Machine Learning Practical (MLP)
Coursework 2
Georgios Pligoropoulos - s1687568

MNIST Digit Classification Problem

Our goal is to classify digits, from the MNIST dataset, as accurately as possible using a classic neural network which takes as input the flat representation of all the 784 (28x28) pixels of each grayscale image.

Grand Experiment 1: Exploring the optimal system architecture in numbers of layers and number of neurons of each layer

Why choosing this experiment and what we are trying to achieve

We are **uncertain of how much complexity is required** of the model of our neural network and thus we want to find the sweet spot between a very simple or a very complex neural network.
We will explore where the neural network breaks/fails for too few layers or too few features and where for too many of them.
In other words **we variate the dimensionality** and also **we variate the number of neurons**.

Note that in this grand experiment **we are are not optimizing for accuracy** neither optimizing for performance. Rather we want to see how the complexity of the model generally affects the accuracy and performance on a dataset as complex as MNIST. In other words we care for comparison rather than absolute values.

Constants in our Neural Network Architecture

We are using the **Gradient Descent Learning Rule with constant learning rate of 1.1** because it performed relatively well, with good accuracy and performance without making the system too unstable.
Note that we are using a learning rate that produced good enough results within 20 epochs for the three layer architecture of 100 dimensionality each in the previous coursework.

$$\Delta wi(t) = -\eta Di(t)$$

$$wi(t) = wi(t-1) + \Delta wi(t)$$

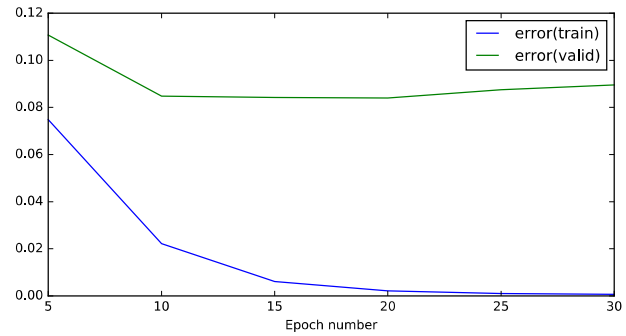
We are **not going to use regularization** rather we will do early stopping if we see that the validation accuracy and validation error have reached optimum levels.

Here we are using **glorot initialization**, a special parameter initialisation scheme for the weights which makes the scale of the random initialisation dependent on the input and output dimensions of the layer, with the aim of trying to keep the scale of activations at different layers of the network the same at initialisation.
We do this because we do not want the weights to get too big too soon and saturate the network.
We also initialise the **biases to zero** as this is not going to affect the gradients.

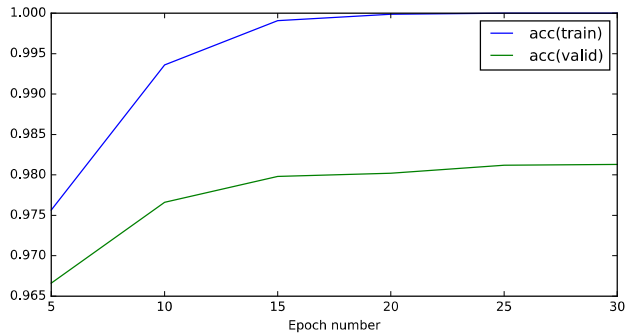
For all experiments **we use a batch size of 50** and we use **affine transformations** which are interleaved with **logistic sigmoid** nonlinearities, and at the end we always have a **softmax output layer**.

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.

Baseline: Three Layers and dimensionality of 100 for each hidden layer



Graph 1: Training & Validation Error - Three Hidden Layers - 100 dimensionality for each hidden layer



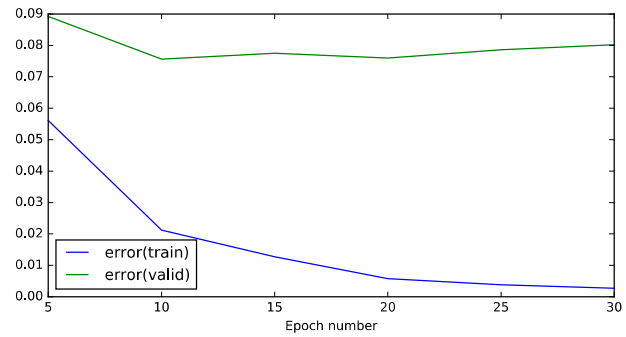
Graph 2: Training & Validation Accuracy - Three hidden layers - 100 dimensionality for each hidden layer

Runtime: `43.7089149952 seconds`

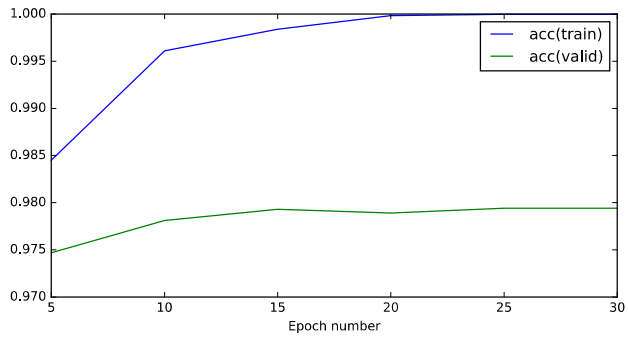
`Final Testing Accuracy (percentage 0 to 1)`	`0.98129999999999906`
`Final Testing Error`	`0.089556102164988771`
`Final Training Accuracy (percentage 0 to 1)`	`1.0`
`Final Training Error`	`0.00070516598919551218`

This is our base reference from first coursework. Within 20 epochs we overfit but as we saw in previous coursework even for 100 epochs the performance does not get significantly better.

Two Layers and dimensionality of 100 for each hidden layer



Graph 3: Training & Validation Error - Two Hidden Layers - Dimensionality 100 for each hidden layer

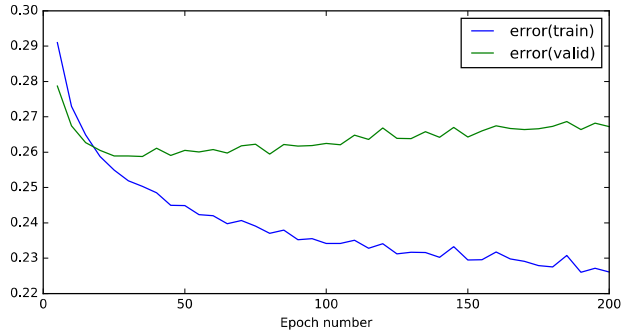


Graph 4: Training & Validation Accuracy - Two Hidden Layers - Dimensionality 100 for each hidden layer

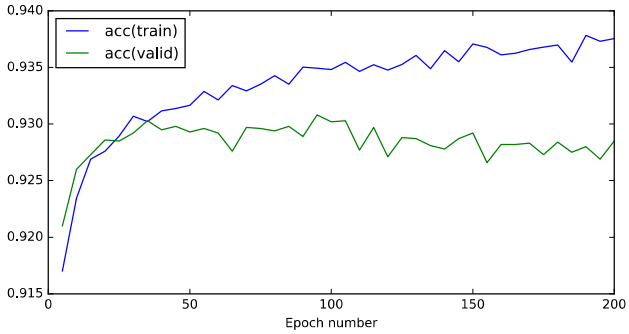
`Final Testing Accuracy (percentage 0 to 1)`	`0.97939999999999872`
`Final Testing Error`	`0.08024213423911615`
`Final Training Accuracy (percentage 0 to 1)`	`0.99997999999999998`
`Final Training Error`	`0.0027497674489311871`

For two layers the accuracy and the performance does not seem so bad. The accuracy has slightly dropped from ~98.1% to 97.9%. The best accuracy is achieved within 15 epochs. Meaning that we could consider keeping this much faster implementation.

One Layer without hidden layers



Graph 5: Training & Validation Error - One Layer



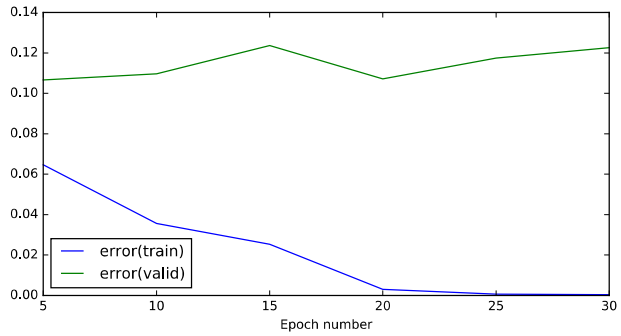
Graph 6: Training & Validation Accuracy - One Layer

Final Testing Accuracy (percentage 0 to 1)	0.92849999999999921
Final Testing Error	0.26725893060469291
Final Training Accuracy (percentage 0 to 1)	0.9375400000000047
Final Training Error	0.22611458848848595

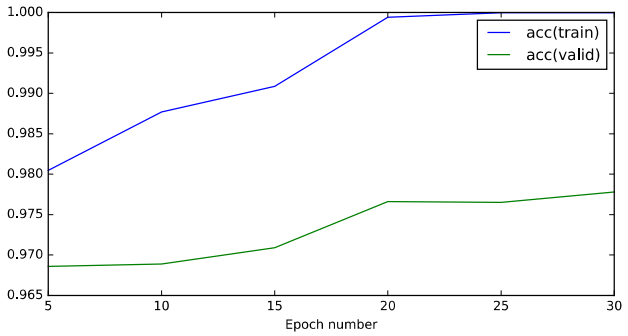
The single layer network is not possible to capture the complexity of the model, even after 200 epochs the training accuracy does not seem to be able to grow larger than 93.7% and the validation error is increasing. Moreover it is apparent that the validation accuracy has even start to decrease. The best validation accuracy achieved is ~93% which is much lower than before. We will consider this a failed case.

Let's see if adding the layers to go from 3 layers to 4 layers will have a positive impact or not or it will have a neutral effect.

Four Layers and dimensionality of 100 for each hidden layer



Graph 7: Training & Validation Error - Four Hidden Layers - dimensionality 100 for each hidden layer



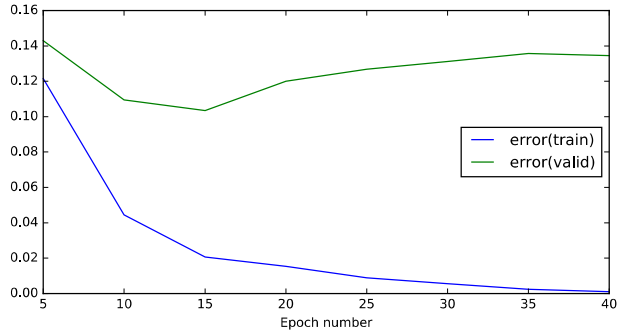
Graph 8: Training & Validation Accuracy - Four Hidden Layers - dimensionality 100 for each hidden layer

Final Testing Accuracy (percentage 0 to 1)	0.97779999999999878
Final Testing Error	0.12262454645853889
Final Training Accuracy (percentage 0 to 1)	0.99997999999999998
Final Training Error	0.00039507227075968177

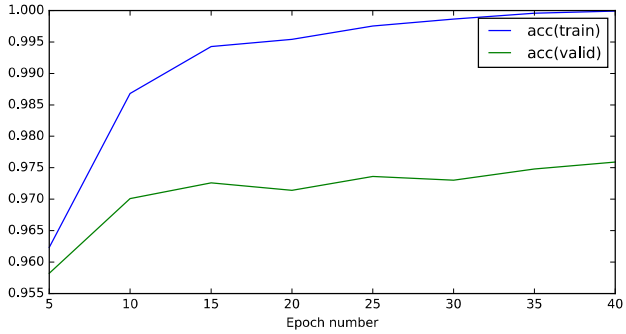
By adding one more layer we do not seem to have achieved anything significant. We are actually overfitting again since the beginning because the validation error is constantly increasing. We have not done anything better neither

in terms of performance or accuracy.

