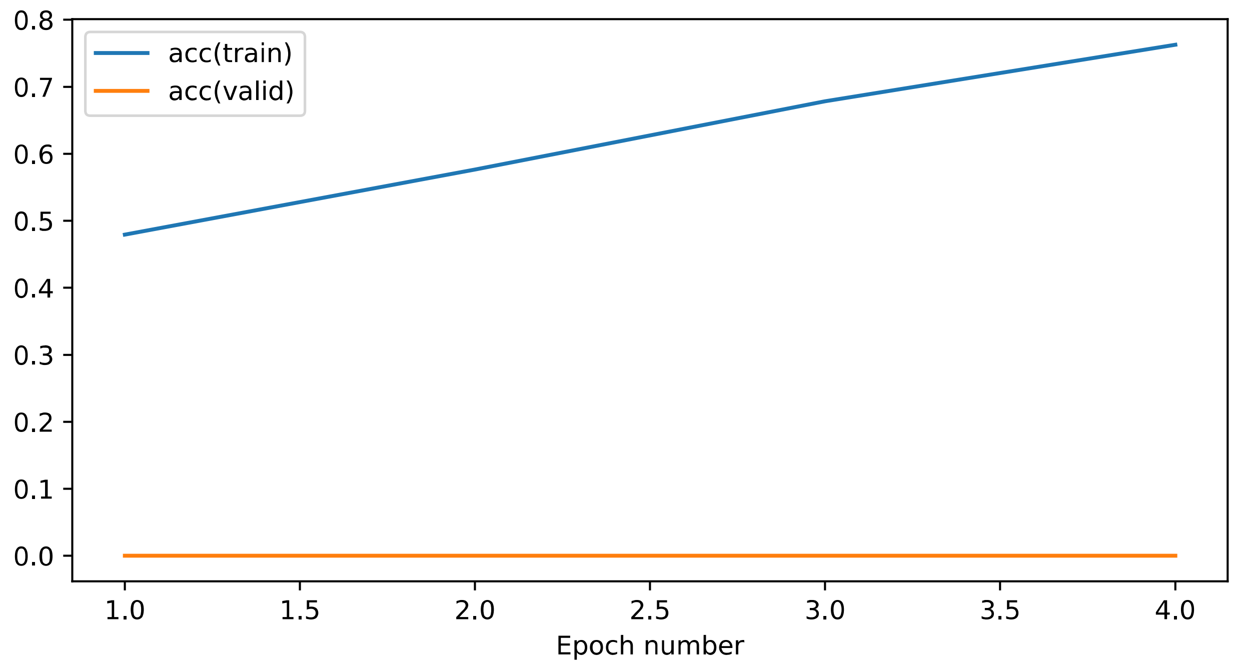
Graph 7: Deep Neural Network – 500-2000 dimensionality at minimum 4 hidden layers and maximum 8 hidden layers – Tanh nonlinearity

Results

We are only showing here the plots of the largest architecture we could implement within our computation restrictions which is 8 layers with 2000 dimensionality for each hidden layer. Only 4 epochs are shown.



Plot 31: Training Error – Deep Neural Net – 8 hidden layers – 2000 dimensionality at all hidden layers – Tanh nonlinearity – (validation error equal to zero because it is not measured)



Plot 32: Training Accuracy – Deep Neural Net – 8 hidden layers – 2000 dimensionality at all hidden layers – Tanh nonlinearity – (validation error equal to zero because it is not measured)

Conclusions

Unfortunately after lots of experiments we have not managed to reach to a plateau within our computational limits. Meaning that using 8 hidden layers of 2000 attributes each was not enough. The Gradient Optimizer with learning rate of 1e-1 was able to train the model. The training was slow and also far from optimal but this experiment did not gave an indication that the neural network would have trouble to train fully if enough epochs were calculated.

Research Question: Does a gradual reduction of the dimensionality of the hidden layers helps the performance or the accuracy?

We want to implement a deep architecture but we do not have the computational resources to have all the layers as wide as we wish. Therefore we need to decrease our dimensionality in a sensible way. Since the input dimensionality is 3000 and the output dimensionality is 10 it makes sense to gradually reduce the dimensionality of the layers asking for less and less neurons to be able and carry the burden of representing the genres of the songs.

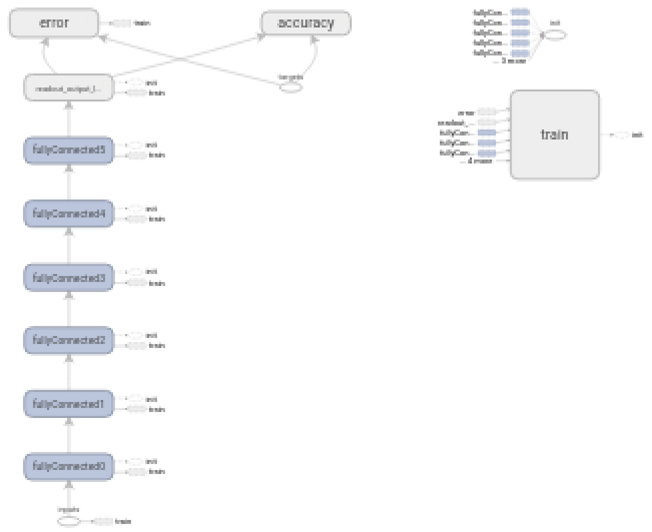
Method: 6 layers of gradually decreasing dimensionality.

The dimensionality starts from 1000 for the first layer and is linearly decreased up to the sixth layer with final dimensionality of 25, and then the readout layer which is 10.
More specifically the dimensionality of the layers from input to ouput are these:

```
3000 → 1000 → 805 → 610 → 415 → 220 → 25 → 10
```

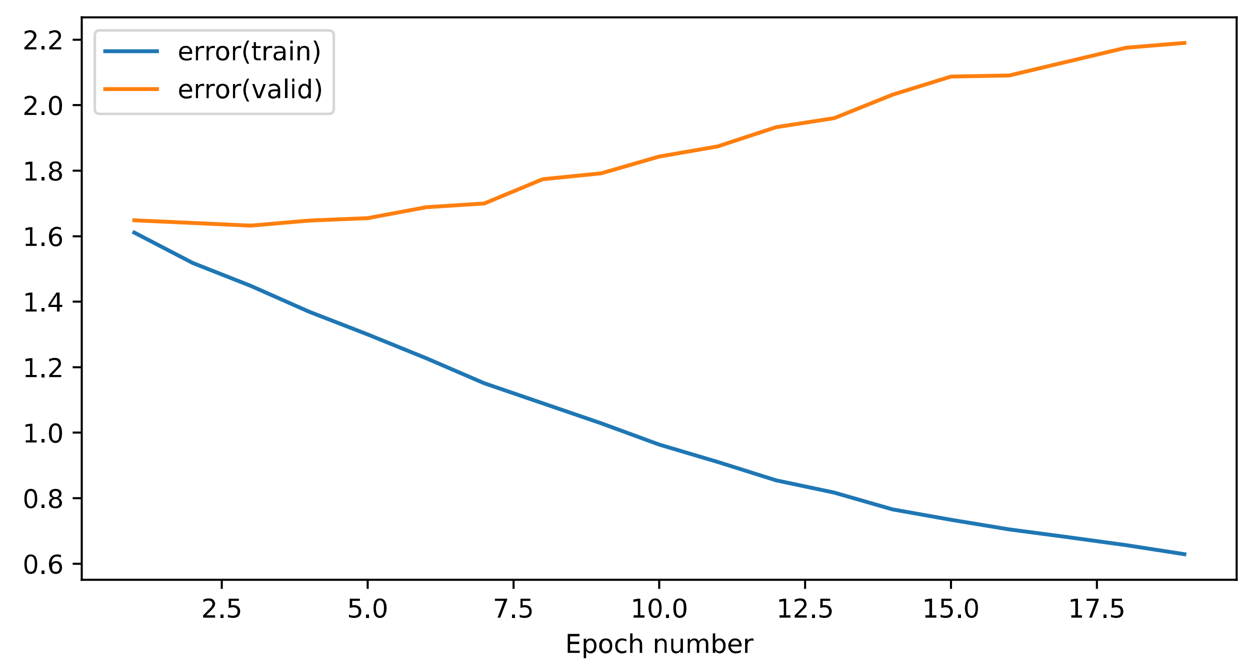
We are not using any regularization yet.

