Machine Learning Practical (MLP)
Coursework 4
Georgios Pligoropoulos - s1687568

Million Song Dataset Classification Problem

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.

We have at our disposal two kinds datasets:

- Fixed input length where we are keeping the central 120 segments. This gives a fixed dimensionality of 3000 features as input.
- Variable input length where the songs vary in the available number of segments. So the number of features vary proportionally. We still have each segment represented by 25 features though. Depending on the sampling that was applied and the length of the song we may have from 120 up to 4000 segments.

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

MSD-10 Classification Problem

A subset of the MSD dataset which contains 40000 instances in the training set and 10000 instances in the validation dataset. In addition we are given 9991 instances as a test set.

Our goal is to classify which of the ten(10) genres correspond to which song, as accurately as possible using artificial neural networks.

MSD-25 Classification Problem

A subset of the MSD dataset that contains 50000 examples for a balanced classification task and each of the 25 classes corresponds to 2000 labelled examples for training and validation. In addition 400 examples are provided for testing.

Our goal is to classify which of the 25 genres correspond to which song, as accurately as possible using artificial neural networks.

Tools

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

Experimentation

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.

Baseline Classifier

In coursework 3 we managed to achieve ~50% validation accuracy on the validation set using our simple dynamic dropout algorithm. However this required a lot of tweaking of newly introduced hyperparameters.

Here we are building a new simpler model with the parameters for the keep probabilities of dropout being static.

So we are targetting MSD-10 classification problem with MLPs and receiving the fixed segments size dataset as input.

In order to be efficient on **development-experiment cycle** we are going to pick a variant of one of the simpler architectures used in coursework3 which provided promising results.

We are choosing to have three hidden layers and the dimensionality from input to output as follows:

(inputs) 3000 → 500 → 500 → 25 → 10 (outputs)

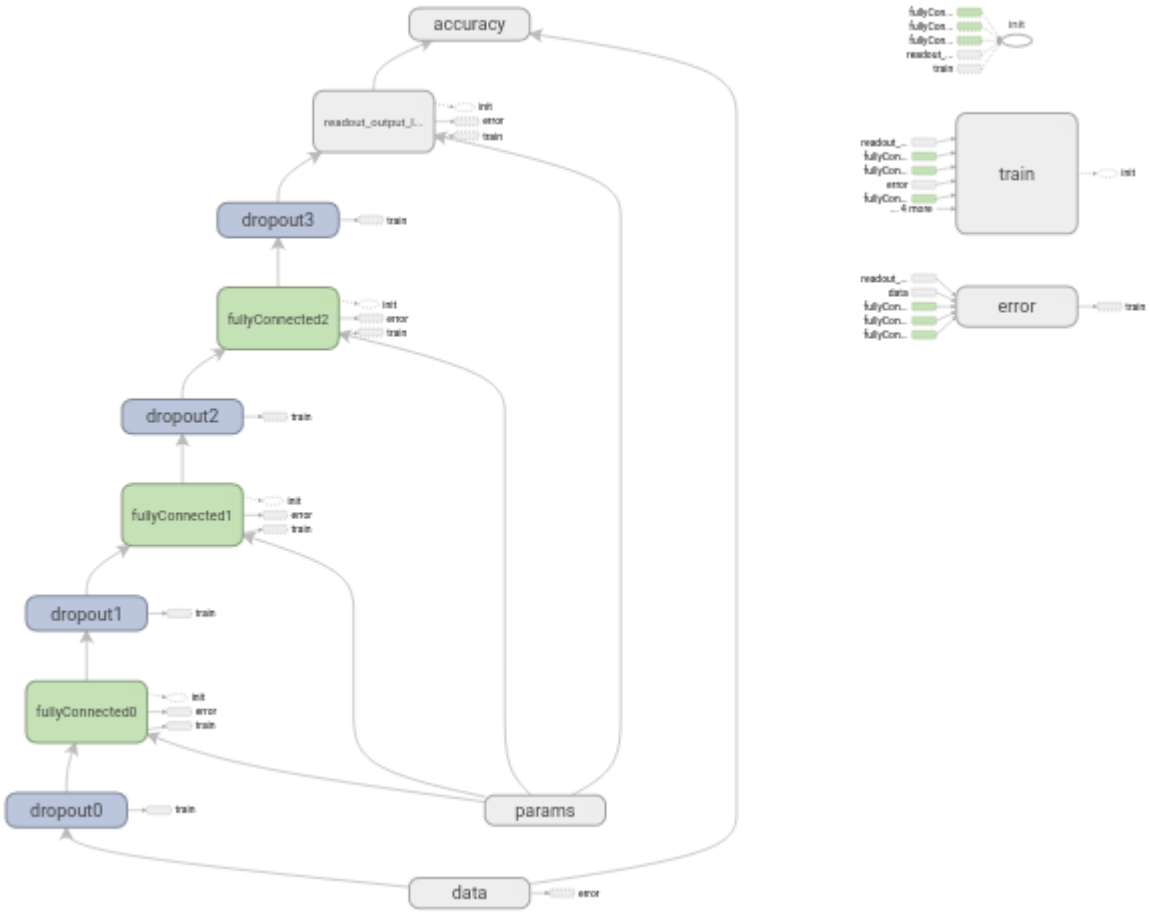
We will be using **affine transformations** which are interleaved with **tanh non-linearity**, and at the end we always have a **softmax output layer**.

Note that we are using **Batch Normalization**, **Dropout** and **L2 regularization** on all layers.

L2 regularization factor is common for all layers and is parametric via a placeholder.

Dropout keep probability of inputs and of hidden layers are also two parameters controlled via two placeholders. Note that the dropout keep probability of all the hidden layers is controlled by the same parameter.

As in coursework 3 the **error metric** is the **mean cross entropy of the batch** which is minimized using **Adam Optimizer**.



Graph 1: Multilayer Perceptron Architecture with dropout, L2 regularization and batch normalization on all layers. Three hidden layers and one readout layer

We are **pretraining** the weights and biases of all hidden layers using the **non-linear stacked autoencoder** from coursework 3. This is a straight forward process without any extra hyperparameters involved.

From coursework 3 we are choosing the **L2 regularization factor** that worked best which was **1e-2**.

We are also keeping the **Learning Rate** at a lower lever than the default value of Adam class in tensorflow. Learning rate is **1e-4**.

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

Bayesian Optimization

Because it would have been difficult to tweak all of the parameters ourselves, we are exploiting the library **scikit-optimize** to get **Bayesian Optimization** to help with finding the optimal dropout keep probabilities.

The space of **dropout keep probabilities** has set to have the **uniform** distribution.

The dropout keep probabilities of the **input layer** are in the real space with **minimum value of 0.7** and maximum of 1.0.

The dropout keep probabilities of the **hidden layers** are in the real space with **minimum value of 0.4** and maximum of 1.0.

The function used for optimization is the **gp_minimize** which includes the hyperparameter **kappa**.
The kappa hyperparameter is a trade-off between exploration versus exploitation and it is set to **1.9**.

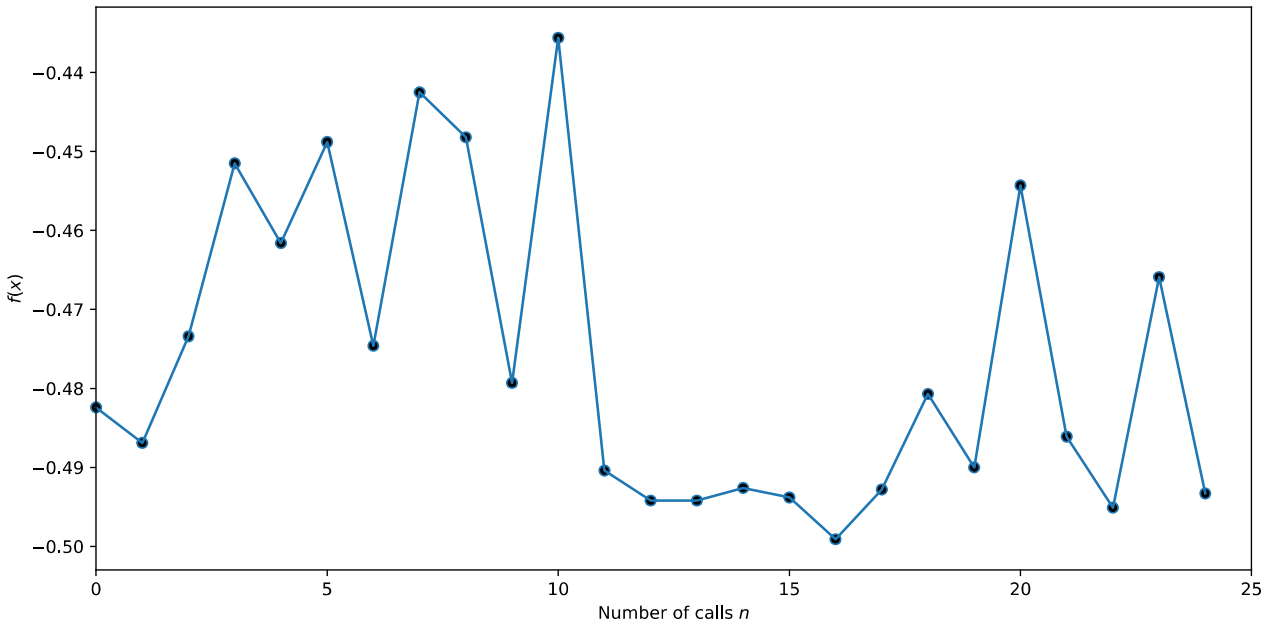
Our objective function is training and validating our deep neural network and we are trying to **maximize** the **maximum validation accuracy** we encountered after **35 epochs**.

We have configured our bayesian optimization to start after executing **5 random initializations**.

We are **evaluating** the objective function **25 times**.

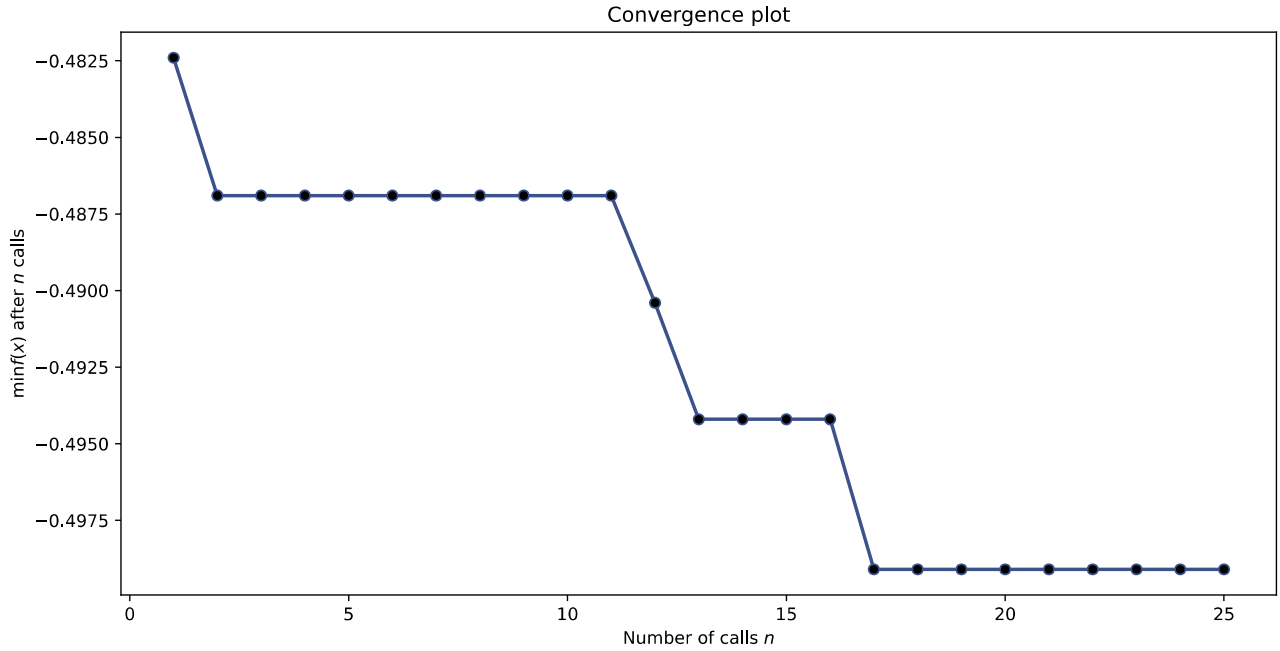
Bayesian Optimization Results

It follows the result of the objective function on every call/evaluation.



Plot 1: Objective Function result of Bayesian Optimization for 35 epochs per call. The objective is the maximum validation accuracy

Here we see the convergence plot which shows the rate at which the bayesian optimization was able to find the minimum value.



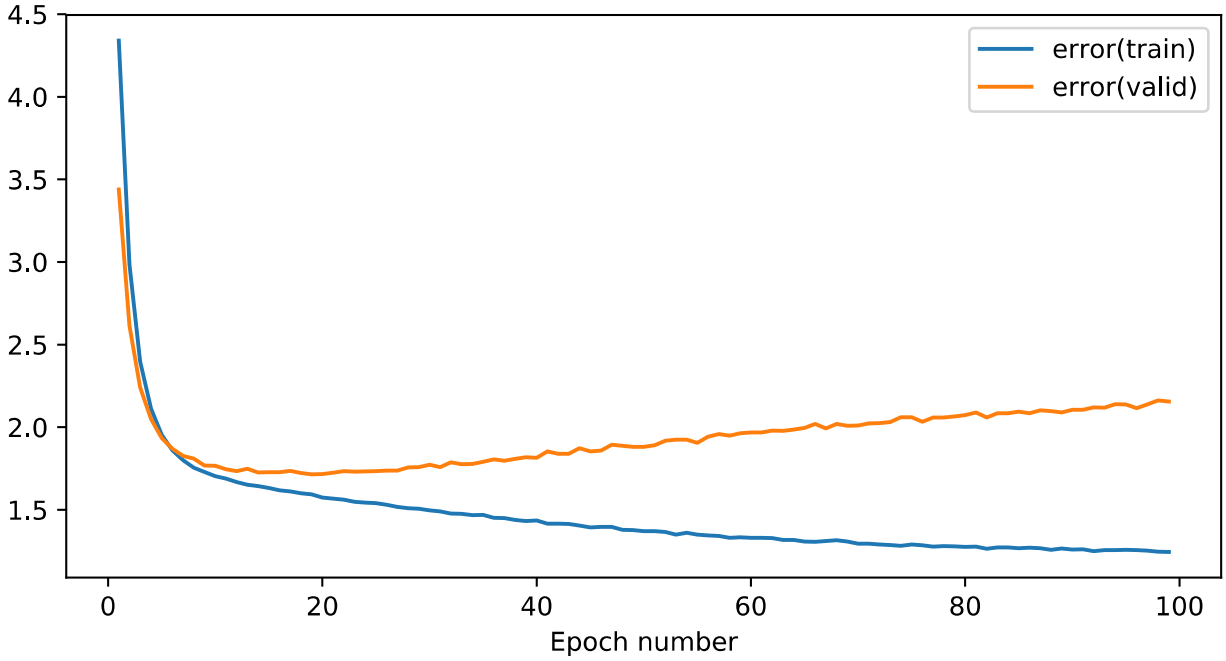
Plot 2: Convergence Plot of Bayesian Optimization for 35 epochs per call. The objective is the maximum validation accuracy. Displays the minimally achieved result for the objective function after every call

So we see that from a ~48% validation accuracy we managed to reach at ~50% through the usage of bayesian optimization. The dropout keep probabilities suggested as the best from the bayesian optimization are:

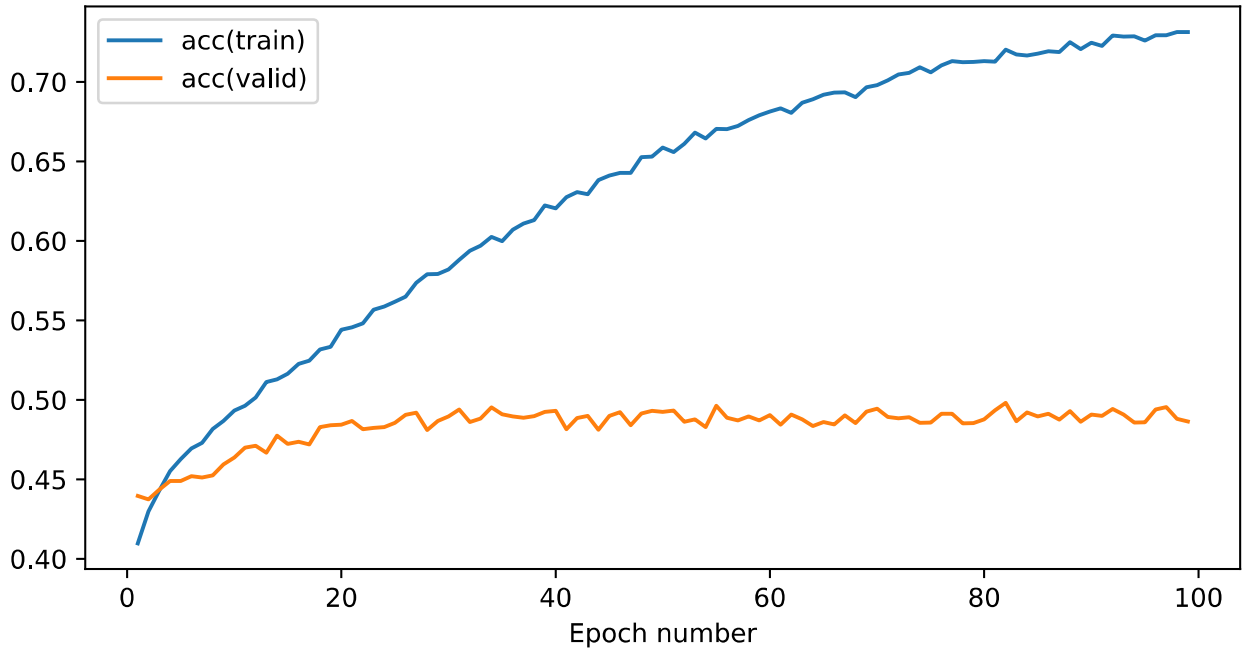
Best dropout keep probability for input layer: 0.7 (or 70%)

Best dropout keep probability for hidden layers: 0.83712466 (or ~84%)

Results by Training Classifier with best parameters



Plot 3: Training & Validation Error – Dropout Keep Prob Input 70% – Dropout Keep Prob Hidden Layers 84% - Layers Dimensionality [3000, 500, 500, 25, 10] – Batch Normalization – Tanh Activation Function – L2 regularization



Plot 4: Training & Validation Accuracy – Dropout Keep Prob Input 70% – Dropout Keep Prob Hidden Layers 84% - Layers Dimensionality [3000, 500, 500, 25, 10] – Batch Normalization – Tanh Activation Function – L2 regularization

Conclusion

We see that we have managed to get a steady validation accuracy performance for the final classifier with the same level of maximum validation accuracy as we had achieved with the dynamic dropout in coursework 3.

However this time there are no extra hyperparameters and there are no such severe oscillations in the plots.

On the other hand we have not achieved optimal regularization since we see a slow increase of the validation error after epoch 20 without being so severe that would impact the classification accuracy.

Research Question: Could Curriculum Learning help the MLP Deep Neural Network to achieve better classification accuracy?

Curriculum Learning is the notion of training a deep neural network gradually going from easier examples to more difficult ones.

In our MSD classification problem the notion of easier versus more difficult is hypothesized to be represented by the cross entropy.
Songs described as easier will have a small cross entropy while songs described as more difficult will have a higher cross entropy.

In a nutshell curriculum learning can be described as follows

- First we train our best version of the neural net
- At the last epoch we get all of the cross entropies for all the instances
- Sort the instances based on the cross entropies.
- Start training the instances from easier to harder

Collect Cross Entropies

We already use cross entropy as an error metric to be minimized so no need to add anything else to the graph.

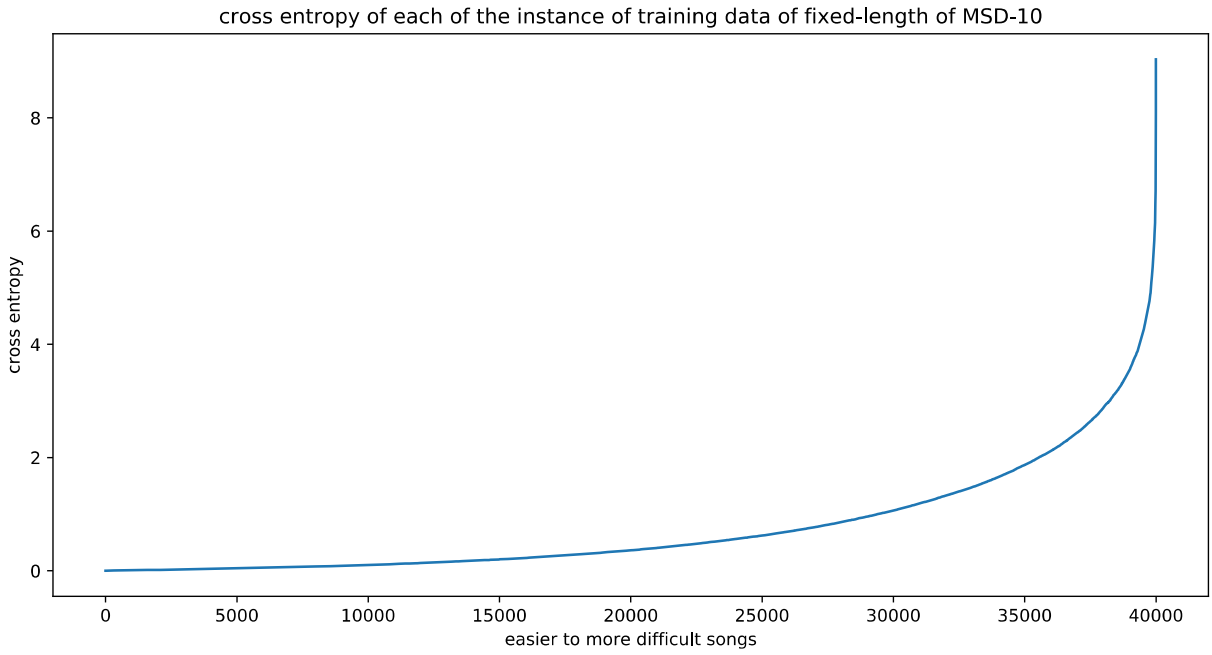
We get our best trained model from the baseline architecture we described above and we run it by collecting all the cross entropies in an array. The indices of this array correspond to the song indices.

Here is the table with the best and worst cross entropies of the training data.

Best Cross Entropies		**Worst Cross Entropies**	
Song ID	**Cross Entropy**	**Song ID**	**Cross Entropy**
36530	0.00018655	4277	6.81745005
22504	0.00040487	20715	6.82365227
23923	0.00047792	10365	7.14912271
14754	0.00050687	21103	7.1554656
36193	0.00051187	35860	7.25000525
5299	0.00063399	19281	7.38539267
8800	0.00075622	33962	7.66791773
7080	0.00076277	22192	8.00147915
23828	0.00079064	17289	8.09997749
21521	0.00084424	22241	9.0351572

Mean value of all cross entropies: `0.7685535595076799`

Variance of all cross entropies: `0.96230627796890367`



Plot 5: Cross Entropies for all of the Song Instances for the fixed-length version of the MSD-10 classification task. They are outputs of the optimal version of the fully trained MLP architecture

Curriculum Learning Data Provider

The `MSD10Genre_Ordered` class takes as input the key_order parameter which is expected to be an array of equal length with the training data containing the specified order of the integers that play the role of the keys on the training dataset.
This is achieved by overriding the `next` function.

The order of the keys can be reversed with the parameter `reverse_order`.
This class does not let `shuffle_order` and `rng` parameters to be set and it always set `shuffle_order` to False at the parent class.

`MSD10Genre_CurriculumLearning` class

This class extends the `MSD10Genre_Ordered` class.

Its constructor accepts the parameter `cross_entropies` which will be the list with cross entropies corresponding to each song unsorted.
Inside the constructor body the cross_entropies get sorted from smaller to larger values and the corresponding keys are passed to the parent constructor.

Parameters:

- `curriculum_step`: We have the flexibility to set how many batches are to be considered in the same group, whether easier or harder. In other words if we set curriculum step to 10 then we consider the first ten batches, or equivalently the 50x10=500 first instances, to belong in the same group of the most easy batches. And then we take the next ten batches and then the next ten and so on.

- `repetitions`: As we train from the curriculum level to the next we might want to persist in the current curriculum level for more than one repetitions. In other words if we are at the first curriculum level and the curriculum step is set to 10 and the repetitions parameter is set to 2 then we are going to train the corresponding 10 batches as many times as the repetitions parameter, in our example twice.
- `repeat_school_class`: This is a boolean parameter which controls whether we are going to keep the instances of the previous curriculum level when transitioning from one curriculum level to the next. In other words if we have 500 instances at the first curriculum level then this boolean control if we are going to train the both the 500 of previous curriculum level and 500 of next curriculum level, entire 1000 instances, on the next curriculum level or only the 500 instances that correspond to the next curriculum level.
- `shuffle_cur_curriculum`: This boolean parameter controls whether the batches that belong in the same curriculum level will be fetched during training sequentially, in the order depended on the cross entropy, or whether they will be shuffled.
- `reverse_order`: This is a boolean parameter to control whether the curriculum learning will happen from easy to hard as is the normal order or from hard to easy, the reverse order.
- `enable_auto_level_incr`: This is a boolean parameter to control whether the curriculum level will increase as we go along the epochs in training, depending on the repetitions parameter, or whether this automatic increment is disabled. By setting it to false the `repetitions` parameter is neglected and we allow the user of this class to call the `updateCurriculumLevel` method to increase the curriculum level manually.

Curriculum Learning Experiment 1

Curriculum Step: 200

Repetitions: 20

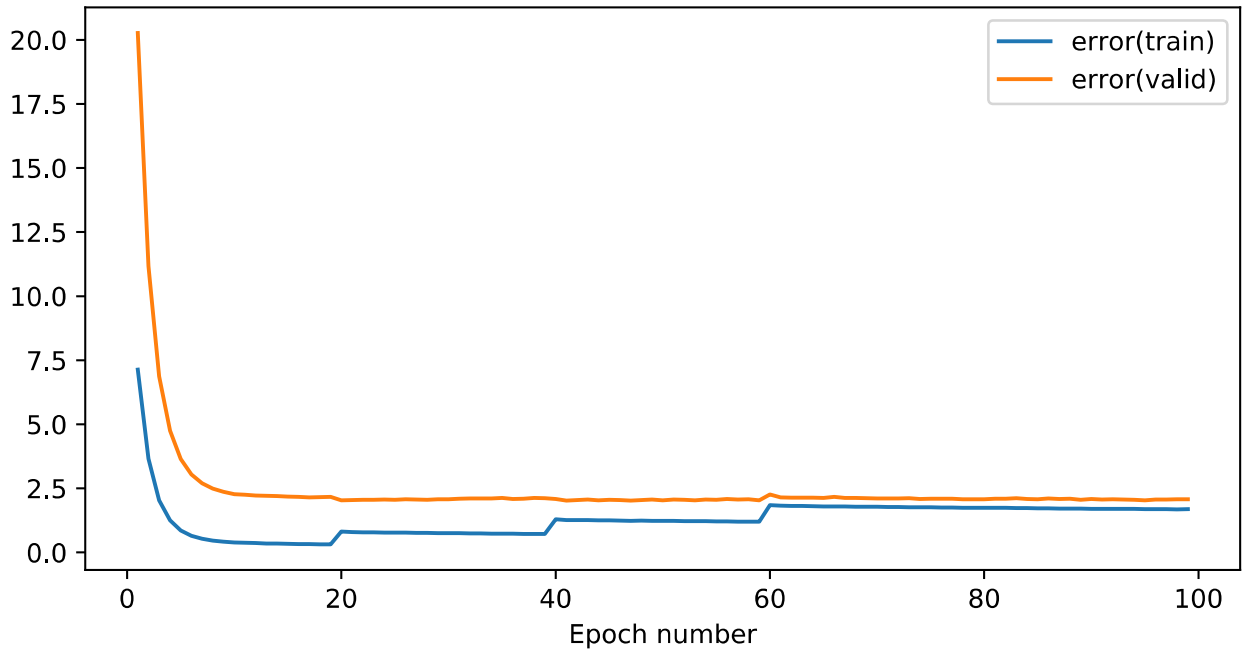
Number of Epochs: 100

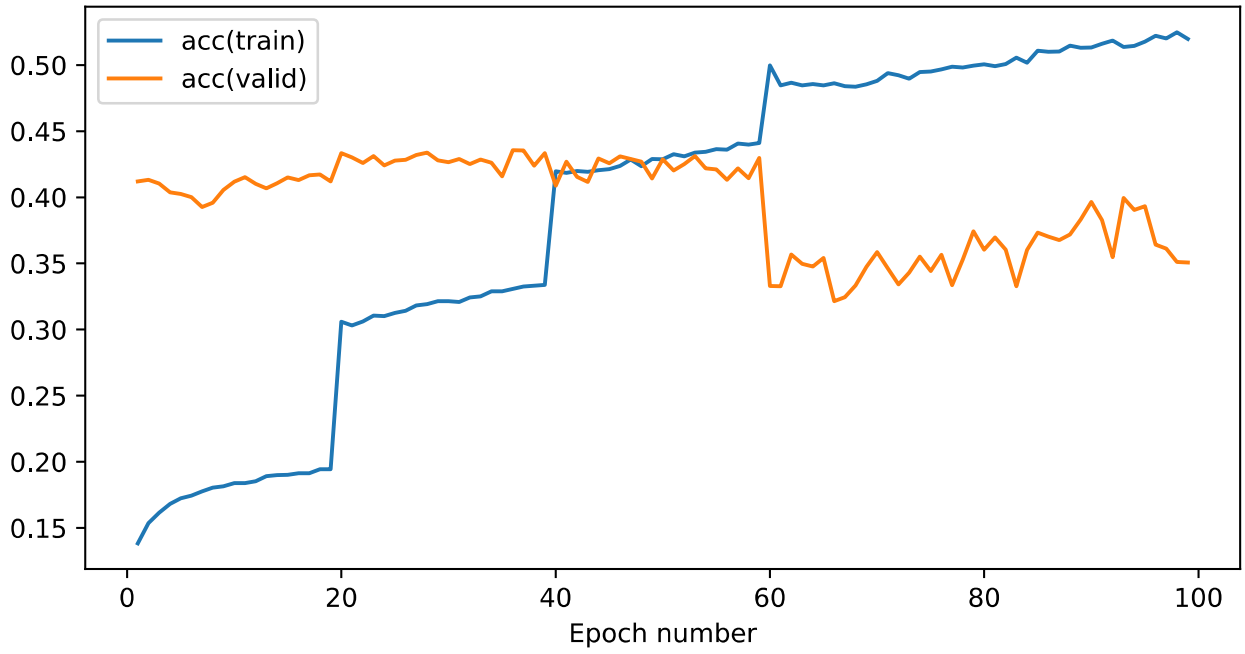
L2 regularization: 1e-1

Initially we are setting a high curriculum step, which breaks the instances in four parts. Also the repetitions of training for each part is set to 20.