Five Layers and dimensionality of 100 for each hidden layer



Graph 9: Training & Validation Error - Five Hidden Layers - dimensionality 100 for each hidden layer

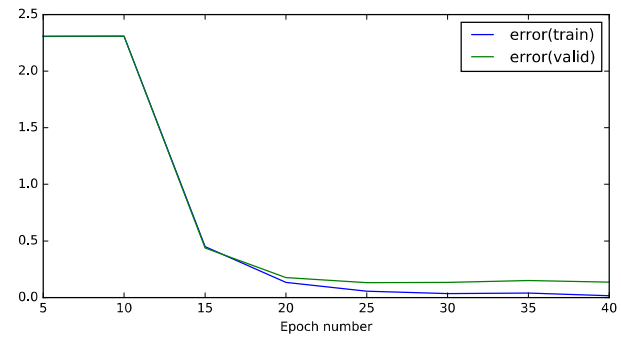


Graph 10: Training & Validation Accuracy - Five Hidden Layers - dimensionality 100 for each hidden layer

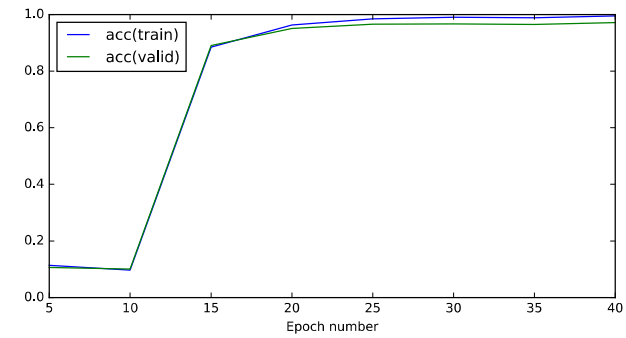
`Final Testing Accuracy (percentage 0 to 1)`	`0.97589999999999943`
`Final Testing Error`	`0.13445038749109026`
`Final Training Accuracy (percentage 0 to 1)`	`0.99990000000000012`
`Final Training Error`	`0.00098406888507099766`

For five layers again slower speed of algorithm and the accuracy has not improved. The validation error starts increasing after 15 epochs as well. We have done worse overall.

Six Layers and dimensionality of 100 for each hidden layer



Graph 11: Training & Validation Error - Six Hidden Layers - dimensionality 100 for each hidden layer



Graph 12: Training & Validation Accuracy - Six Hidden Layers - dimensionality 100 for each hidden layer

`Final Testing Accuracy (percentage 0 to 1)`	`0.97219999999999929`
`Final Testing Error`	`0.13549318830813875`
`Final Training Accuracy (percentage 0 to 1)`	`0.99558000000000246`
`Final Training Error`	`0.016967568263340702`

And finally with six layers we see a different picture. The neural network has a hard time to adjust all the weights to achieve something useful at the first ten epochs. The model is slowly learning as the weights are adjusting trying to find a balance. Afterwards we see that the accuracy is at the same terms as before. This means that even though we have all this extra complexity to our model of 6 layers we are not taking advantage any of them. It seems that some

features are just being replicated from layer to layer and the model has already found its minimum on the third layer.

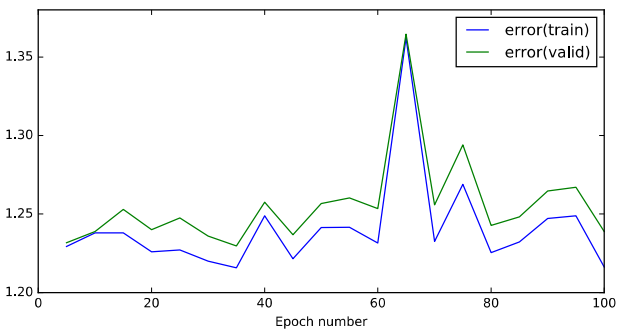
Comparing Layers with dimensionality of 100 for each hidden layer

	Final Testing Accuracy	Final Testing Error	Final Training Accuracy	Final Training Error
One Layer	0.92849999999999	0.26725893060469	0.93754000000000	0.22611458848848
Two Layers	0.97939999999999	0.08024213423911	0.99997999999999	0.00274976744893
Three Layers	0.98129999999999	0.08955610216498	1.000000000000000	0.00070516598919
Four Layers	0.97779999999999	0.12262454645853	0.99997999999999	0.00039507227075
Five Layers	0.97589999999999	0.13445038749109	0.99990000000000	0.00098406888507
Six Layers	0.97219999999999	0.13549318830813	0.99558000000000	0.01696756826334

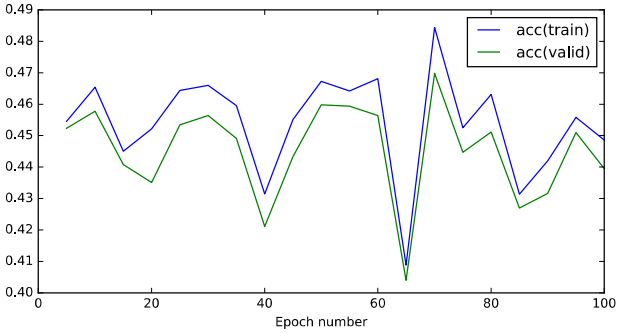
Taking into account both our plots above, where the behavior of the neural network can be displayed, and this comparison table we conclude that adding more than three layers have helped nothing else but slowing the execution time. Even the difference between the two layers and three layers does not seem that significant.

Let's go back to our experiment with 2 layers that was fast enough and with good enough accuracy and let's try to play with the dimensionality to see what we can achieve with less or more dimensionality on the hidden layer.

Two Layers and dimensionality of 2 for the hidden layer



Graph 13: Training & Validation Error - Two Layers - dimensionality of 2 for the hidden layer

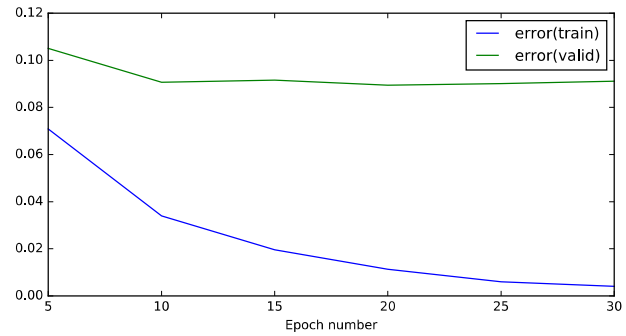


Graph 14: Training & Validation Accuracy - Two Layers - dimensionality of 2 for the hidden layer

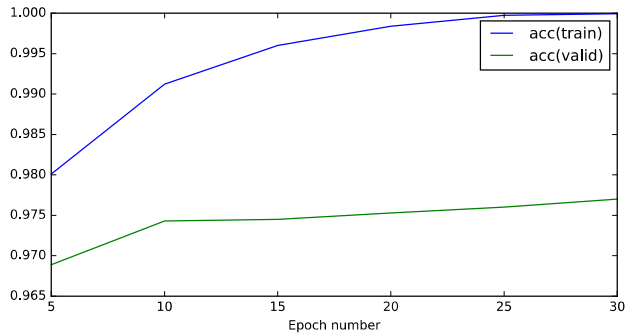
Final Testing Accuracy (percentage 0 to 1)	0.43949999999999995
Final Testing Error	1.238846999618503
Final Training Accuracy (percentage 0 to 1)	0.44859999999999839
Final Training Error	1.2162632009070424

Obviously when the features were reduced to only 2 features it would be silly of us to believe that only two features could represent 10 digits from 0 to 9. We executed this experiment to verify our intuition. And the numbers agree with us with a final validation accuracy of ~43%

Two Layers and dimensionality of 1000 for the hidden layer



Graph 15: Training & Validation Error - Two Layers - dimensionality of 1000 for the hidden layer



Graph 16: Training & Validation Accuracy - Two Layers - dimensionality of 1000 for the hidden layer

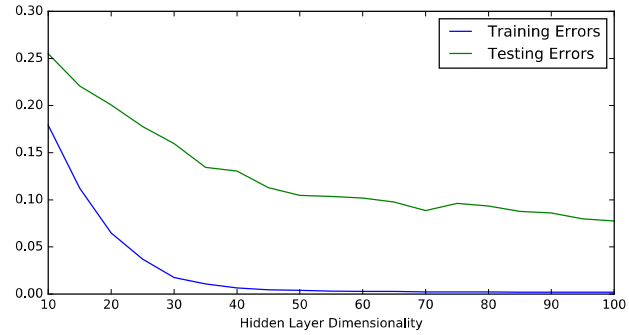
Final Testing Accuracy (percentage 0 to 1)	0.97699999999999931
Final Testing Error	0.09109709503057789
Final Training Accuracy (percentage 0 to 1)	0.99994000000000005
Final Training Error	0.0040190342463076881

When we increased the dimensionality to 1000 we got an accuracy of ~97.7% which means that we didn't really do any better than the dimensionality of 100 and we overfitted at ~25 epochs. The system also became extremely slow because of the number of connections. Therefore in this experiment achieved nothing special.

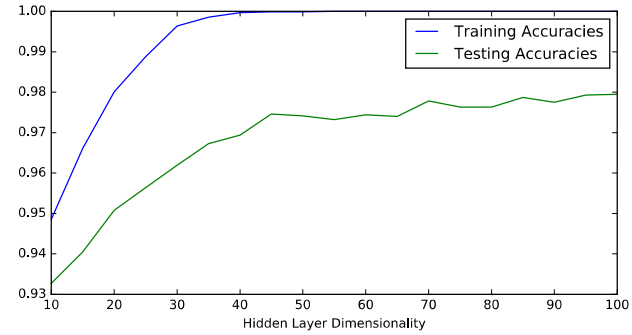
Next we are going to do a grid search to try and find the best dimensionality for a neural network of two layers.

Two Layers and dimensionality of 10 to 100 for the hidden layer with steps of 5

We are executing the experiment from 10 to 100 neurons for the hidden layer of a two layer network to see how the final validation accuracy behaves. Each experiment runs for 40 epochs because 40 epochs for a learning rate of 1.1 is enough to get an estimate if the neural network is capable to be trained or not.



Graph 17: Final Training and Validation Error for Two Layers by varying the dimensionality from 10 to 100 with steps of 5



Graph 18: Final Training and Validation Accuracy for Two Layers by varying the dimensionality from 10 to 100 with steps of 5

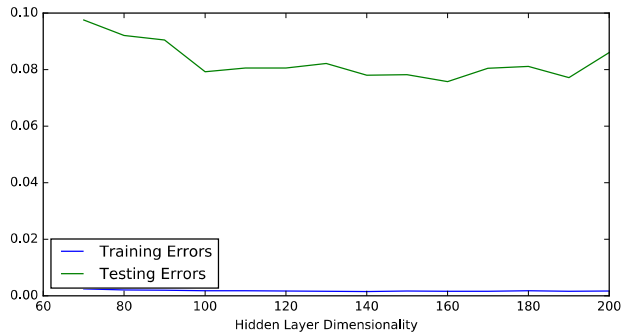
From the above two plots we can see how the model becomes more capable of capturing the complexity of the input when the dimensionality of the hidden layer is increasing

By following the exact same procedure it worths to try and see if for larger dimensionalities, again for one

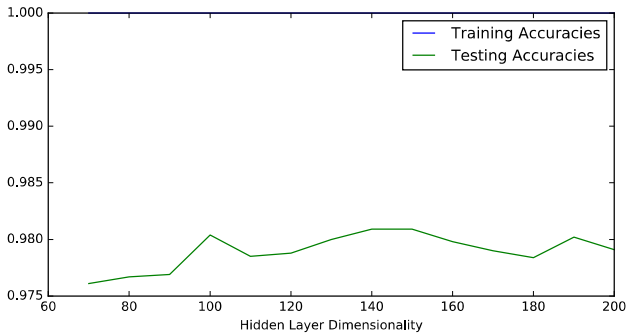
hidden layer the accuracy is increased.

Two Layers and dimensionality of 70 to 200 for the hidden layer with steps of 10

We are repeating the previous experiment but now we are ranging from 70 to 200 neurons for the hidden layer with steps of 10 mainly to see how the final validation accuracy behaves. Again we are using 40 epochs for each experiment.