The graph of tensorflow is as follows:

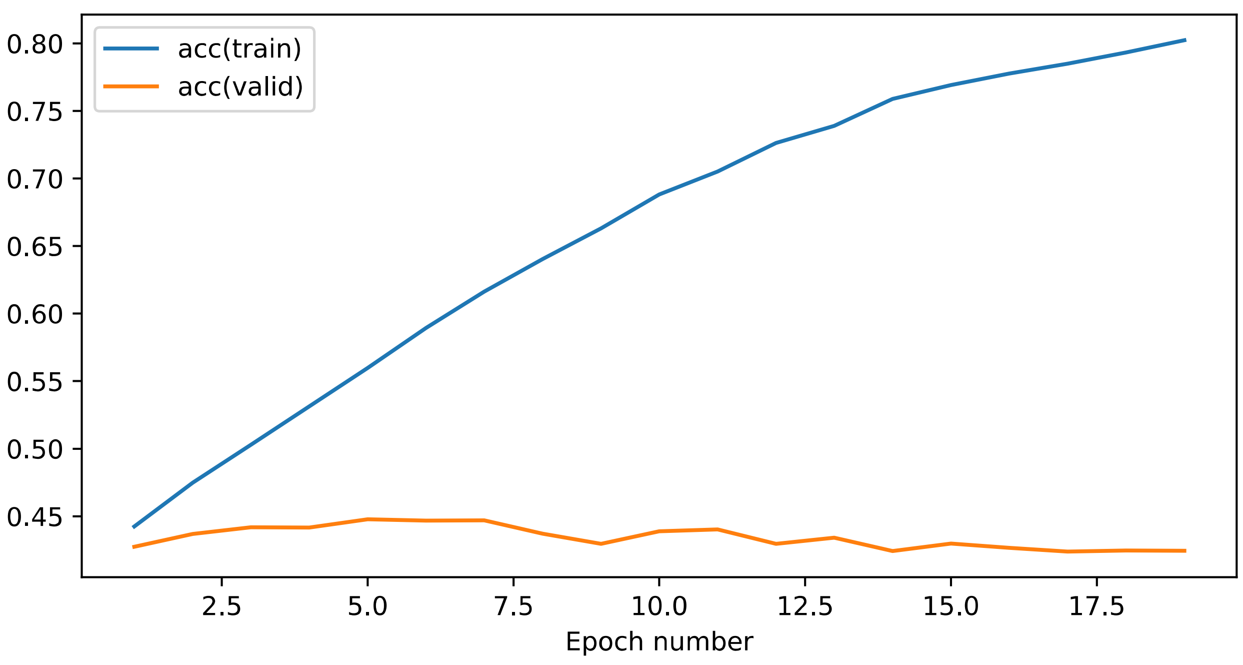


Graph 8: Deep Neural Network – 6 hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25]

Results



Plot 33: Training & Validation Error - Deep Neural Network – 6 hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25]



Plot 34: Training & Validation Accuracy - Deep Neural Network – 6 hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25]

Conclusions

We should note that the deep neural network has an improved accuracy to begin with which these are signs of a better classifier. As the epochs go by the overfitting effects are apparent which is expected.
We will use these results as a basis for our deep neural network.

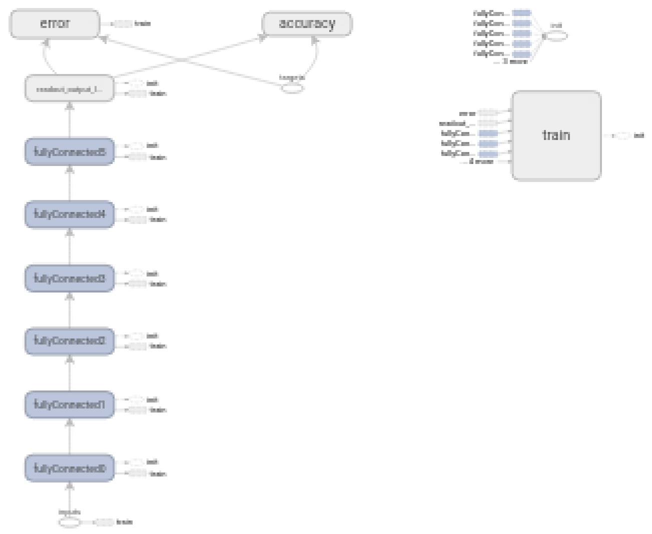
Research Question: Will pretraining yield better results for the deeper neural network?

The first attempt for pretraining did not yield any wanted results at the shallow neural network. Now that we have a deeper neural network we might get some added value from pretraining.

Method: Stacked Autoencoder Pretraining for all layers

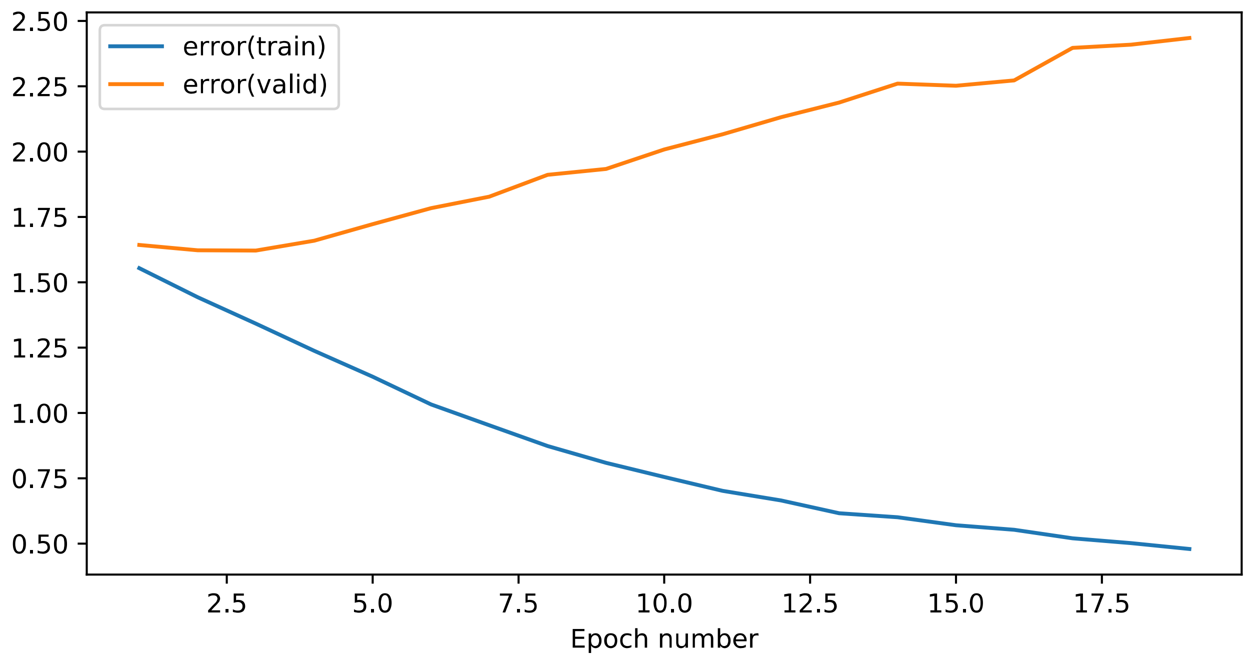
We are using the exact same methodoly we used above but now the stacked autoencoder is executed iteratively for multiple layers

The tensorflow graph is available below:

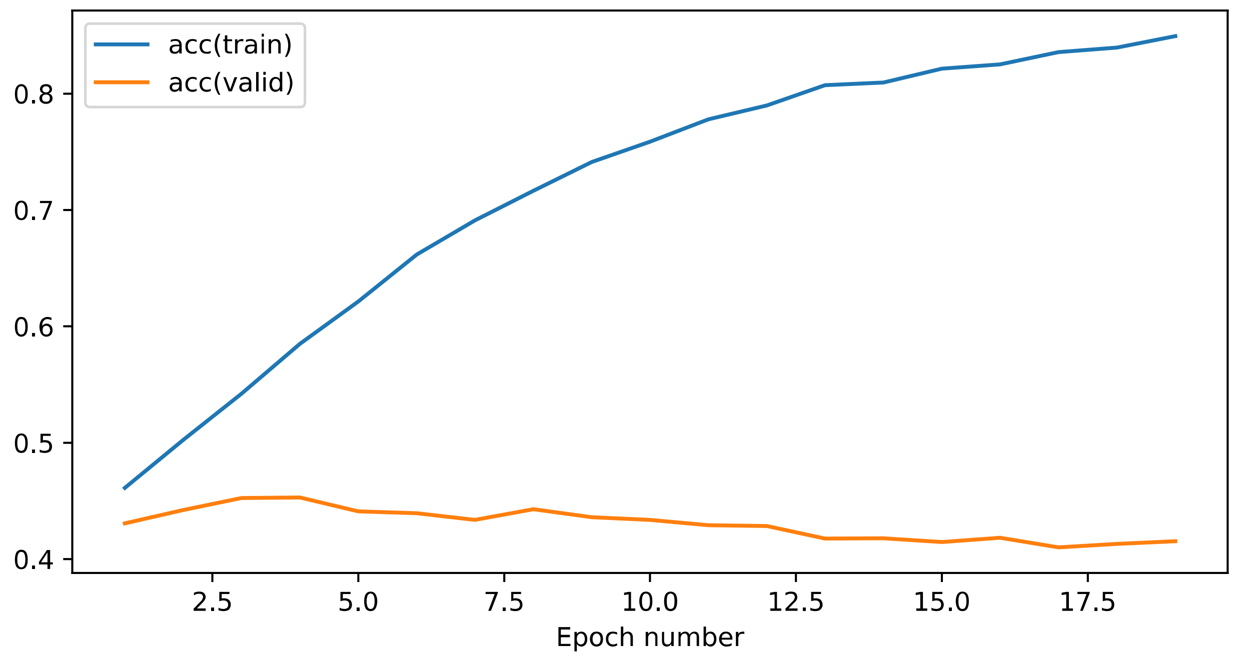


Graph 9: Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder

Results



Plot 35: Training & Validation Error - Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder



Plot 36: Training & Validation Accuracy - Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder

Conclusions

After pretraining we get a maximum of ~45% which is not different than before.