We are allowing the network to run for 20 more epochs after the last curriculum level

Results





Conclusions

We notice that having only the 200 easier batches at the beginning is enough to bring the validation accuracy to ~43% which is not too bad in comparison to previous experiments. As the curriculum level increases we note these jumps at the training accuracy and error. Training accuracy is getting better with more instances but we note that the training error is getting worse.

The validation error seems steady but this is not the same with the accuracy. We note that especially when the fourth set of the most difficult songs comes along the accuracy has a sudden drop and it does not seem to be able to increase at the next 40 epochs. There are lots of oscillations which means that the neural network has shifted from a not so good place to a worse one.

The higher L2 regularization that was used in comparison with the baseline provided a steady validation error but could have also be blamed for underfitting the model.

Curriculum Learning Experiment 2

Curriculum Step: 200

Repetitions: 3

Number of Epochs: 32

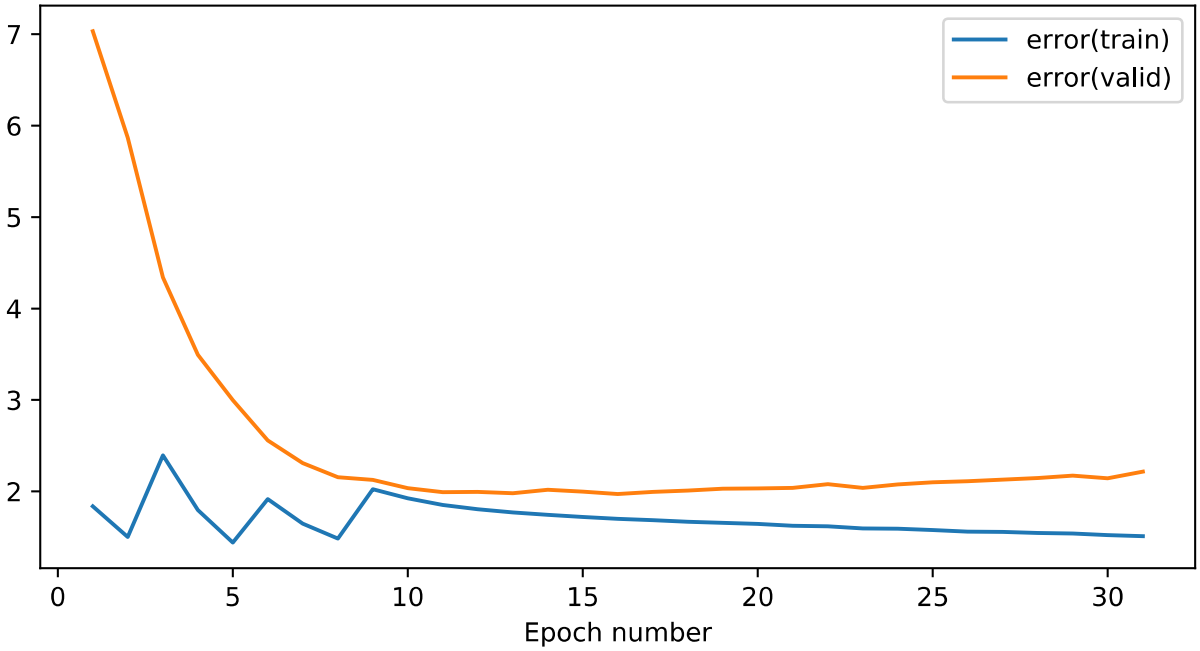
Again we are allowing the training to run for 20 more epochs after the end of the training of the last curriculum level.

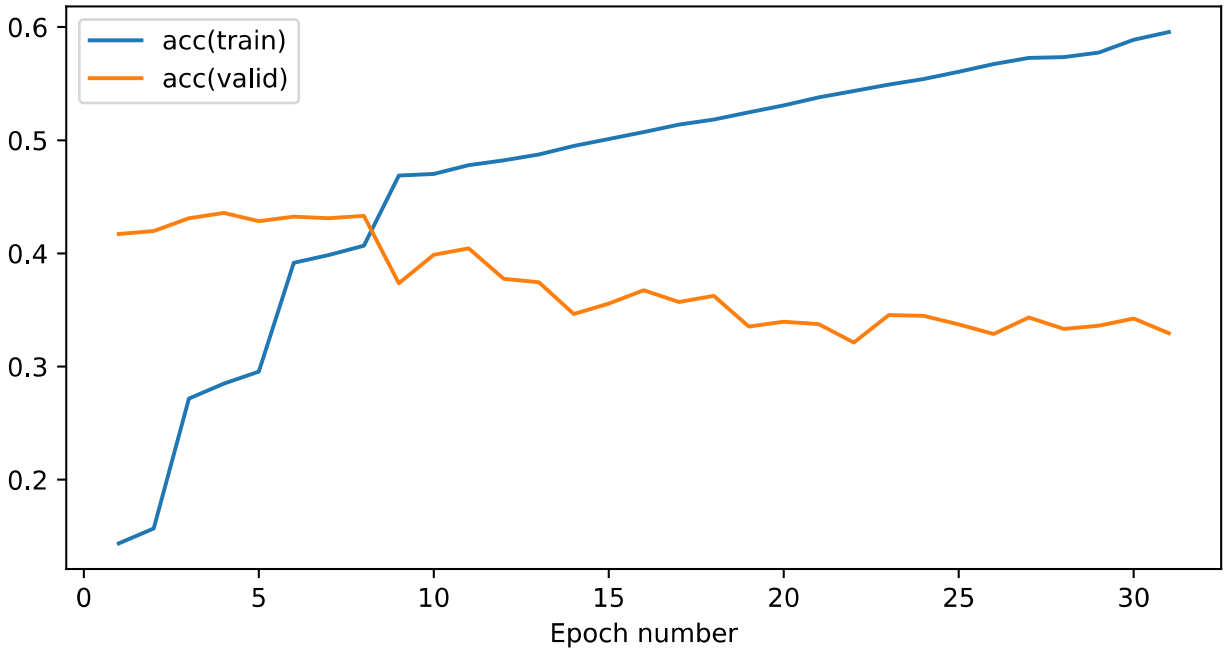
Note that the total number of epochs is proportional to the `repetitions` parameter.

Note that **L2 regularization** is back to the value of **1e-2** as is in the baseline.

In the second experiment we would like to decrease the number of repetitions to avoid giving the chance to the neural network to overfit to the features of the easier example, hypothesizing that this might be the case of the unwanted final results of the previous experiment.

Results





Conclusions

Unfortunately a similar story of the previous experiment is unravelled in the above plots as well but in a smaller scale this time because we have a smaller number of repetitions.

It seems that this pretraining of the neural network with the easier examples which is what it is in fact achieved through curriculum learning does not work as we wanted it. When the neural network is trained with the most difficult examples it is already pretrained in a way that causes the neural network to find an local minimum that is not desirable in terms of regularization. The final effect is that the validation accuracy is worse than our baseline.

The smaller L2 regularization factor, in comparison with the previous experiment, did not provide sufficient regularization and the model suffers from overfitting effects from epoch 15 onwards.

Curriculum Learning Experiment 3

Curriculum Step: 200

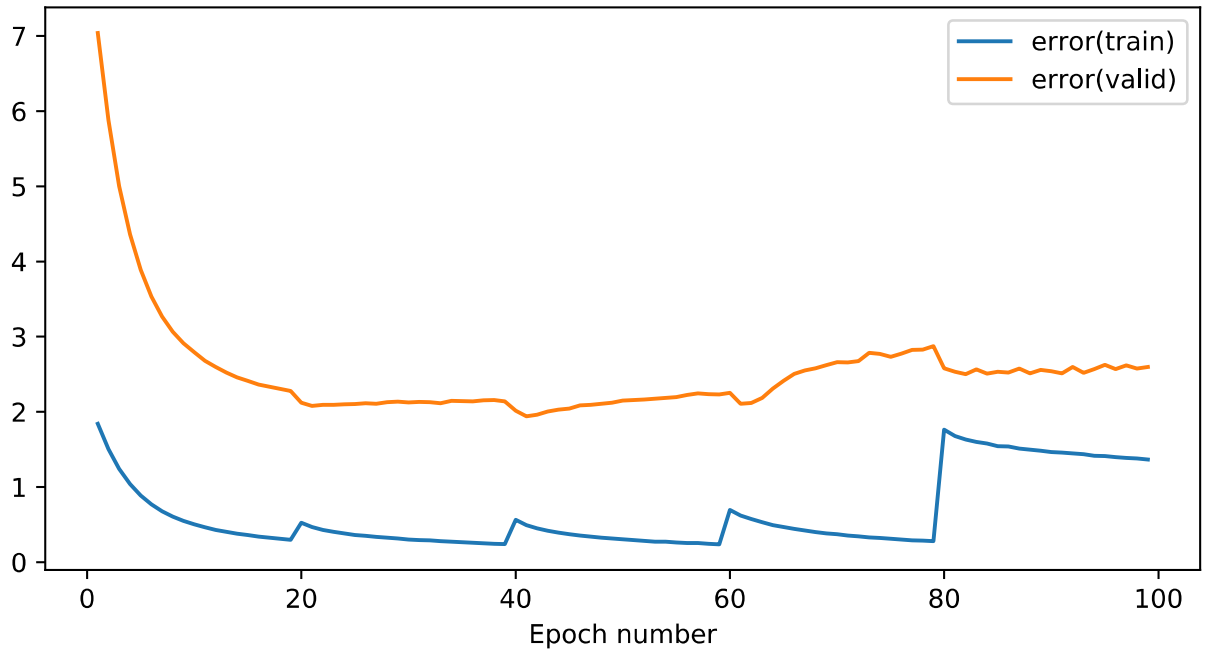
Repetitions: 20

Number of Epochs: 100

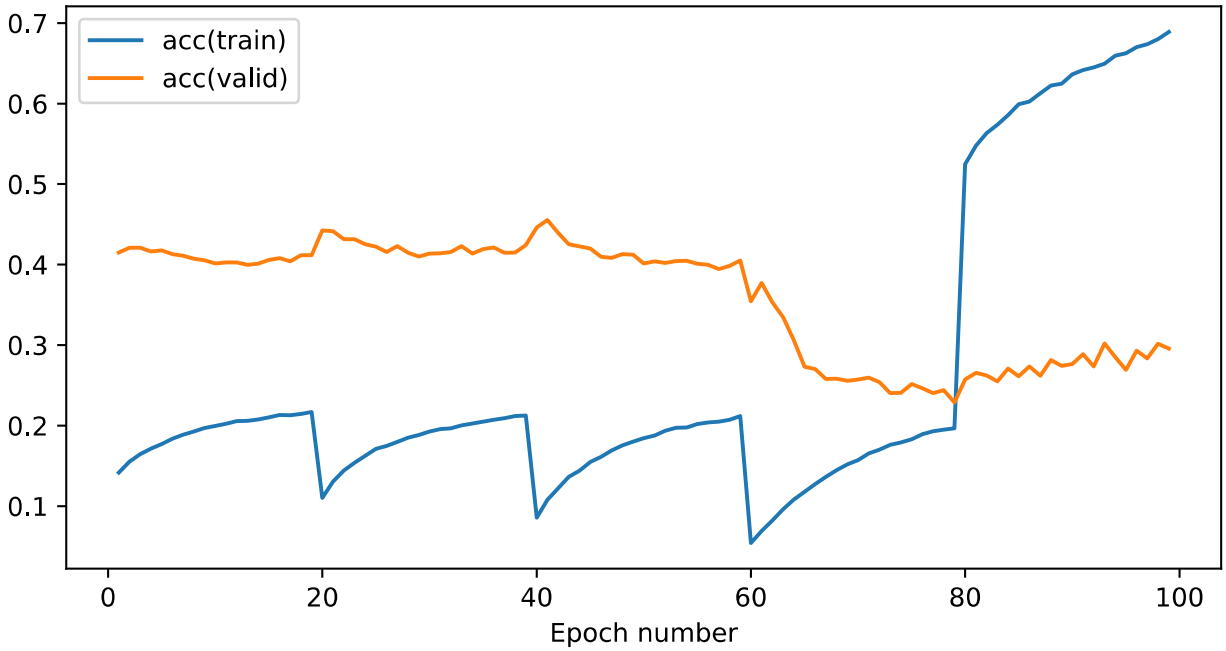
Repeating School Class: False

Note that this is the same as experiment 1 but with two differences. First of all we are letting the L2 regularization be small, because even if there are some overfitting effects we might supress the model with a higher L2 regularization factor and we do not want to do that right now because we are not trying to tweak, rather we are trying to debug.
And secondly we are setting the repeating of school class to false. We are testing to see if we might get a better performance than previous curriculum learning experiments by training only the batches of the current curriculum level at a time and then running 20 epochs by including all the batches.

Results



Plot 6: Training & Validation Error – Curriculum Learning – Curriculum Step: 200 – Repetitions: 20 – Number of Epochs: 100 – Repeating School Class: False



Plot 6: Training & Validation Accuracy – Curriculum Learning – Curriculum Step: 200 – Repetitions: 20 – Number of Epochs: 100 – Repeating School Class: False

Conclusions

Here in this experiment we see that when transitioning from first or second curriculum level to the next, the training error comes with a small rise but not a big one, meaning that the neural network is adequately pretrained to handle the next more difficult set of songs.

We notice that after the 80 epochs where the curriculum training is finished and the neural network is asked to optimize in regards of all the songs in the training dataset, the training error has a very big rise which means that the

pretraining did not work well to train an optimal model in regards to training. We could consider that as not in such thing because we are mostly interested on unseen data and the validation error comes with a small drop and the validation accuracy is not affected a lot which in fact it has a small rising trend.

Curriculum Learning Experiment 4

Curriculum Step: 200

Repetitions: 20

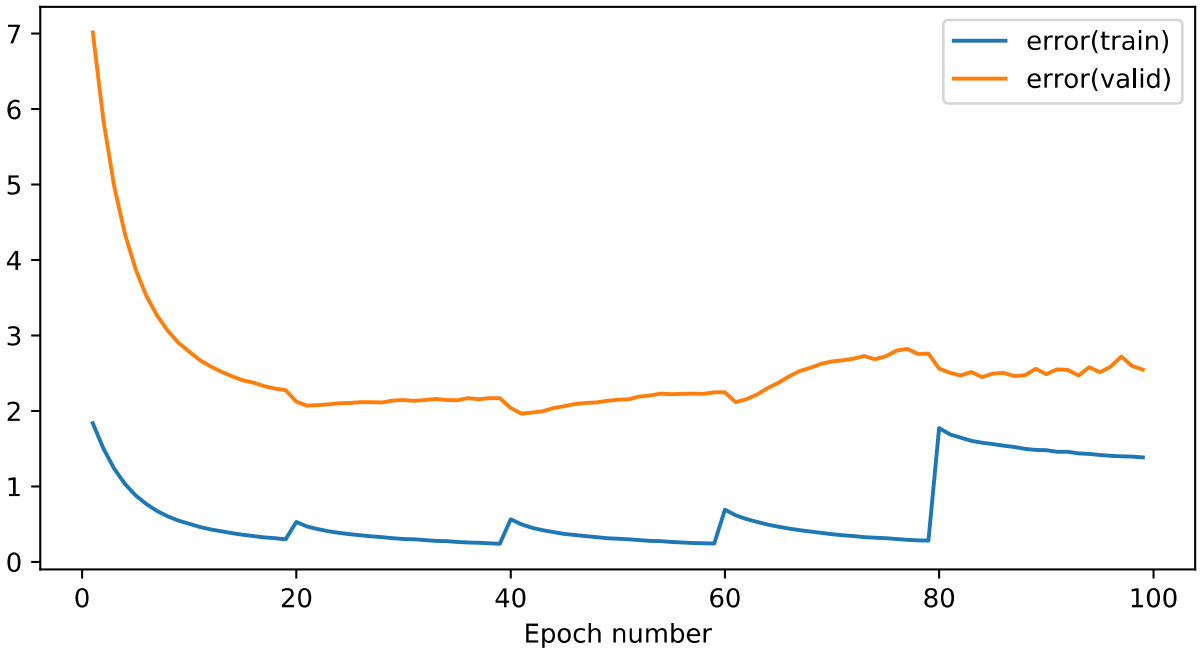
Number of Epochs: 100

Repeating School Class: False

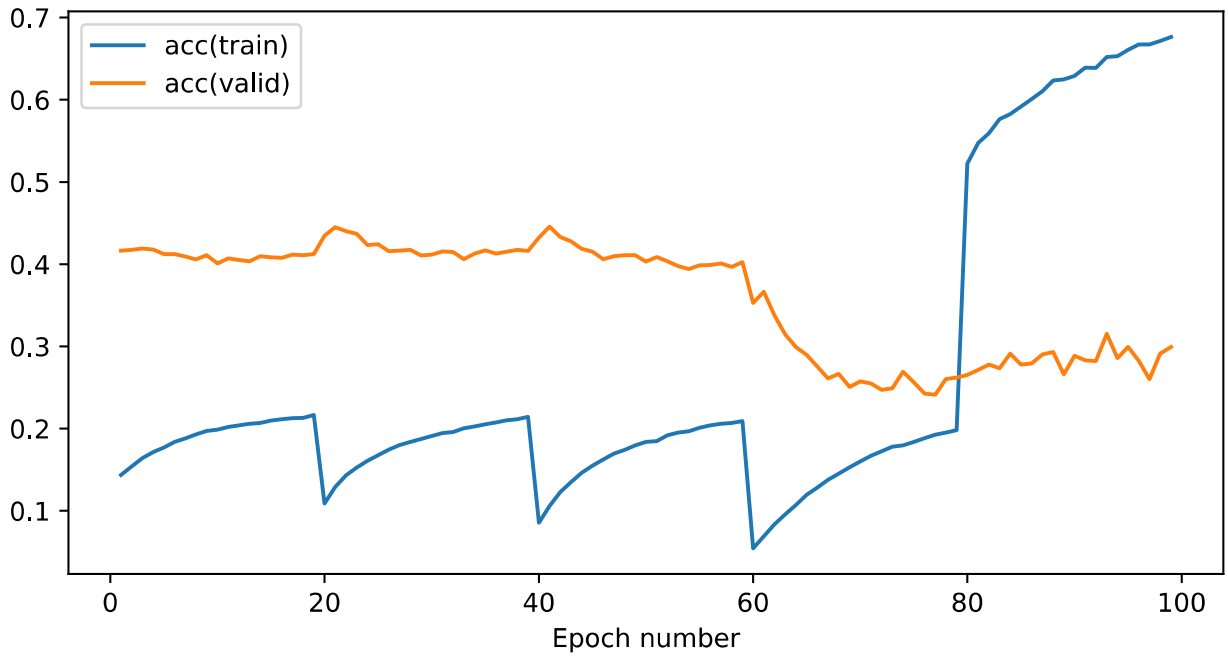
Shuffling Instances of Current Curriculum Level: True

Here we are repeating the above experiment but we are not iterating over the batches, of the same curriculum level, sequentially but rather we are shuffling them on every epoch.

Results



Plot 7: Training & Validation Error – Curriculum Learning – Curriculum Step: 200 – Repetitions: 20 – Number of Epochs: 100 – Repeating School Class: False – Shuffling Instances of Current Curriculum Level: True



Plot 8: Training & Validation Accuracy – Curriculum Learning – Curriculum Step: 200 – Repetitions: 20 – Number of Epochs: 100 – Repeating School Class: False – Shuffling Instances of Current Curriculum Level: True

Conclusions

We see that shuffling the instances plays a vital role on generalizing because with only shuffling we are able to see a slight increase of the validation accuracy the last 20 epochs where all the instances are present in the training process.

However the overall conclusion from the last two experiments where the repeat school class parameter is set to false and not all instances are present it looks to perform even worse than before. The lack of instances finally result in a worse case for the trained model. We will avoid setting this parameter to False in next experiments.

Curriculum Learning Experiment 5

Curriculum Step: 200

Repetitions: 5

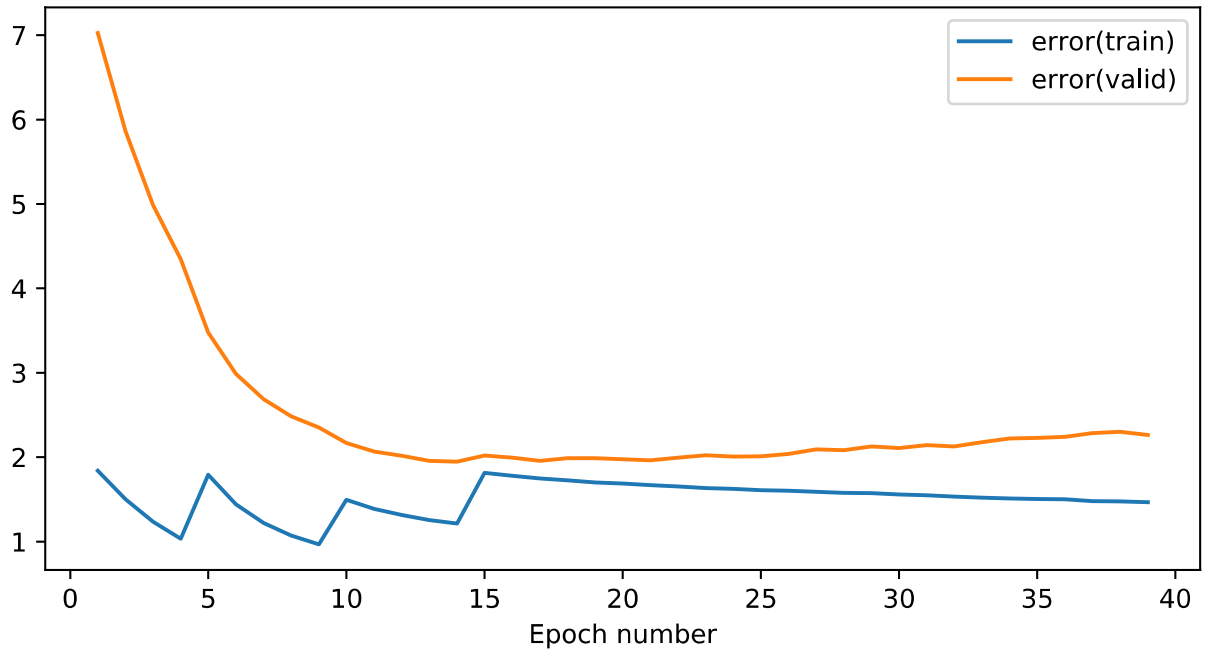
Number of Epochs: 40

Shuffling Instances of Current Curriculum Level: True

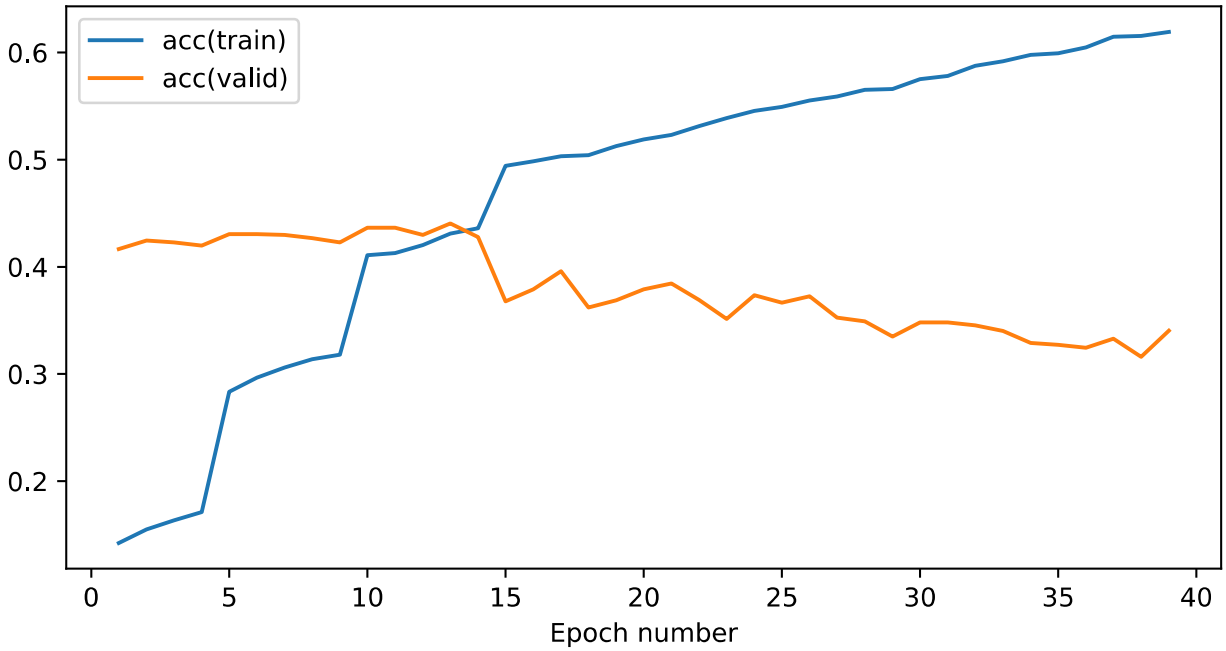
Now we are turning the **Repeating School Class** parameter back to its default which is **True** but we keep shuffling enabled.

Note that we try to also reduce the number of repetitions as we hypothesize that too many repetitions might lead to overfitting to the instances that correspond to a particular curriculum level.

Results



Plot 9: Training & Validation Error – Curriculum Learning – Curriculum Step: 200 – Repetitions: 5 – Number of Epochs: 40 - Shuffling Instances of Current Curriculum Level: True – Repeating School Class: True



Plot 10: Training & Validation Accuracy – Curriculum Learning – Curriculum Step: 200 – Repetitions: 5 – Number of Epochs: 40 - Shuffling Instances of Current Curriculum Level: True – Repeating School Class: True

Conclusions

Trying the experiment within these fewer epochs does not yield as bad results as before as the validation accuracy remains above 30% for the most part. However the same behavior we have seen in previous experiments is repeated here with the drop of validation accuracy after introducing the songs that correspond to the most difficult curriculum level.

Curriculum Learning Experiment 6

Curriculum Step: 50

Repetitions: 10

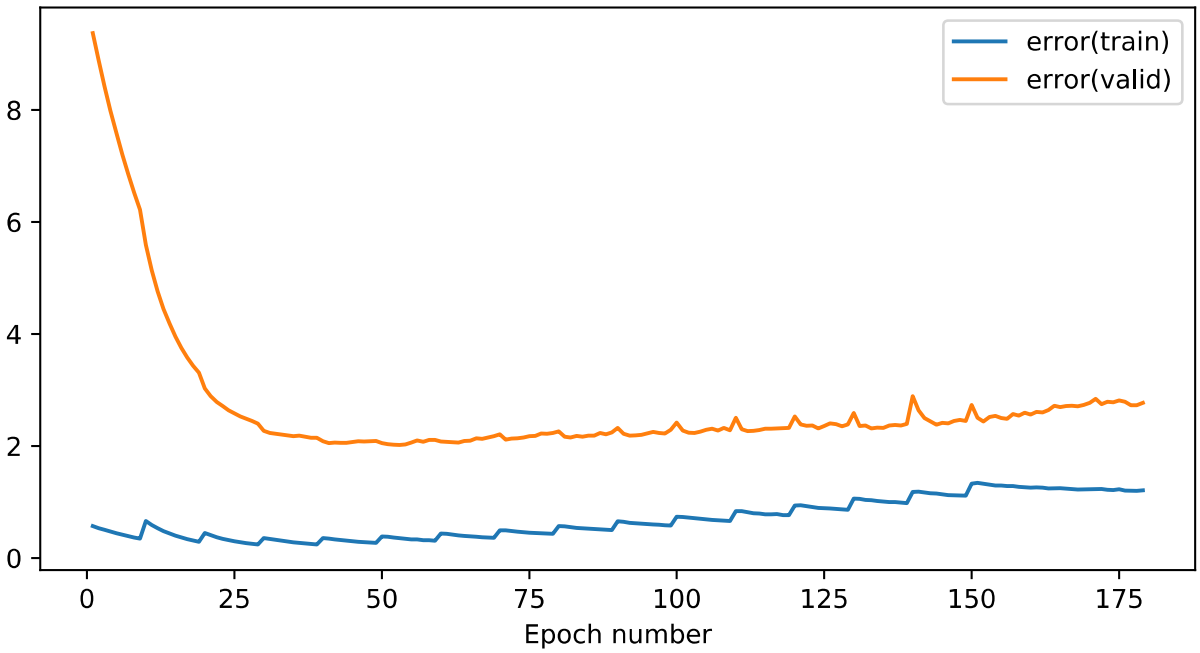
Number of Epochs: 180

Shuffling Instances of Current Curriculum Level: True

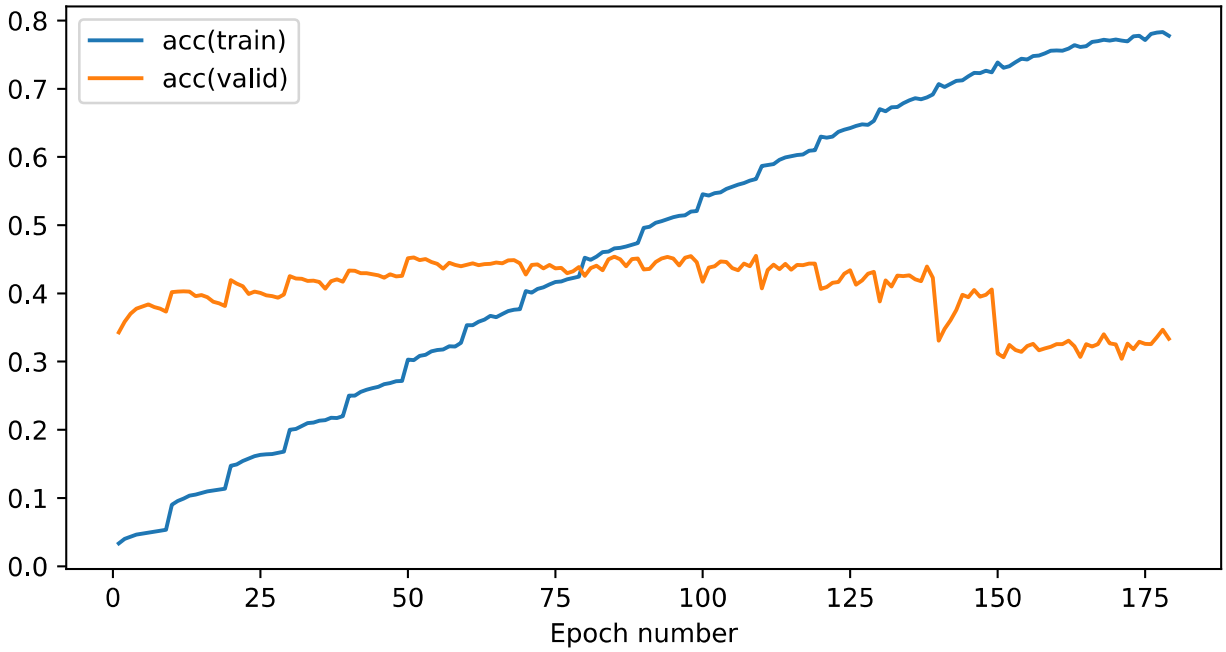
One thing we might give credit for the poor performance is that we might have set a very large curriculum step and we need to train the neural network more slowly by providing more curriculum levels. We do this by setting the curriculum step 4 times smaller than before which effectively creates a total of 16 curriculum levels.