Graph 19: Final Training and Validation Error for Two Layers by varying the dimensionality from 70 to 200 with steps of 10

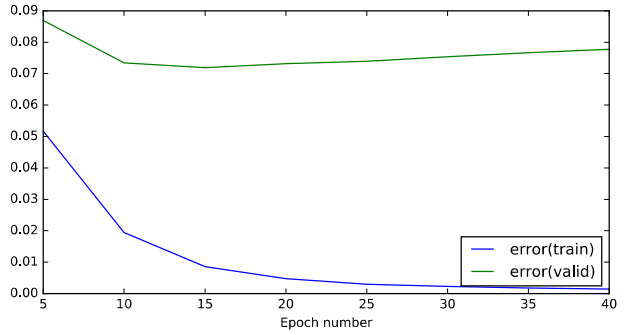


Graph 20: Final Training and Validation Accuracy for Two Layers by varying the dimensionality from 70 to 200 with steps of 10

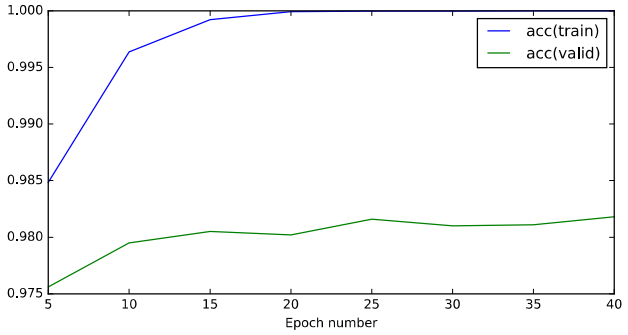
We see that for larger dimensionality we have a better capture of the complexity of the model. We peaked at hidden dimensionality = 190 and 140 has almost equal performance.

We are going to repeat that experiment for this particular dimensionality of 190 to compare its speed with the three layers when the dimensionality was 100 for each layer.

Two Layers and dimensionality of 190 for the hidden layer



Graph 21: Training & Validation Error - Two Layers - dimensionality of 190 for the hidden layer



Graph 22: Training & Validation Accuracy - Two Layers - dimensionality of 190 for the hidden layer

Runtime: `81.0337300301 seconds`

`Final Testing Accuracy (percentage 0 to 1)`	`0.98179999999999878`
`Final Testing Error`	`0.077739857983879238`
`Final Training Accuracy (percentage 0 to 1)`	`1.0`
`Final Training Error`	`0.0014331900425222032`

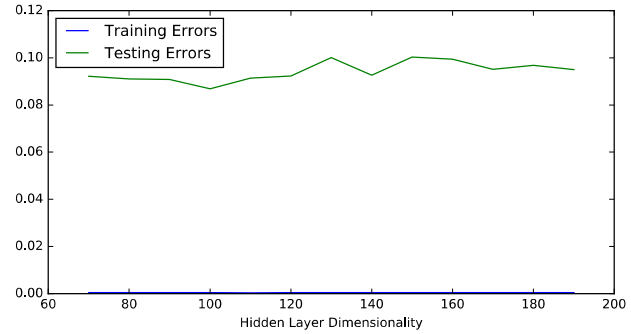
We got an optimal training accuracy of approximately ~98.1% within 25 epochs. However the 40 epochs run took in

total ~81 seconds which is almost half the speed of the calculation in comparison to the 100x100 two hidden layers (three layers in total) which took only ~43 seconds for a total of 100 epochs.

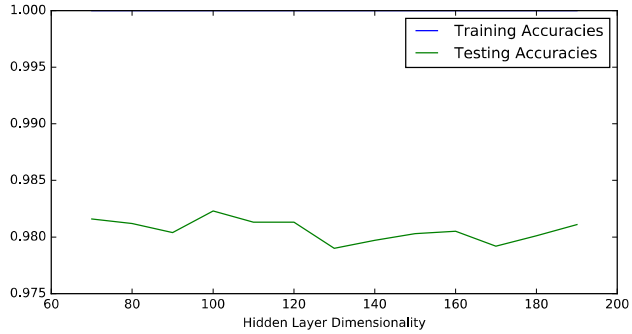
We will do one final experiment with two hidden layers where the dimensionality of the first hidden layer is going to be 140 which had a good enough accuracy, only slightly smaller than the experiment with 190 neurons. This is because it seems that too many neurons are required to get a good performance for the two layer model. We might get better accuracy if we bring in more levels in the neural network to get more levels of abstractions and since we had seen that three levels was working better than two levels in the experiments above.

Then for the second hidden layer we are going to do a grid search for the dimesionality within the range of 70 to 190. But this time we are going to keep track both of the accuracy/error and the runtime because apart of the accuracy we also care for the speed on which we can perform the experiments. Because the faster we can iterate through our experiments the faster we are going to be able to reach to meaningful conclusions.

Three Layers with 140 dimensionality of first hidden layer and a range of 70 to 200 for the second hidden layer with step of 10



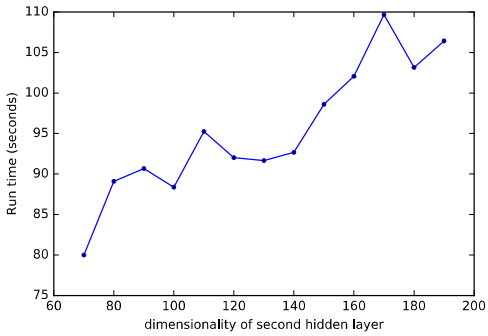
Graph 23: Final Training and Validation Error for Three Layers with first hidden layer of dimensionality of 140 and by varying the dimensionality of the second hidden layer from 70 to 200 with steps of 10



Graph 24: Final Training and Validation Accuracy for Three Layers with first hidden layer of dimensionality of 140 and by varying the dimensionality of the second hidden layer from 70 to 200 with steps of 10

We see that we have achieved the best accuracy for 100 dimensions in the second hidden layer.

Let's see how is performance, in terms of running time, for each case.



Graph 25: Runtime for Three Layers with first hidden layer of dimensionality of 140 and by varying the dimensionality of the second hidden layer from 70 to 200 with steps of 10

Comparing Metrics of best of Two Layers with best of Three Layers

Dimensionality of hidden layers	**190 (two layers)**	**140x100 (three layers)**
Runtime (seconds) for 40 epochs	81.0337300301	88.37469506263733
Final Testing Accuracy (percentage 0 to 1)	0.98179999999999878	0.98229999999999873
Final Testing Error	0.077739857983879238	0.08683500828069636
Final Training Accuracy (percentage 0 to 1)	1.0	1.0
Final Training Error	0.0014331900425222032	0.00037837813292187177

We have found out that an architecture of three layers has not provided any significant advantage in comparison with the two layers.

Training error for three layers is smaller than training error for two layers which means that the three layer model captures better the complexity of the training data.

However the performance of the accuracy has only increased ~0.1% from ~98.18% to ~98.23%

By combining two hidden layers we achieved similar results with the two layers model (only one hidden layer). More particularly the final validation accuracy for three layers is 98.12% which is very similar to 98.17% of the two layers with 190 neurons.

However the performance of the neural network with three layers is worse than our most-computationally-expesive case of the two layers of 190 neurons. We expect the two layers neural network to be faster for 140 or 150 neurons where the validation accuracy was comparable to the 190 neurons.

Conclusions

It depends of whether you are looking for maximum performance or you are looking for maximum accuracy.

For the MNIST problem you may choose 2 or 3 layers (one or two hidden layers respectively)

When choosing 2 layers network the best range is between 100 to 150 neurons and again the criteria of accuracy vs performance is going to determine the exact number of neurons.

On the other hand when choosing 3 layers and you have optimized the first hidden layer to have 140-150 neurons then you need to pick between 100-120 neurons for the second hidden layer to have an extra advantage on terms of final error and accuracy.

Grand Experiment 2: Annealed Dropout with Maxout

Why choosing this experiment and what we are trying to achieve

Dropout originally is exploited as a method for regularization since the inputs of each layer change randomly in a probabilistic way and therefore it is more "difficult" for the neural network to ever fit exactly to the peculiarities of the training data, or in other words overfit.
Quote from the paper Annealed Dropout Training of Deep Networks: "Dropout training discourages the detectors in the network from co-adapting, which limits the capacity of the network and prevents overfitting."

However in annealing dropout the point is to start from a low inclusive probability, probability of input being included in output in dropout layer, and **gradually increase the probability** as the epochs go by until 100%.

So in this case if we let the training run for lots of epochs for 100% after annealing dropout is finished then indeed it is expected to overfit.

We know that **dropout simulates approximately an exponential number of neural networks** so as we start from a lower probability we have only a few neurons, randomly chose, play the role of adapting to the model each time. Of course these small models 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**.

The ultimate goal is to see if annealing dropout always yields the **highest performance and the lowest error rate** in comparison with a simple dropout procedure.

Note that in this grand experiment **we are are not optimizing for accuracy** neither optimizing for performance. Rather we want to see how annealing dropout yield higher results in comparison with a simple dropout procedure.

Neural Network Architecture specifications

We are going to be using **Momentum Learning Rule** and **RMSProp Learning Rule** and it is going to be mentioned which one, and with which parameters is used in each case. We will use Momentum Learning Rule because we have already executed experiments with Momentum Learning Rule using Max Pooling that we will use as a baseline. Secondly we are going to use the RMSProp Adaptive Learning Rule because it yielded very good and quick results in the experiments of coursework 1 and needed little attention to the careful selection of its parameters.

Dropout itself will handle the **regularization** part but in the annealing dropout case early stopping is necessary.

We are using **glorot initialization**, a special parameter initialisation scheme for the weights which makes the scale of the random initialisation dependent on the input and output dimensions of the layer, with the aim of trying to keep the scale of activations at different layers of the network the same at initialisation.
We do this because we do not want the weights to get too big too soon and saturate the network.
We also initialise the **biases to zero** as this is not going to affect the gradients.

For all experiments **we use a batch size of 50** and we use **affine transformations** which are interleaved with **max pooling** nonlinearities, and at the end we always have a **softmax output layer**.

Note that we are going to use an architecture that is in consice with the conclusions from Grand Experiment 1.
We are going to use the **two layer model** which is much faster in terms of performance and it will allow us to execute more experiments. This model is going to have a **dimensionality of 140 for the hidden layer**.

Implementing Annealing Dropout as a Scheduler named AnnealedDropoutScheduler

Because annealing dropout changes the inclusion probability as a function of the epochs we introduced a minor hack where we created a new Scheduler class that has a reference to the model and can change the probability of the dropout layers.

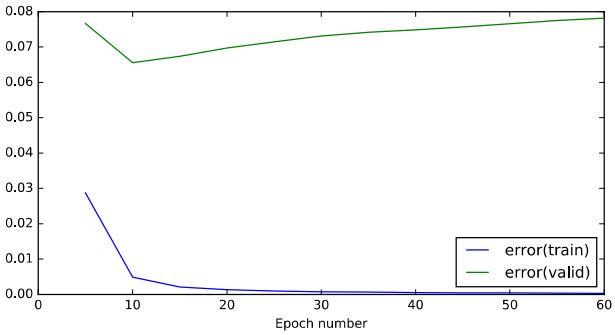
The **update_learning_rule** method which is essential in the interface of all the schedulers as it is called inside the optimizer is being exploited because it provides as a parameter the **epoch number**. We neglect the learning rule parameter of the method as it is not necessary.

The class is initialized with five attributes (the constructor has five parameters):

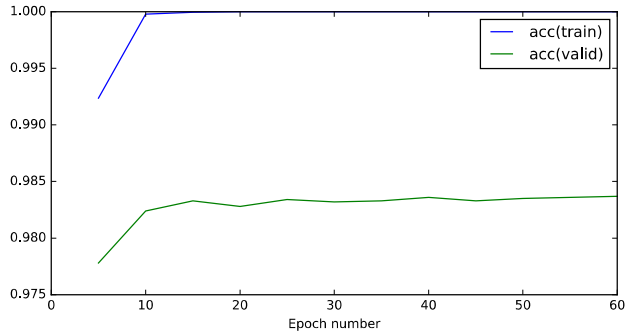
- **model**: this must the a reference to the model with all the layers. This scheduler has no effect if the model does not contain any dropout layers at all
- **startInputInclProb**: This is the initial inclusive probability for the input. We want to treat input with a different probability than the output
- **startHiddenInclProb**: This is the initial inclusive probability for all the hidden layers of the neural network.
- **epochs**: The total number of epochs of the experiment that the model is going to be used
- **holdOnPercent**: The final probability is always 100% or 1 and this percentage expresses how much percent of all epochs the probability is going to be equal to 1.

Probabilities are being increased in a linear way as the epochs go by. This is true both for the input probability and the rest of the probabilities for the hidden layers.