Of course after a few epochs due to overfitting the accuracy is gradually decreasing.

The training is low with maximum training accuracy of ~80% after 20 epochs.

Research Question: Will Batch Normalization increasing the learning rate of the Deep Neural Network?

As the data goes through the neural network the weights and biases are adjusted making them too big or too small. This is called the “internal covariate shift”. So by normalizing the data in each batch the problem is avoided.

This helps the learning rate because the weights do not have large and noisy fluctuations. In most cases you also get a higher accuracy as well.

Method: Batch Normalization in all layers with scaling and shifting factors

We are applying batch normalization to all outputs of the affine layers in order to have normalization all over the place.

Note that we are including a scaling and a shifting factor which are both trainable so that we will not confine our representations to only normalized values.

Here the graph that includes batch normalization is slightly different than before because it includes the **tf.cond** which is a way of tensorflow to express an if-statement in the graph according to a boolean placeholder.

We recall that batch normalization works differently on training and validation/testing scenario. We need to use the batch mean and the batch variance during training while we need to use the population mean and population variance at validation/testing cases.

The population mean and variance are computed as the exponential moving average of the batch mean and variance accordingly. They are derived from the following formulas:

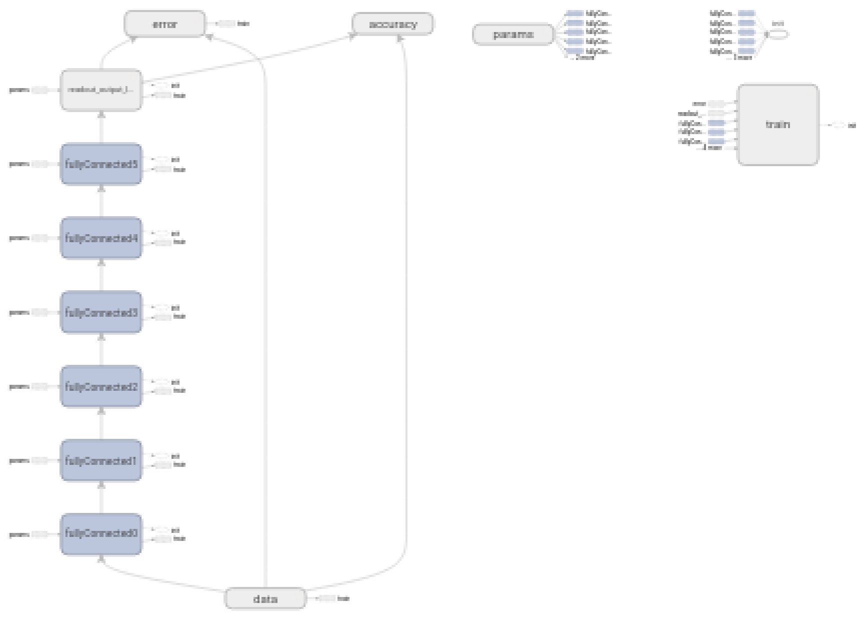
$$population\,mean = population\,mean * 0.999 + batch\,mean * 0.001$$

$$population\,variance = population\,variance * 0.999 + batch\,variance * 0.001$$

The decay factor is set to 0.999 to give a higher weight at the population mean and lower weight at the batch mean. This works ok for cases where we have lots of batches like this one.

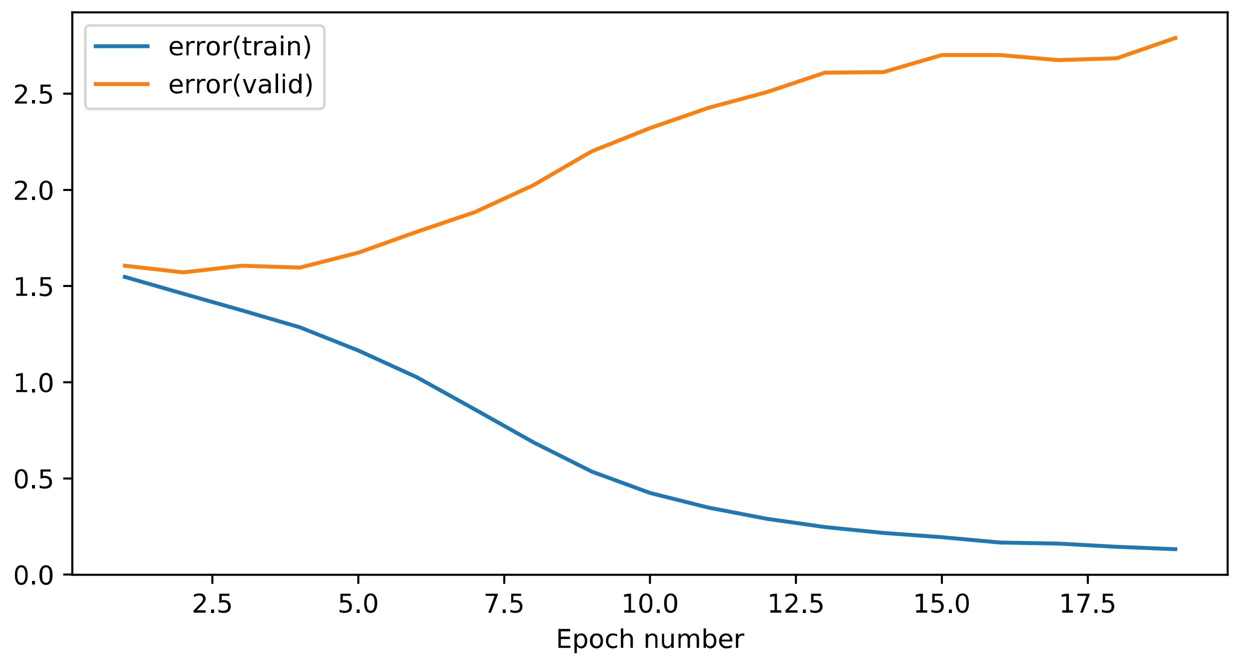
Note that the population mean is initialized to zeros and population variance is initialized to ones.

Here is the tensorflow graph:

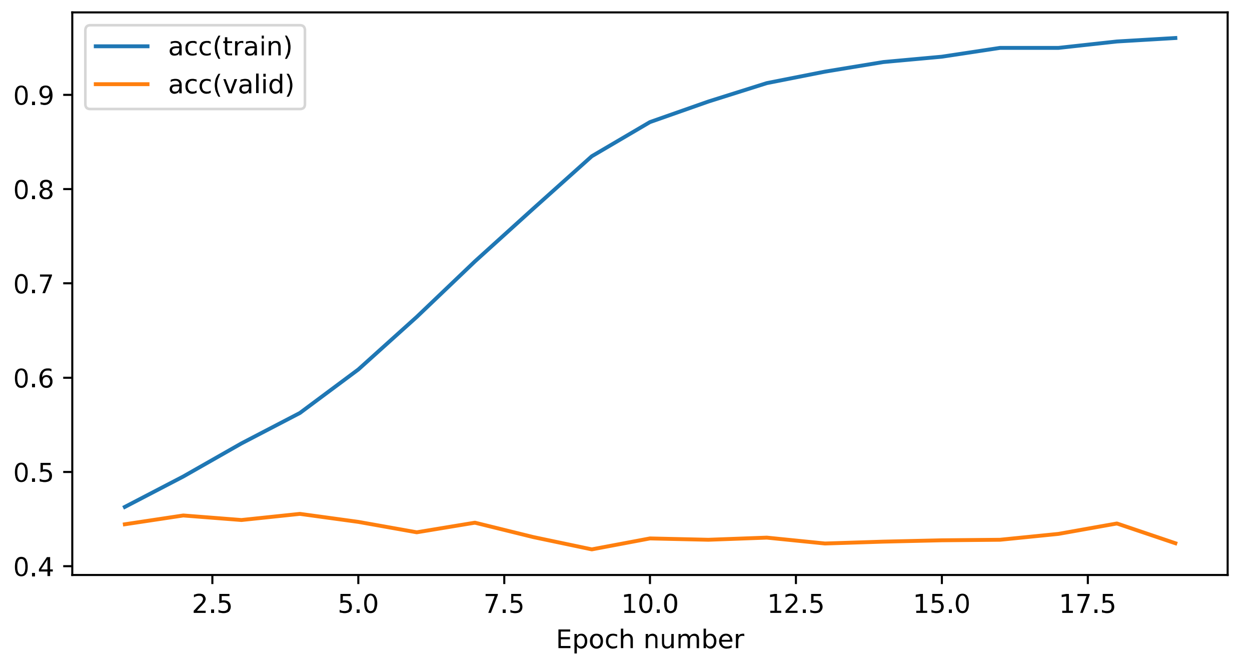


Graph 10: Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder - Batch Normalization on all layers

Results



Plot 37: Training & Validation Error - Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder - Batch Normalization on all layers



Plot 38: Training & Validation Accuracy - Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder - Batch Normalization on all layers

Conclusions

The results of this experiment were as expected and wanted.
Batch normalization helped the learning rate, resulted in higher overall training accuracy within 20 epochs and the validation accuracy peaked at a higher level, before the overfitting effects reduce it.

Research Question: Will Dropout help increase the maximum validation accuracy and regularize the deep neural network?