We also increase the number of repetitions because we would like to see if any kind of negative effects we have seen before appear here as well. We feel that 5 epochs per curriculum level is not going to provide the necessary resolution of what is the current behavior.

Results



Plot 11: Training & Validation Error – Curriculum Learning – Curriculum Step: 50 – Repetitions: 10 – Number of Epochs: 180 – Shuffling Instances of Current Curriculum Level: True



Plot 12: Training & Validation Accuracy – Curriculum Learning – Curriculum Step: 50 – Repetitions: 10 – Number of Epochs: 180 – Shuffling Instances of Current Curriculum Level: True

Conclusions

The validation accuracy above peaks at 46% which is the first one from the experiments so far to reach at a higher level closer to the one we had achieved without curriculum learning.

However while achieving high accuracy with having considered only less than the total amount of songs we see that as new songs come along the validation accuracy slightly drops and when ever harder to classify songs come along the validation accuracy drops even further coming to similar results as the ones we saw in the experiments above.

Curriculum Learning Experiment 7

Curriculum Step: 50

Repetitions: 10

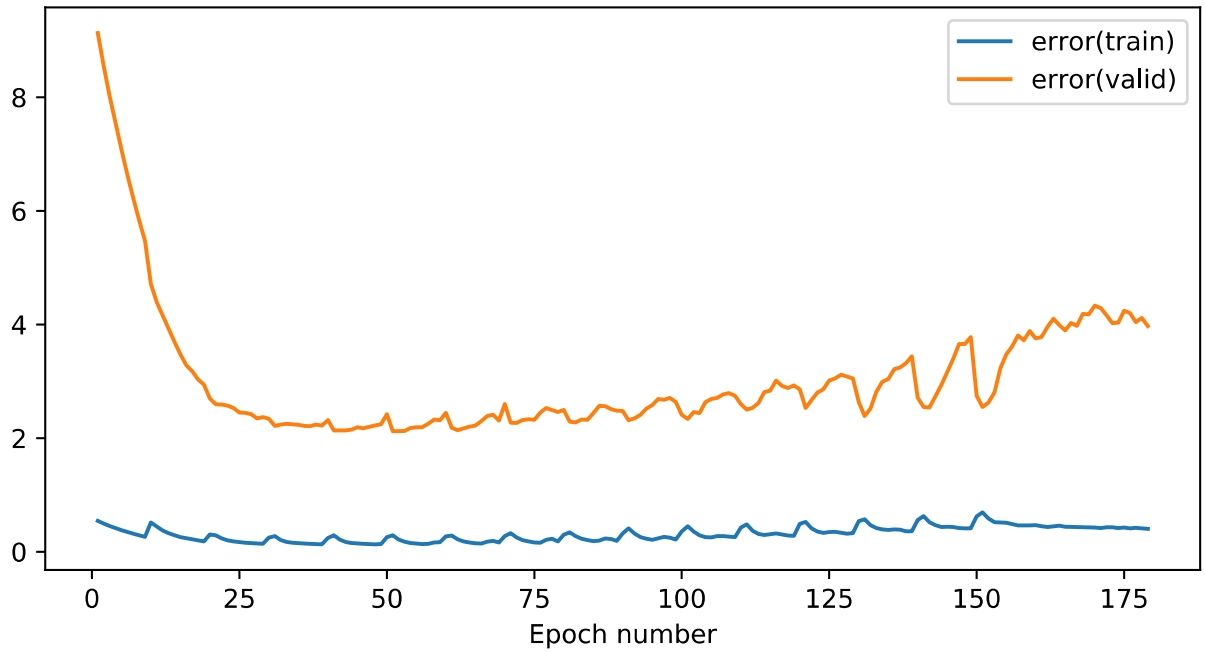
Number of Epochs: 180

Shuffling Instances of Current Curriculum Level: True

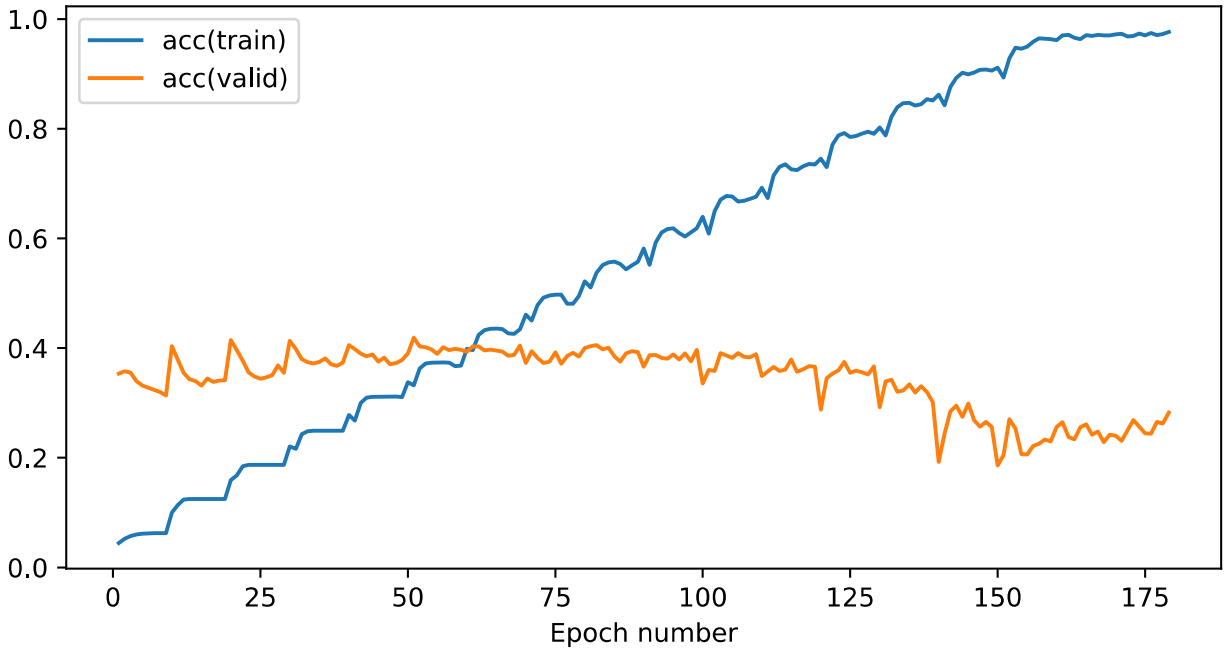
Dropout Keep Probabilities: Both 100%

We see here that both dropout probabilities are set to 100% which means that we have effectively disabled dropout. The role of this experiment is to see how the network behaves without dropout feature because dropout introduces some noise in the layers which might make it too hard to converge to anything useful within the few epochs provided for each curriculum level.

Results



Plot 13: Training & Validation Error – Curriculum Learning – Curriculum Step: 50 – Repetitions: 10 – Number of Epochs: 180 – Shuffling Instances of Current Curriculum Level: True – Dropout Keep Probabilities: Both 100%



Plot 14: Training & Validation Accuracy – Curriculum Learning – Curriculum Step: 50 – Repetitions: 10 – Number of Epochs: 180 – Shuffling Instances of Current Curriculum Level: True – Dropout Keep Probabilities: Both 100%

Conclusions

We can safely conclude comparing this and the previous experiment that dropout is necessary in curriculum learning. With lack of dropout overfitting effects are quite vivid even by leaving l2 regularization enabled. The final validation accuracy is much worse that we would expect.

Curriculum Learning Experiment 8

Curriculum Step: 50

Repetitions: 5

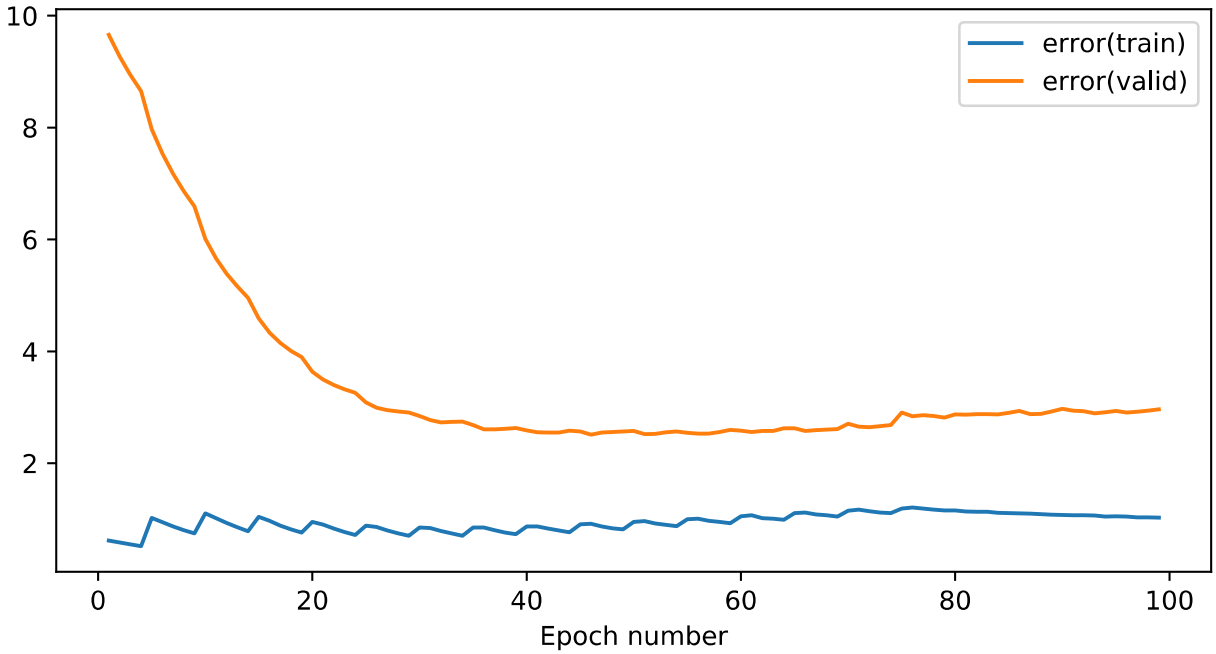
Number of Epochs: 100

Shuffling Instances of Current Curriculum Level: True

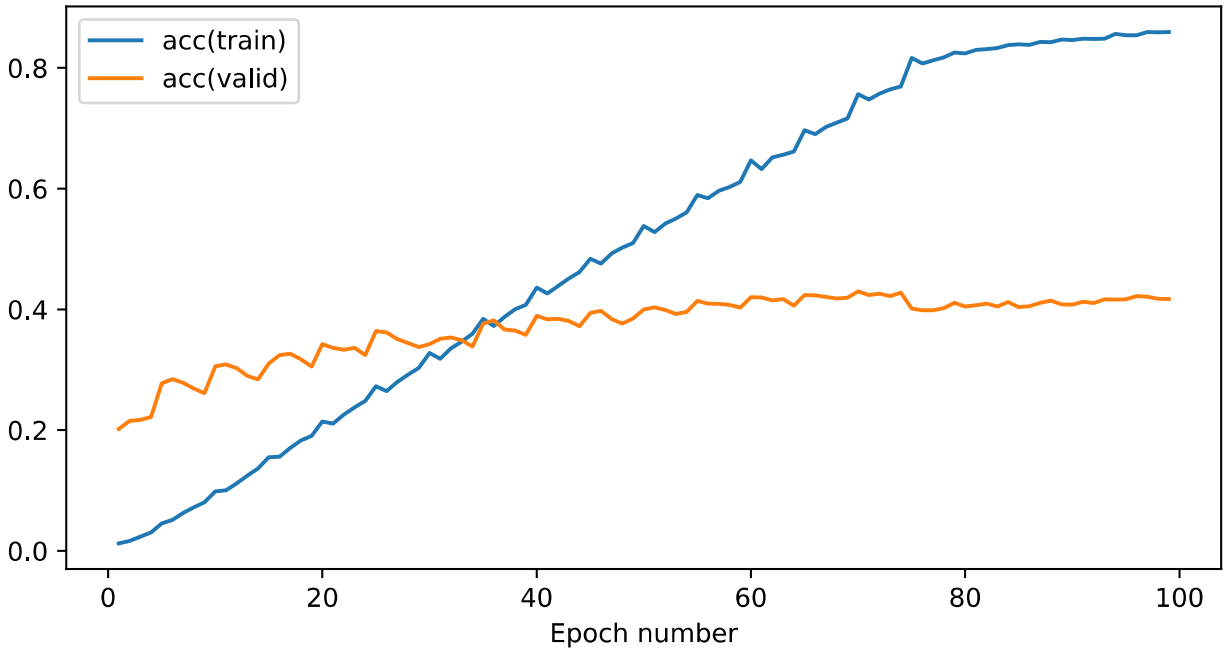
Dropout Keep Probabilities: Both 90%

Since Dropout is vital we are introducing again dropout regularization with relatively high keep probabilities of 90% both for the input and the hidden layers. We are also lowering the number of repetitions to not let the model train too much on a particular curriculum level but keep learning as new curriculum levels come along.

Results



Plot 15: Training & Validation Error – Curriculum Learning – Curriculum Step: 50 – Repetitions: 5 – Number of Epochs: 100 – Shuffling Instances of Current Curriculum Level: True – Dropout Keep Probabilities: Both 90%



Plot 16: Training & Validation Accuracy – Curriculum Learning – Curriculum Step: 50 – Repetitions: 5 – Number of Epochs: 100 – Shuffling Instances of Current Curriculum Level: True – Dropout Keep Probabilities: Both 90%

Conclusions

Note that in comparison to previous experiments our chosen hyperparameters are not too bad in creating a more stable system. The validation error has a small rising trend which suggests overfitting but only on a small scale. However in terms of generalization the model, even if perfoming steadily, it results in a low validation accuracy. The upside is that there are not so big drops on the validation accuracy as the new more difficult songs come along.

Self Paced

Experiments below neglect the repetitions parameter and they follow a self-paced approach. With this approach we do not set a specific number of epochs for the experiment to run but we stop the run at a curriculum level based on conditions. Here the condition is a simple one where the current validation error must be lower than the previous validation error. In other words any kind of sign that we are overfitting by having the validation error increase is considered the end of the current curriculum level and we go to the next one.

Curriculum Learning Experiment 9

Curriculum Step: 50

Number of Epochs: 94

Shuffling Instances of Current Curriculum Level: True

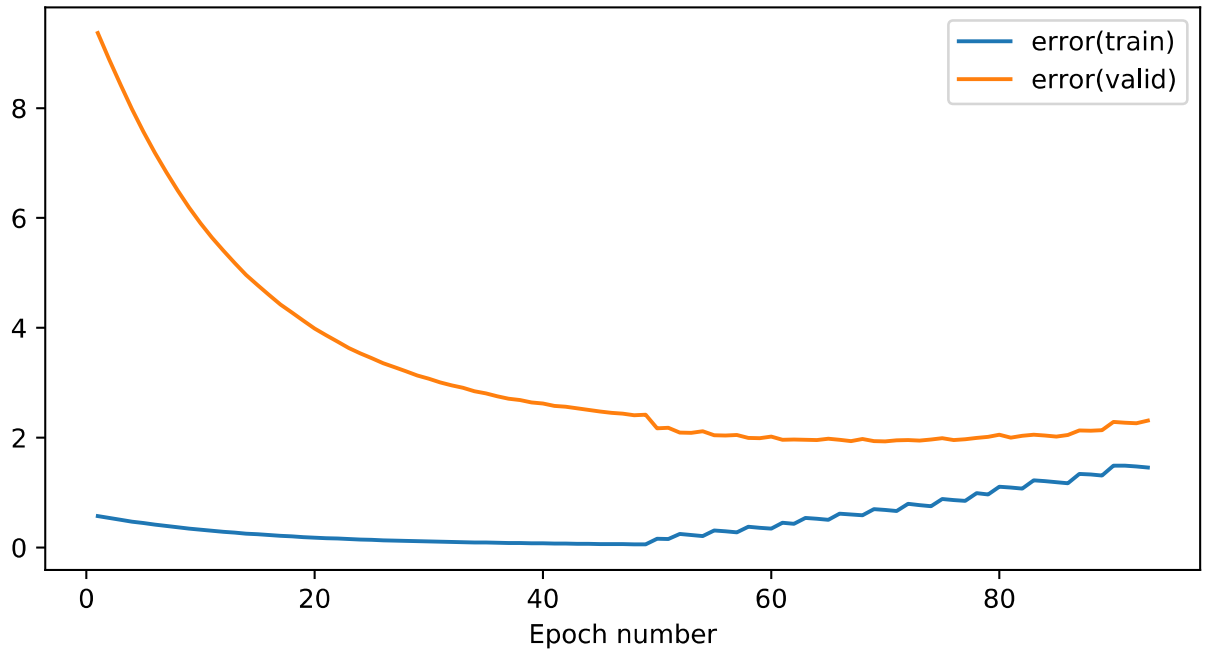
Dropout Keep Probabilities: Both 80%

Self Paced: True

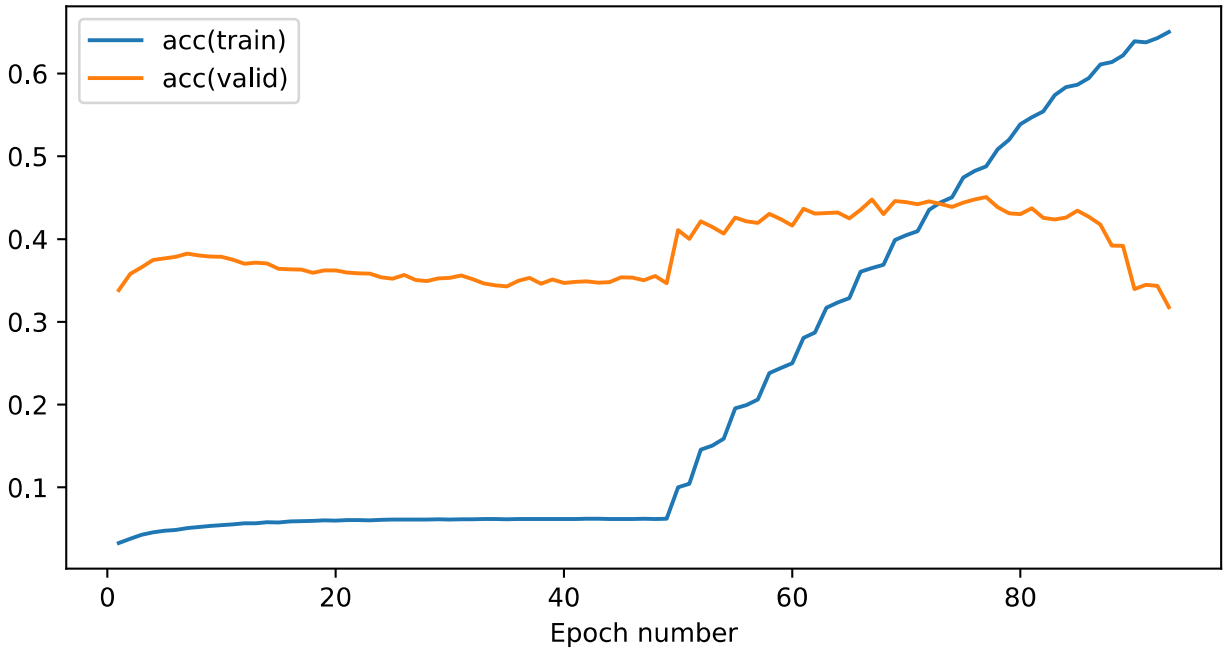
Dropout keep probability is being lowered closer to the values that were set for our baseline.

We want to see the behavior of self-paced curriculum learning.

Results



Plot 17: Training & Validation Error - Curriculum Learning - Curriculum Step: 50 – Number of Epochs: 94 – Shuffling Instances of Current Curriculum Level: True – Dropout Keep Probabilities: Both 80% – Self Paced: True



Plot 18: Training & Validation Accuracy - Curriculum Learning - Curriculum Step: 50 – Number of Epochs: 94 – Shuffling Instances of Current Curriculum Level: True – Dropout Keep Probabilities: Both 80% – Self Paced: True

Conclusions

Note how the system trains on the first curriculum level for several epochs before it starts increasing the curriculum level. Within this period, the first ~50 epochs, even though validation error has dropped significantly there is also a drop in the validation accuracy which means that the system is not particularly getting better.

We can assume that different magnitudes of regularization are required for different levels of the curriculum level. So even being self-paced we are considering that the problem is getting a lot harder because we need either a

smaller dropout keep probability to introduce more artificial noise in the inputs and this dropout keep probbility must be just right, not above or below, or altenatively we need to dynamically set the dropout probabilities and/or the L2 regularization factor.

Summarized Conclusions for Curriculum Learning for Experiments 1-9

So far we have not managed to achieve any metric better with curriculum learning in comparison with previous experiments. So it seems that at the end of the training the neural network is encountered with the most difficult songs, songs it has trouble to classify. These songs cause the largest changes in the variables of the neural network and this may be the cause of a gradient explosion, even if this is short term.

Another way to look at what curriculum learning is trying to achieve is to see it as pretraining. What actually is happening is that we have a good enough trained network with the first group of easy instances and then the network is pretrained in a way that theoritically would be better prepared for the most difficult instances that come along. The key here is the phrase “good enough trained network”. Because it seems that our originally globally optimized hyperparameters as the L2 regularization factor and the Dropout keep probabilities were chosen for another network. Also we do not have an easy way of dynamically changing the hyperparameters as we go along. To be honest we tried this approach in coursework 3 with our simple dynamic dropout algorithm but this did not result in a more stable system.

So we hypothesize that curriculum learning could in fact bring better results but only if in every curriculum level we could achieve a generalized model that would classify unseen data with the same accuracy or better than the previous curriculum level.

(Reverse) Curriculum Learning Experiment 10

Reverse Order: True

Curriculum Step: 50

Number of Epochs: 109

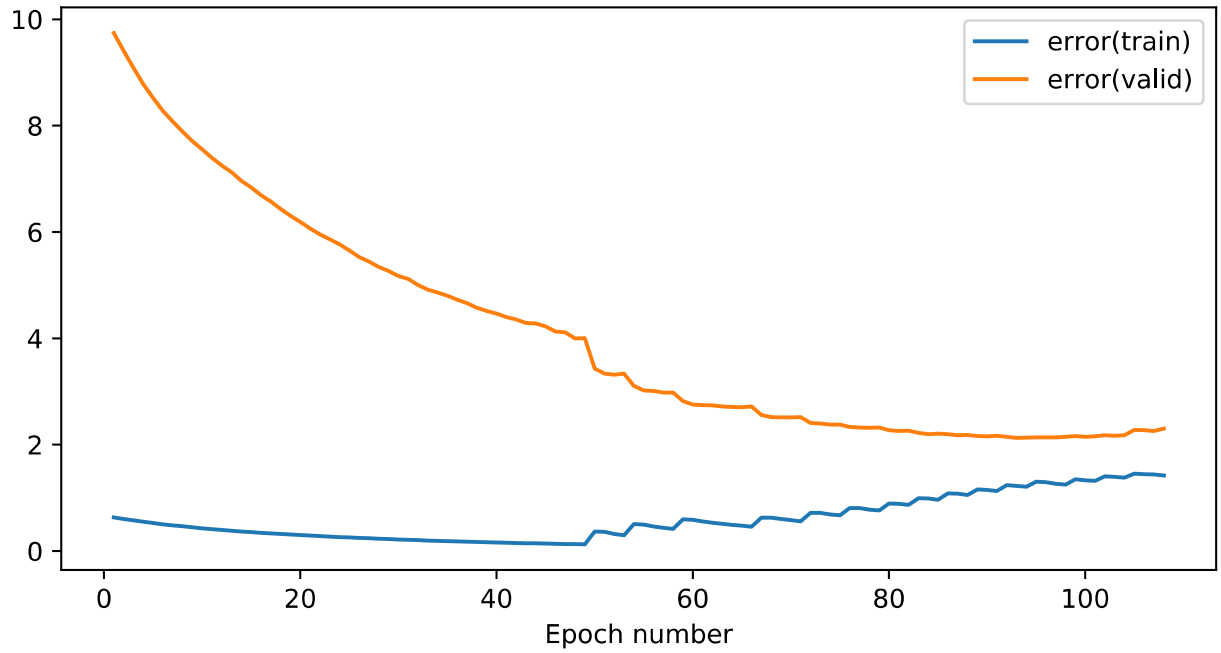
Shuffling Instances of Current Curriculum Level: True

Dropout Keep Probabilities: Both 80%

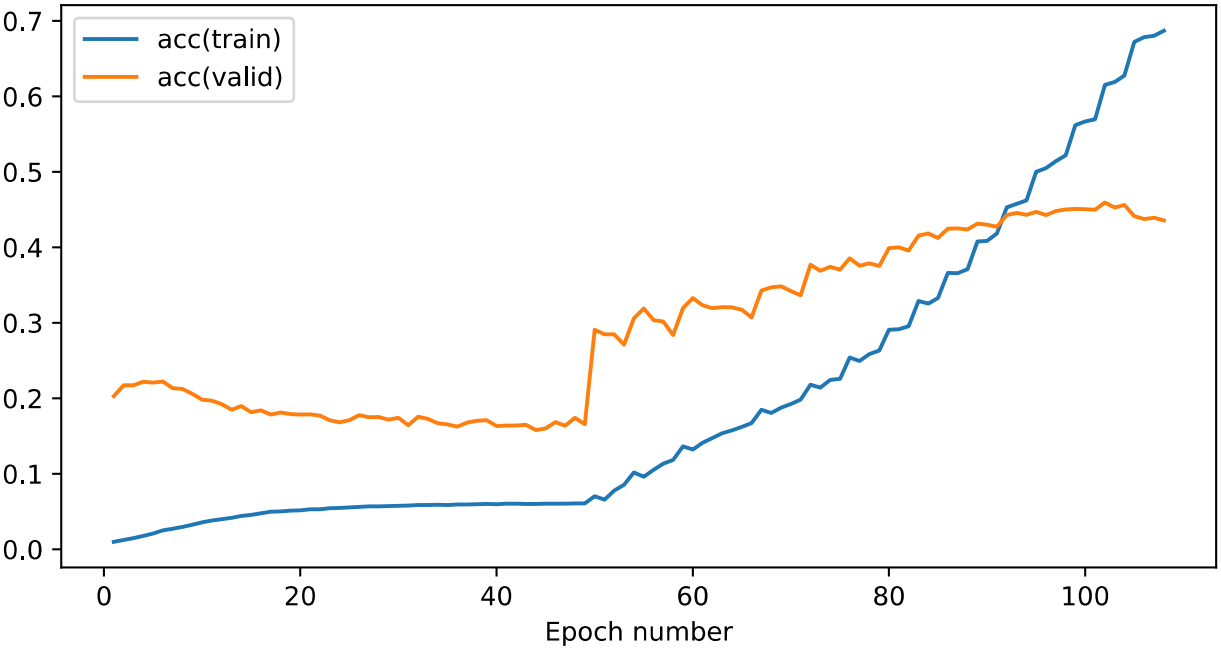
Self Paced: True

Here we are looking curriculum learning from another perspective. We are **reversing the entire procedure** by starting learning from the most difficult instances and moving towards the easier instances. In a sense this could be seen to have similarities with **annealing dropout** because we are starting from instances which are very noisy and we are gradually moving towards instances which are less noisy and easier for our classifier.

Results



Plot 19: Training & Validation Error – Curriculum Learning – Reverse Order: True – Curriculum Step: 50 – Number of Epochs: 109 – Shuffling Instances of Current Curriculum Level: True – Dropout Keep Probabilities: Both 80% – Self Paced: True



Plot 20: Training & Validation Accuracy – Curriculum Learning – Reverse Order: True – Curriculum Step: 50 – Number of Epochs: 109 – Shuffling Instances of Current Curriculum Level: True – Dropout Keep Probabilities: Both 80% – Self Paced: True

Conclusions

From a first look just by reversing the order of the curriculum we have better results in the experiments because we have an almost constant rising trent at the validation accuracy and an almost constant descreasing trend at the validation error. In addition we have achieved a peak at the validation accuracy of 46% which is relatively a good

validation accuracy even though not better than what we have already achieved in previous experiments to make this process of reversed curriculum learning worthwhile.