Baseline: Max Pooling
pool size: 2
Momentum Learning Rule learning rate: 0.02 and coefficient: 0.9



Graph 1: Training & Validation Error - Max Pooling - pool size: 2 - Momentum Learning Rule - learning rate: 0.02 and mom coefficient: 0.9



Graph 2: Training & Validation Accuracy - Max Pooling - pool size: 2 - Momentum Learning Rule - learning rate: 0.02 and mom coefficient: 0.9

Runtime: 389.950199127 seconds

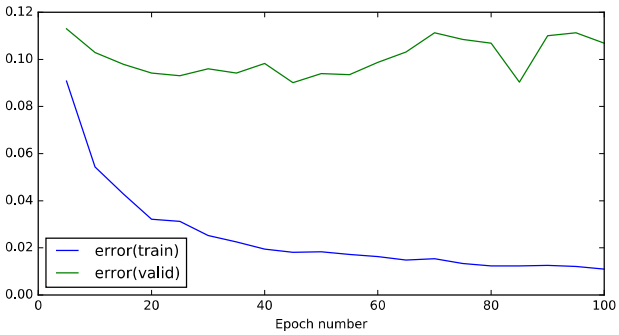
Final Testing Accuracy (percentage 0 to 1)	0.98369999999999902
Final Testing Error	0.078179863515805789
Final Training Accuracy (percentage 0 to 1)	1.0
Final Training Error	0.00032944897215494046

Even with smaller learning rate the momentum drives the learning properly to achieve an accuracy of higher than ~98% within 10 epochs.

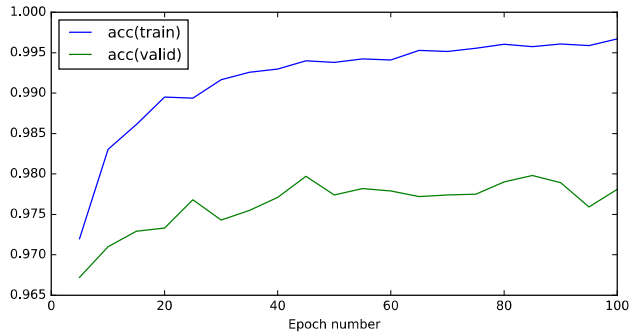
Validation error has increased a lot within 60 epochs while training accuracy has reached perfection and training error is almost zero which means that we have overfitted.

If we care for faster learning of the neural network then we might increase the learning rate and this will result to more overfitting. That's why we need to use a regularization method and here we use dropout to compare later its performance with Annealed Dropout.

Max Pooling and Dropout
pool size: 4
Momentum Learning Rule learning rate: 0.02 and coefficient: 0.9
Initial Inclusive Input Probability: 0.9
Inclusive Probability for hidden layers: 0.6



Graph 3: Training & Validation Error - Max Pooling and Dropout - pool size: 4 - Momentum Learning Rule - learning rate: 0.02 and coefficient: 0.9 - Initial Inclusive Input Probability: 0.9 - Inclusive Probability for hidden layers: 0.6



Graph 4: Training & Validation Accuracy - Max Pooling and Dropout - pool size: 4 - Momentum Learning Rule - learning rate: 0.02 and coefficient: 0.9 - Initial Inclusive Input Probability: 0.9 - Inclusive Probability for hidden layers: 0.6

Runtime: 775.867121935 seconds

`Final Testing Accuracy (percentage 0 to 1)`	`0.97809999999999919`
`Final Testing Error`	`0.10682848345968722`
`Final Training Accuracy (percentage 0 to 1)`	`0.99670000000000214`
`Final Training Error`	`0.01097044801313264`

With a much higher learning rate dropout managed to achieve good regularization and stabilized the validation error instead of leaving it to increase. However we didn't manage to achieve a high validation accuracy as before over 98%.

Note that we used a max pooling layer with pool size of 4 which was expected to yield optimal results as the previous experiments in the labs have shown. However this must be accounted as a significant factor for making the run time even a bigger number.

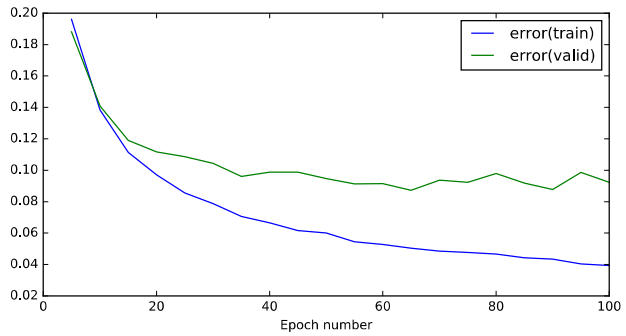
One major problem is that we have introduced two more parameters to our problem here which is choosing the best inclusive probability for the input as well choosing the best inclusive probability for the rest of the hidden layers given that we have a dropout layer in all cases.

One more important factor is that we have iterated for 100 epochs, 40 epochs more than before, but the

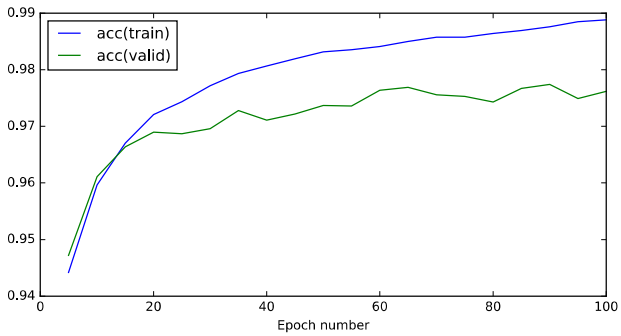
learning is still increasing in a slow rate which means we need even more epochs to achieve optimal results which makes the entire procedure very slow.

Next we are going to remove the momentum learning rule and use the RMSprop learning rule instead which proved to be better overall in coursework 1. Meaning that this adaptive learning rate was flexible enough to allow even small learning rates to be adapted quickly and efficiently.

Max Pooling and Dropout
pool size: 4
RMSProp Adaptive Learning Rule, learning rate: 1e-4, beta: 0.9
Initial Inclusive Input Probability: 0.9
Inclusive Probability for hidden layers: 0.6



Graph 5: Training and Validation Error - Max Pooling and Dropout - pool size: 4 - RMSProp Adaptive Learning Rule – learning rate: 1e-4 – beta: 0.9 - Initial Inclusive Input Probability: 0.9 - Inclusive Probability for hidden layers: 0.6



Graph 6: Training and Validation Accuracy - Max Pooling and Dropout - pool size: 4 - RMSProp Adaptive Learning Rule - learning rate: 1e-4 – beta: 0.9 - Initial Inclusive Input Probability: 0.9 - Inclusive Probability for hidden layers: 0.6

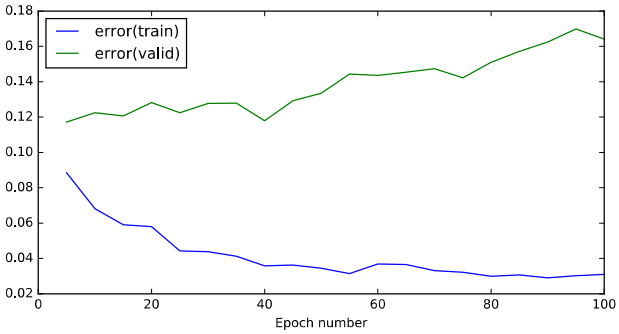
Runtime: 1281.10038996 seconds

Final Testing Accuracy (percentage 0 to 1)	0.97619999999999896
Final Testing Error	0.092352949149768368
Final Training Accuracy (percentage 0 to 1)	0.98880000000000567
Final Training Error	0.039424900776501695

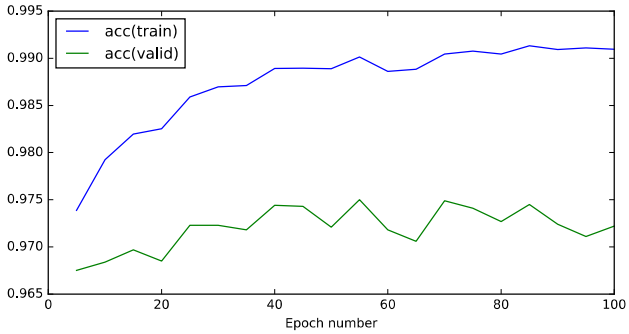
We see that adding the RMSProp did not produce any better results in comparison with the previous experiment and the execution time was increased significantly.
However we must note that the performance is more stable, the plot is more smooth than the previous experiment.

Let's repeat the experiment for max pooling size of two because it is very important to be able and run the experiments as fast as possible to be able and experiment a lot and derive some meaningful conclusions.
We will also increase the learning rate parameters of RMSProp by a factor of ten to give the model a better chance of reaching optimum validation accuracy and error within 100 epochs. This is because a max pooling size of two is expected to perform worse than the max pooling size of four.

Max Pooling and Dropout
pool size: 2
RMSProp Adaptive Learning Rule, learning rate: 1e-3, beta: 0.9
Initial Inclusive Input Probability: 0.9
Inclusive Probability for hidden layers: 0.6



Graph 7: Training and Validation Error - Max Pooling and Dropout - pool size: 2 - RMSProp Adaptive Learning Rule - learning rate: 1e-3 - beta: 0.9 - Initial Inclusive Input Probability: 0.9 - Inclusive Probability for hidden layers: 0.6



Graph 8: Training and Validation Accuracy - Max Pooling and Dropout - pool size: 2 - RMSProp Adaptive Learning Rule - learning rate: 1e-3 - beta: 0.9 - Initial Inclusive Input Probability: 0.9 - Inclusive Probability for hidden layers: 0.6

Runtime: 572.393479109 seconds

`Final Testing Accuracy (percentage 0 to 1)`	`0.97219999999999918`
`Final Testing Error`	`0.16408024330393034`
`Final Training Accuracy (percentage 0 to 1)`	`0.9909600000000045`
`Final Training Error`	`0.030960291560478217`

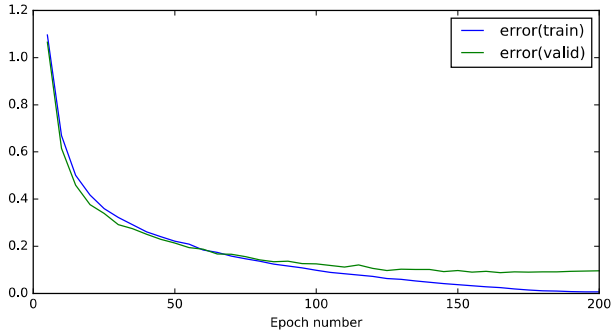
As expected with smaller max pooling size of two we have a significant increase in performance but the model does not seem flexible enough to reach the better accuracy of both training and validation set of the previous experiment where max pooling size was four.

Next we are going to use Annealed Dropout.

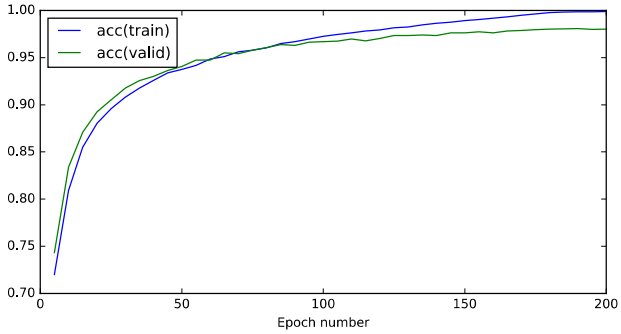
Note that Annealed dropout at first training iterations it learns simpler explanations of the data and gradually increases the capacity of the model to learn more complex explanations for all these details that cannot easily be explained.

With annealed dropout we are expecting to reach at an optimal level with minimal validation error and maximum validation accuracy. We are expecting not to have the issues of the simple dropout where the accuracy was not optimized or the optimal place required too many epochs to reach.

Starting input inclusive probability from 0.8 to 1
Starting hidden layer inclusive probability from 0.1 to 1
Percentage of epochs in the end where probability is equal to 100% constantly is 0.1
Number of epochs: 200
Max Pooling size 2



Graph 9: Training & Validation Error - Starting input inclusive probability 0.8 - Starting hidden layer inclusive probability 0.1 - Percentage of epochs in the end where probability is equal to 100% constantly is 0.1 - Max Pooling size 2



Graph 10: Training & Validation Accuracy - Starting input inclusive probability 0.8 - Starting hidden layer inclusive probability 0.1 - Percentage of epochs in the end where probability is equal to 100% constantly is 0.1 - Max Pooling size 2

Runtime: 1354.82056618 seconds

`Final Testing Accuracy (percentage 0 to 1)`	`0.98029999999999862`
`Final Testing Error`	`0.09619017831096531`
`Final Training Accuracy (percentage 0 to 1)`	`0.99876000000000054`
`Final Training Error`	`0.0061450748950512037`

Here we are training for 200 epochs which seem a lot but give the annealing this slow rate of learning to optimize the final result.