We are attempting to use Dropout as our only means for regularization.

Method: Dynamic Dropout in all layers

We are using the exact same algorithm we used before for dynamically configuring the dropout probabilities of the input and hidden layers.

The initial dropout probabilities are 50% for all layers.

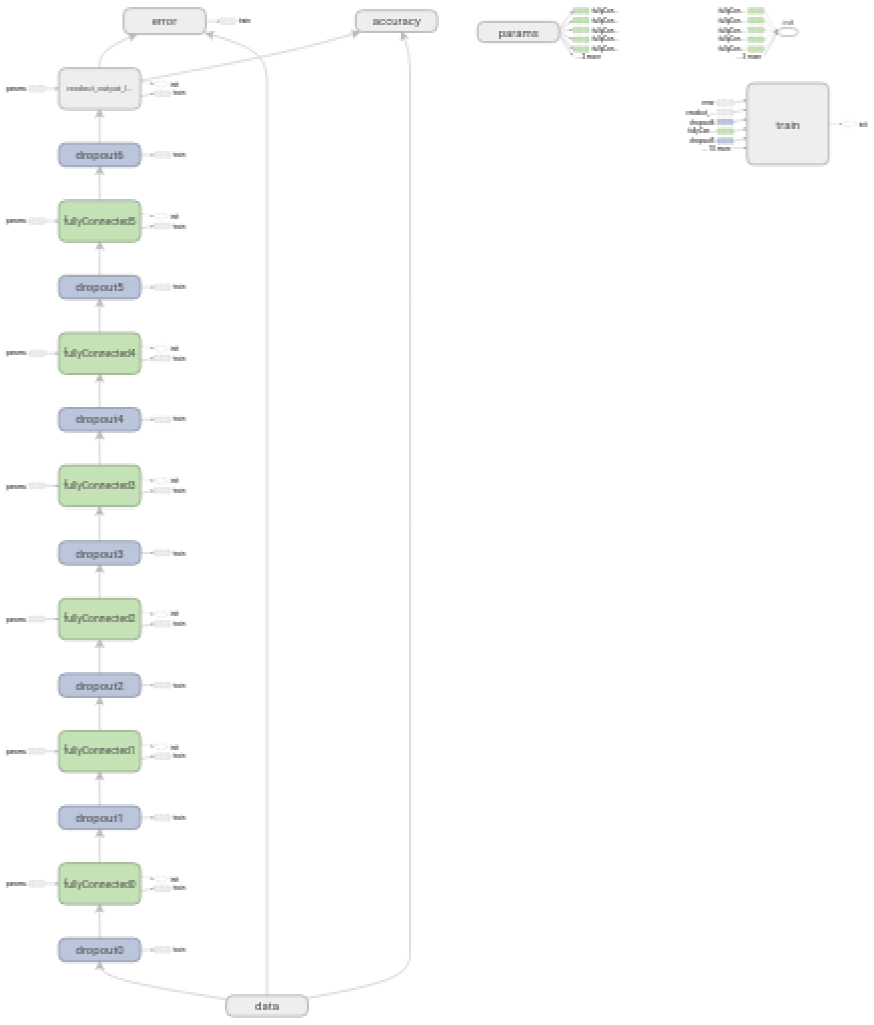
The minimum dropout probabilities are 50% for the input layer and 10% for all the other layers

Maximum dropout probability for all layers is 100%.

Resolution hyperparameter is 10

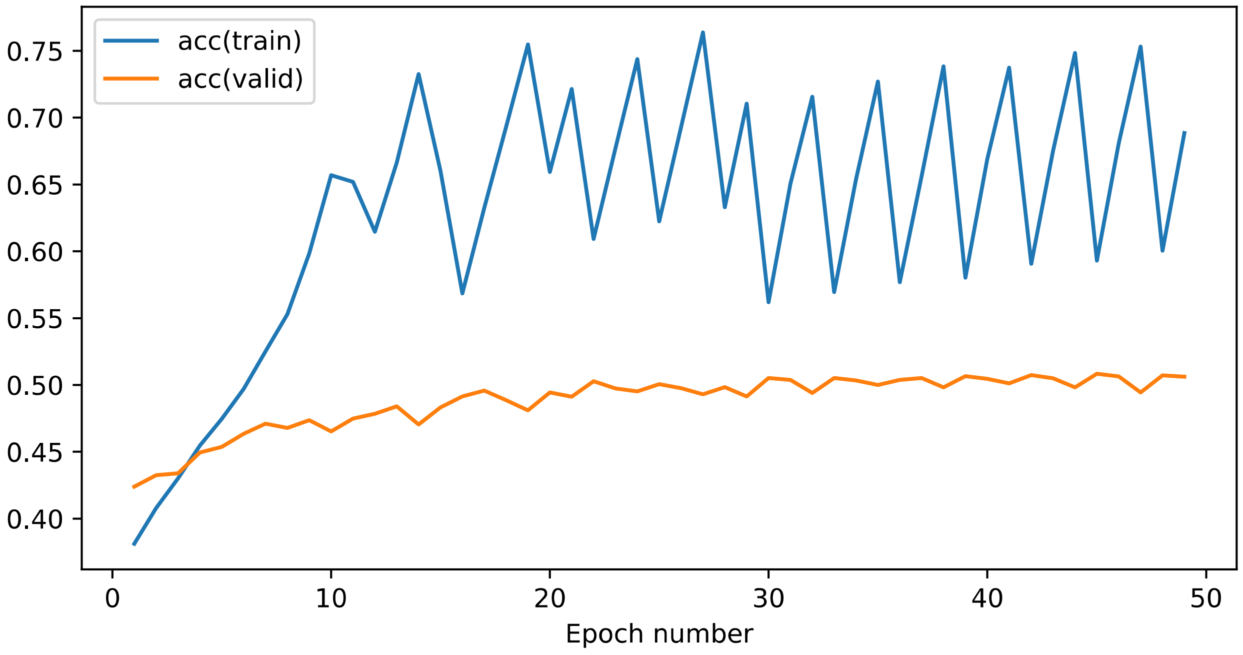
Gamma hyperparameter is 0.6.

The tensorflow graph is as follows:

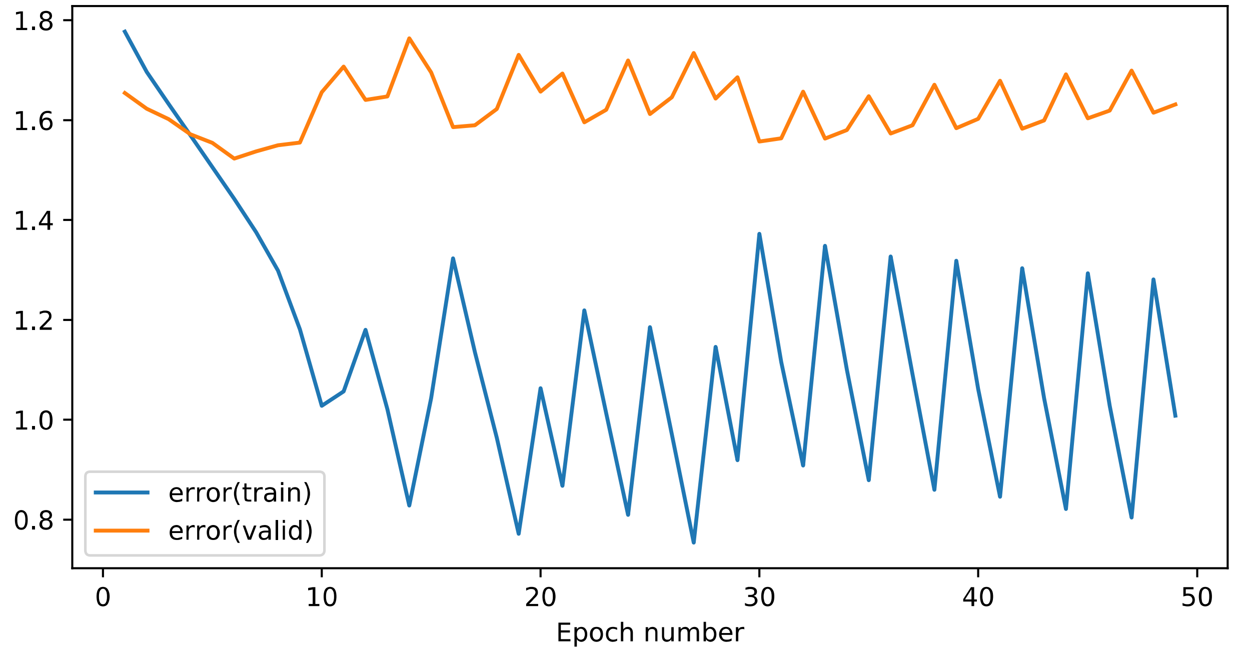


Plot 39: Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder - Batch Normalization at all layers – Dropout layer before all layers with keep probabilities as placeholders

Results



Plot 41: Training & Validation Accuracy - Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder - Batch Normalization at all layers – Dropout layer before all layers with keep probabilities dynamic – min input prob 50% - min hidden prob 10% - Resolution hyperparameter 10 - Gamma hyperparameter 0.6



Plot 40: Training & Validation Error - Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder - Batch Normalization at all layers – Dropout layer before all layers with keep probabilities dynamic – min input prob 50% - min hidden prob 10% - Resolution hyperparameter 10 - Gamma hyperparameter 0.6

Conclusions

Due to the varied dropout probability caused by our dynamic algorithm we can see that the validation error has remained constant, on average, for the 50 epochs without any trend of increase and the 50 epochs.