(Reverse) Curriculum Learning Experiment 11

Reverse Order: True

Curriculum Step: 5

Number of Epochs: 701

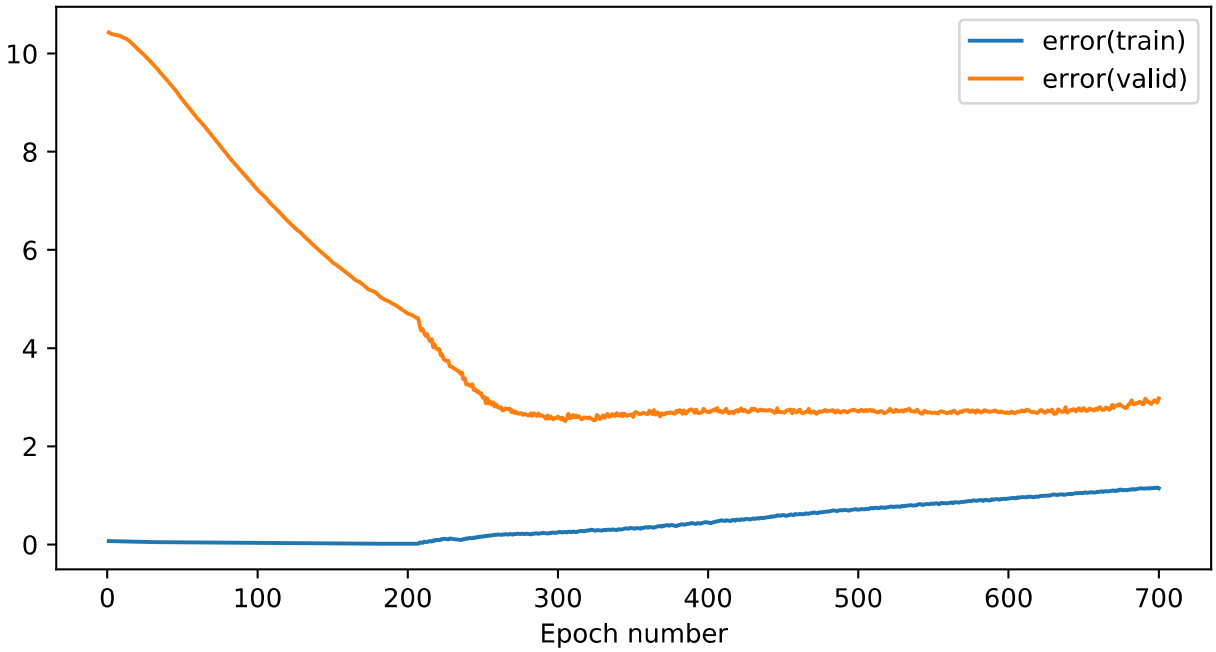
Shuffling Instances of Current Curriculum Level: True

Dropout Keep Probabilities: 70% for input dropout keep probability, 83.7125% for dropout keep probability of hidden layers

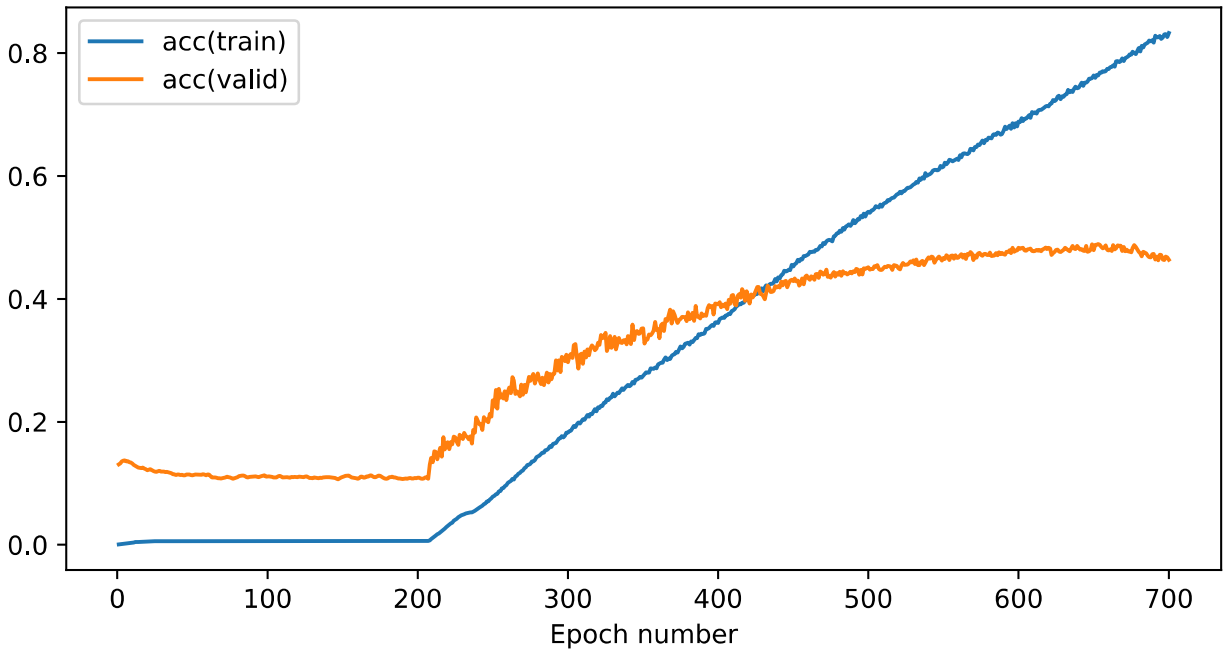
Self Paced: True

Because of the promising results of the previous experiment here we are considering a very similar experiment where the dropout keep probabilities have been set as were the original ones on our baseline classifier. The main difference though is that we have set a much smaller curriculum step of only five(5). This means that the process is going to be much slower and the entire process lasted for 701 epochs.

Results



Plot 21: Training & Validation Error – Curriculum Learning – Reverse Order: True – Curriculum Step: 5 – Number of Epochs: 701 – Shuffling Instances of Current Curriculum Level: True – Dropout Keep Probabilities: 70% for input dropout keep probability, 83.7125% for dropout keep probability of hidden layers – Self Paced: True



Plot 22: Training & Validation Accuracy – Curriculum Learning – Reverse Order: True – Curriculum Step: 5 – Number of Epochs: 701 – Shuffling Instances of Current Curriculum Level: True – Dropout Keep Probabilities: 70% for input dropout keep probability, 83.7125% for dropout keep probability of hidden layers – Self Paced: True

Conclusions

Following a more thorough approach where the curriculum step was quite smaller the results are better than before since for several epochs the validation accuracy peaked at 49% which is equivalent to what we had achieved with our baseline classifier. The drawback is that this process was much slower and it did not achieve better results to make it more attractive in any way.

Unfortunately still we lack either the time resources or the computational resources to be able and properly regularize our neural network, in a brute force way, at every curriculum step and thus provide the best possible result.

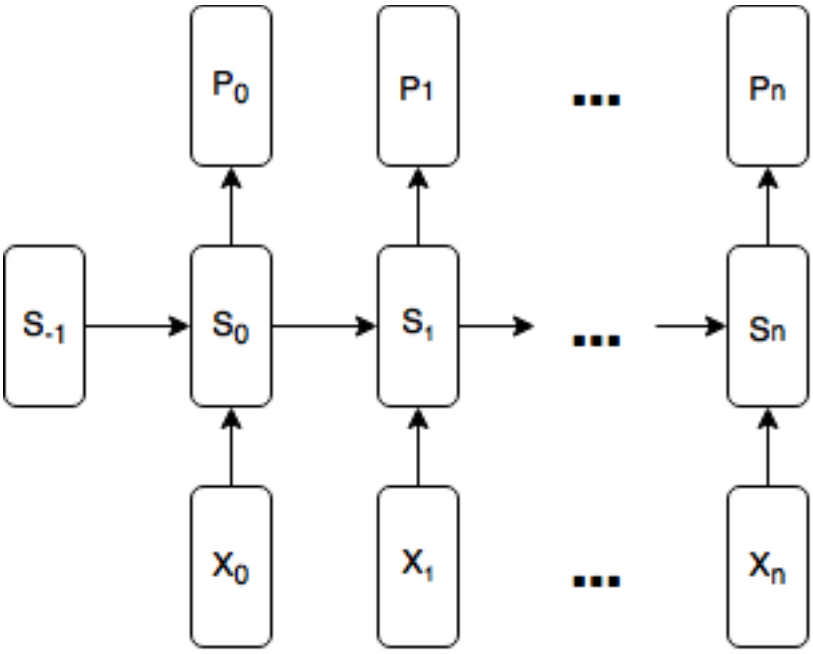
On the other hand it is difficult to say if we have reached the global optimum for our MLP classifier and blame that of reaching our upper bound of classification performance with MLP architecture.

Since songs are series of time depended segments then it makes sense to try an architecture which takes advantage of time dependencies such as the Recurrent Neural Network.

Research Question: Could the basic Recurrent Neural Network help to achieve better accuracy for MSD classification task?

Since the information of a song is played sequentially through time it is expected that there will be temporal dependencies between parts of the song that are close on the timeline of the song.

We are expecting that if we can exploit these temporal differences between the segments of the song that we are going to build a classifier that will perform better. One architecture for deep neural networks that provides this capability of exploiting sequentially depended inputs are the Recurrent Neural Networks. The architecture is as the following diagram:



Graph 2: Basic Recurrent Neural Network

Using RNNs natively implemented in Tensorflow – Short Discussion

Here we are explaining shortly what was attempted using Tensorflow's native RNN functionality and why this did not work as expected.

We will not be including the multiple experiments that were based on this implementation because they were executed under wrong assumptions on how Tensorflow `tf.nn.rnn` function (https://www.tensorflow.org/versions/r0.12/api_docs/python/nn/recurrent_neural_networks#rnn) really works.

So we are removing all the pages that are related with the description of the related architecture and experiments in order to avoid providing any misleading results.

Under those terms do not take into account the model related to these experiments implemented in the class `MyBasicRNN` found in python module `rnn.MyBasicRNN`.

The Basic RNN and the implementation of the entire RNN can be implemented using basically only two lines:

```
cell = tf.nn.rnn_cell.BasicRNNCell(state_size)  # tanh is default activation
rnn_outputs, final_state = tf.nn.rnn(
    cell, rnn_inputs,
    initial_state=init_state, sequence_length=np.repeat(num_steps, self.batch_size)
)
```

Quoted from documentation: *"If the sequence_length vector is provided, dynamic calculation is performed. This method of calculation does not compute the RNN steps past the maximum*

sequence length of the minibatch (thus saving computational time), and properly propagates the state at an example's sequence length to the final state output.”

Conclusions

Unfortunately `sequence_length` parameter is only vaguely described in the documentations without any examples of how it can be used properly. Lessons learned from this process was that documentation matters a lot when source code is being used by people other than the ones who originally wrote it and also documentation without examples is inefficient and usually incomplete.
Statements backup by this paper: Bloch, Joshua. "How to design a good API and why it matters." *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006.
In addition the well known in the software development community stackoverflow.com website is supporting the same principles for documentations which are found in this link: http://stackoverflow.com/documentation.

RNNs manually implemented using Tensorflow

The advantage of using native Tensorflow RNN functions would be to reduce any boilerplate code and exploit the optimized for speed performance RNN implementation.
Since this approach did not work as we wanted it, we will fallback into creating our own RNN implementation where we will handle manually all the RNN cells, by having an unfolded RNN network with the cells share their weights.

We will implement manually the truncated recurrent neural network where backpropagation is limited to the number of RNN steps. In order to achieve that we will manually handle the connection of the final state of the previous part of the song with the initial state of the next part of the song. We are defining as part the group/set of segments we are taking into account at once on every run of the session due to the limited number of recurrent steps which our RNN will support.

So we have a fixed number of *n+1* steps for the recurrent neural network equal to the number of inputs, here segments, that we pass to the RNN. We also see that we have the possibility of *n+1* outputs.
The first state S_0 is feeded with the output of the previous set of segments unless this is the first set which in that case the S_{-1} is an array of zeros.
To be more explicit we are explaining with an example that if we take RNN with 10 steps then we are dividing the 120 segments of each song by 10 which gives 12 sets of segments.

In order for the Recurrent Neural Network to work we need to feed the segments of the song to our inputs and not the flat structure of all the 120 segments as we did in previous experiments that is why we need a custom Data Provider for this case.

We created `MSD10Genre_120_rnn_DataProvider` class to use as a data provider suitable for the RNN case. On initialization we calculate how many parts are needed for the number of steps for the RNN which is provided as parameter to the constructor.
The `next` method is overridden to provide the batches accordingly. The way that this is handled is that we first grab the inputs batch and the targets batch as provided from the parent class. The targets batch is left intact, while the inputs batch is reshaped from 50x3000 to 50 **x** number of parts **x** number of RNN steps **x** length of each segment. As the next method is being called we are iterating over each part from the ones we created until all the parts have been consumed. After all the parts have been consumed, we call again the `next` method of the parent class to fetch the next song of data and so on.

We are always using as outputs the targets for our classification task since we do not have anything else to use as output. In other words there is no symmetry between input and output. We have multiple inputs while only one output, the class that this song belongs to.

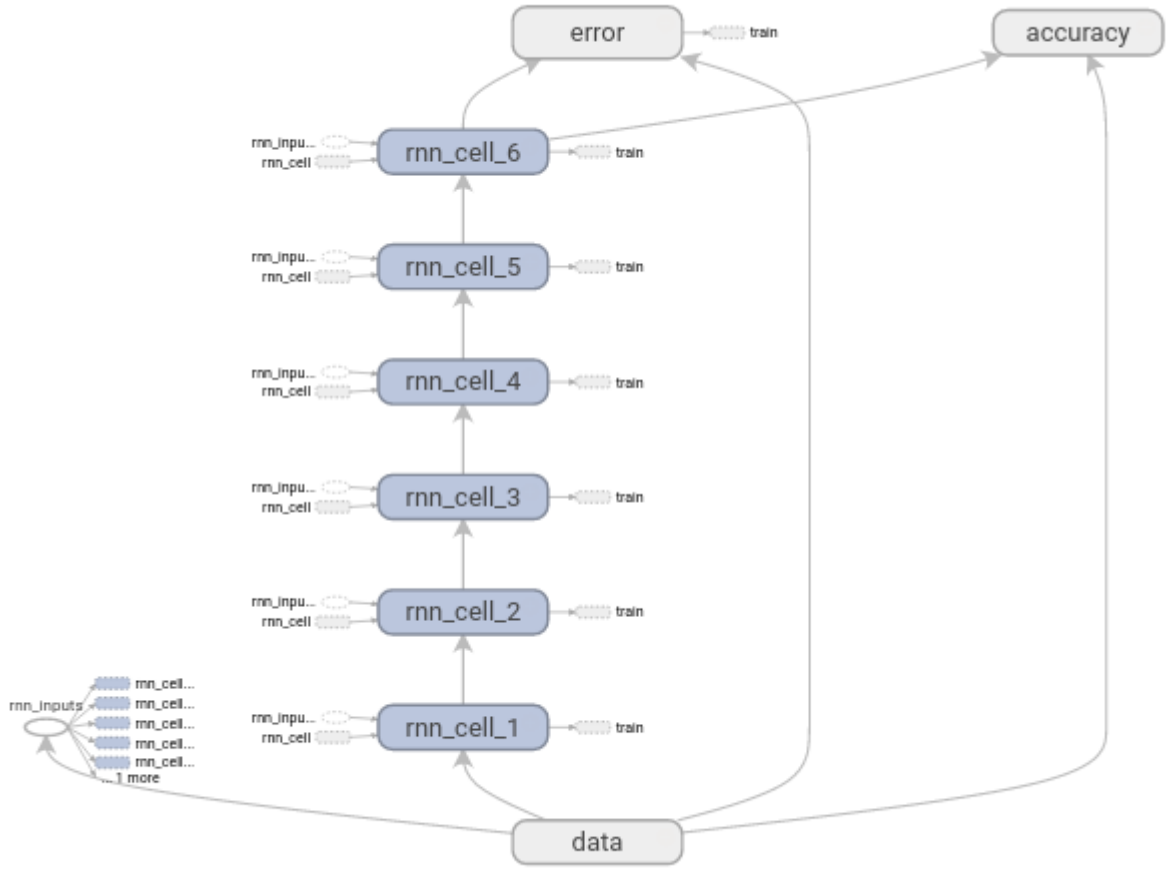
Note that this has the implication that we are going to ask from our neural network to classify each part of the song. However we expect that since the final state of each part is feeded as the initial state of the next part that the last part will contain all the necessary information and we are going to keep only this classification neglecting the rest. There are other architectured that could be followed but this is our current approach.

The above logic requires a custom `trainingEpoch` and `validateEpoch` methods. Both of them now take into account the `cur_state` which is a variable, originally initialized with zeros and then being written with the final state of the recurrent neural network. This same variable then gets feeded to the placeholder to be the initial state of the next set of segments.
Note that from every song we are taking into account for the training and validation error and accuracy only the output of the final part of each song.

For building the Tensorflow graph we have an RNN cell created by the corresponding `rnn_cell` method which we call equal number of times with the number of RNN steps that we want.
As in the diagram above each RNN cell takes as input the concatenation of the ouput of the previous state and the inputs from the data.
The non-linearity used at the output is the activation function `tanh`.
Note that the characteristic of the Basic RNN architecture is the sharing of variables among RNN cells. This is why the tensorflow `get_variable` method is being used instead of creating a `Variable` for each cell.

Note that our current implementation does not include batch normalization, nor dropout, nor L2 regularization.

For an RNN with parametric step size, here six(6), we have the following graph:

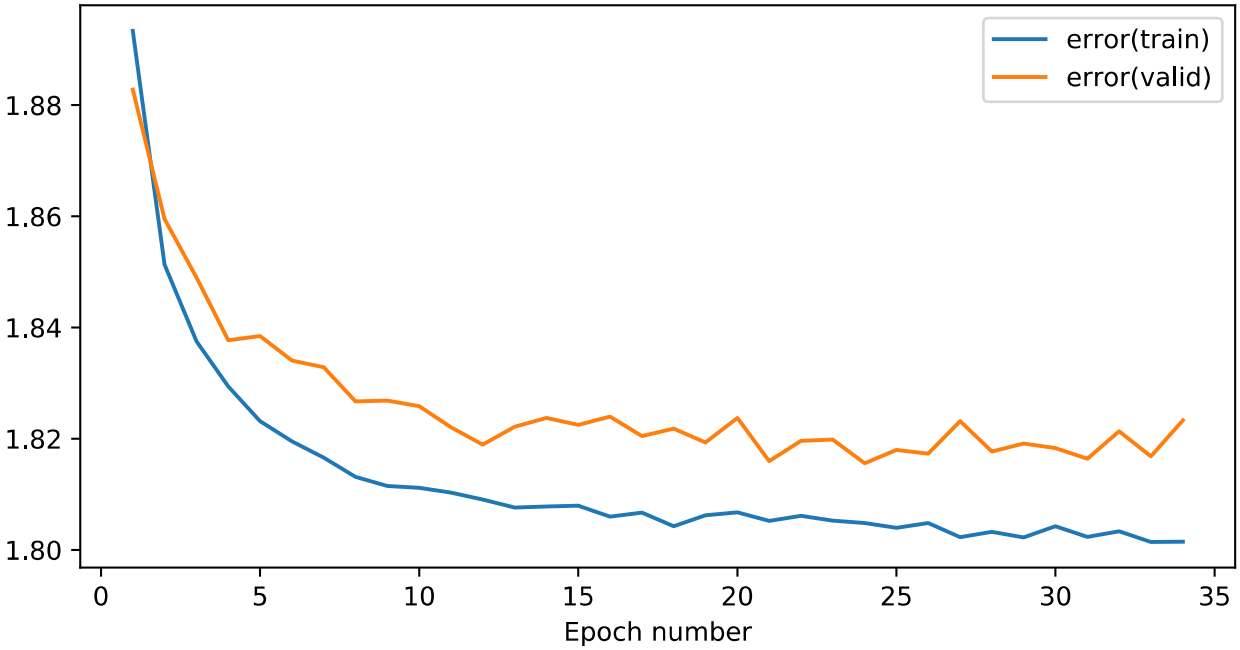


Graph 3: Basic Recurrent Neural Network unfolded to each RNN cell with shared weights – RNN steps: 6

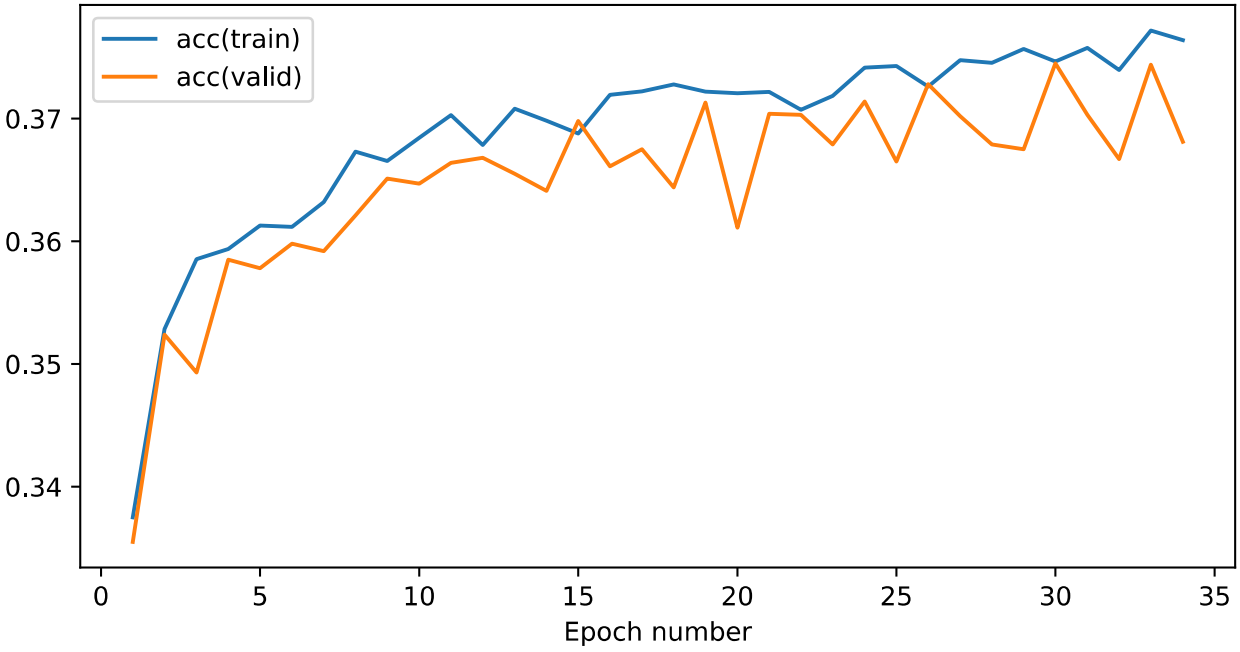
Note that this is a simplistic model where the state size is exactly equal with the number of classes included in our classification task, here ten(10), that is why we are able to apply a softmax directly to the outputs of the final state. This is of course very restrictive but this is only a first basic architecture of our RNN.

We are setting the number of RNN steps to number six(6) as shown in the graph above and we are training for 35 epochs.

Results



Plot 23: Training & Validation Error – Basic Recurrent Neural Network – RNN steps: 6 – State size: 10 – no readout layer



Plot 24: Training & Validation Accuracy – Basic Recurrent Neural Network – RNN steps: 6 – State size: 10 – no readout layer

Conclusions

Within 35 epochs we do not see any signs that of overfitting but we are also have not achieved a very high validation accuracy. Validation accuracy seems to have converged around 37%.
This is expected since we have a small state size of only ten.
We are going to be using this as a baseline for the next RNN experiments.

Grid Search for Best RNN number of steps

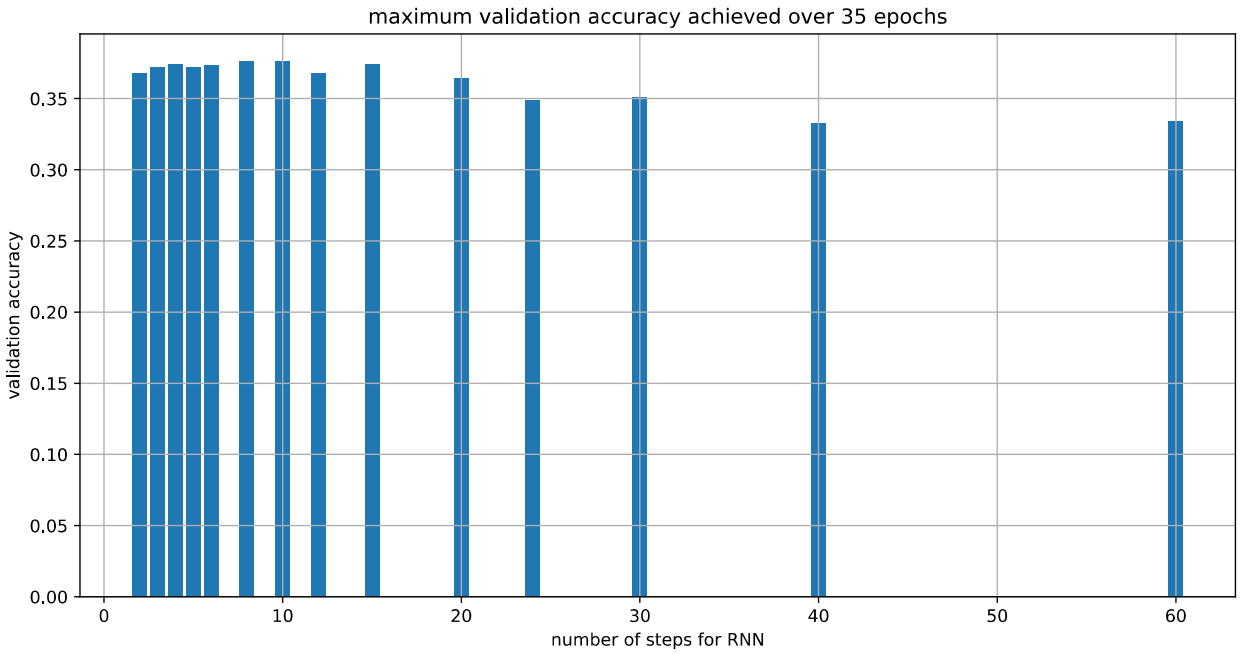
Since the 120 segments can be divided exactly by only a fixed set of number of steps we may as well search all of the following cases in order to get a hint for the optimal number of steps for our current dataset.

This is a list with the number of steps that divide evenly 120 segments:

```
2,  3,  4,  5,  6,  8, 10, 12, 15, 20, 24, 30, 40, 60
```

Results

This is the bar chart after running training for 35 epochs in all of the cases in the list above

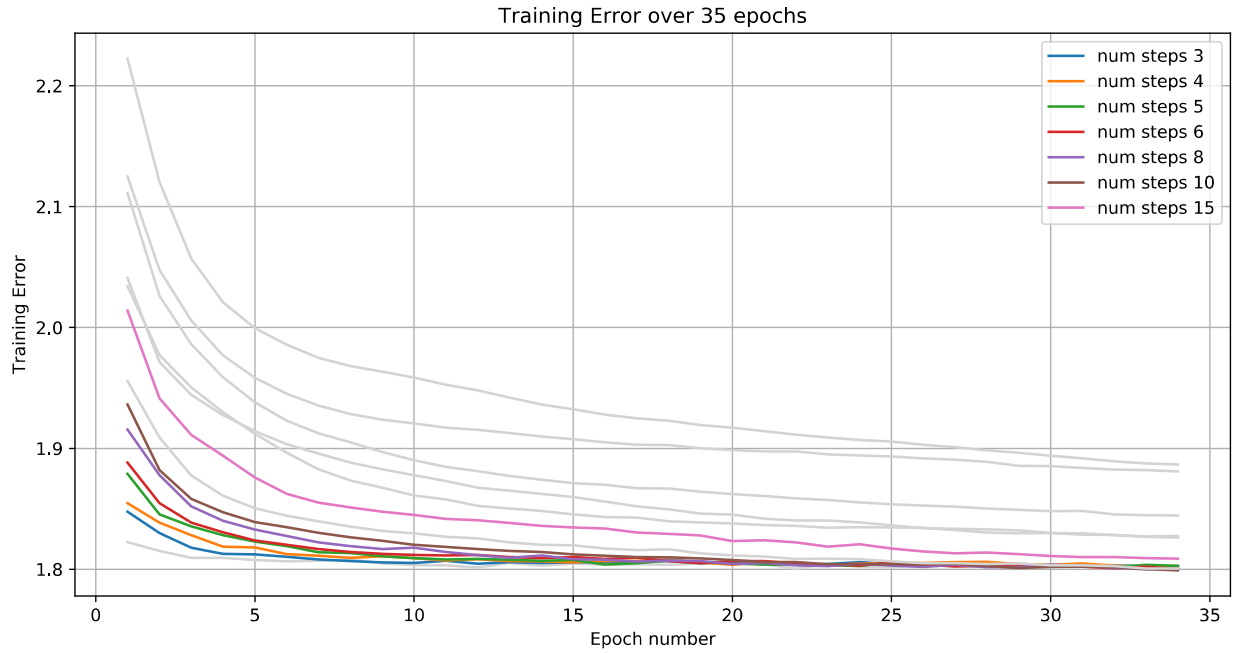


Plot 25: Maximum Validation Accuracy achieved within 35 epochs for various RNN steps for Basic Recurrent Neural Network without readout layer with State size: 10

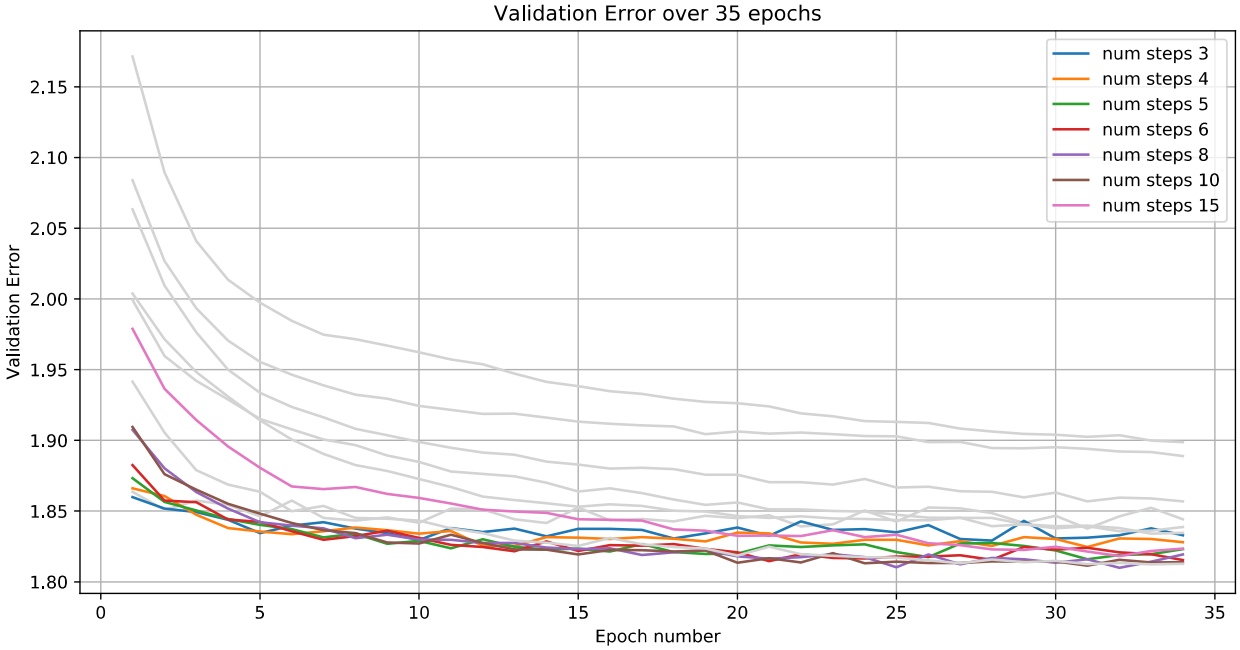
The seven number of steps as hyperparameter which correspond to the highest validation accuracy are ordered from higher to lower:

```
8,  10,  15,  4,  6,  3, 5
```

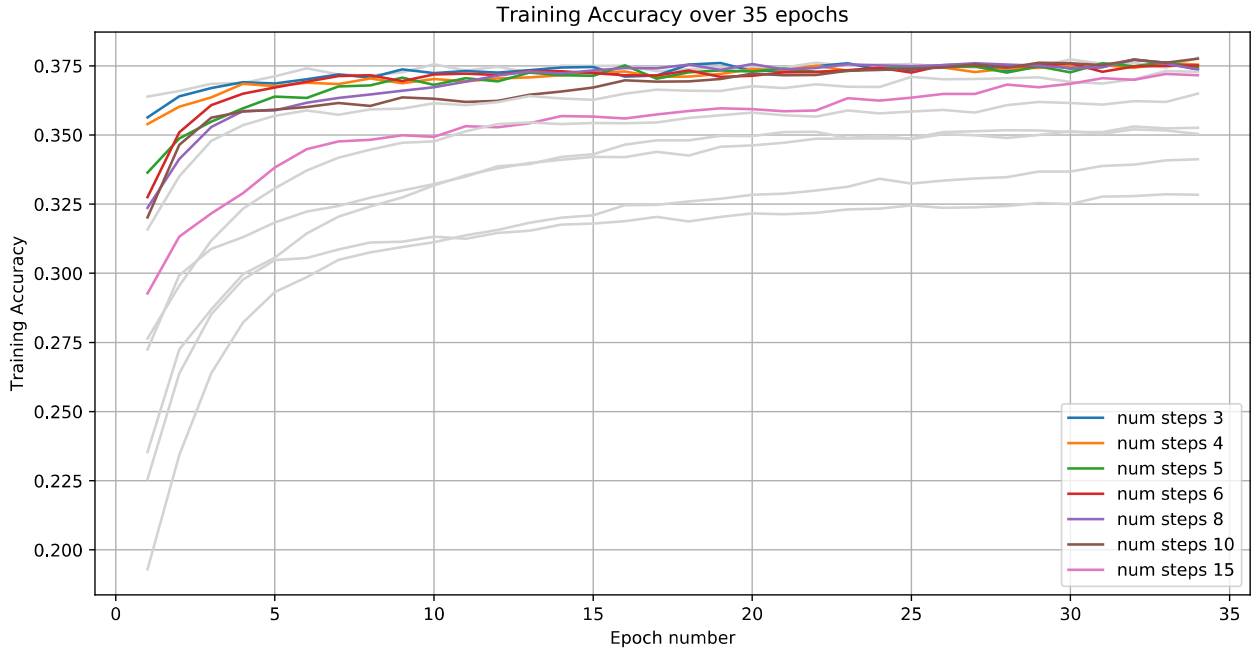
Below we are plotting the Training and Validation Error as well as the Training and Validation Accuracy of the seven best number of steps to examine and compare their behavior.