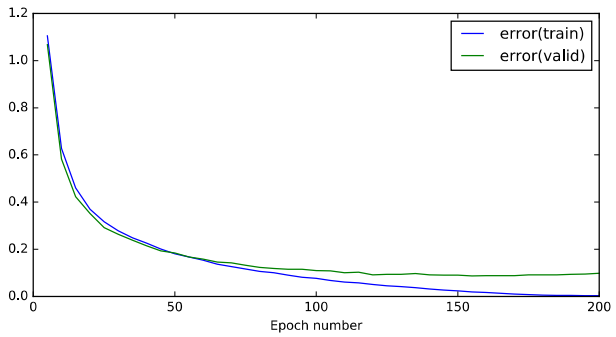
The final validation accuracy is over 98% which is an indicator of a good finally tuned model even with a max pooling of size two. Remember from previous experiments that we had to increase max pooling size in order to optimize the performance.

In addition the testing error is one of the lower testing errors in comparison to previous experiments.

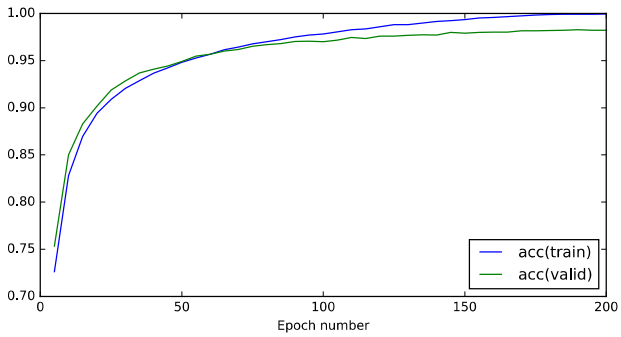
However the training running time was approximately one third of an hour which is significant.

We need to run more experiments before stating any conclusions.
Next let’s execute the same experiment but for max pooling size of 4.
If Annealed Dropout “guarantees” the best possible model we should see an increase in the testing accuracy.

Starting input inclusive probability from 0.8 to 1
Starting hidden layer inclusive probability from 0.1 to 1
Percentage of epochs in the end where probability is equal to 100% constantly: 0.1
Number of epochs 200
Max Pooling size 4



Graph 11: Training & Validation Error - Starting input inclusive probability from 0.8 to 1Starting hidden layer inclusive probability from 0.1 to 1Percentage of epochs in the end where probability is equal to 100% constantly: 0.1Number of epochs 200Max Pooling size 4



Graph 12: Training & Validation Accuracy - Starting input inclusive probability from 0.8 to 1Starting hidden layer inclusive probability from 0.1 to 1Percentage of epochs in the end where probability is equal to 100% constantly: 0.1Number of epochs 200Max Pooling size 4

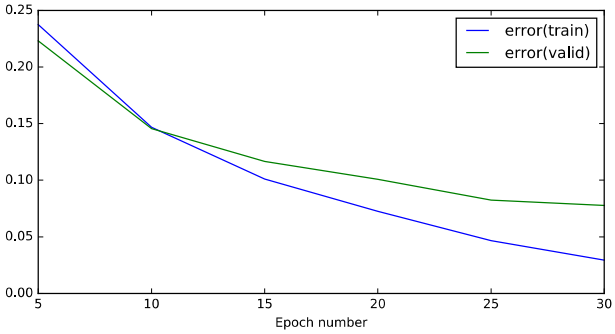
Runtime: 3312.20865393 seconds

Final Testing Accuracy (percentage 0 to 1)	0.9820999999999992
Final Testing Error	0.098154630284018191
Final Training Accuracy (percentage 0 to 1)	0.99938000000000038
Final Training Error	0.0028349326601966609

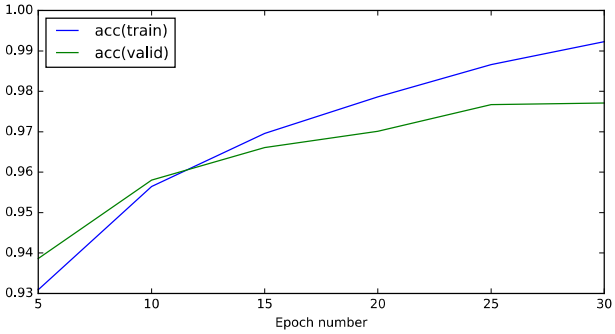
And yes here annealing dropout resulted in one the highest validation accuracies we have seen. We must note that we don't expect annealing dropout to produce remarkable results in comparison to simple dropout nor that it is expected to yield a higher performance because it does nothing special in contributing to the architecture or the complexity or the simplicity of the model. However what annealing dropout does is maximize the potential of the model in hand by avoid the convergence towards local minima.

Because Annealing Dropout is a very computationally intensive procedure it worths doing some experiments with fewer number of epochs and different initial inclusive probabilities to see if we can get the same good results but faster.

Starting input inclusive probability from 0.8 to 1
Starting hidden layer inclusive probability from 0.5 to 1
Percentage of epochs in the end where probability is equal to 1 constantly: 0.05
Number of epochs 30
Max Pooling size 4



Graph 13: Training and Validation Error - Starting input inclusive probability from 0.8 to 1 - Starting hidden layer inclusive probability from 0.5 to 1 - Percentage of epochs in the end where probability is equal to 1 constantly: 0.05 - Number of epochs 30 - Max Pooling size 4



Graph 14: Training and Validation Accuracy - Starting input inclusive probability from 0.8 to 1 - Starting hidden layer inclusive probability from 0.5 to 1 - Percentage of epochs in the end where probability is equal to 1 constantly: 0.05 - Number of epochs 30 - Max Pooling size 4

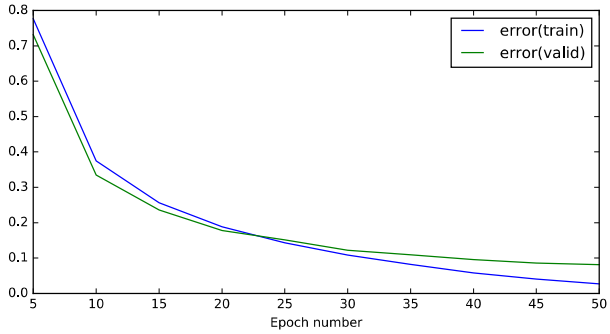
Runtime: 395.63371706

Final Testing Accuracy (percentage 0 to 1)	0.97709999999999908
Final Testing Error	0.077706504433064316
Final Training Accuracy (percentage 0 to 1)	0.99230000000000385
Final Training Error	0.029451786118968395

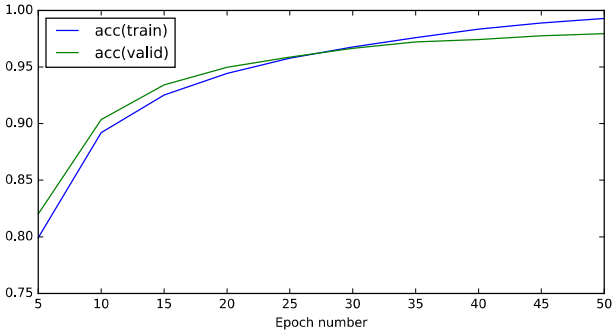
Note that we had chosen to decrease the percentage of epochs where probability is equal to 100% at the end because 30 epochs are not enough for something like that. Also the initial inclusive probability for the hidden layers was chosen to be 0.5 because going from 0.1 to 1 in 30 epochs would have been very quick for annealing to work properly.

This experiment did not yield the expected results. Seems like 30 epochs is not slow enough for the model to go from simpler to more complex representations and annealing did not gave us the best model.

Starting input inclusive probability from 0.8 to 1
Starting hidden layer inclusive probability from 0.1 to 1
Percentage of epochs in the end where probability is equal to 1 constantly: 0.05
Number of epochs 50
Max Pooling size 4



Graph 15: Training and Validation Error - Starting input inclusive probability from 0.8 to 1 - Starting hidden layer inclusive probability from 0.1 to 1 - Percentage of epochs in the end where probability is equal to 1 constantly: 0.05 - Number of epochs 50 - Max Pooling size 4



Graph 16: Training and Validation Accuracy - Starting input inclusive probability from 0.8 to 1 - Starting hidden layer inclusive probability from 0.1 to 1 - Percentage of epochs in the end where probability is equal to 1 constantly: 0.05 - Number of epochs 50 - Max Pooling size 4

`Final Testing Accuracy (percentage 0 to 1)`	`0.97949999999999904`
`Final Testing Error`	`0.081238606177604145`
`Final Training Accuracy (percentage 0 to 1)`	`0.99300000000000366`
`Final Training Error`	`0.027229031752991996`

Here we have not achieved any remarkable results in comparison to previous experiments but it is interesting to note that with 20 more epochs we were able to derive a better final model with higher validation accuracy than before. Note that here we have started from 0.1 initial inclusive probability for the hidden layers but this did not have any slowing-down or other unwanted effect.
This experiment is a good indication that we should better start from lower probabilities rather than higher ones for the hidden layer.

Conclusions

Annealing dropout seems like a simple and yet very powerful technique to optimize the model in hand in terms of classification accuracy by avoiding local minima and going slowly from simpler models, where only a few important features of the problem have been captures and it builds on top of them to end-up with a more complex model that can capture the full complexity of the input, these small details that make a difference in the final classification accuracy.

To summarize it the experiments suggest that Annealed Dropout guarantees the best possible model with the current architecture.

It also works without requiring any pretraining to start-off from a good place rather than what the Glorot initialization of the weights might give you. This is because from where it starts makes little difference how we choose the initial weights. These initial simpler models are going to quickly converge to capture the more basic features of our problem. In other words at the beginning we have clearly underfitted. The only thing to avoid is, as always not set too large weights in the beginning to avoid saturation effects.

Grand Experiment 3: Stacked Autoencoders Pretraining

Why choosing this experiment and what we are trying to achieve

Autoencoders are useful in many problems either to remove noise from images or other representations or when you want to do dimensionality reduction either because PCA is not enough because it lacks non-linearities or just because you want to plot high dimensional data in a 2D space in a way that makes sense. They are a way of representing an input with many features into another dimensional space that makes sense.

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.

In this experiment we are using stacked autoencoders to optimize the pretraining of our neural network.

Our intentions is a pretraining that is going to have two positive effects:

1) For large architectures with many layers there is the issue that the backpropagation is more "slow". This is understandable because we have many layers and in order for the error to propagate properly backwards and set the weights at a good level we might have saturations or other effects that cause very slow learning at the first epochs.

We are going to test that by running an experiment with six or more layers where the slow-learning effect is apparent and then repeat the experiment with our stacked autoencoders pretraining. The expectation is that our pretraining will not go through this phase of slow learning and the learning process will converge more quickly.

2) We should get a very good performance always in comparison to random initialization of the weights.

We are going to test that by repeating the same experiment a few times for various random initializations of the weights. Then for the same architecture we are going to pretrain the network using stacked autoencoders and run to see the final error and accuracy of both training and validation set. We must see that this pretraining yields better or equally good results as the best of the cases with the random initializations.

We will experiment with two kinds of stacked autoencoders: **Linear Stacked Autoencoders** and **Non-Linear Stacked Autoencoders**.

Neural Network Architecture specifications

We are going to be using **Gradient Descent Learning Rule** because the adaptive learning rules might disguise the effects that we are hoping to get with and without pretraining.

For simplicity and because of low resources we will not do any **regularization** other than early stopping.

We are using **glorot initialization**, a special parameter initialisation scheme for the weights which makes the scale of the random initialisation dependent on the input and output dimensions of the layer, with the aim of trying to keep the scale of activations at different layers of the network the same at initialisation.
We do this because we do not want the weights to get too big too soon and saturate the network.
We also initialise the **biases to zero** as this is not going to affect the gradients.

For all experiments **we use a batch size of 50** and we use **affine transformations** which are interleaved with **Sigmoid** nonlinearities, and at the end we always have a **softmax output layer**.

We are going to use multiple architectures in terms of how many layers we use in order to see and explore the effects of pretraining in deep neural networks. Based on our experiences from the experiments of this and the past coursework we will use the **dimensionality of the hidden layer** to always be **100**.