We can also see that the validation is on average slowly increasing until epoch ~30 and then remained still at ~50%. It peaked at 50.8% which is the largest accuracy value we have seen so far.

Research Question: Is L2 regularization going to be stable to regularize the deep neural network?

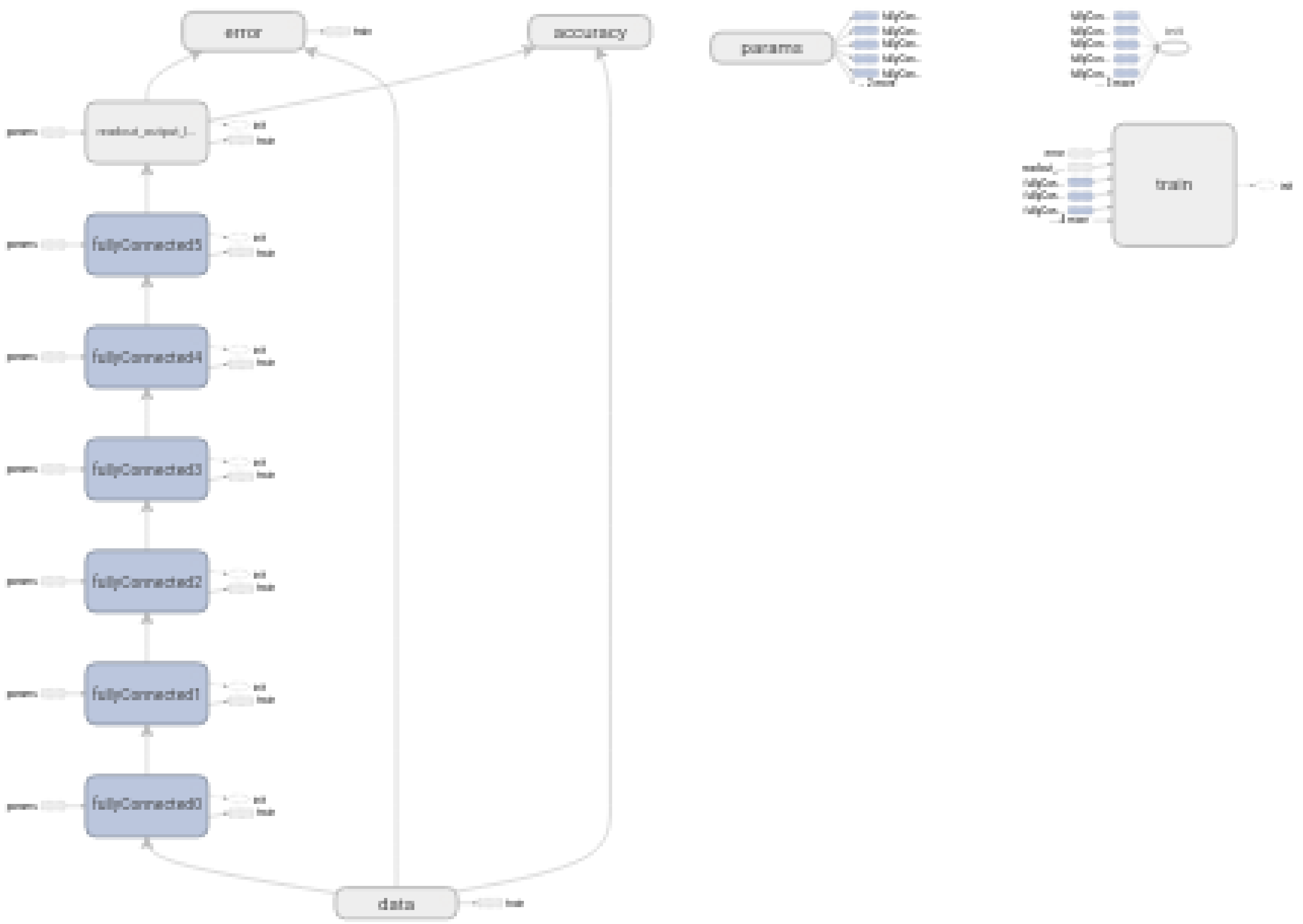
So far we have only considered Dropout as our means of regularization. Next we will check if L2 regularization is useful as well.

Method: L2 regularization

We are using the architecture we used above but we have removed all the dropout layers and the source code related to dynamic dropout.

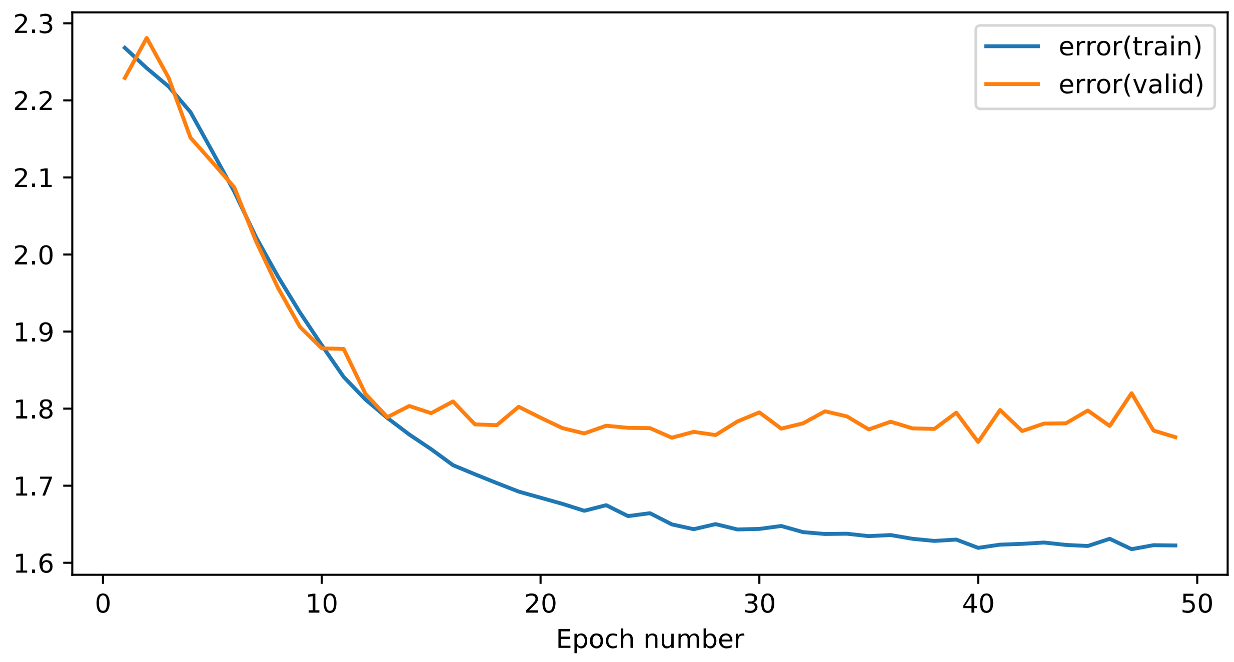
Instead we have got the L2 loss for all of the weights of the fully connected layers of our deep neural network.
No regulation for the biases.
This loss is added to the total loss/error that is passed as input to our Adam Optimizer.
The L2 losses of all the weights are multiplied by a common **lambda factor equal to 1e-2**.

Tensorflow graph is as follows:

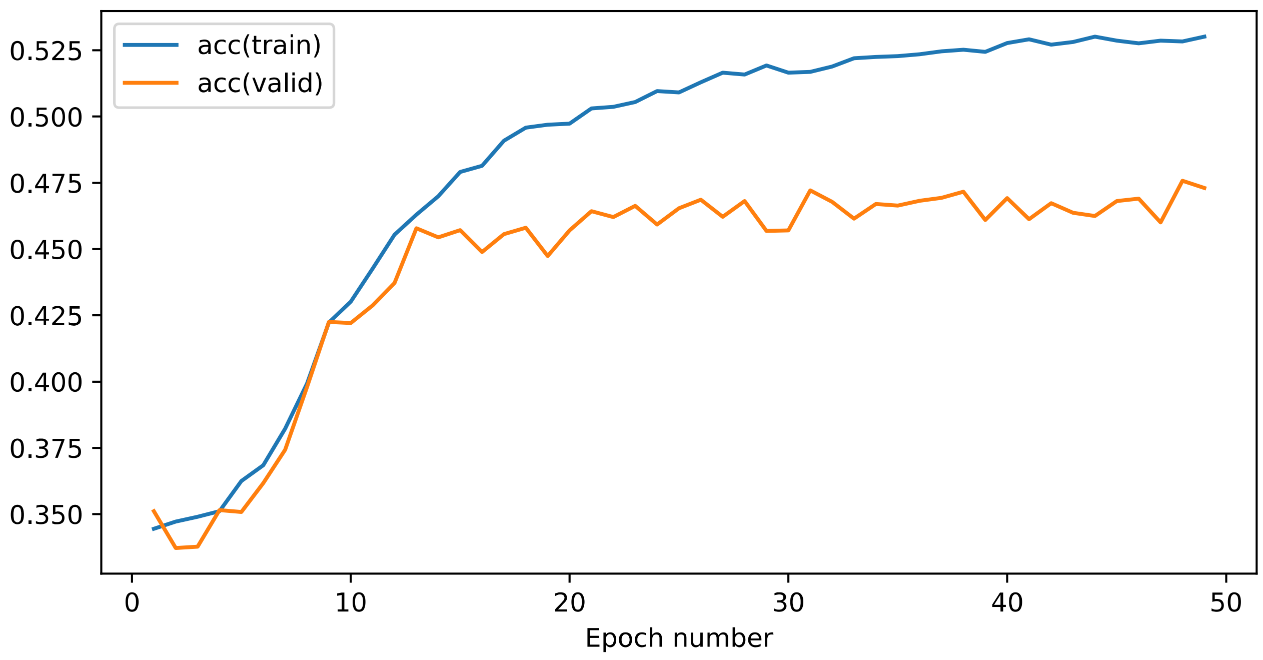


Graph 11: Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder - Batch Normalization at all layers – L2 regularization with factor 1e-2