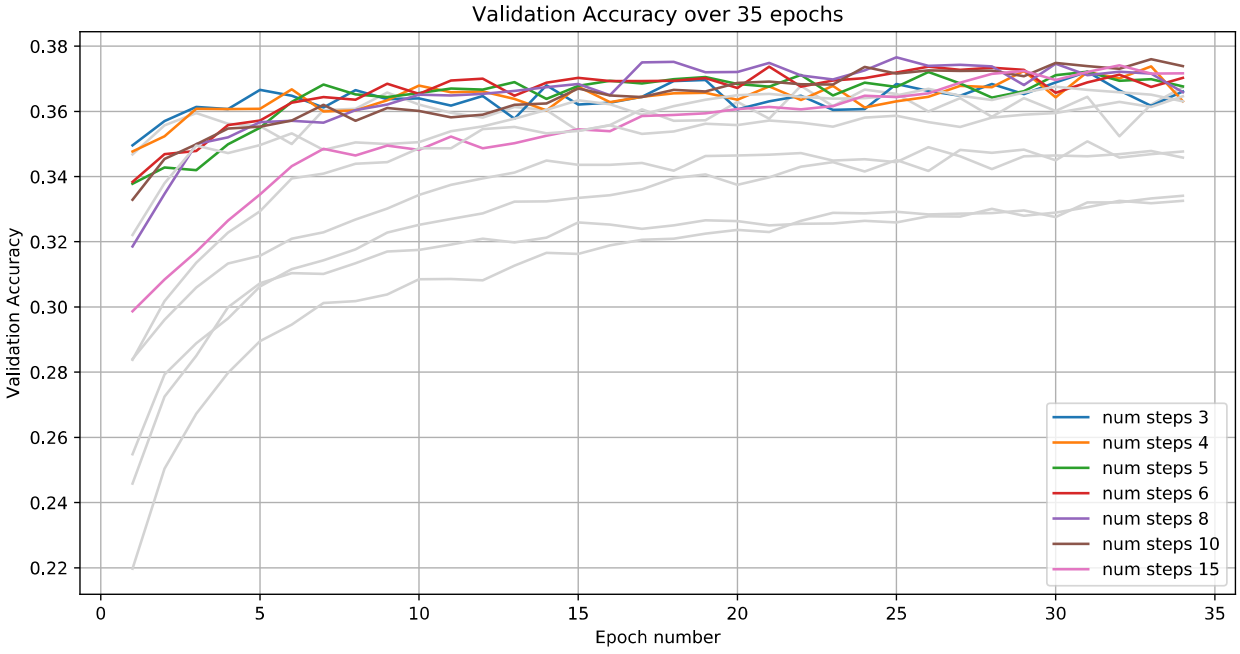
Plot 26: Training Error for various RNN steps for Basic Recurrent Neural Network without readout layer and with State size: 10



Plot 27: Validation Error for various RNN steps for Basic Recurrent Neural Network without readout layer and with State size: 10



Plot 28: Training Accuracy for various RNN steps for Basic Recurrent Neural Network without readout layer and with State size: 10



Plot 29: Validation Accuracy for various RNN steps for Basic Recurrent Neural Network without readout layer and with State size: 10

Conclusions

We see that we do not have a huge variation among the implementations. The barchart suggests that the bigger the number of steps the worse is the overall validation accuracy. This is expected because RNNs with large number of steps are suffering from exploding / vanishing gradient issues.

The number of steps from 4 up to 15 do not seem to have a real difference and the maximum validation accuracy reported above for the range of these steps should not be taken very seriously into account because the plots reveal

that there is a lot of variance of the validation accuracy around 37%. This means that given more epochs we could easily see a slightly different ranking of the best rnn step.

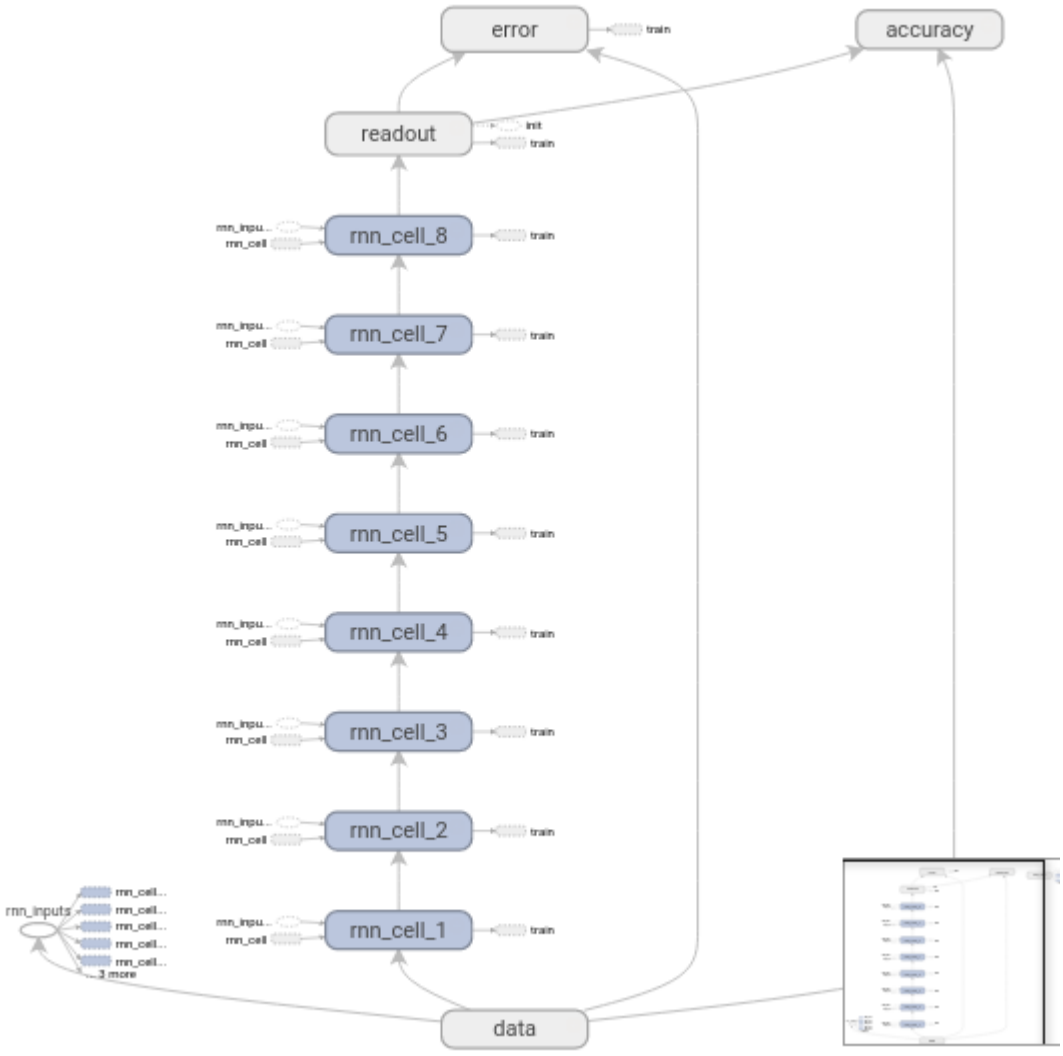
Exploration of best combination of state size and RNN number of steps with Bayesian Optimization

Here we are trying to explore with combination of state size and RNN number of steps yields the best classifier.

Before delving into the details of our implementation we must note that for an arbitrary state size we need to create one extra MLP layer that will combine the final state of the RNN to dimensionality of ten, as is the number of our classes for the MSD-10 classification task and be able to apply softmax and do cross entropy. We solve this by adding an extra readout.

Note that we have organized this model by putting the implementation into the `ManualRNN` class, which is found in the `rnn.manual_rnn` module, because we can handle more easily the large number of variables involved in setting up of the experiment with tensorflow. These variables now are not used in a functional way where they are returned by a function as we applied before but they are contained in the class from where they are easily accessible to all the methods.

So by adding the readout layer the Tensorflow graph becomes as follows:



Graph 4 Basic Recurrent Neural Network with Readout Layer – RNN cells unfolded (shared weights) – RNN steps: 8

For Bayesian Optimization we are using the following parameters:

- **Space of State Size**: 15 to 500 (integer)
- **Space of Number of RNN steps**: 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60 (categorical)

Minimum state size is set to 15 because we have already specified from the previous experiment that size of 10 is insufficient to model appropriately.
Maximum state size is set to 500 due to computational restrictions.

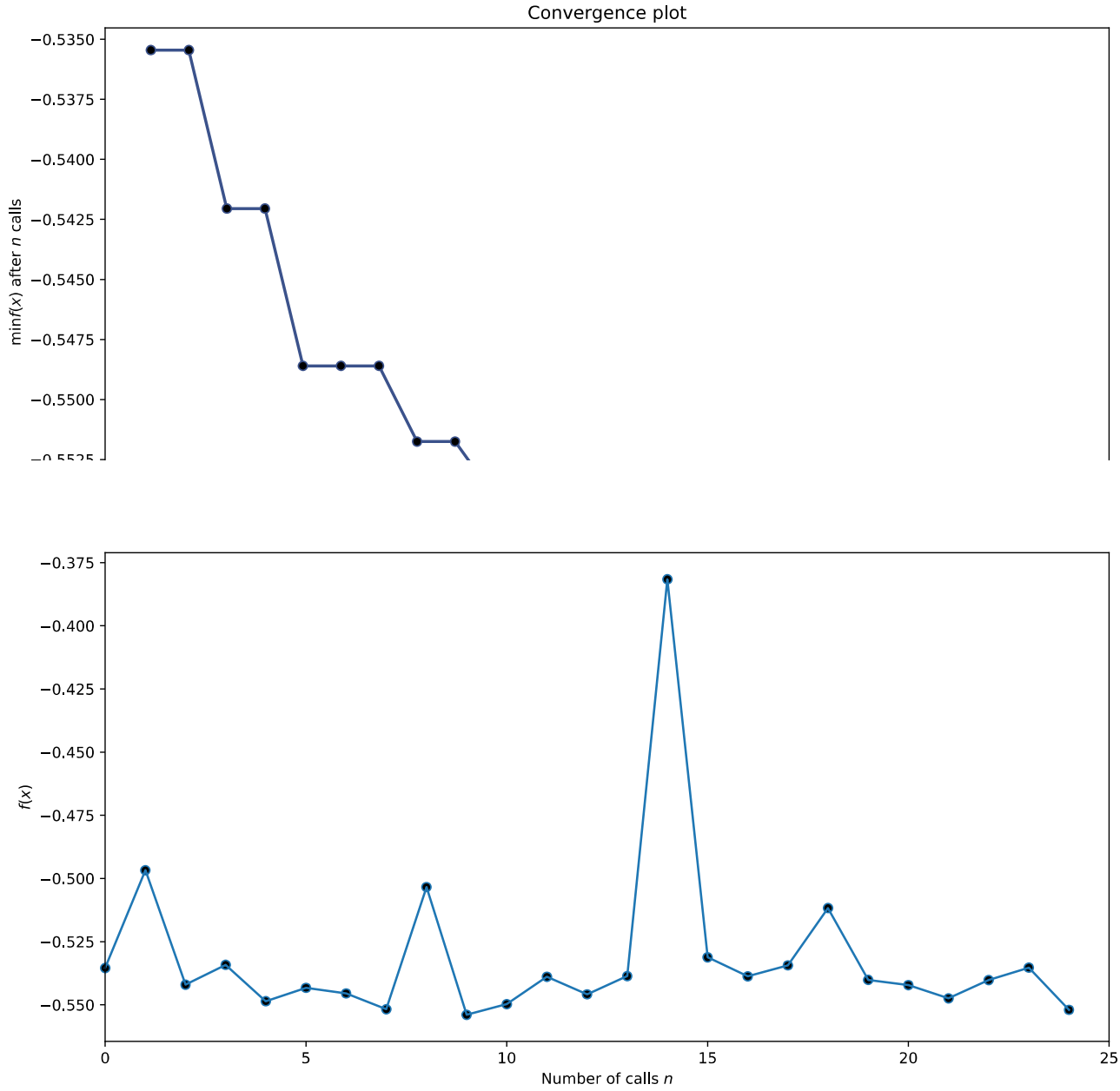
Hyperparameter Kappa, the trade-off between exploration and exploitation is set to 1.9.

We are executing only one random start and 25 calls of the bayesian optimization which gives a total of 26 executions. We cannot afford more executions due to restrictions in computational power.

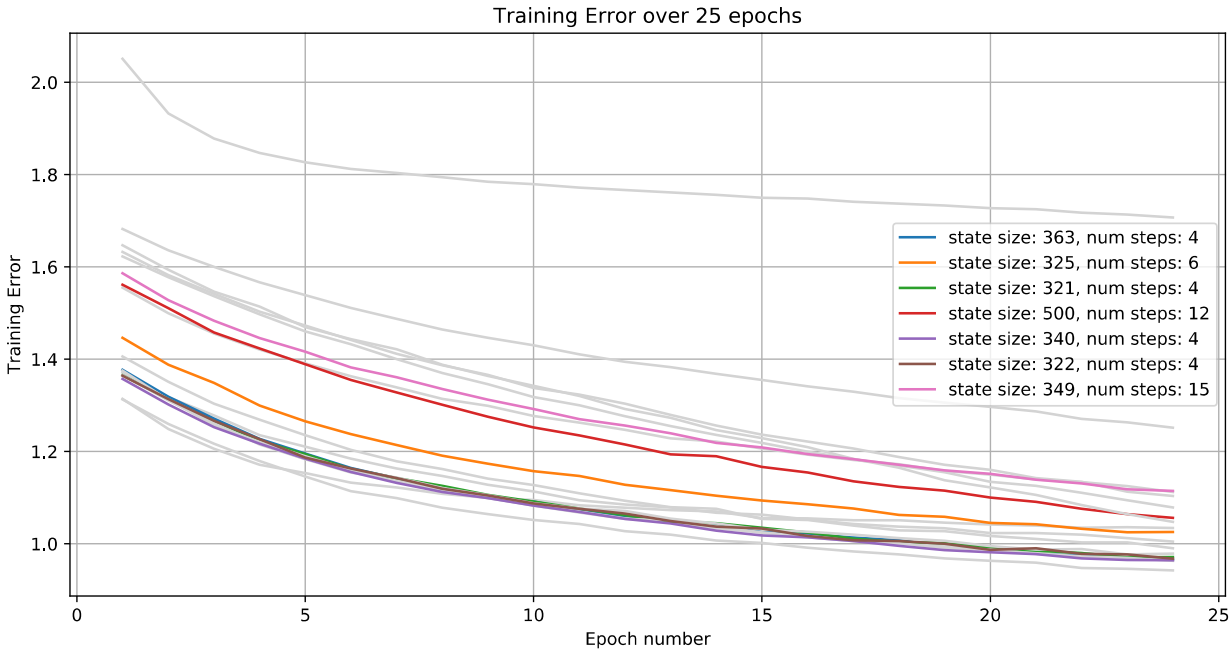
For similar reason each experiment is executed for 25 epochs.

Our objective function is returning the mean of the 10% of the best validation accuracies, here the mean of the top 2

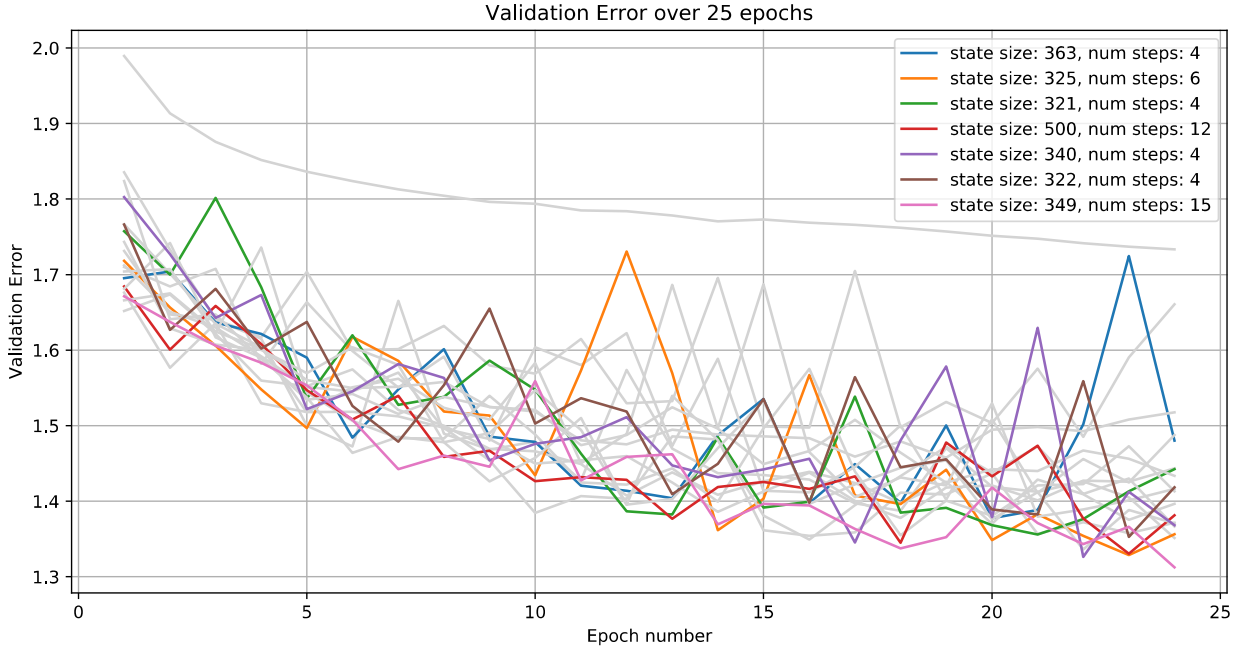
Results



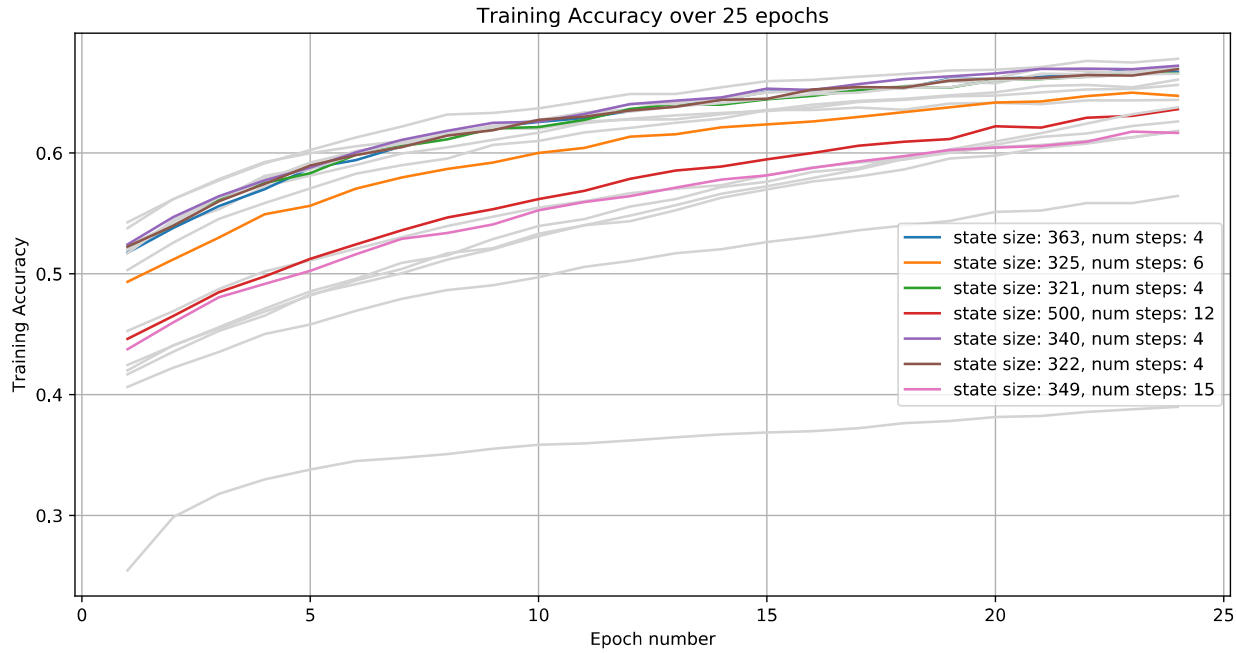
Plot 31: Objective Function output of Bayesian Optimization trying to find the best RNN steps and best State Size for Basic Recurrent Neural Network – Objective Function returns the average of the top two validation accuracies for training of 25 epochs



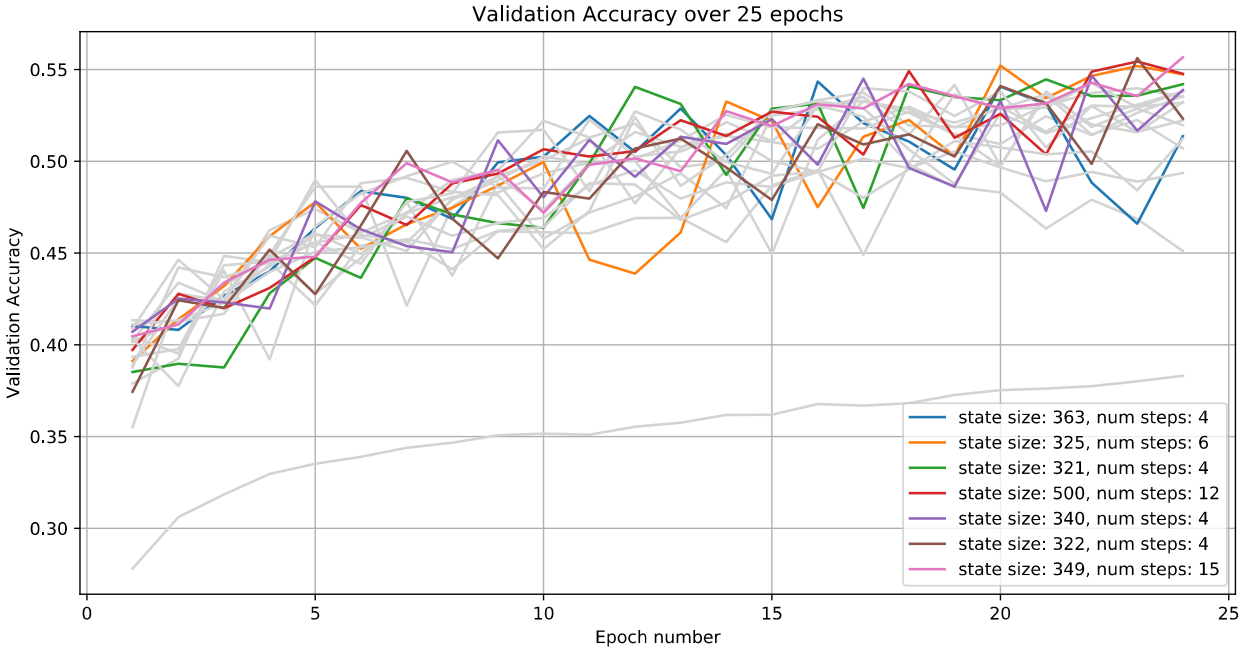
Plot 32: Training Error – Bayesian Optimization trying to find the best RNN steps and best State Size for Basic Recurrent Neural Network – Objective Function returns the average of the top two validation accuracies for training of 25 epochs



Plot 33: Validation Error – Bayesian Optimization trying to find the best RNN steps and best State Size for Basic Recurrent Neural Network – Objective Function returns the average of the top two validation accuracies for training of 25 epochs



Plot 34: Training Accuracy – Bayesian Optimization trying to find the best RNN steps and best State Size for Basic Recurrent Neural Network – Objective Function returns the average of the top two validation accuracies for training of 25 epochs



Plot 35: Validation Accuracy - Bayesian Optimization trying to find the best RNN steps and best State Size for Basic Recurrent Neural Network – Objective Function returns the average of the top two validation accuracies for training of 25 epochs

Best Parameters:

- **State Size**: 341
- **Number of RNN steps**: 4

Conclusions

We notice that bayesian optimization was able to find the best case quite fast within 10 epochs and then could not find a better case within the rest of 15 epochs, even though all of the values contributed so that Bayesian Optimization will return to us the best parameters for the current architecture.

Note that in the Validation Error/Accuracy plots we have state sizes of around ~350 with 4 RNN steps to be the best of various of the experiments ran by the bayesian optimization. No wonder why the best parameters returned at the end are near these values.

However we should state that this is local optimum and by having more time or computational resources at our disposal the bayesian optimization could perhaps find a better local optimum.

The positive side is that through this process we have experienced validation accuracies of ~56% which is better than our best validation accuracy so far.

RNN Experiment with Best Parameters

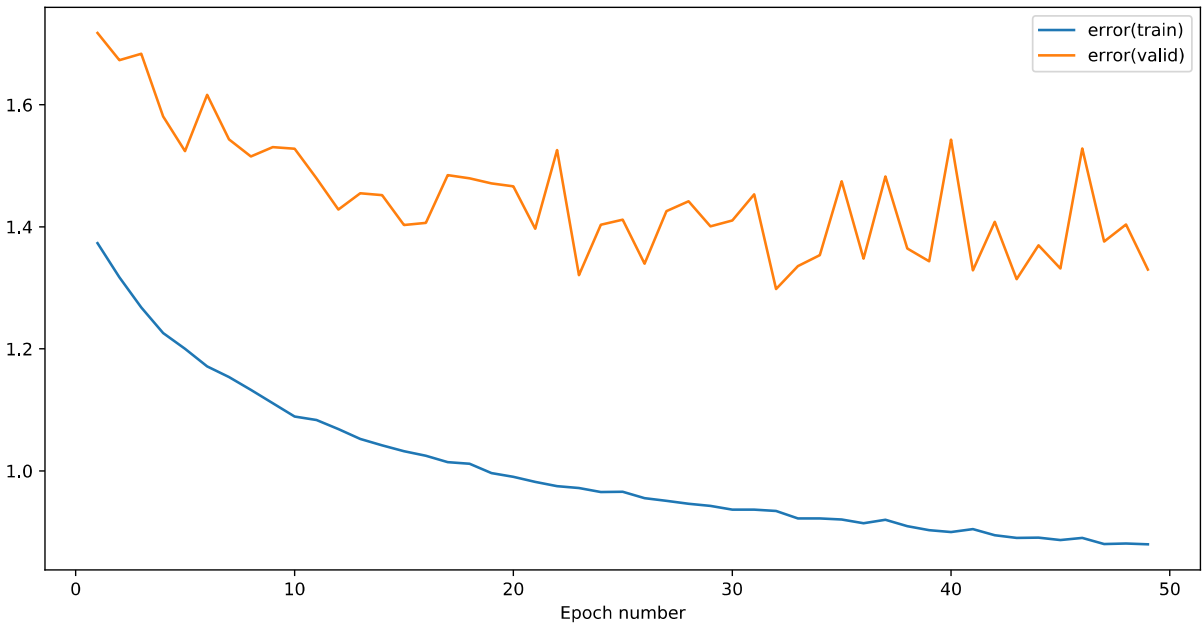
Next step is to take advantage of the best parameters of the Bayesian Optimization and run an experiment for 50 epochs to have the chance and examining the behaviour of our RNN classifier with the best parameters.

Also it is a good chance to check if the parameters suggested by the Bayesian Optimization algorithm yield indeed an optimal state, depended of course from the optimality as described by our objective function.

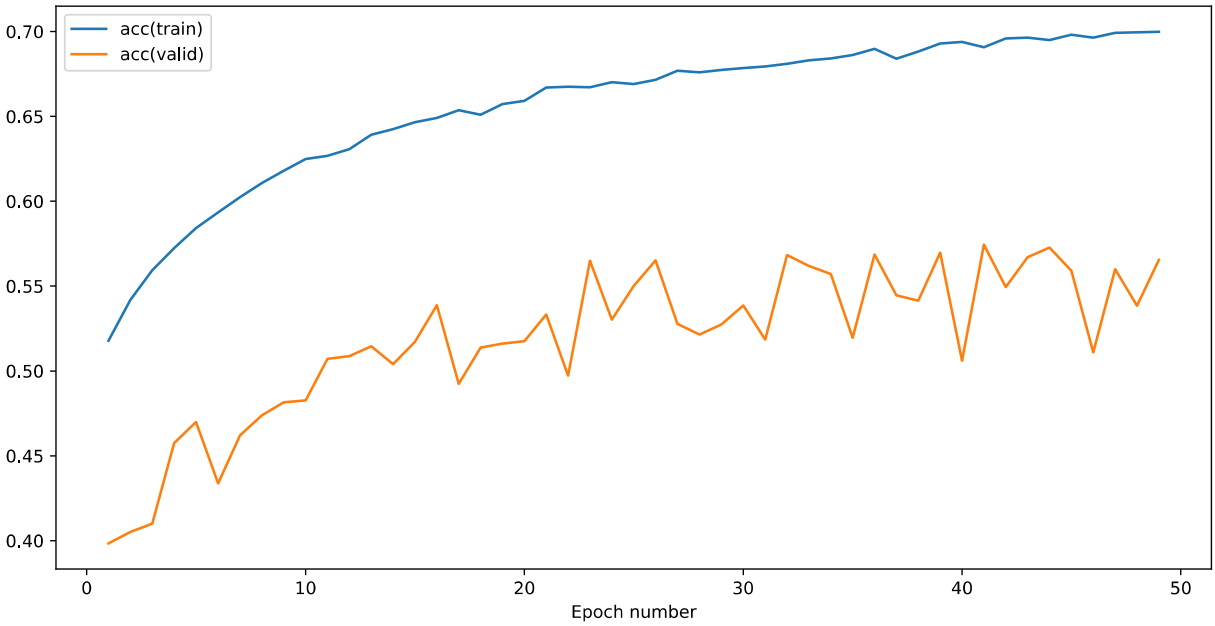
So we execute the experiment according to the following parameters:

- **State Size**: 341
- **Number of RNN steps**: 4
- **Epochs**: 50

Results



Plot 36: Training & Validation Error – Basic Recurrent Neural Network – State Size: 341 – Number of RNN steps: 4 – Training Epochs: 50



Plot 37: Training & Validation Accuracy – Basic Recurrent Neural Network – State Size: 341 – Number of RNN steps: 4 – Training Epochs: 50

Conclusions

We can assume that the Bayesian Optimization has provided sufficiently good hyperparameters since we have achieved a new record with the validation accuracy peaking at 57.44%

In addition we have verified that the recurrent neural network could take advantage of the temporal dependencies among the segments of the songs in order to build a superior classifier.