Implementing Stacked Autoencoders

It is easy with our current implementations to create an Autoencoder because we already have a data provider to provide the input as the 784 pixels of a 28x28 image.

But in order to create a second layer in our neural network pretrained by an autoencoder we need the first layer to play the role of the input (and output) for this autoencoder and this requires a new kind of Data Provider.

StackedAutoEncoderDataProvider

This is a subclass of MNISTDataProvider and it provides as input and output the batch of the inputs after they have been processed through a neural network up until the level we want.

To achieve that our Data provider has access to the current model that we are pretraining and it uses its fprop method to get a representation of the image for the layers that we have so far pretrained.

As an example if we want to pretrain the second layer of our network then the model passed as parameter in the Data Provider class must have the first layer already pretrained and this layer is used to transform the 784 dimensional inputs of our image to the representation of that layer which it will have features with 100 dimensionality in our case.
Recall that this layer has been pretrained to provide an as accurate as possible representation of the input. And this representation is that we want to have as input the second iteration of our stacked autoencoder where an autoencoder is used to build the second layer by using the first.

If model is set as *None* in python then the StackedAutoEncoderDataProvider falls back to a simple MNISTDataProvider which returns the inputs as inputs and ouputs.

Linear Stacked Autoencoder

The Linear Stacked Autoencoder does not include any non-linearities.
It is composed by two affine layers where the input and output layer have the same dimensionality since we want the input to match the output and the hidden layer is the layer that we want to pretrain.

As an error we are using the Sum of Squared Differences Error because we want to match the input with the output.

We are using our best Adaptive Learning Rule which is Adam Learning Rule with default parameters.

When pretraining is finished the affine layer that refers to the ouput is of no concern to us and we are discarding it to keep first affine layer as our pretrained layer.

Non-Linear Stacked Autoencoder

The Non-Linear Stacked Autoencoder is composed by two affine layers interleaved by two non-linearities which here we have chosen the Sigmoids.

The input and output layer have the same dimensionality since we want the input to match the output and the hidden layer is the layer that we want to pretrain.

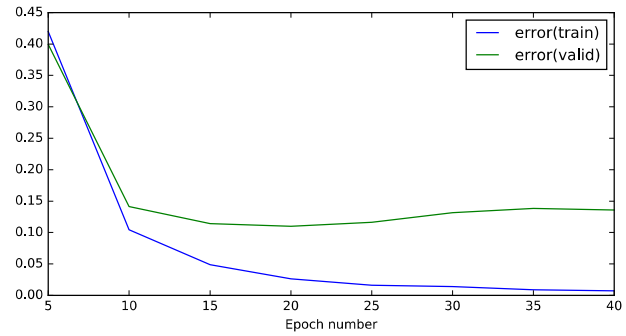
As an error we are using the Sum of Squared Differences Error because we want to match the input with the output.

We are using our best Adaptive Learning Rule which is Adam Learning Rule with default parameters.

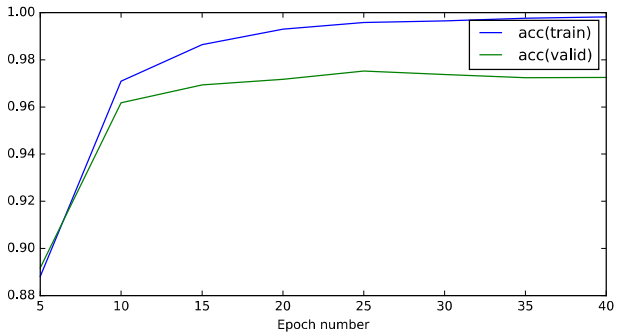
When pretraining is finished the affine layer and the sigmoid non-linearity that refers to the ouput is of no concern to us and we are discarding both of them to keep only the first affine layer as our pretrained layer. Note here that if we keep or not the Sigmoid non-linearity does not make a difference since this layer does not contain parameters, does not store any state.

Experiment A: avoiding slow learning effects on multi-layer deep neural networks

Six Layers without pretraining



Graph 1: Training & Validation Error – Six Layers without Pretraining



Graph 2: Training & Validation Accuracy – Six Layers without Pretraining

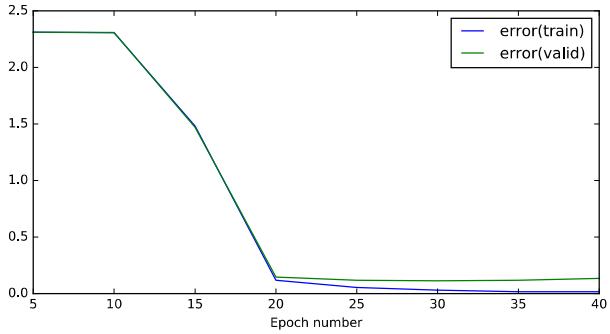
Runtime: `111.428107977 seconds`

`Final Testing Accuracy (percentage 0 to 1)`	`0.97249999999999903`
`Final Testing Error`	`0.13592290449311858`
`Final Training Accuracy (percentage 0 to 1)`	`0.99816000000000116`
`Final Training Error`	`0.0069894214987917744`

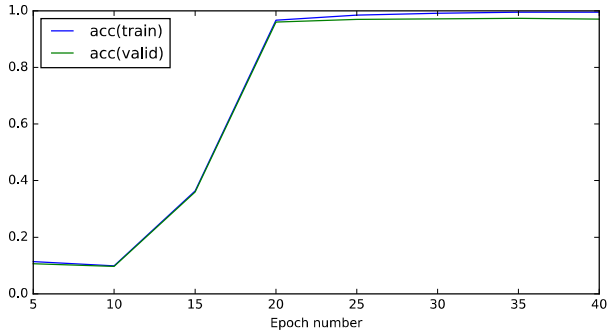
We see in the plots that we start from low accuracy and higher error because of the too many layer of the model. Actually we peak at the 25 epochs and then because of overfitting the metrics are worse resulting in a relatively bad final accuracy.

Now we will construct a model of six layers with 100 dimensionality in each hidden layer via a Linear Stacked AutoEncoder.
This will give us six affine layers which we interleave with Sigmoid Layers in order to have a proper neural network with non-linearities.

Six Layers with Linear Stacked Autoencoder Pretraining



Graph 3: Training & Validation Error – Six Layers with Linear Stacked Autoencoder Pretraining



Graph 4: Training & Validation Accuracy – Six Layers with Linear Stacked Autoencoder Pretraining

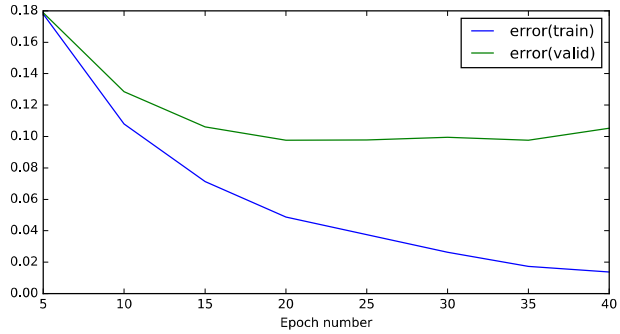
Runtime: `110.861319065` seconds

`Final Testing Accuracy (percentage 0 to 1)`	`0.9709999999999992`
`Final Testing Error`	`0.135004313417807`
`Final Training Accuracy (percentage 0 to 1)`	`0.99542000000000253`
`Final Training Error`	`0.015544239697404282`

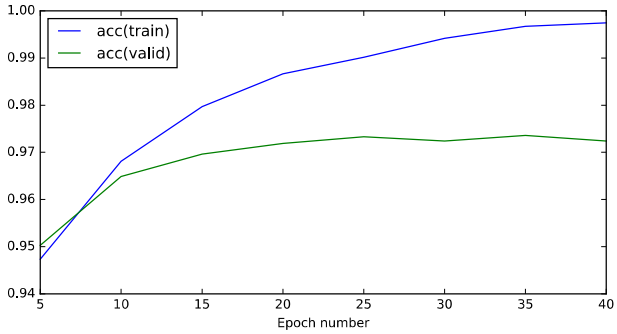
Our linear stacked autoencoder seems to have done a poor job in pretraining the network. It seems that we have evoked the effect that we were trying to avoid!

Let’s try and repeat the same experiment using the Non-Linear Stacked Autoencoder for Pretraining

Six Layers with Non-Linear Stacked Autoencoder Pretraining



Graph 5: Training & Validation Error – Six Layers with Non-Linear Stacked Autoencoder Pretraining



Graph 6: Training & Validation Accuracy – Six Layers with Non-Linear Stacked Autoencoder Pretraining

Runtime: `110.105275154` seconds

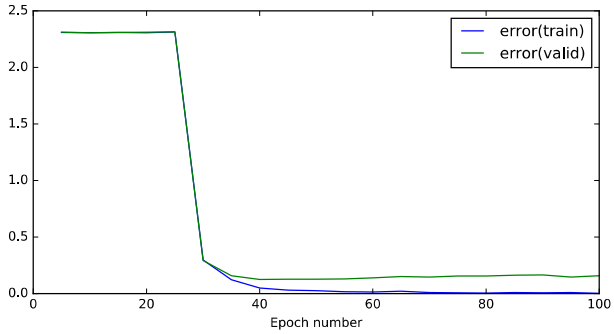
`Final Testing Accuracy (percentage 0 to 1)`	`0.97239999999999904`
`Final Testing Error`	`0.10528195949162089`
`Final Training Accuracy (percentage 0 to 1)`	`0.99748000000000137`
`Final Training Error`	`0.01367461917368043`

The Non-Linear Stacked Autoencoder Pretraining did a much better job than the Linear Stacked Autoencoder.

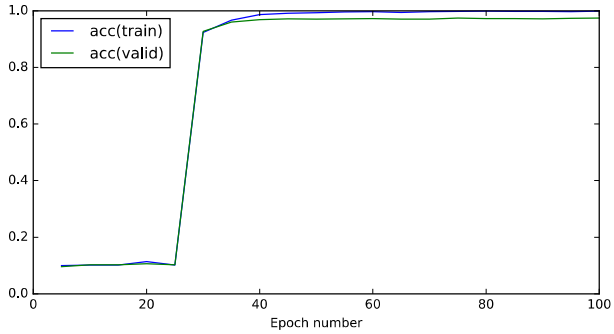
Gladly this experiment is in consice with our original intentions. Here we don't see a slow learning rate at the beginning but rather an immediate increase of the accuracy and decrease of the error.

The final accuracy is better than the experiment with the Linear Stacked Autoencoder.

Seven Layers without pretraining



Graph 7: Training & Validation Error – Seven Layers without Pretraining



Graph 8: Training & Validation Accuracy – Seven Layers without Pretraining

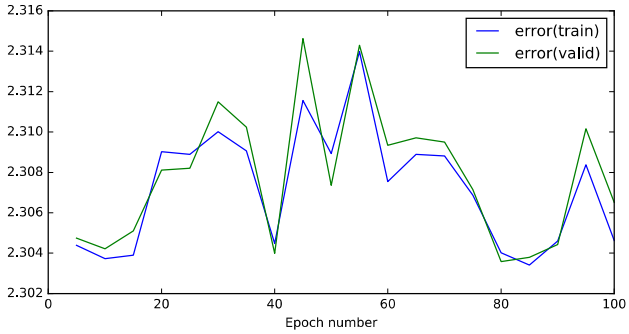
Runtime: `319.806434155` seconds

`Final Testing Accuracy (percentage 0 to 1)`	`0.97459999999999836`
`Final Testing Error`	`0.15829059585158961`
`Final Training Accuracy (percentage 0 to 1)`	`0.99946000000000024`
`Final Training Error`	`0.002222706411481464`

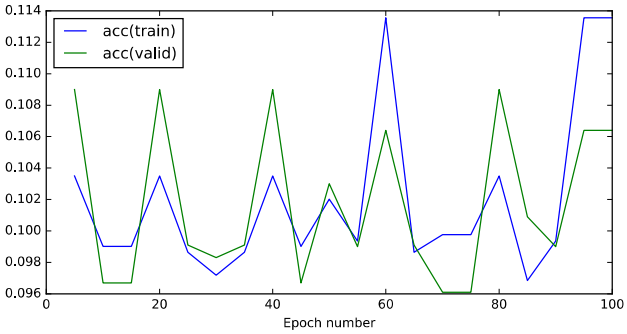
With 7 layers the effect is more severe. The learning is really slow in the beginning and we have to go to more than 40 epochs so that the neural network really starts learning

Now let's construct a model of seven layers with 100 dimensionality in each hidden layer via a Linear Stacked AutoEncoder and see if we are going to alieviate this unwanted effect.