Results



Plot 42: Training & Validation Error - Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder - Batch Normalization at all layers – L2 regularization with factor 1e-2



Plot 43: Training & Validation Accuracy - Deep Neural Network – six hidden layers – Tanh nonlinearity – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Pretraining with Stacked Autoencoder - Batch Normalization at all layers – L2 regularization with factor 1e-2

Conclusions

L2 regularization is successful when the lamda2 hyperparameter is set to 1e-2.

It is noticable how the validation error is maintained within a certain region, even if there are small oscillations, and the validation accuracy reaches to a steady value of 47%, again with some oscillations.

Protocol followed for the MSD-10 classification task:

Through the process of trying to achieve better classification accuracy for the MSD-10 classification task we have followed a certain protocol:

- We have first pretrained the network with Stacked Autoencoder to start from a good place.
- Then we picked a deep neural network with dimensionality that was slowly decreasing.
- We replaced ReLU with Tanh as our activation functions.
- We introduced Batch Normalization at the output of each affine layer.
- We introduced Dropout layers before the input of all hidden layers with a shared keep-probability. We also used a typically higher Dropout probability for the input layer.
- We used an algorithm to calibrate the dropout probabilities according to the validation error in order to achieve the best possible dropout probabilities that would keep the validation error as low as possible.

Research Question: What kind of results will the current protocol have on the MSD-25 classification task?

We are using the same protocol that we used for MSD-10 task.

Method: Deep Neural Network with 8 layers and same protocol as MSD-10 classification task

For MSD-25 we will follow the same protocol as we did for MSD-10 but we will use a deeper architecture since we expect that a more rich model will be needed to represent the 25 genres.

The hidden dimensionality from input to ouput layer by layer is as follows:

```
3000 → 1000 → 860 → 721 → 582 → 442 → 303 → 164 → 25
```

The dynamic dropout used here has the following hyperparameters:

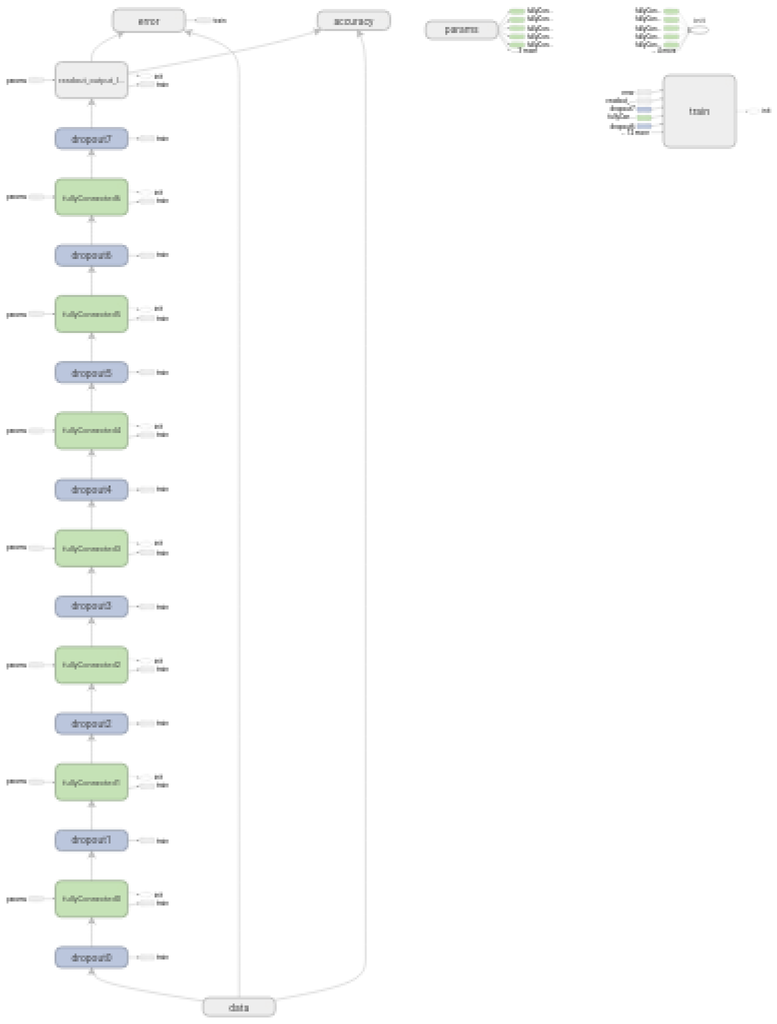
Minimum Input Dropout Probability: 50%

Minimum Hidden Dropout Probability = 10%

Resolution = 10

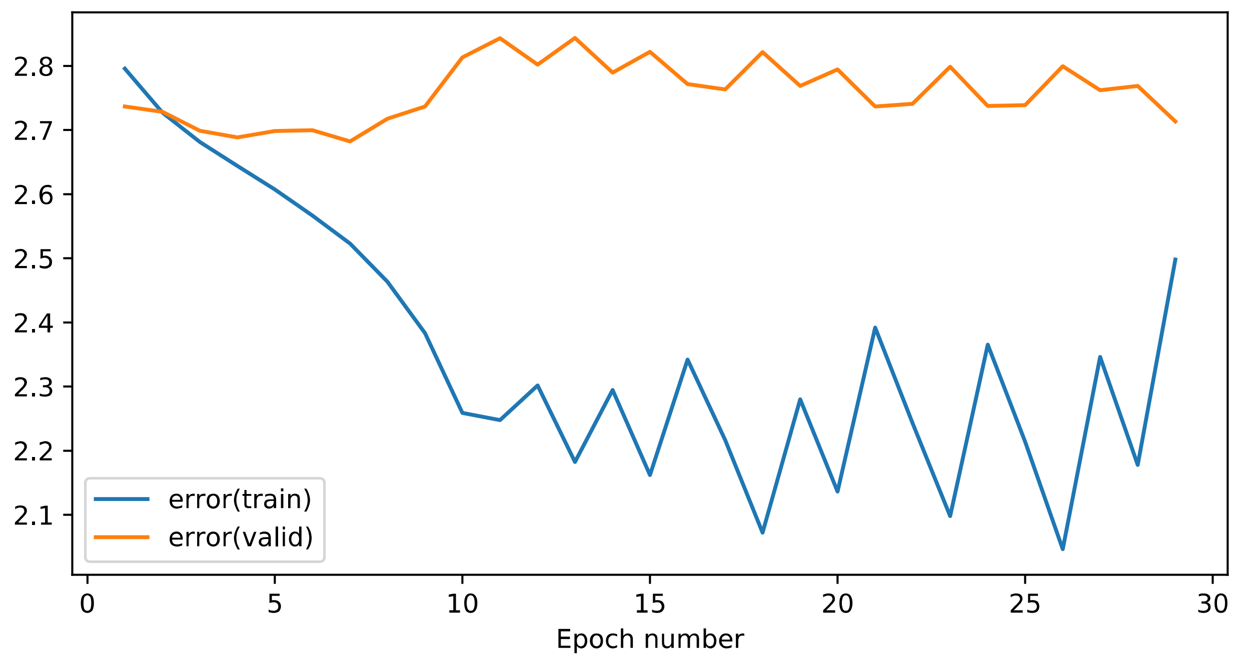
Gamma = 0.6

The tensorflow graph is available below:

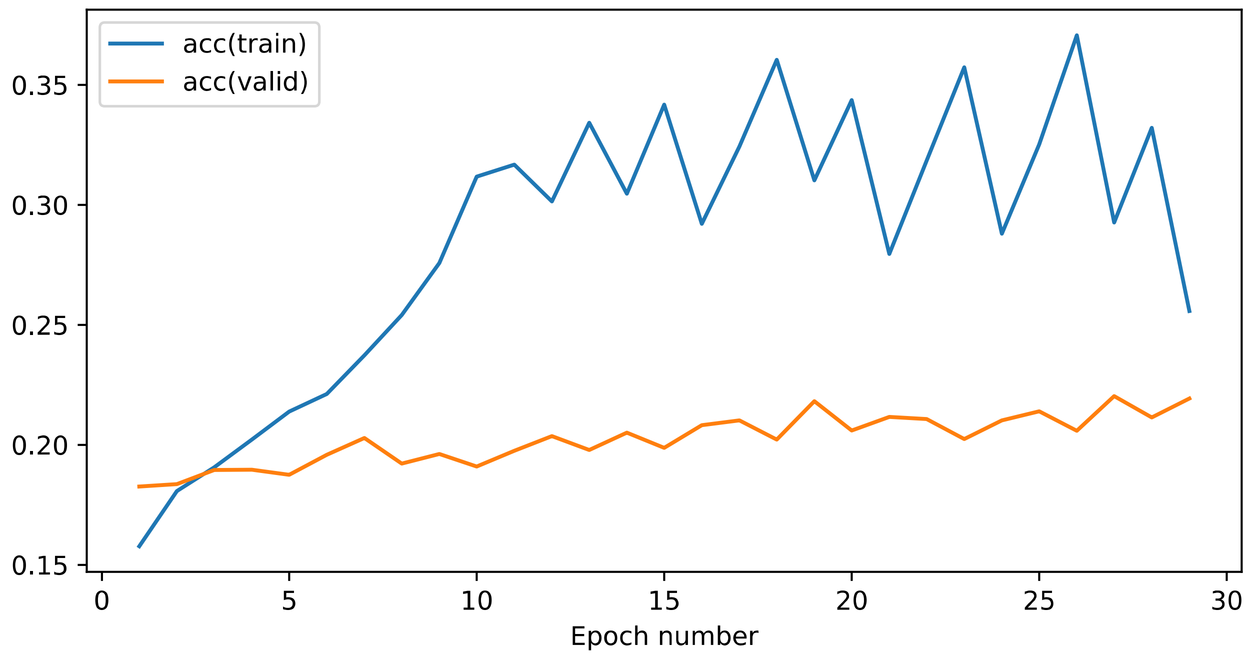


Graph 12: Deep Neural Network – 8 layers – Hidden dimensionalities [1000, 860, 721, 582, 442, 303, 164] – Tanh nonlinearity – Pretraining with Stacked Autoencoder – Batch Normalization at all layers – Dropout at all layers with keep probabilities as placeholders

Results



Plot 44: Training & Validation Error - Deep Neural Network – 8 layers – Hidden dimensionalities [1000, 860, 721, 582, 442, 303, 164] – Tanh nonlinearity – Pretraining with Stacked Autoencoder – Batch Normalization at all layers – Dynamic Dropout with min input keep prob. 50% and min input keep prob 10% - Gamma hyperparameter 0.6 – Resolution hyperparameter 10



Plot 45: Training & Validation Accuracy - Deep Neural Network – 8 layers – Hidden dimensionalities [1000, 860, 721, 582, 442, 303, 164] – Tanh nonlinearity – Pretraining with Stacked Autoencoder – Batch Normalization at all layers – Dynamic Dropout with min input keep prob. 50% and min input keep prob 10% - Gamma hyperparameter 0.6 – Resolution hyperparameter 10

Conclusions

We see that the classification of the 25 genres is a much more difficult task for an even deeper neural network. More computational power is required and the learning rate is slower.

More resources are needed timewise as well because we should let the algorithm run for longer periods to get satisfactory results.

Thankfully our protocol has performed well in terms of regularization since we see that the validation error has remained stable.

The maximum validation accuracy recorded is ~22% which is less than the half for the validation accuracy of the simpler MSD-10 classification task.

Research Question: Will pretraining with the parameters of a model optimized for a similar classification task yield better results?

We are exploring another kind of pretraining based on neural networks we have trained before.

Method: Pretraining using the model parameters of the MSD-10 classifier

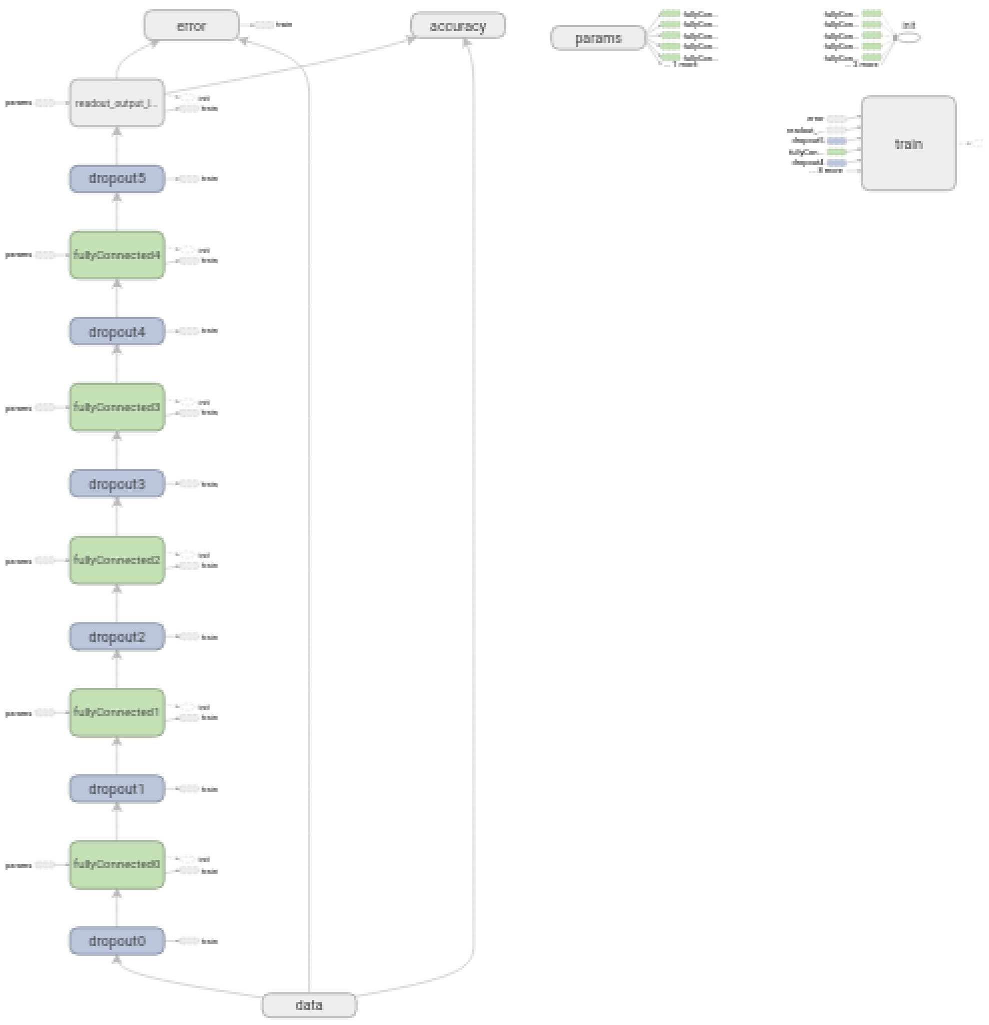
Here we are going to use the same neural network as above and the same protocol but we are pretraining it by asking it to solve the MSD-10 classification task first.

All the variables from all the layers of the weights, biases, along with the trainable parameters of batch normalization (beta offset and scale gamma) are used to pretrain/initialize the layers of the MSD-25 classifier.

Note that the current experiment is not in a direct comparison with the above experiment because we are using a different architecture of 6 layers, as opposed with the 8 layers used above. This is because we had optimize the msd-10 classification task for six layers.

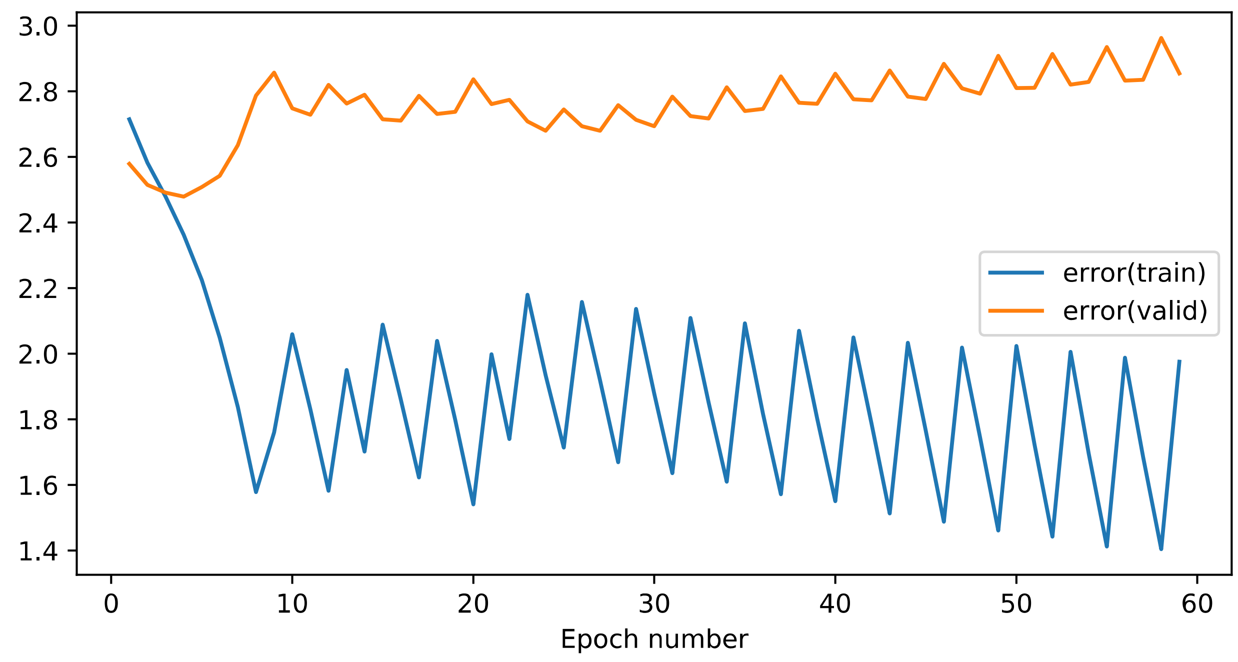
So layer dimensionalities are these: `3000 → 1000 → 805 → 610 → 415 → 220 → 25`