However we must note that there are lots of oscillations both in the validation accuracy and the validation error which means that the neural network has difficulty to converge to a place where it expresses a good generalized classifier for the MSD-10 task.

Research Question: Could we train a shallow neural network to be as good as our basic Recurrent Neural Network in terms of classification accuracy?

We have achieved a higher validation accuracy with our basic RNN but the model has multiple layers and this is a much larger model, in terms of parameters, than the shallow neural networks we tested when first encountered the MSD-10 classification task.
This bring a speed performance issue when we would like to classify a million songs that would have to pass through this network. The question is whether we could train a shallow neural network to replicate the behavior of the more complex deep recurrent neural network, by following the so called Teacher-Student architecture.

The Teacher-Student architecture is based on the presentation slides found on the following link.
Mimicking deep neural networks with shallow and narrow networks: http://web.eng.tau.ac.il/deep_learn/wp-content/uploads/2016/12/Mimicking-deep-neural-networks-with-shallow-and-narrow-networks.pptx

The first step in order to build the Teacher-Student architecture is to pass unlabelled data through the deep recurrent neural network and collect all the outputs. Note that we care for the logits and not for the processed softmax outputs in order to have values which are within a constrained range rather than exponential values produced by the softmax function. This will help us more easier to pick a suitable cost function for our student model later.

The issue is that until now in our MSD-10 dataset we have considered the train dataset as having the labelled data and the validation dataset as having the unlabelled data. However we need the outputs of as much unlabelled data as we can in order to have enough data to train the Student model later.

Cross Validation

Our solution will be **Cross-Validation** but we need to make a number of changes to fully implement it.

First we are building the `CrossValidator` class found in the `rnn.cross_validator` module. To get the instances of each fold of the k-fold cross validation, of training and validation instances, we are exploiting the `StratifiedKFold` class from the `model_selection` module of scikit-learn library.
Note that we need to use Stratified KFold and not the simple KFold because we have a balanced classification task and we need to maintain this state in order to avoid having to use techniques that need with imbalanced classification tasks.

First we have created a small python script named concat_train_valid.py found inside the data folder which is responsible for merging the training and thevalidation datasets under the same file, in our case *msd-10-genre-train_valid.npz* file. Concatenation merges 40000 with 10000 songs to a total of 50000 songs.

Note that StratifiedKFold process is set to shuffle the instances. So if we set a k-fold of k=10 we are going to have 45000 shuffled instances as training set and the rest 5000 instances as validation set, also shuffled.

This brings two new requirements for our data providers:

- Our data provider needs to be able to filter the instances according to a list of indices which correspond to the ids of the instances that we need to consume.
- Our data provider needs to keep the references of the original ids of the instances even if it is working with a filtered dataset.

So if we need to have 45000 instances only, for example for the training dataset, we need to keep only these 45000 from the full 50000 instances and discard the rest.
We have enhanced `MSD10Genre_120_rnn_DataProvider` class with an extra parameter at the constructor named `indices`. The indices are the indices of the instances that we wish to keep. Whenever indices is set to None

we fallback to the original implementation where all indices are being kept.
Next step is to enhance the `MSD10GenreDataProvider` class. Its constructor has a `data_filtering` callback parameter which should be a function that takes inputs and targets as inputs, process them and returns them as outputs.
The implementation of `data_filtering` inside the `MSD10Genre_120_rnn_DataProvider` class is an one liner implemented with lambdas. We are keeping the inputs and targets that correspond to the indices parameter is indices is not None and it is an array.

For the second requirement we have included an extra parameter at the constructor our base `DataProvider` class named `initial_order`.
Until now we were considering the ids of the inputs and targets to be a serial number from zero to the length of the dataset minus one but now we need to accommodate custom ids which are initiated with a specific order. This requires two changes.
Firstly the `_current_order` attribute of the class is initialized with `initial_order` unless the `initial_order` is None which is then initialized with numpy's `arange` function as it was the original implementation.
The second and most difficult change is to find the inverted permutation of the current order in case we want to use the reset method. Here we take advantage of numpy's `argwhere` function in order to find in which indices the values of the `initial_order` are now located on the permutated `_current_order`.

Logits Gathering

Our purpose here however it is not to do cross-validation for the classification task but rather to gather logits from unlabelled data the best model in each fold of the k-fold cross-validation process.

For this purpose we have created the `LogitsGatherer` class found in `rnn.logits_gatherer` module. This implements the `getLogits` method which iterates over all of the instances of the data provider passed as parameter and is building a python dictionary.
The keys of the dictionary are the ids of the instances as provided from _current_order for the way the instances are currently permuted.
The values of the dictionary are the vector of the, ten(10) here, logits from the neural network.

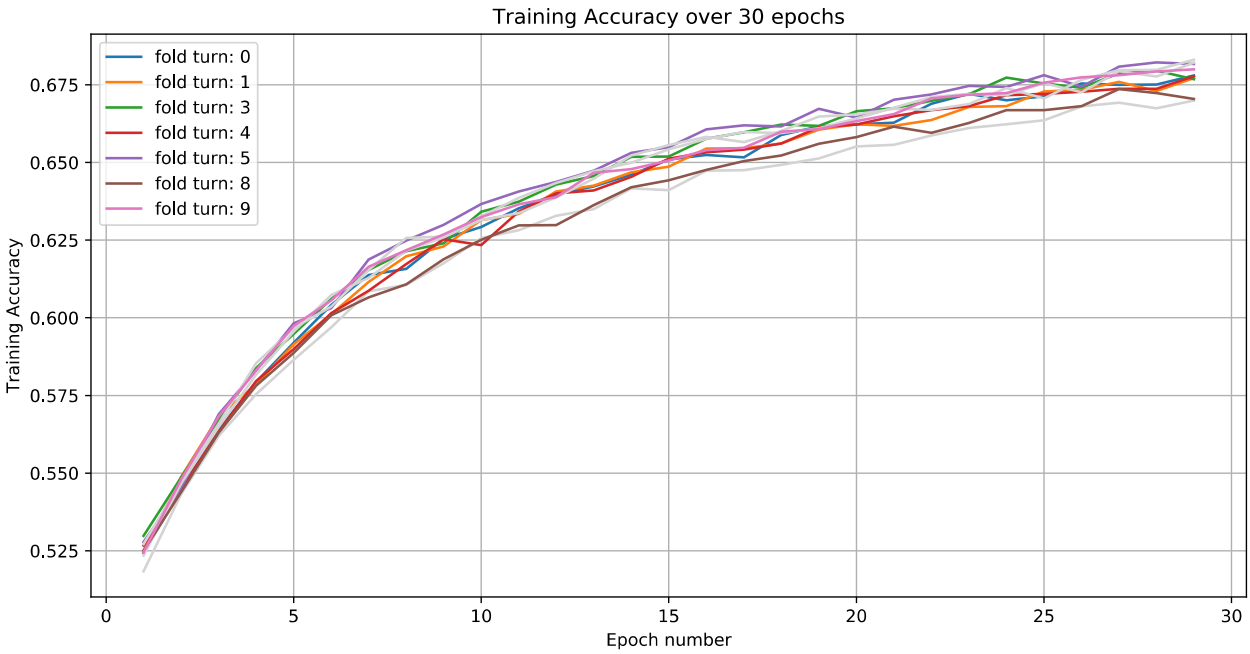
When training/validating our Recurrent Neural Network we are calling getLogits method on the first epoch but not on all consecutive epochs. We are calling getLogits again upon the condition that the newly calculated validation accuracy is larger than any of the previous ones.

Cross Validation and Logits Gathering execution

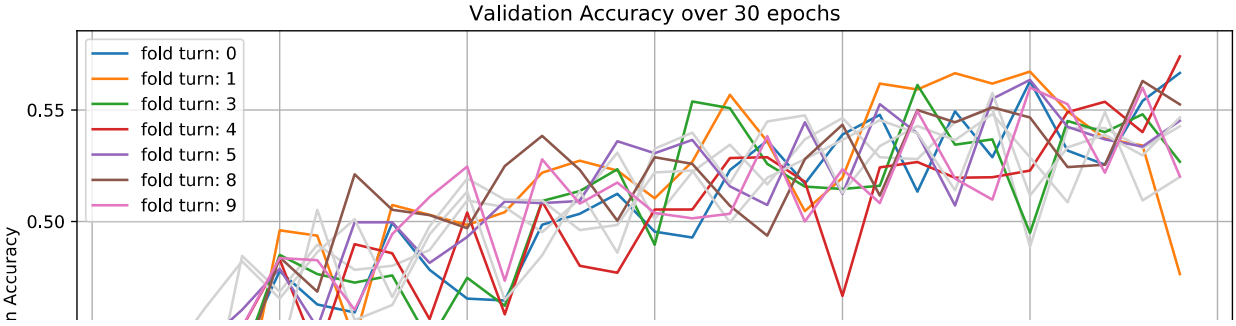
We used **k=10 fold Stratified** to have a good enough balance which is commonly used as a rule of thumb of ~90-10% percent of training and validation sets respectively.

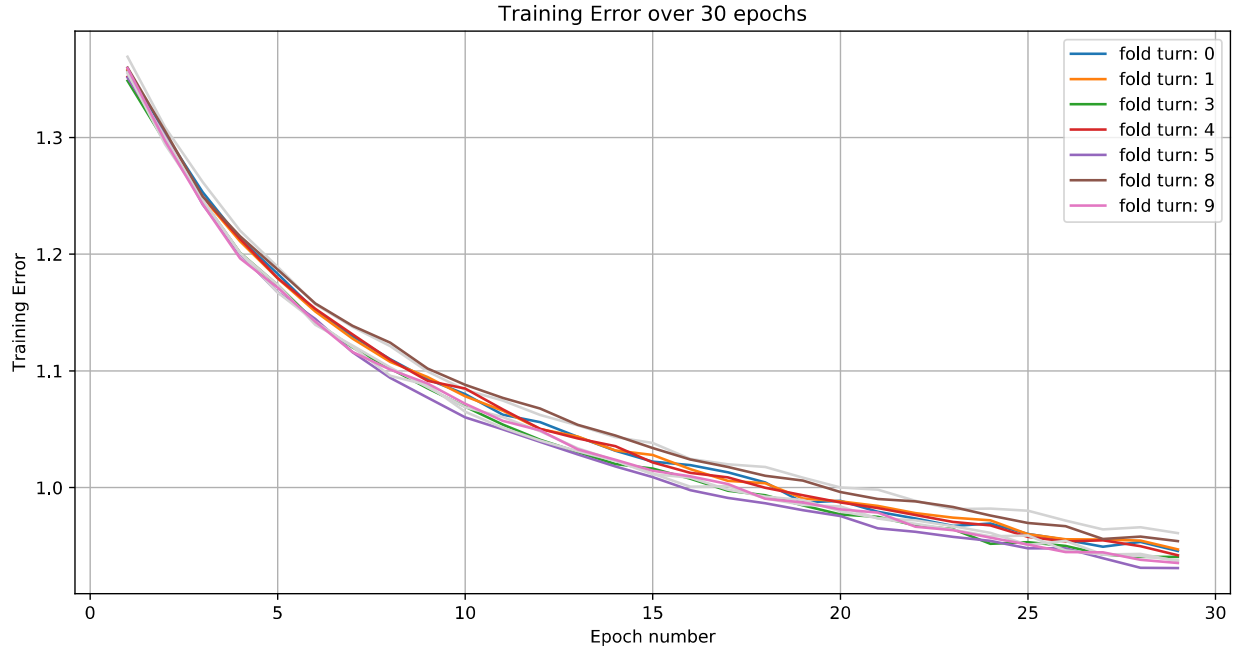
Each training was executed for **30 epochs**.

Results

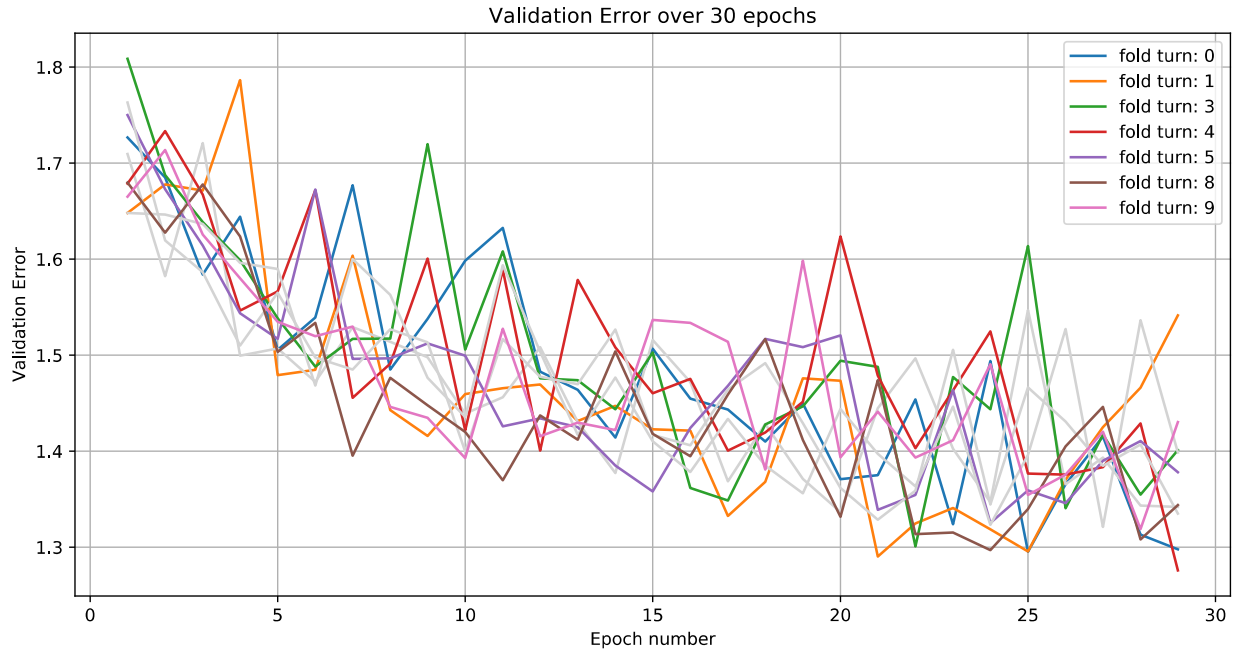


Plot 38: Training Accuracy – Stratified 10-Fold Cross-Validation for 30 Epochs – Basic Recurrent Neural Network – State Size: 341 – RNN steps: 4



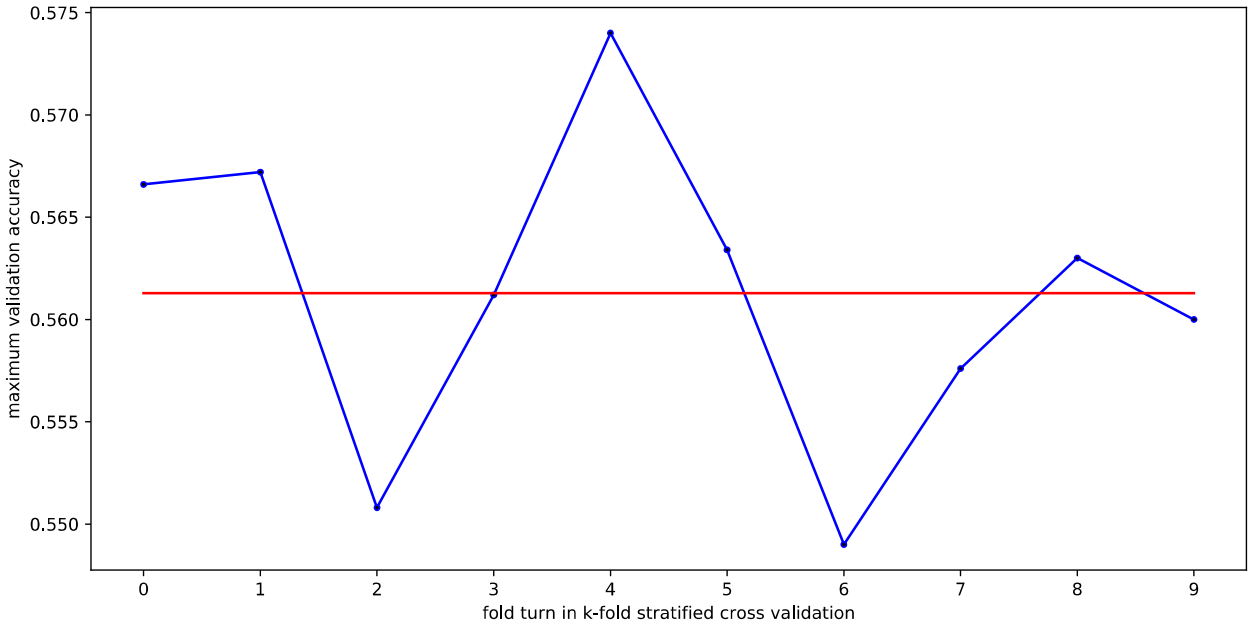


Plot 40: Training Error – Stratified 10-Fold Cross-Validation for 30 Epochs – Basic Recurrent Neural Network – State Size: 341 – RNN steps: 4



Plot 41: Validation Error – Stratified 10-Fold Cross-Validation for 30 Epochs – Basic Recurrent Neural Network – State Size: 341 – RNN steps: 4

//…



Plot 42: Maximum Validation Accuracy per fold Turn for Stratified 10-Fold Cross-Validation – 30 Epochs – Basic Recurrent Neural Network – State Size: 341 – RNN steps: 4 – Red Line represents the average value of all the max validation accuracies

Conclusions

From the plots above we see that running our Recurrent Neural Network without any regularization or batch normalization is the cause of a high variance for the weights and as a result we have the validation accuracy and error fluctuating instead of easily converging. This is valid for all fold turns of k-fold stratified cross-validation.

The fluctuation is more visible from the last plot where the maximum validation accuracy among fold turns ranges from **54.9% to 57.4%** which is a substantial difference. The mean value is noted with the red line and it is **56.13%**. Note that each one of the models that correspond to these maximum recorded accuracies are being used to generate the logits of the unlabeled data to be used as outputs of the Teacher Model and later to be used to the Student Model.

Building the Student Model

Student model is a shallow artificial neural network which contains only **one hidden layer** of arbitrary, usually large, dimensionality.
The hidden layer is followed by a **non-linearity** of our choosing, which here is the *tanh* activation function.
Finally we have a **readout affine layer** which reduces the dimensionality to the outputs corresponding to the dimensionality of the logits from the Teacher model.

The training dataset is going to use as inputs all of the 50000 we used above when we were doing cross-validation. The training targets are going to be the outputs/logits from the Teacher model.

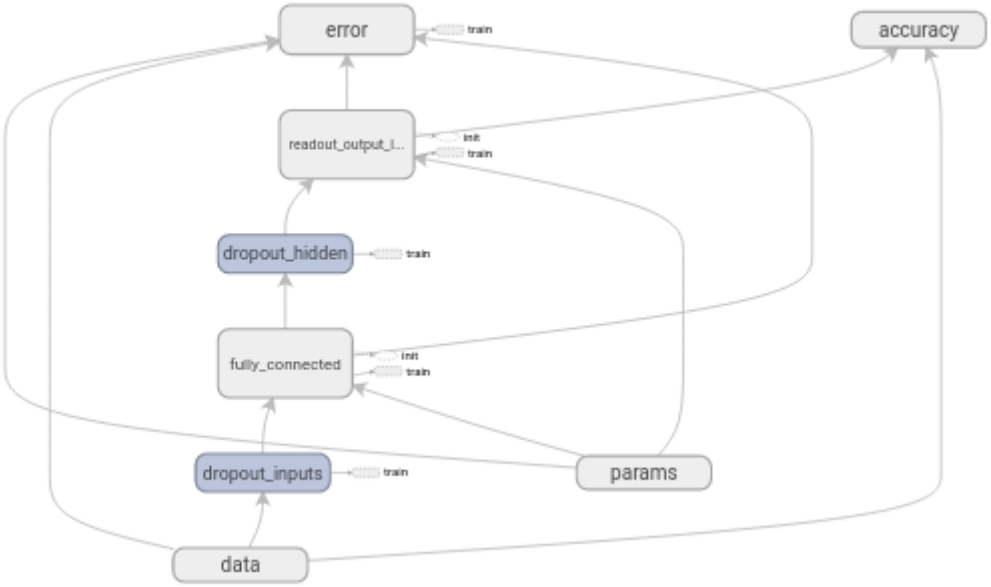
As error function we are going to use the Mean Squared Error between the outputs of the Student Models and the already processed and saved outputs of the Teacher Model.

$Loss = \frac{1}{T}(\sum_t ||\beta f(Wx) - z||^2)$ where β is the readout affine layer matrix, f is the non-linear activation function, W is the matrix of the hidden layer, x are the inputs, z are the logits outputted from the Teacher model and T is the length of the dataset.

In order to be able and validate the Student model we are going to use the testing dataset. Note that this is not ideal since we would like to leave the testing dataset out of the validation process as a way to measure the “real” performance of our model on unseen data. However due to computational restrictions we are not going to use cross-validation, as it would have been the alternative, and “sacrifice” the testing dataset to be used as our validation dataset here. This can be considered ok for the current system since there is no evidence that we have overfitted our validation dataset and thus that we have the need to use a testing dataset for more objective evaluation.

We need a new data provider for the Student model. We have implemented `MSD10Genre_Teacher_DataProvider` class in `models.teacher_student_nn` module. The difference from its base, which is `DataProvider`, is that there are two parameters at the constructor, `dataset_filename` and `logits_filename` which control the sources of the inputs and the targets of the data provider respectively.

The Student model is implemented by the `StudentNN` class found in `models.teacher_student_nn` module. The graph implemented by the StudentNN is rendered below.



Graph 5: Student model – MLP Neural Network – Dropout Layer for Inputs – Dropout Layer for Hidden Layer – Batch Normalization – L2 Regularization – One Hidden Layer plus Readout Layer

Note that there are two kinds of errors being calculated depending on the training placeholder.
If training is True then we are at training stage and we have as error the Mean Square Error of the outputs against the logits of the Teacher model.
Else if training is False then we are at the validation stage and we pass the outputs through softmax function and then take the cross entropy as we have done multiple times in previous experiments.

Also note that Dropout, L2 Regularization and Batch Normalization are included in all layers of the shallow neural network of the Student model.

Bayesian Optimization of the hyperparameters of the Student model

We would like to run multiple experiments to determine the optimal values of dropout keep probabilities, L2 regularization factor and the magnitude of the hidden dimensionality with the help of Bayesian optimization.

The space of these hyperparameters is as follows:

- Keep Dropout Probability of Input layer: Minimum 50%, Maximum 100%
- Keep Dropout Probability of Hidden Layer: Minimum 50%, Maximum 100%
- Number of Neurons for the Hidden Layer: Integer with minimum 20 and maximum 2000
- L2 regularization factor: minimum 1e-3 and maximum 10 with log-uniform distribution

We ran the bayesian optimization twice. We will provide both results and then at the end we are going to provide conclusions taking into account all experiments

Bayesian Optimization with Learning Rate 1e-5

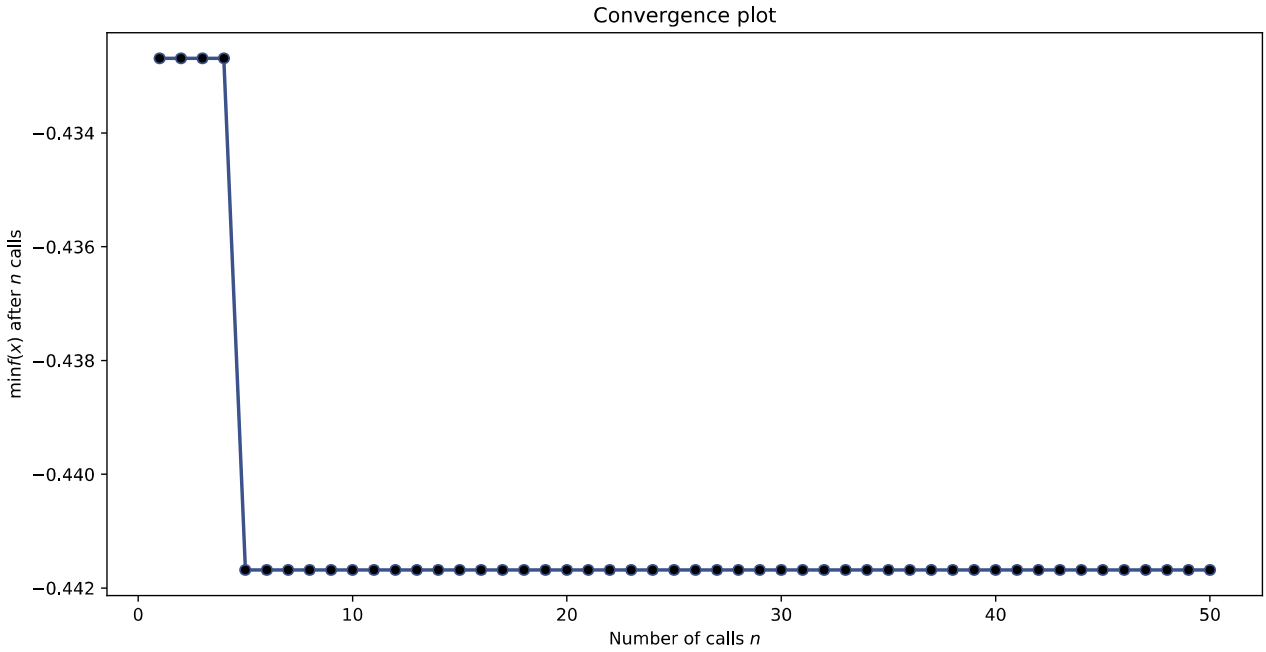
Here learning rate for optimizer is set to a low value of **1e-5**

Training runs for **40 epochs** on every call of the objective function.

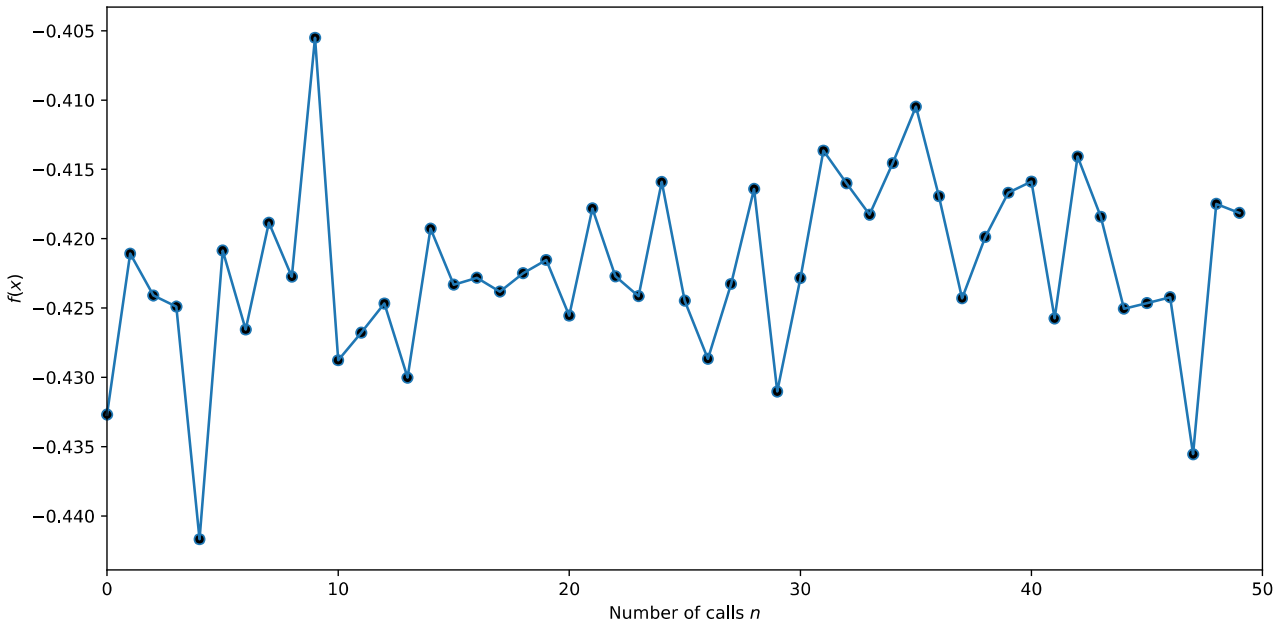
Number of calls is equal to fifty(50) and at the initialization before we start modelling we call the objective function with **random hyperparameters for five(5) times**.

Note that the objective function is trying to maximize the mean value of the 10% of the best validation accuracies which here is the top four(4) values.