Seven Layers with Linear Stacked Autoencoder Pretraining



Graph 9: Training & Validation Error – Seven Layers with Linear Stacked Autoencoder Pretraining



Graph 10: Training & Validation Accuracy – Seven Layers with Linear Stacked Autoencoder Pretraining

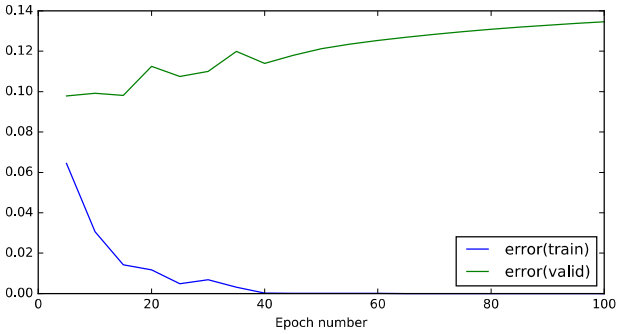
Runtime: `345.946224928` seconds

`Final Testing Accuracy (percentage 0 to 1)`	`0.10639999999999988`
`Final Testing Error`	`2.3065173136848172`
`Final Training Accuracy (percentage 0 to 1)`	`0.11355999999999987`
`Final Training Error`	`2.3046307375203297`

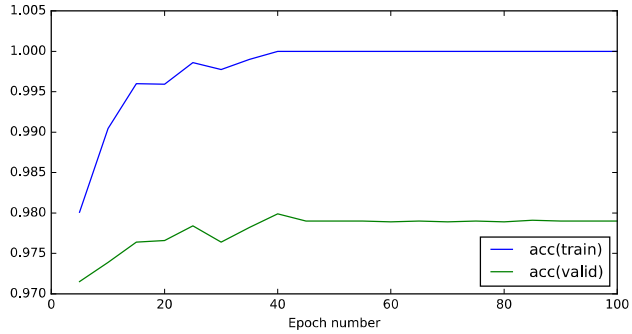
On this experiment the Linear Stacked Autoencoder Pretraining resulted in really bad results for the seven layers. The model is not able to converge and the metrics show that we have not really succeeded any training. The training and testing accuracy is near to 10% which is like the prior probability.

Next we will try again with the Non-Linear Stacked Autoencoder to try and yield better results.

Seven Layers with Non-Linear Stacked Autoencoder Pretraining



Graph 11: Training & Validation Error - Seven Layers with Non-Linear Stacked Autoencoder Pretraining



Graph 12: Training & Validation Accuracy - Seven Layers with Non-Linear Stacked Autoencoder Pretraining

Runtime: `337.20222497` seconds

`Final Testing Accuracy (percentage 0 to 1)`	`0.97899999999999876`
`Final Testing Error`	`0.13459701113216441`
`Final Training Accuracy (percentage 0 to 1)`	`1.0`
`Final Training Error`	`2.2924539333227827e-05`

We see that the non linear stacked autoencoder yields better results. Even with 7 layers there is no phase where we have slow-training. The training reachs optimal error and accuracy at ~40 epochs.

Conclusions on Experiment A

The above experiments show that the Linear Stacked Autoencoder is not a reliable system to use. The massive failure of our experiments could be explained by our architecture where we have consecutive hidden layers of the exact same dimensionality.
So for example in our case if you have as input and output for a Linear Autoencoder 100 features and you are asking to train the optimal hidden layer of again 100 features you are more likely to get the identity matrix which means that lots of weights are saturated. This approach could cause the unwanted effects we experienced above.

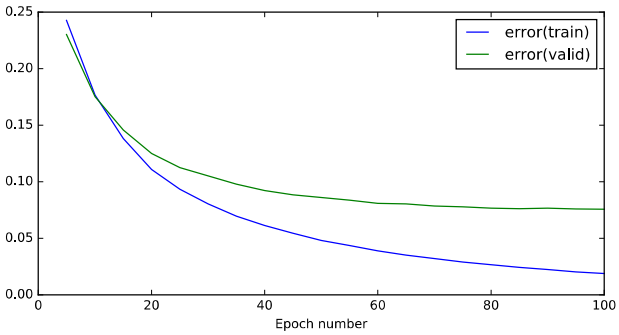
On the other hand a Non-Linear Stacked Autoencoder alleviated the issue of slow learning for deep neural network and the pretraining brought the initial state of the neural network at a place that is suitable for our classification task. Therefore the Non-Linear Stacked Autoencoder is recommened for multilayer deep neural networks.

Experiment B: yielding smaller error and higher accuracy by pretraining instead of randomly initializing weights

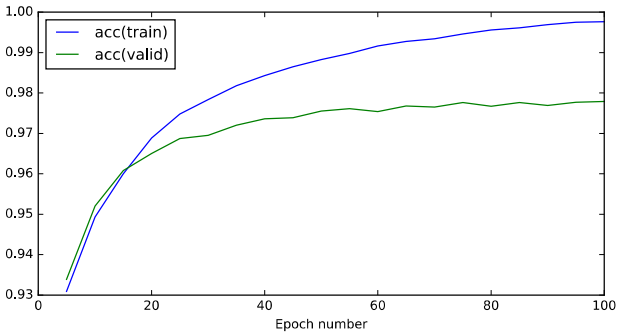
Our goal is to initialize the randomly generated weights with different seeds and see what kind of results we are getting. Then use Pretraining with Stacked Autoencoders and compare the final performance and accuracy.

Two Layer Model with random initialization of weights with Glorot Uniform Initialization

Let's see how our two layer model behaves within 100 epochs



Graph 13: Training & Validation Error - Two Layers without Pretraining

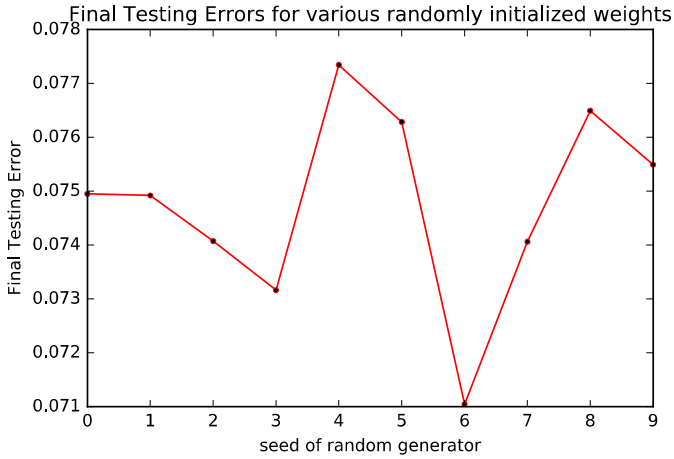


Graph 14: Training & Validation Accuracy - Two Layers without Pretraining

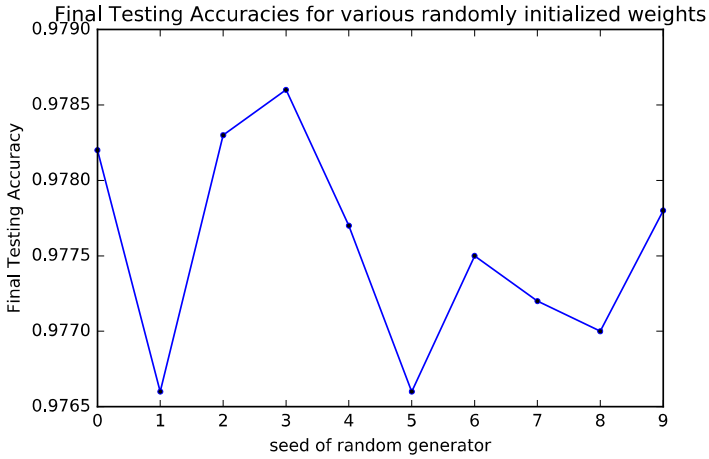
```
Runtime: 126.894783974 seconds
```

Final Testing Accuracy (percentage 0 to 1)	0.97789999999999777
Final Testing Error	0.075733012679254696
Final Training Accuracy (percentage 0 to 1)	0.99764000000000153
Final Training Error	0.018958982953952288

Two Layer Model multiple experiments with various random initializations of weights with Glorot Uniform Initialization



Graph 15: Final Validation Error for various randomly initialized weights for a two layer deep neural network



Graph 16: Final Validation Accuracy for various randomly initialized weights for a two layer deep neural network

Maximum Testing Accuracy	0.97859999999999903
Mean Testing Accuracy	0.97754999999999903
Minimum Testing Accuracy	0.97659999999999914
Maximum Testing Error	0.077343073407462298
Mean Testing Error	0.074782804560505595
Minimum Testing Error	0.071047685963312984

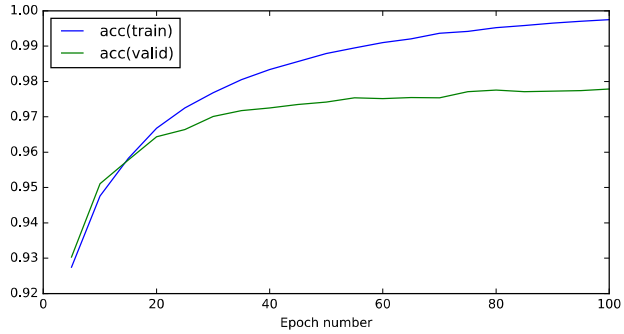
The final accuracy is between ~97.66% to ~97.86%

and the validation error is between ~0.071 to ~0.077

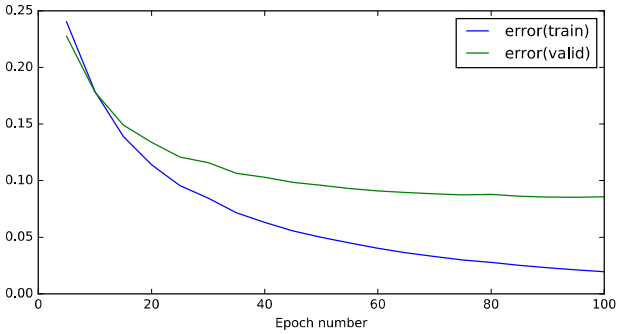
We will use this information for comparisons below

Next we will repeat the experiment by pretraining with a Linear Stacked Autoencoder.

Two layers with Linear Stacked AutoEncoder Pretraining



Graph 17: Training & Validation Error - Two layers with Linear Stacked AutoEncoder Pretraining



Graph 18: Training & Validation Accuracy - Two layers with Linear Stacked AutoEncoder Pretraining

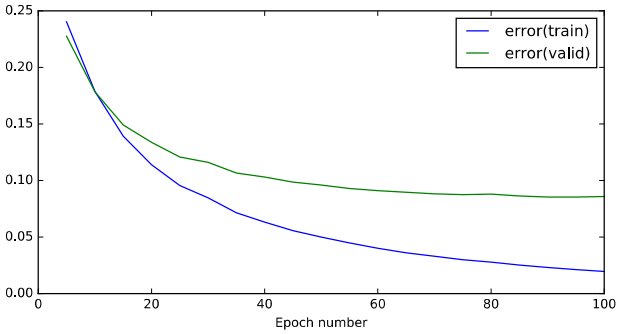
`Runtime: 129.11024189 seconds`

`Final Testing Accuracy (percentage 0 to 1)`	`0.9778999999999991`
`Final Testing Error`	`0.071578519487560929`
`Final Training Accuracy (percentage 0 to 1)`	`0.99748000000000137`
`Final Training Error`	`0.019062135396377036`

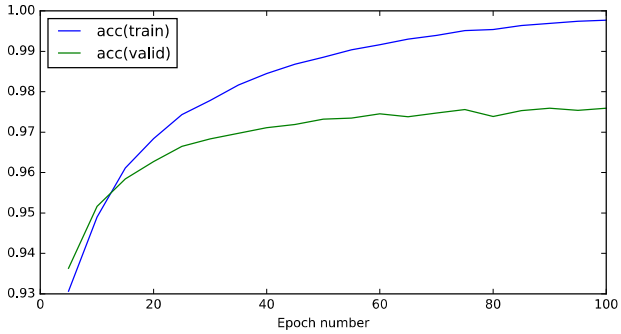
The pretraining with the linear stacked autoencoder yielded in general good results.
The validation accuracy is near the upper end of the range when using random initializations.
Also the validation error is near the lower end of the range when using random initializations.
The results can be described as good but not spectacular.