The tensorflow graph is as follows:

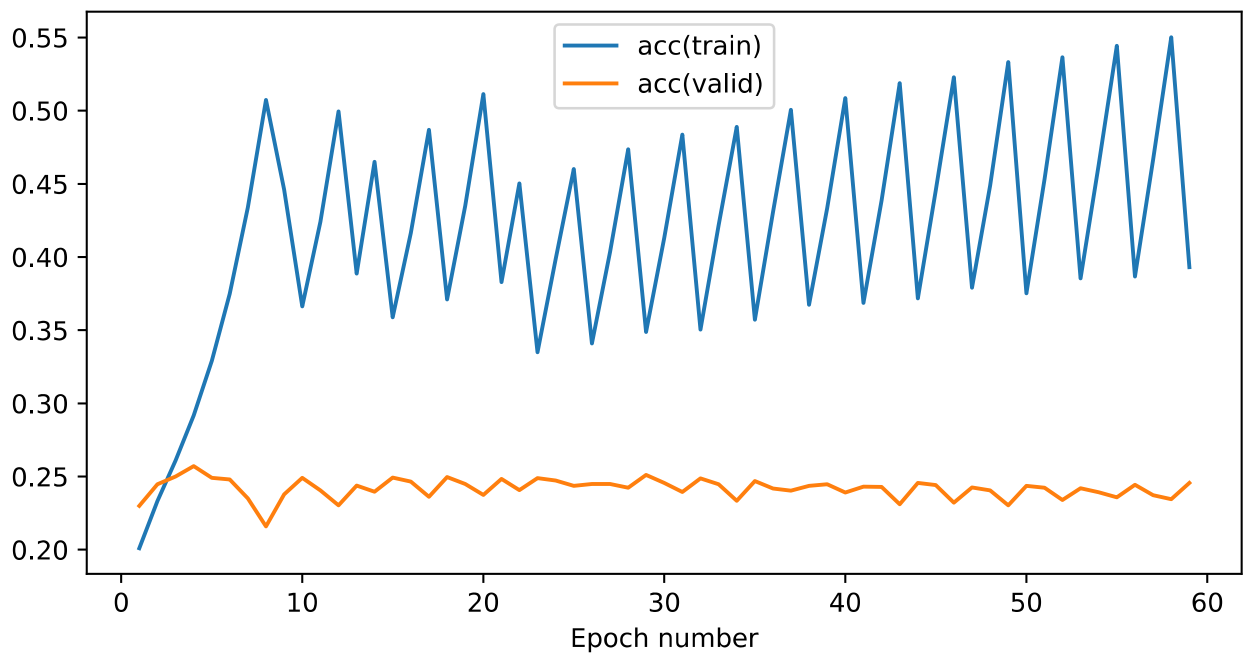


Graph 13: Deep Neural Network – 6 layers – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Tanh nonlinearity – Batch Normalization at all layers – Dropout at all layers with keep probabilities as placeholders - Pretraining with model params of MSD-10 classification task

Results



Plot 46: Training & Validation Error - Deep Neural Network – 6 layers – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Tanh nonlinearity – Batch Normalization at all layers – Dynamic Dropout with min input keep prob. 50% and min input keep prob 10% - Gamma hyperparameter 0.6 – Resolution hyperparameter 10 – Pretraining with model params of MSD-10 classification task



Plot 47: Training & Validation Accuracy - Deep Neural Network – 6 layers – Hidden dimensionalities [1000, 805, 610, 415, 220, 25] – Tanh nonlinearity – Batch Normalization at all layers – Dynamic Dropout with min input keep prob. 50% and min input keep prob 10% - Gamma hyperparameter 0.6 – Resolution hyperparameter 10 – Pretraining with model params of MSD-10 classification task

Conclusions

Pretraining by model parameters of MSD-10 classifier worked better than expected. We started with a very good validation accuracy and we peaked over 25% which is the best we have seen so far for MSD-25 classification task.

However our dynamic dropout algorithm did a poor job on converging and we can see an overall increasing trend of the validation error, meaning that it did not work as an optimal regularizer.

Further Work

- Even though our simple algorithm for dynamic dropout yield some promising results, it includes a lot of resources both timewise and computationally to make it converge. So for final coursework 4 we will not continue using it. Instead we will try and combine L2 regularization with dropout of static keep-probability and use that as our baseline.
- We are going to build our own custom Data Provider in order to exploit the song data with segments of **variable length**. We will utilize **Recursive Neural Networks** to build a classifier for the variable length segments.
- We will try and use **Curriculum Learning** in order to train first the most easier to be classified samples and then to train the more “difficult” ones.
- Given that the MSD-25 classes are a specialization of the more generic classes of the MSD-10, we will train a network for the MSD-25 classification task and then use the learned classes to derive the simpler classes of the MSD-10 classification task and see how well our classifier performs then.
- More experiments to drive the classification accuracy of unseen data/songs as high as possible

Tensorflow & Python Technical Details Appendix

- Pretrained model variables are saved in `.npz` files with the `np.savez` method whenever necessarry and then loaded with the `np.load` method
- We are not using the tensorflow default graph, rather we are creating our own graph and use that on each experiment via the `graph = tf.Graph()` command
- `batch_norm.py` file contains functionality useful for batch normalization
    - `batchNormWrapper_byExponentialMovingAvg` is a wrapper function which sets four variables. Two trainable variables will be our *beta offset* and the *scale gamma* which are added and multiplied respectively. We use two more variables for the population mean and variance and exponential moving average is used to approximate it from the mean and variance of the batches. Note that the `with tf.control_dependencies` is used to force tensorflow to calculate the mean and variance before proceeding with calculating the actual batch normalization step
    - `fully_connected_layer_with_batch_norm` is a function that created a fully connected affine and a nonlinearity putting a batch normalization layer in between
- `jupyter_notebook_helper.py` file contains functionality for plotting graphs and plots embedded in jupyter and in tensorbard
    - `show_graph` is the function provided to show a tensorflow graph inside jupyter notebook

    - `getRunTime` is a function to calculate the duration of the execution of a callback function passed as a parameter
    - `getWriter, getTrainWriter` and `getValidWriter` are three functions that work with the convention that the log directory for summaries for tensorboard will be the concatenation of an arbitrary path folder, the timestamp with second precision and a key value
    - `initStats` is a function to initialize the statistics we collect on every epoch
    - `gatherStats` is a helper function to set each passed parameter to the right index at the two dimensional `stats` array
    - `plotStats` is a function to render the statistics as a line plot inside jupyter notebook
- `py_helper.py` file contains a collection of functions usually generically useful for any python program
    - `merge_dicts` is a helper function to easily merge an arbitrary number dictionaries. Key conflicts are solved by giving precedence to the dictionaries that are of lower order in the stack. For example the dictionary at the tail has higher precedence than the dictionary before that etc.
- `tf_helper.py` file contains methods generally useful across tensorflow applications
    - `tfRMSE` calculates the Root Mean Sum of Square Residuals
    - `tfMSE` calculates the Mean Sum of Square Residuals
    - `fully_connected_layer` adds to a tensorflow graph a combinations of an affine layer along with a nonlinearity
    - `trainEpoch` and `validateEpoch` are helping to reduce the boilerplate code for training and validating a neural network for an entire epoch. Both functions take into account the keep probabilities of dropout layers that could be used as placeholders and an extra feed dictionary for passing extra parameters.
- `stacked_autoencoder_pretrainer.py` file contains useful functionality for stacked autoencoder pretraining. This module is mainly the same functionality that was used for coursework2 but refactored to work for tensorflow framework.
    - `executeNonLinearAutoencoder` is a function that builds a graph where there is one hidden layer and the error function is provided as parameter but it is typically the Mean Sum of Square Residuals, since we are trying to make the output match the input
    - `buildGraphOfStackedAutoencoder` is a function that creates a graph with all the layers that are already pretrained and put onto the stack. This is necessary to get inputs that have passed through the pretrained network.
    - `constructModelFromPretrainedByAutoEncoderStack` is a function which makes the stacked autoencoder as a black box. You only need to provide a set of dimensionalities for all hidden layers and a data provider class capable of processing a batch before returning the inputs/outputs set (which in the case of autoencoding are the same).
- In `data_provider.py` file in mlp folder you will find the `MSD10Genre_StackedAutoEncoderDataProvider` and `MSD25Genre_StackedAutoEncoderDataProvider` classes.
  Their implementations are identical and they are mainly extending the corresponding class to provide a way to process an input batch.