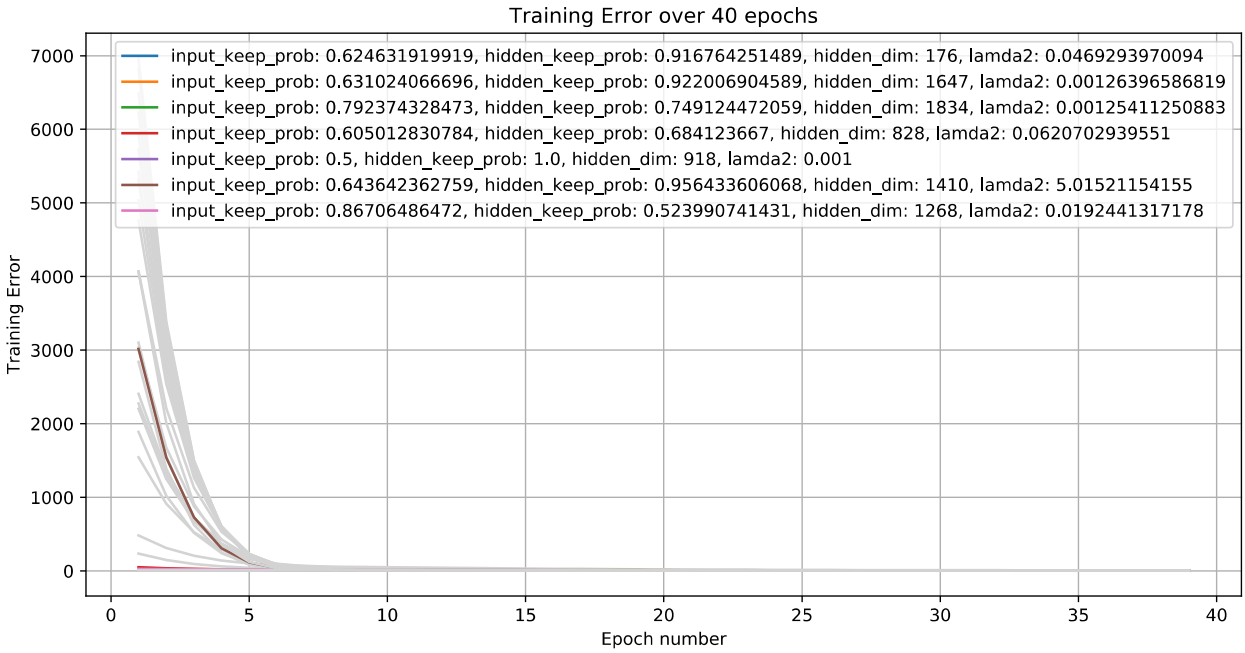
Results



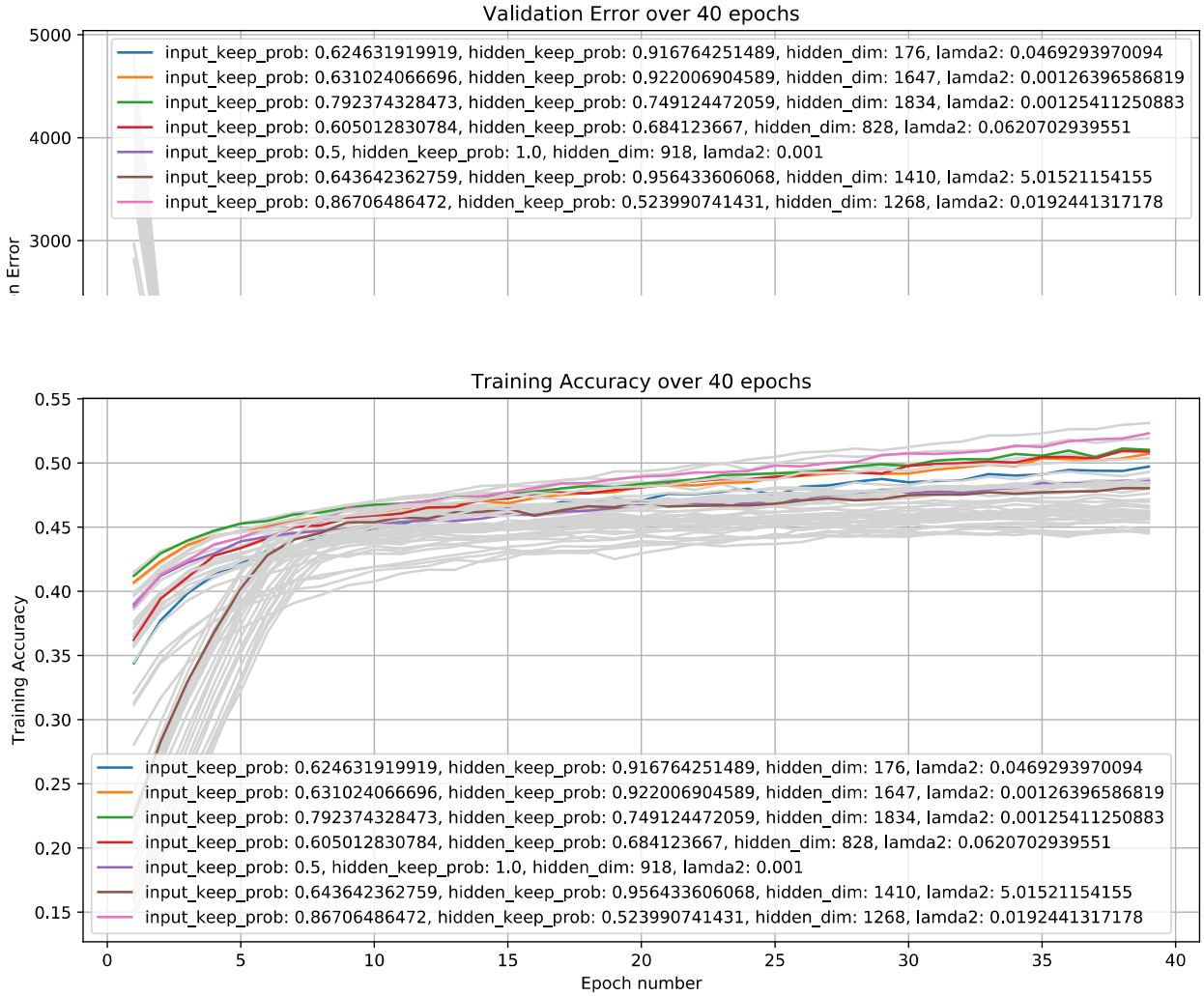
Plot 43: Convergence Plot for Bayesian Optimization trying to find optimal Hidden Dimensionality, Dropout Keep Probability of Input and Dropout Keep Probability of Hidden Layer for Student Model with Learning Rate 1e-5



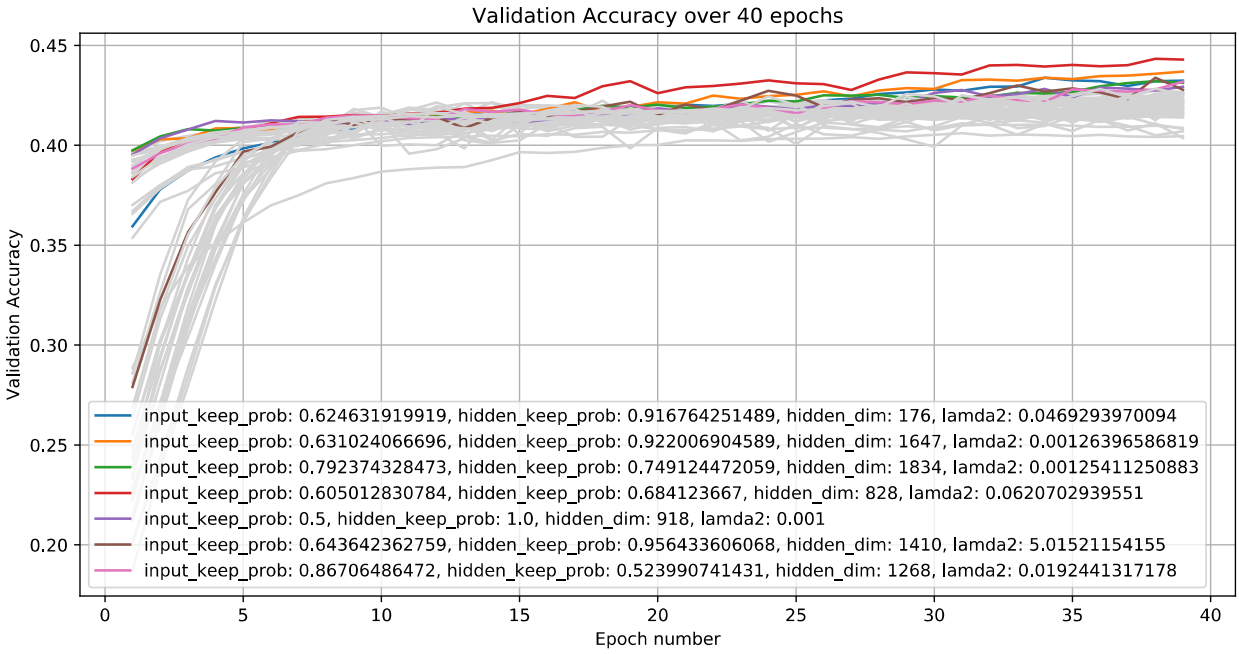
Plot 44: Output of Objective Function for Bayesian Optimization trying to find optimal Hidden Dimensionality, Dropout Keep Probability of Input and Dropout Keep Probability of Hidden Layer for Student Model with Learning Rate 1e-5



Plot 45: Training Error over 40 epochs per call for Bayesian Optimization trying to find optimal Hidden Dimensionality, Dropout Keep Probability of Input and Dropout Keep Probability of Hidden Layer for Student Model with Learning Rate 1e-5



Plot 47: Training Accuracy over 40 epochs per call for Bayesian Optimization trying to find optimal Hidden Dimensionality, Dropout Keep Probability of Input and Dropout Keep Probability of Hidden Layer for Student Model with Learning Rate 1e-5



Plot 48: Validation Accuracy over 40 epochs per call for Bayesian Optimization trying to find optimal Hidden Dimensionality, Dropout Keep Probability of Input and Dropout Keep Probability of Hidden Layer for Student Model with Learning Rate 1e-5

Best Parameters suggested from this Bayesian Optimization:

- Keep Dropout Probability of Input layer: 60.5%
- Keep Dropout Probability of Hidden Layer: 68.41%
- Number of Neurons for the Hidden Layer: 828
- L2 regularization factor: 0.062

Bayesian Optimization with Learning Rate 1e-4

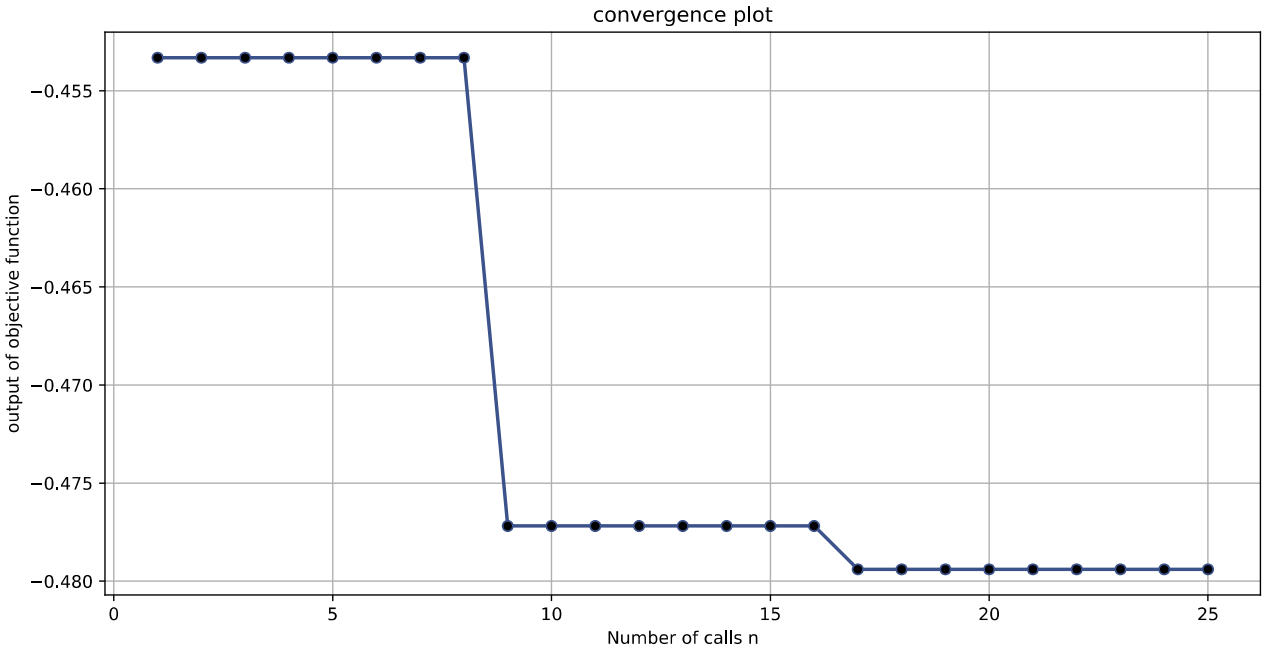
Now the learning rate of the optimizer is set ten time larger than before.

Training runs for **20 epochs** on every call of the objective function. We have used half the epochs than before because from the plots of the previous bayesian optimization we can see that at 20 epochs each model has shown evidence of being better or worse than the rest of the models.

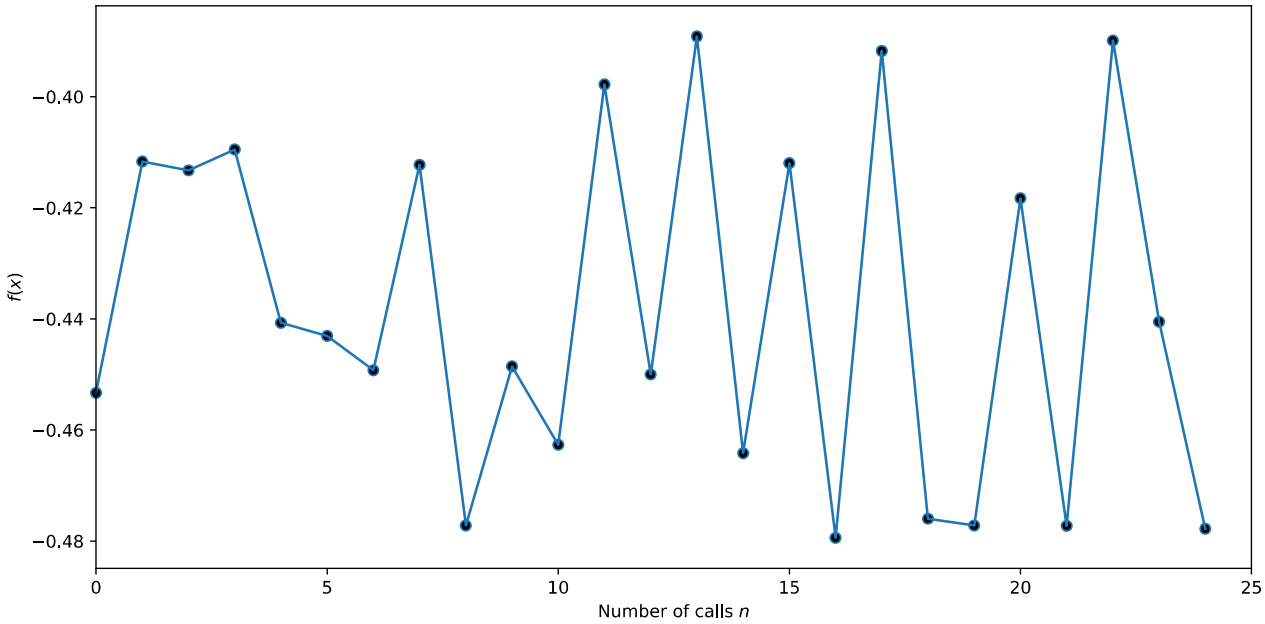
Number of calls is equal to 25. Note that we have cut in-half the number of calls because we saw that with the previous bayesian optimization the converge plot was able to converge in less than 10 calls, so to save computational resources we are reducing the number of calls.
At the initialization before we start modelling we call the objective function with **random hyperparameters for five(5) times**.

Note that the objective function is trying to maximize the mean value of the 10% of the best validation accuracies which here is the top two(2) values.

Results

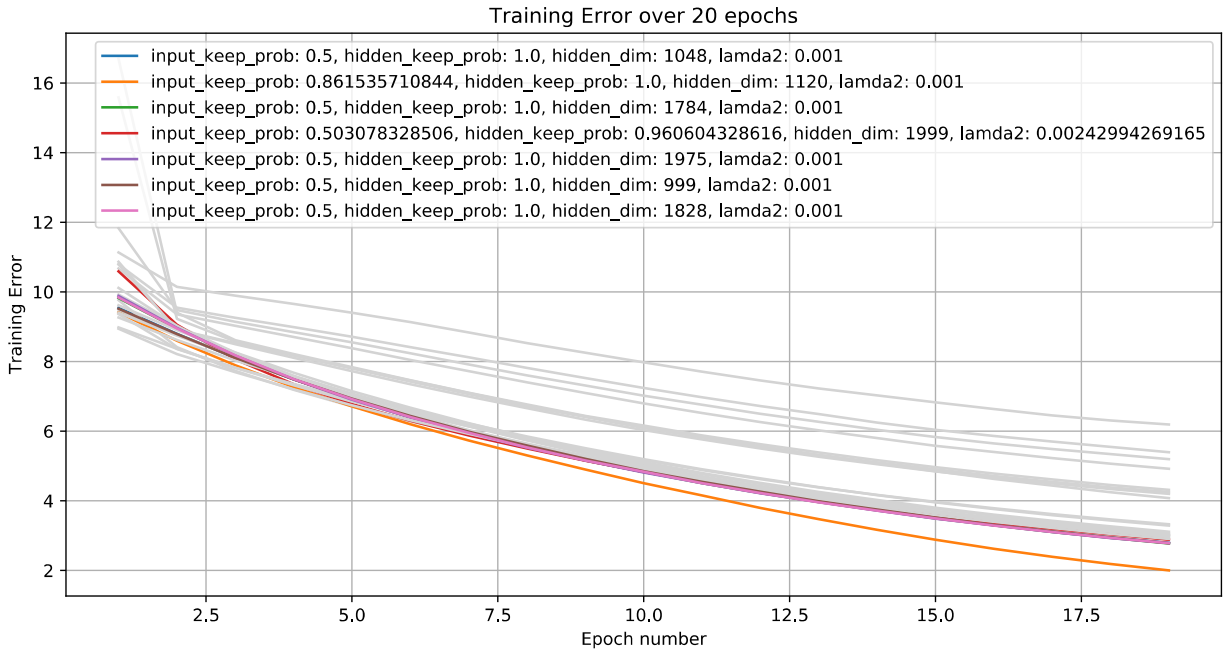


Plot 49: Convergence Plot for Bayesian Optimization trying to find optimal Hidden Dimensionality, Dropout Keep Probability of Input and Dropout Keep Probability of Hidden Layer for Student Model with Learning Rate 1e-4

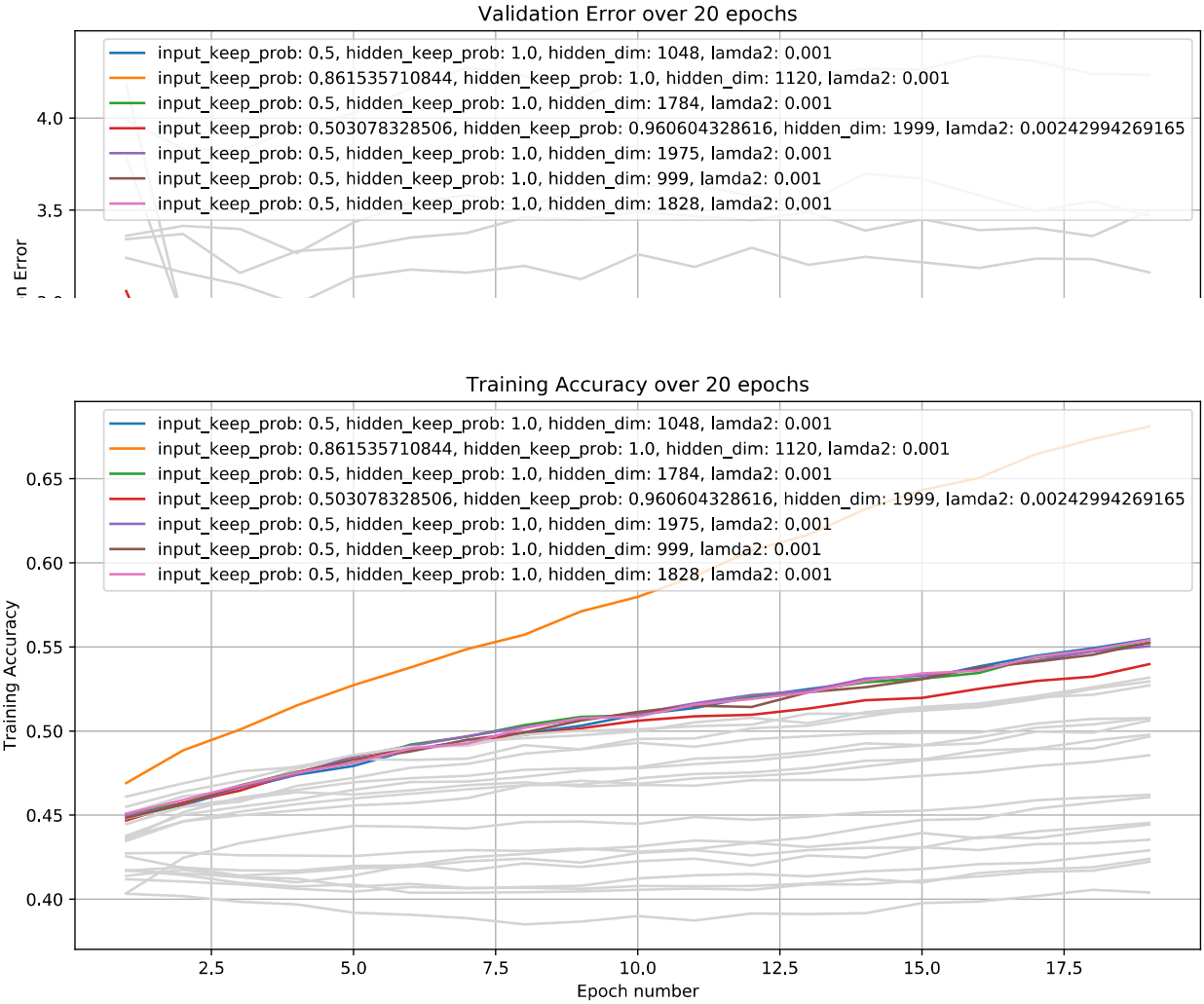


Plot 50: Output of Objective Function for Bayesian Optimization trying to find optimal Hidden Dimensionality, Dropout Keep Probability of Input and Dropout Keep Probability of Hidden Layer for Student Model with Learning Rate 1e-4

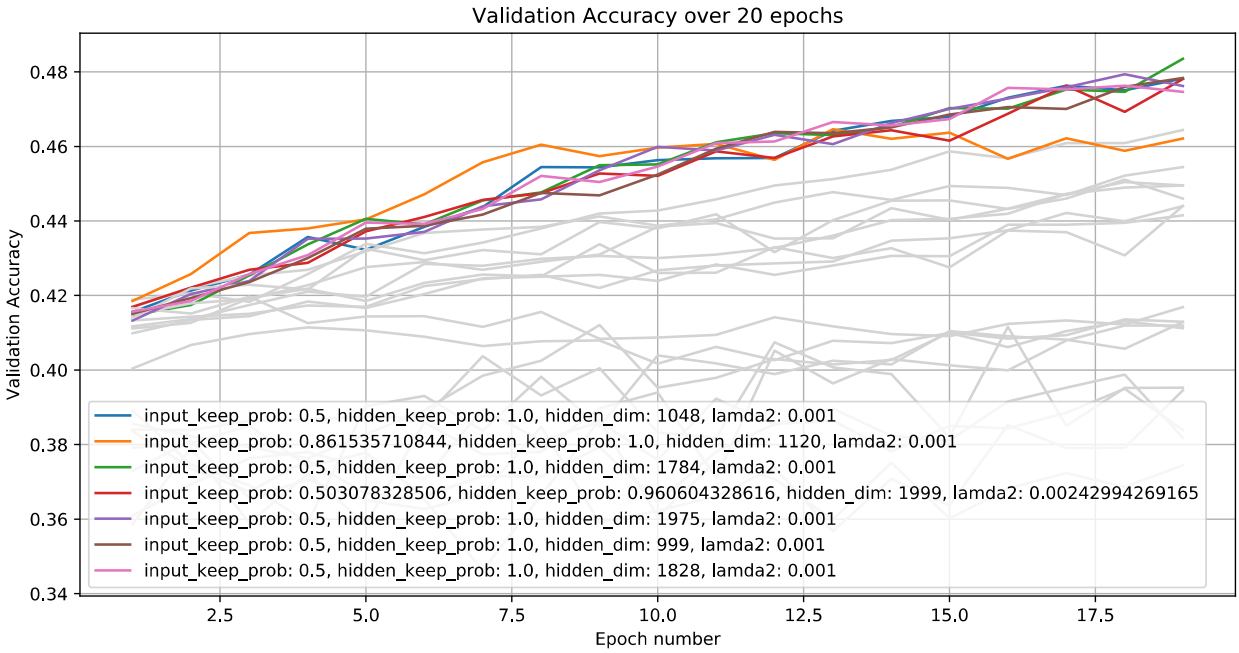
//…



Plot 51: Training Error over 40 epochs per call for Bayesian Optimization trying to find optimal Hidden Dimensionality, Dropout Keep Probability of Input and Dropout Keep Probability of Hidden Layer for Student Model with Learning Rate 1e-4



Plot 53: Training Accuracy over 40 epochs per call for Bayesian Optimization trying to find optimal Hidden Dimensionality, Dropout Keep Probability of Input and Dropout Keep Probability of Hidden Layer for Student Model with Learning Rate 1e-4



Plot 54: Validation Accuracy over 40 epochs per call for Bayesian Optimization trying to find optimal Hidden Dimensionality, Dropout Keep Probability of Input and Dropout Keep Probability of Hidden Layer for Student Model with Learning Rate 1e-4

Best Parameters suggested from this Bayesian Optimization:

- Keep Dropout Probability of Input layer: 50%
- Keep Dropout Probability of Hidden Layer: 100%
- Number of Neurons for the Hidden Layer: 1784
- L2 regularization factor: 0.001

Conclusions

First of all we note that Bayesian Optimization is able to find an overall better model, even if fewer runs and epochs are provided, with the larger learning rate than with the small learning rate.
Top validation accuracy is 48% with learning rate at 1e-4 while it was only 46% with learning rate at 1e-5.

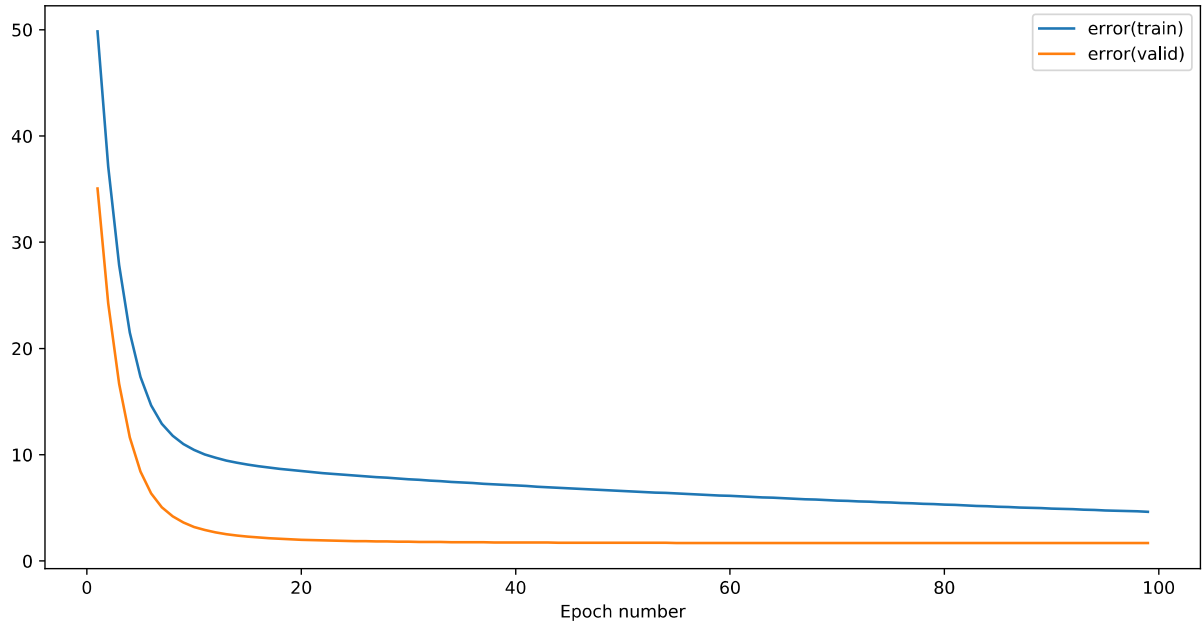
Also note that there is a higher variance for the results of the objective function for the learning rate 1e-4 case. In other words we see the objective function fluctuating between the lowest and the highest values as it tries different hyperparameters.
On the other hand when the learning rate was set to 1e-5 the convergence was really quick, in less than 10 runs the bayesian optimization had found the (locally) optimal parameters.

It is worth mentioning that the learning rate plays two roles. One role is to determine how fast the learning is going to progress but the other role is to act as an exploration parameter in the error space.
In that respect, when the learning rate was set in a small value equal to 1e-5 we had little exploration and the optimization was easily focussed on a locally optimal point in the error space missing the bigger picture.
No wonder then that the optimal dropout probabilities were found to be much smaller when the learning rate was set to 1e-5, in order to compensate for the exploration role that was missing. So in the case where learning rate was set to 1e-4 the dropout probability for the hidden layer was set to 100%.