We will repeat the same experiment using a Non-Linear Stacked Autoencoder

Two layers with Non-Linear Stacked AutoEncoder Pretraining



Graph 19: Training & Validation Error - Two layers with Non-Linear Stacked AutoEncoder Pretraining



Graph 20: Training & Validation Accuracy - Two layers with Non-Linear Stacked AutoEncoder Pretraining

Runtime: `131.107201099 seconds`

`Final Testing Accuracy (percentage 0 to 1)`	`0.97589999999999943`
`Final Testing Error`	`0.085712887076613342`
`Final Training Accuracy (percentage 0 to 1)`	`0.99770000000000147`
`Final Training Error`	`0.019567442932861615`

Even though we expected the Non-Linear Stacked Autoencoder to perform better based on previous experiments with Non-Linear Stacked Autoencoder in fact here the metrics after the pretraining are worse.

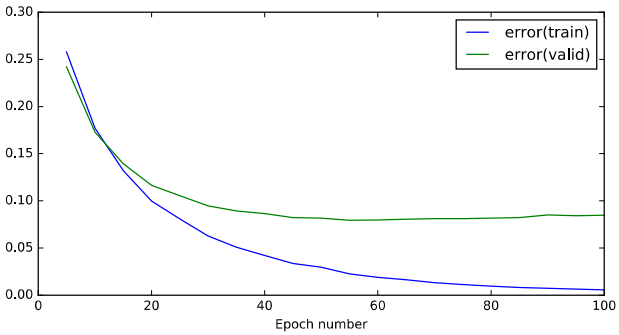
Comparison Table for Two Layers

Two Layers	Random Initialization of Weights		Linear Stacked Autoencoder	Non-Linear Stacked Autoencoder
Testing Accuracy	`Maximum`	`0.97859999999999`	`0.97789999999999`	`0.97589999999999`
	`Mean`	`0.97754999999999`		
	`Minimum`	`0.97659999999999`		
Testing Error	`Maximum`	`0.07734307340746`	`0.07157851948756`	`0.08571288707661`
	`Mean`	`0.07478280456050`		
	`Minimum`	`0.07104768596331`		

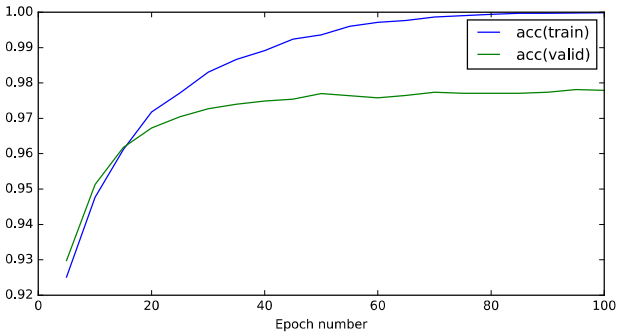
The conclusion is that the pretraining for two layers using either Linear or Non-Linear Stacked Autoencoders does not guarantee better results nor does it have any other significant impact.

We will try to see if any effects emerge when the neural network is deeper with more layers.

Three Layer Model with random initialization of weights with Glorot Uniform Initialization



Graph 21: Training & Validation Error - Three Layer Model with random initialization of weights with Glorot Uniform Initialization



Graph 22: Training & Validation Accuracy - Three Layer Model with random initialization of weights with Glorot Uniform Initialization

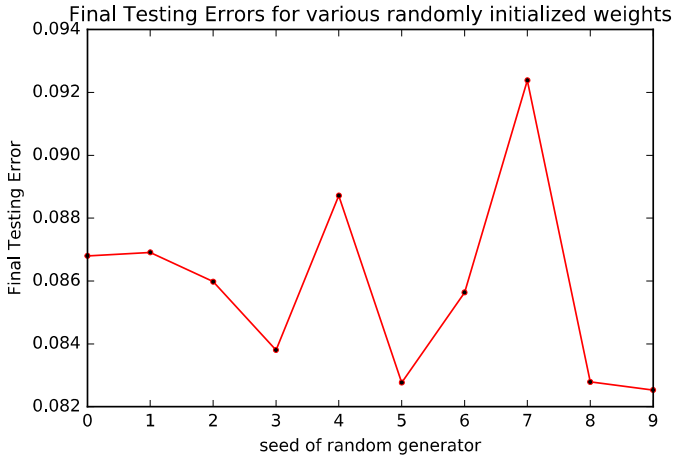
Runtime: `173.823220015 seconds`

`Final Testing Accuracy (percentage 0 to 1)`	`0.9778999999999991`
`Final Testing Error`	`0.084736826625688905`
`Final Training Accuracy (percentage 0 to 1)`	`0.99982000000000015`
`Final Training Error`	`0.0055768883911021475`

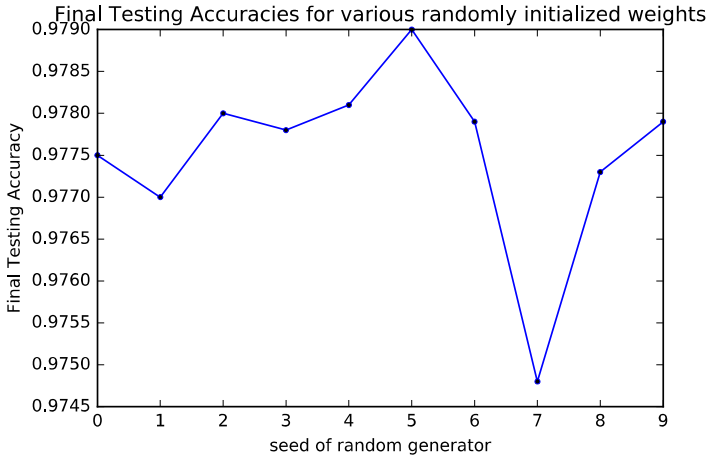
After 100 epochs we have reached at ~97.78% of accuracy for our classification task

We will repeat the experiment for various random initializations and explore the range of testing error and testing accuracy.

Three Layer Model multiple experiments with various random initializations of weights with Glorot Uniform Initialization



Graph 23: Final Testing Errors - Three Layer Model multiple experiments with various random initializations of weights with Glorot Uniform Initialization



Graph 24: Final Testing Accuracies - Three Layer Model multiple experiments with various random initializations of weights with Glorot Uniform Initialization

Maximum Testing Accuracy	0.97899999999999876
Mean Testing Accuracy	0.97752999999999912
Minimum Testing Accuracy	0.97479999999999867
Maximum Testing Error	0.092384759482505838
Mean Testing Error	0.08583166386151371
Minimum Testing Error	0.082530967717292694

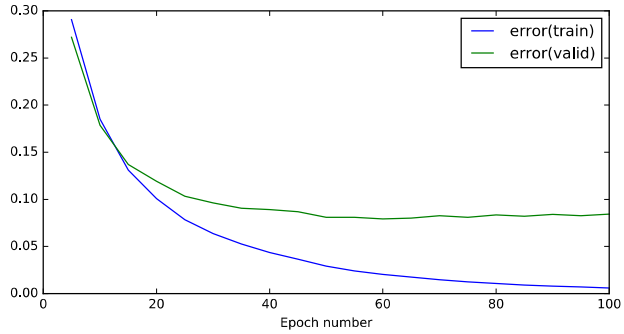
The final accuracy is between ~97.48% to ~97.90%

and the validation error is between 0.082 to 0.092

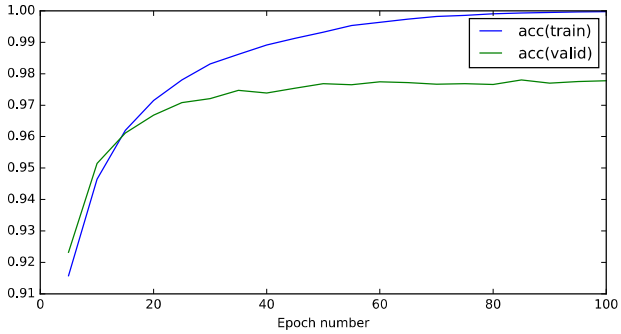
We will use this information for comparisons below

Next we will repeat the experiment by pretraining with a Linear Stacked Autoencoder.

Three layers with Linear Stacked AutoEncoder Pretraining



Graph 25: Training & Validation Error - Three layers with Linear Stacked AutoEncoder Pretraining



Graph 26: Training & Validation Accuracy - Three layers with Linear Stacked AutoEncoder Pretraining

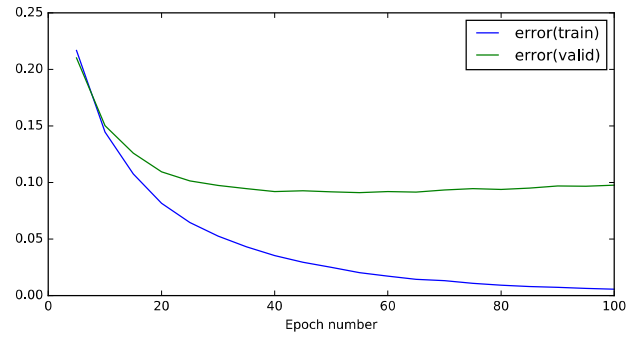
```
Runtime: 168.464905977 seconds
```

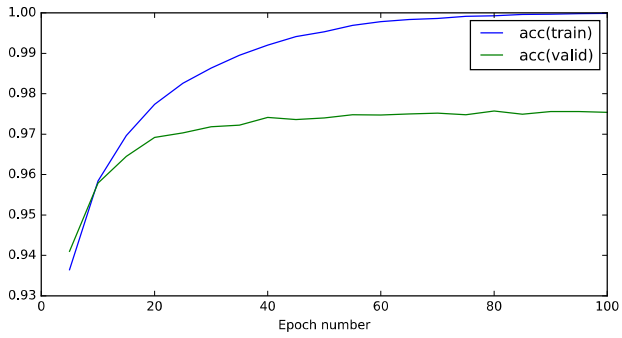
Final Testing Accuracy (percentage 0 to 1)	0.977799999999999
Final Testing Error	0.084256629397097479
Final Training Accuracy (percentage 0 to 1)	0.99974000000000007
Final Training Error	0.0058863214910023351

The final testing accuracy and final testing error are above and below the mean respectively in comparison to experiment with various initialization of the weights which is a first but not particularly strong indicator of the effects of pretraining with a linear stacked autoencoder.

We will repeat the same experiment using a Non-Linear Stacked Autoencoder

Three layers with Non-Linear Stacked AutoEncoder Pretraining





Runtime: `163.474694967 seconds`

`Final Testing Accuracy (percentage 0 to 1)`	`0.97539999999999893`
`Final Testing Error`	`0.09755379924948486`
`Final Training Accuracy (percentage 0 to 1)`	`0.9998800000000001`
`Final Training Error`	`0.0055896778343472566`

Nothing special for the Pretraining with Non-Linear Stacked Encoder in comparison to previous experiments

Comparison Table for Three Layers

Three Layers	Random Initialization of Weights		Linear Stacked Autoencoder	Non-Linear Stacked Autoencoder
Testing Accuracy	`Maximum`	`0.97899999999999`	`0.97779999999999`	`0.97539999999999`
	`Mean`	`0.97752999999999`		
	`Minimum`	`0.97479999999999`		
Testing Error	`Maximum`	`0.09238475948250`	`0.08425662939709`	`0.09755379924948`
	`Mean`	`0.08583166386151`		
	`Minimum`	`0.08253096771729`		

The conclusion is that the pretraining with either Linear or Non-Linear Stacked Autoencoder failed to yield better performance than the random initialization of the weights.

Conclusions for Experiment B

For the MNIST dataset and our current architecture we have proven from previous experiments of this coursework that the metrics are optimal when using three layers. But on this Grand Experiment we saw that pretraining made a difference when the architecture is chosen to be more complex than it is required. This is extremely useful when we have enough computational power to execute an experiment with a very deep and wide neural network but we lack the computational power to repeat this experiment multiple times to find which is the optimal architecture. On that perspective Non-Linear Stacked Autoencoder Pretraining is a useful tool.

However the experiments with pretraining did not yield any spectacular metrics in comparison to the ones with the random initialisation of the weights. In other words we did not achieve the outcome that we were aiming for before these experiments.

The next logical step is to test how pretraining would behave to a more complex problem that would require a deeper neural network.