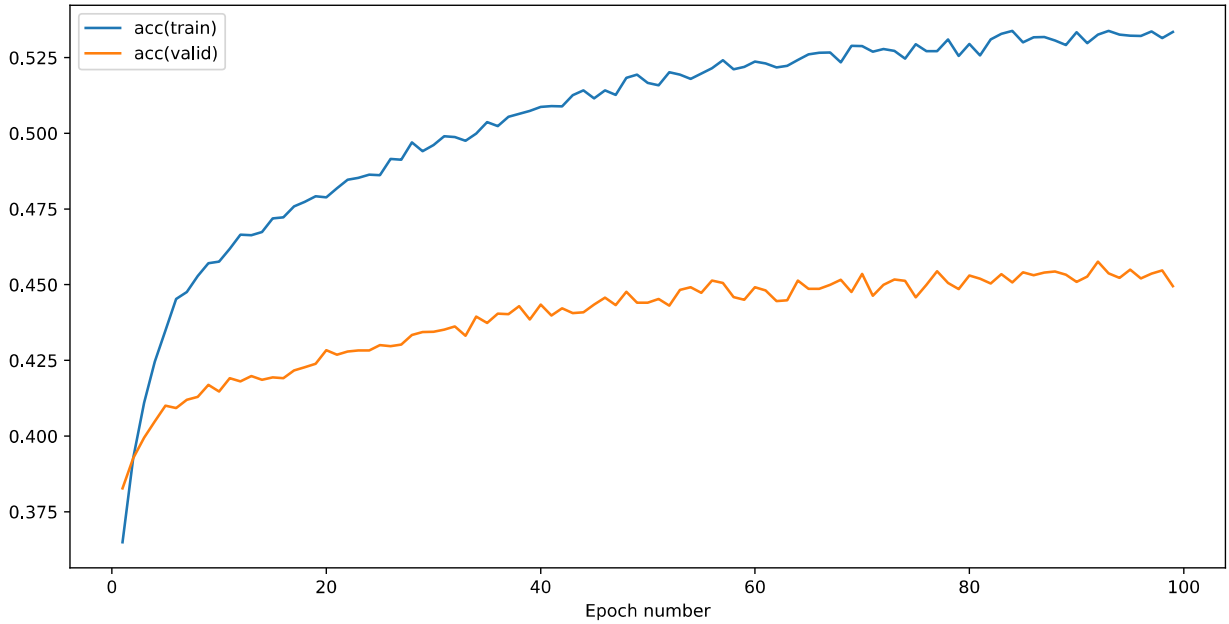
Full training of Student Shallow Neural Network with learning rate 1e-5

L2 regularization factor: 0.062070
Hidden dimensionality: 828
Learning Rate: 1e-5
Epochs: 100
Dropout Keep Probability of Input Layer: 0.605013
Dropout Keep Probability of Hidden Layer: 0.684124

Results



Plot 55: Training & Validation Error of Student Model from Teacher logits – L2 regularization factor: 0.062070 – hidden dimensionality: 828 – learning rate: 1e-5 – epochs: 100 – Dropout Keep Probability of Input Layer: 60.5% – Dropout Keep Probability of Hidden Layer: 68.4%

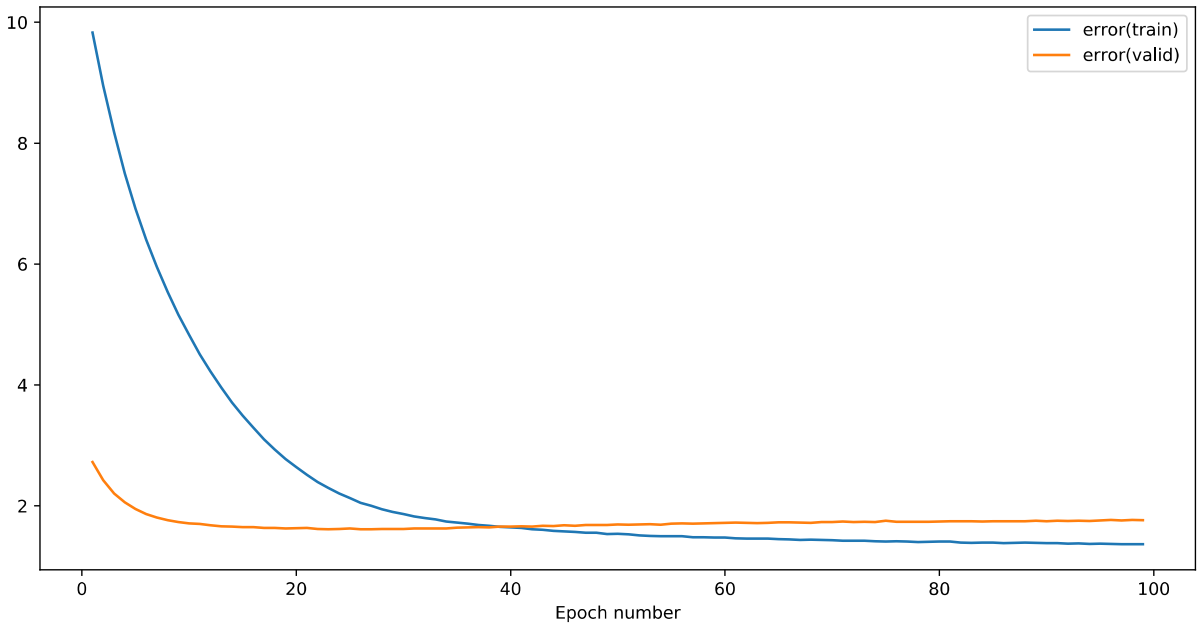


Plot 56: Training & Validation Accuracy of Student Model from Teacher logits – L2 regularization factor: 0.062070 – hidden dimensionality: 828 – Learning Rate: 1e-5 – epochs: 100 – Dropout Keep Probability of Input Layer: 60.5% – Dropout Keep Probability of Hidden Layer: 68.4%

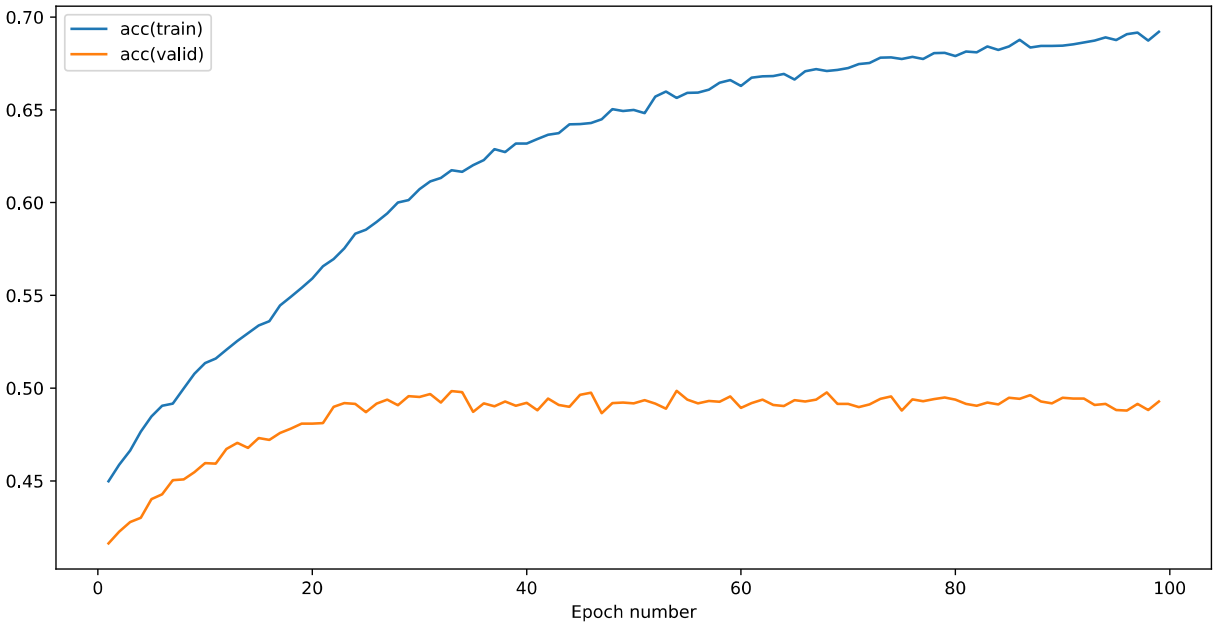
Full training of Student Shallow Neural Network with learning rate 1e-4

L2 regularization factor: 0.001000
Hidden dimensionality: 1784
Learning Rate: 1e-4
Epochs: 100
Dropout Keep Probability of Input Layer: 0.5
Dropout Keep Probability of Hidden Layer: 1.0

Results



Plot 57: Training & Validation Error of Student Model from Teacher logits – L2 regularization factor: 0.001 – hidden dimensionality: 1784 – Learning Rate: 1e-4 – epochs: 100 – Dropout Keep Probability of Input Layer: 50% – Dropout Keep Probability of Hidden Layer: 100%



Plot 58: Training & Validation Accuracy of Student Model from Teacher logits – L2 regularization factor: 0.001 – hidden dimensionality: 1784 – Learning Rate: 1e-4 – epochs: 100 – Dropout Keep Probability of Input Layer: 50% – Dropout Keep Probability of Hidden Layer: 100%

Conclusions

We notice that even for as many as 100 epochs the Student with the small learning rate of 1e-5 is not able to perform a validation accuracy of more than ~45% or barely 46%. This comes as a big or smaller trade-off when comparing the validation accuracy of Teacher model which is able to achieve the ~58% validation accuracy. This obviously depends on the business needs of accuracy versus speed.

The larger learning rate of the 1e-4 and the optimal parameters that come with it according to bayesian optimization give a Student model which takes twice the time than the Student model of the previous paragraph on the one hand but with greater validation accuracy of 50%.

Comparing the Validation Speeds of Teacher versus Student

The train_valid, the concatenation of training and validation datasets, contains in total 50 thousands instances of song data. So for running our validation method on the fully trained model for 20 epochs we will have validated 50.000 x 20 = 1 000 000 or 1 million songs.

Note that for all experiments below you will not be able to reproduce the results because those are dependent on the hardware and the operating system being used. This is not very important because we only care for comparison.

Validating using Teacher

Executing the experiment for our basic Recurrent Neural Network, our Teacher model with best parameters:

- State Size: 341
- Num Steps: 4

Results:

- Mean Time per epoch: 45.941 secs

- Min Time per epoch: 45.458 secs
- Max Time per epoch: 46.672 secs
- Variance of Time per epoch: 0.11426

As provided by the `%%time` special command of jupyter notebook:

```
CPU times: user 20min 31s, sys: 1min 57s, total: 22min 28s
Wall time: 15min 18s
```

Validating using Student

Executing the experiment for our Shallow MLP Neural Network, our Student model with best parameters:

- Learning Rate for Adam Optimizer: 1e-4
- Dropout Keep Probability: 100% (because on validating we always set dropout to 100%)
- Hidden Dimensionality: 1784
- L2 regularization factor: 1e-3

Results:

- Mean Time per epoch: 9.420 secs
- Min Time per epoch: 9.304 secs
- Max Time per epoch: 10.015 secs
- Variance of Time per epoch: 0.0256

As provided by the `%%time` special command of jupyter notebook:

```
CPU times: user 1min 28s, sys: 10.4 s, total: 1min 38s
Wall time: 3min 11s
```

Conclusions

The conclusions from the results of the two experiments above show that we have achieved a ~**5x** improvement in speed as all metrics suggest.
This could should be taken into account when building a working systems that would need to classify potentially millions of songs.

Further work for Student-Teacher model (not implemented)

Further work from this, it could be to have a system where both the Teacher and the Student model are being used. A brief description of such an implementation would include passing the song data through the Student and if the cross entropy was bigger than a certain threshold, which we would had set as a hyperparameter, then the Teacher model would be called to handle this classification. Of course at the end we would pick the one of the two with the smaller cross entropy.
Setting this threshold hyperparameter at an appropriate level would give a continuous trade-off between speed and classification accuracy.
We will not implement this system on coursework4.

Research Question: How well the basic Recurrent Neural Network architecture used for MSD-10 classification task will perform on the MSD-25 classification task?

We would like to explore whether this basic RNN architecture is able to give us an improvement on the MSD-25 classification task in comparison to the MLP achitecture of coursework3.

Extending RNN class functionality

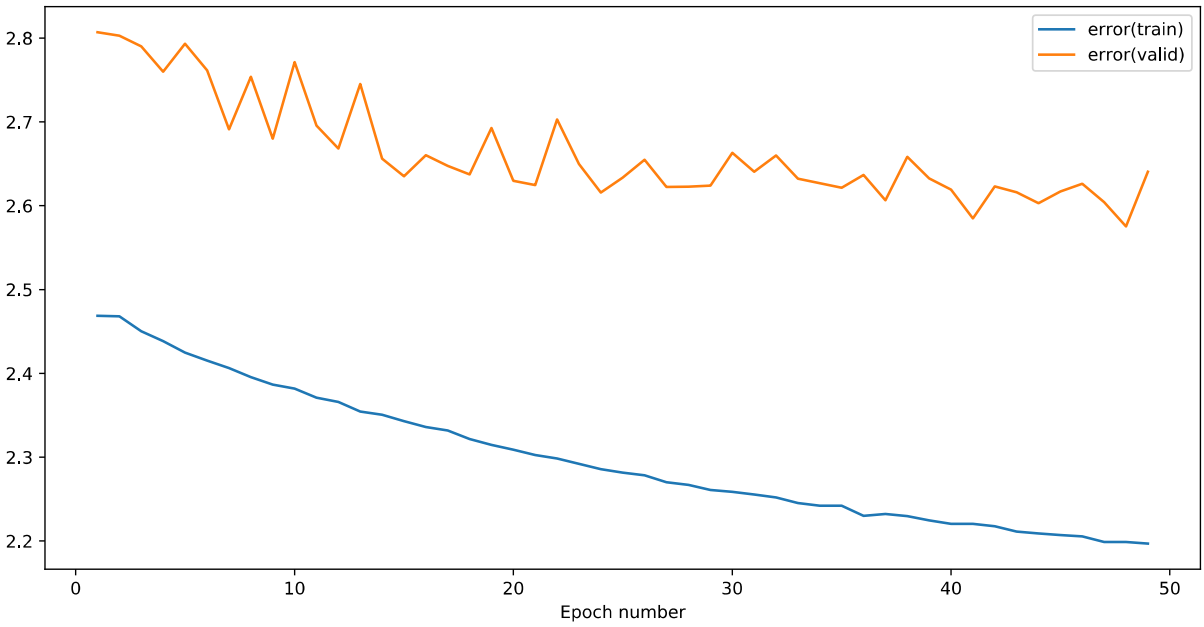
In order to test on MSD-25 class we need to update our `ManualRNN` class. The constructor receives an extra parameter which by default is 10 and can also be set to the value 25.

We also had to generalize the data providers to work with an arbitrary number of classes. For this reason a new module was created found under the path `rnn.data_providers`.
This module contains the class `MSD120RnnDataProvider` which is contains all of the functionality of the `MSD10Genre_120_rnn_DataProvider` that was previously used but by now being agnostic to the number of classes being used. The `MSD120RnnDataProvider` class is now the default class being used for the role of the data provider in `ManualRNN` class.
Note that to complete the functionality we had to re-create the base class of `MSD120RnnDataProvider` class to also be agnostic to the number of classes. So inside `rnn.data_providers` module the `MSDDataProvider` class can be found. This works as long as the number of classes are encoded as the corresponding integer at the prefix of the file which contains the actual data. In other words the filename should begin like *msd-<XX>-...* where *<XX>* should be substituted with the number of the classes.

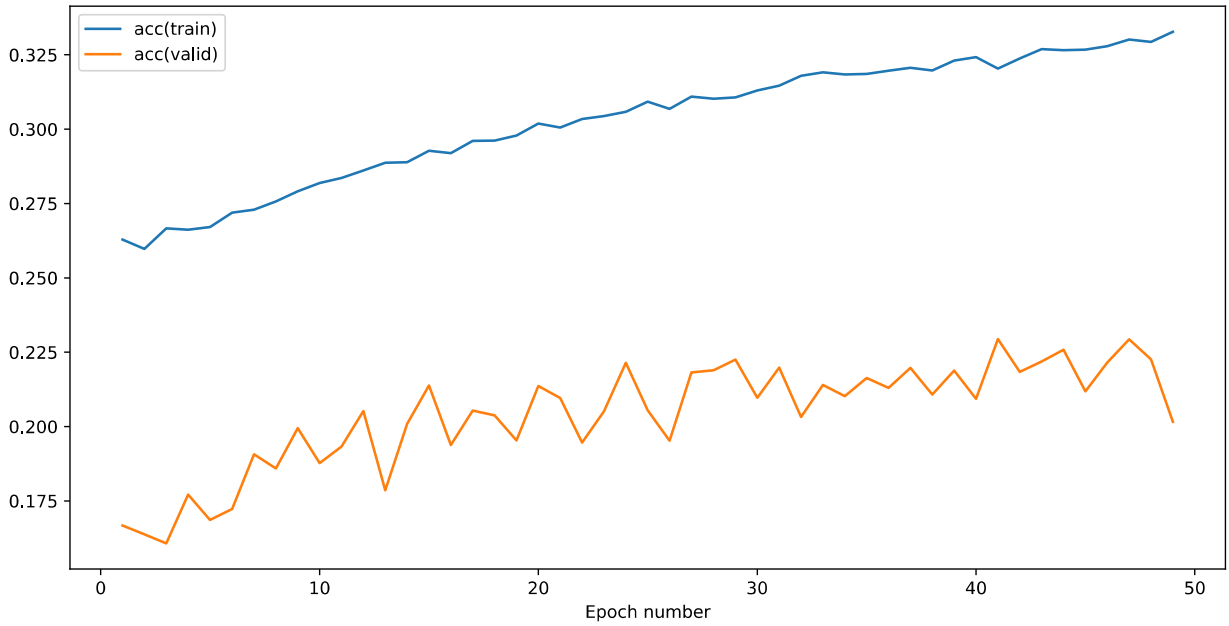
The graph and corresponding architecture is exactly the same as the one used for MSD-10 case.

We are executing the experiment for 50 epochs using the same parameters that worked best for the MSD-10 case:
State Size: 341
Number of Steps: 4

Results



Plot 59: Training & Validation Error – Recurrent Neural Network – State Size: 341 – Num Steps: 4 – MSD 25 classification task – Learning Rate: 1e-4



Plot 60: Training & Validation Accuracy – Recurrent Neural Network – State Size: 341 – Num Steps: 4 – MSD 25 classification task – Learning Rate: 1e-4

Conclusions

We notice that there is a lot of oscillation on the validation accuracy and error which means that the training of our basic RNN model has a hard time to find an equilibrium.

The performance of the validation accuracy is at 22.94% which is considered much lower than the 25% we achieved in coursework3.

We could perhaps achieve a better validation accuracy by performing bayesian optimization on the hyperparameters for the MSD-25 classification task.
However we put in higher priority to address the issue of the high variance of the validation error/accuracy which is apparent in both MSD-10 and MSD-25 classification tasks and which we hypothesize originates from the high variance of the weights of the neural network and as a result it makes training more difficult.

Research Question: Could batch normalization have a positive effect on the classification performance of our Recurrent Neural Network?

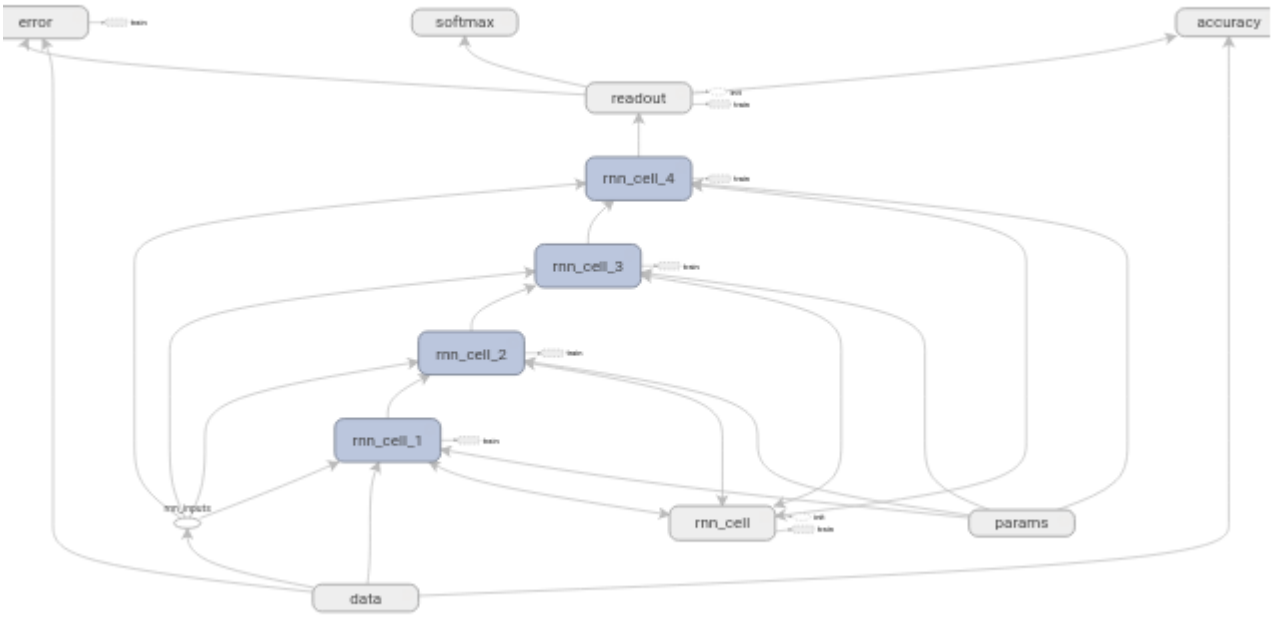
Now that we have achieved a fast working system it makes sense to seek how could we achieve a higher validation accuracy.

From previous experiments in coursework3 we have seen the benefits of batch normalization in terms of performance for both speed and higher classification accuracy for the MLP case of the Neural Network. We have also seen that batch normalization reduces the variance of the weights of the corresponding layer which is the factor responsible for the benefits it provides.
It seems that our current implementation of our basic Recurrent Neural Network suffers from exactly this issue of high variance of the validation error and accuracy. We hypothesize that batch normalization within the RNN cell will help mitigate this issue.

Implementing Recurrent Neural Network with Batch Normalization

The Recurrent Neural Network with the support of Batch Normalization is implemented in `RNNBatchNorm` class found in `rnn.rnn_batch_norm` module. This class extends the `ManualRNN` class and overrides functions related to batch normalization.

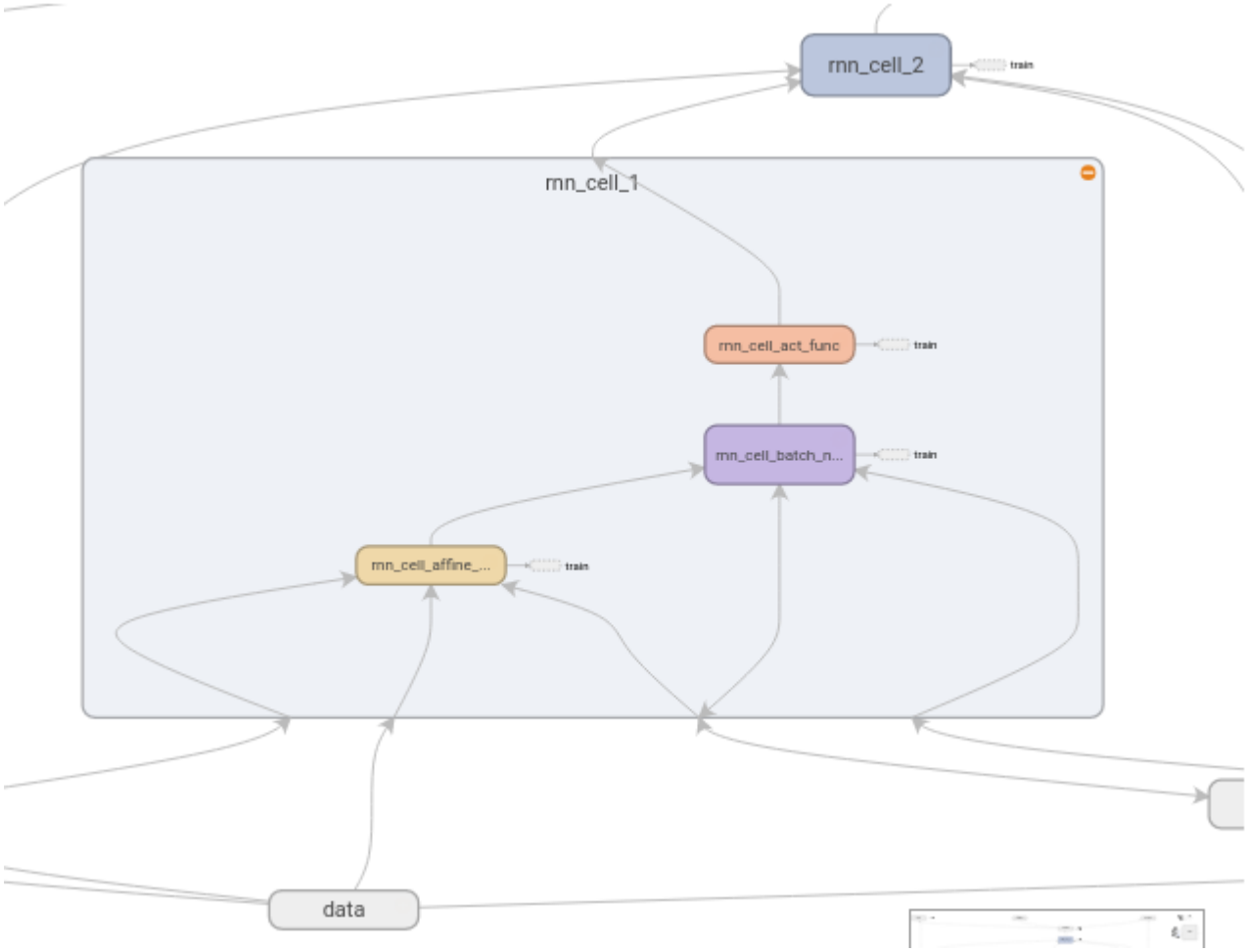
First of all we override `getGraph` method because we want to implement a new tensorflow graph that includes a batch normalization. The new graph is similar to the one we used for the basic RNN and it is show here:



Graph 6: Basic Recurrent Neural Network with Batch Normalization – RNN steps: 4

Note that the Softmax node is not connected anywhere because it is used only to produce the predictions for the Kaggle in-class competition.

The batch normalization layer is being placed between the affine layer and the non-linearity, here tanh, as shown here:



Graph 7: RNN cell of Basic Recurrent Neural Network with Batch Normalization

Note that the `RNNBatchNorm` class overrides the function `batchNormWrapper_byExponentialMovingAvg` found in the `mylibs.batch_norm` module in order to provide an implementation of batch normalization which works with shared variables by using the `tf.get_variable` method from Tensorflow. Note that the hyperparameter epsilon of batch normalization stays constant at 1e-3.

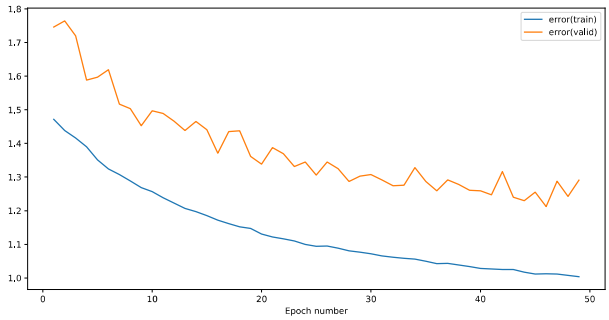
Experiments with RNN with Batch Normalization

We are executing the experiment for **50 epochs** with the best parameters we had for our basic recurrent neural network:

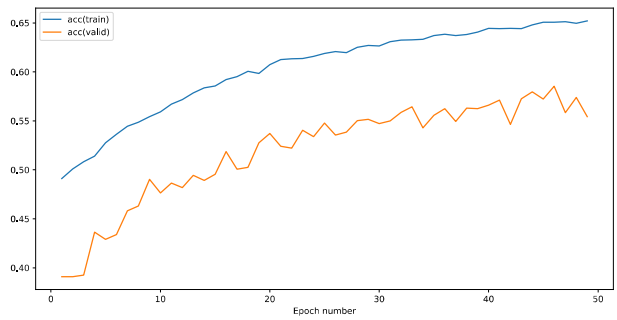
- **State Size**: 341
- **Number of RNN steps**: 4

Results

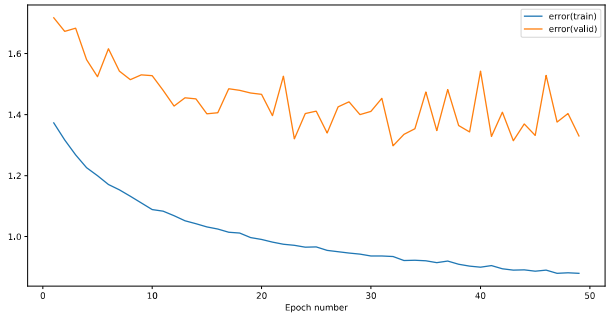
Note that we are putting the results without Batch Normalization underneath for comparison purposes.



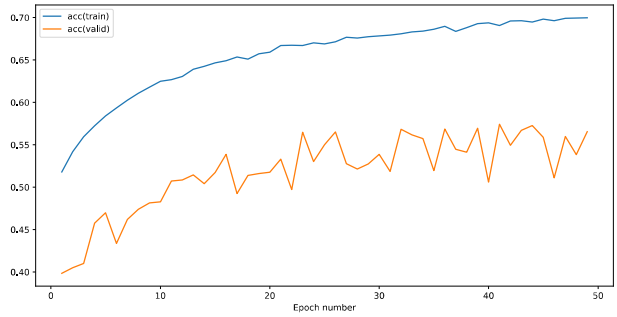
Plot 61: Training & Validation Error - Basic Recurrent Neural Network with Batch Normalization



Plot 62: Training & Validation Accuracy - Basic Recurrent Neural Network with Batch Normalization



Plot 63: Training & Validation Error - Basic Recurrent Neural Network without batch normalization



Plot 64: Training & Validation Accuracy - Basic Recurrent Neural Network without batch normalization

Conclusions

Comparing the plots above it is obvious from both the validation error and validation accuracy that the architecture which lacks batch normalization has a larger variance than the architecture which includes batch normalization.

Lower variance has the benefit of easier learning and this is reflected at the higher validation accuracy achieved within the same epochs which peaked over 58% while previous we were only above 57%.

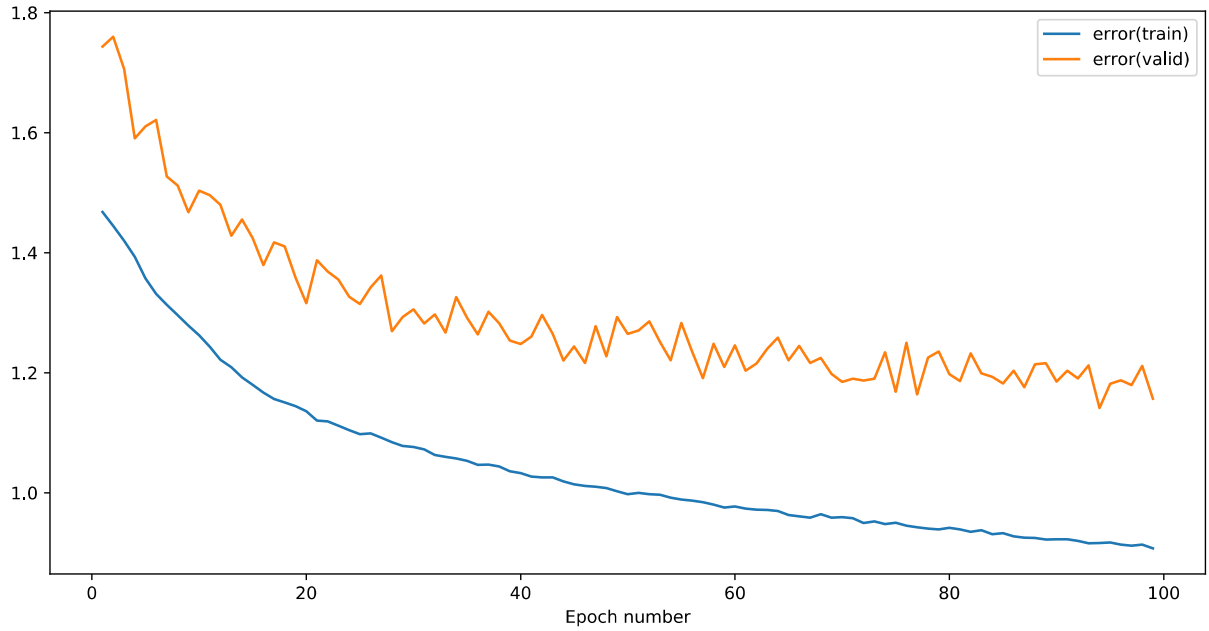
Now that the increasing trend is more obvious it makes sense to run training for more epochs to see how high it could reach.

RNN with Batch Normalization Experiment for 100 epochs

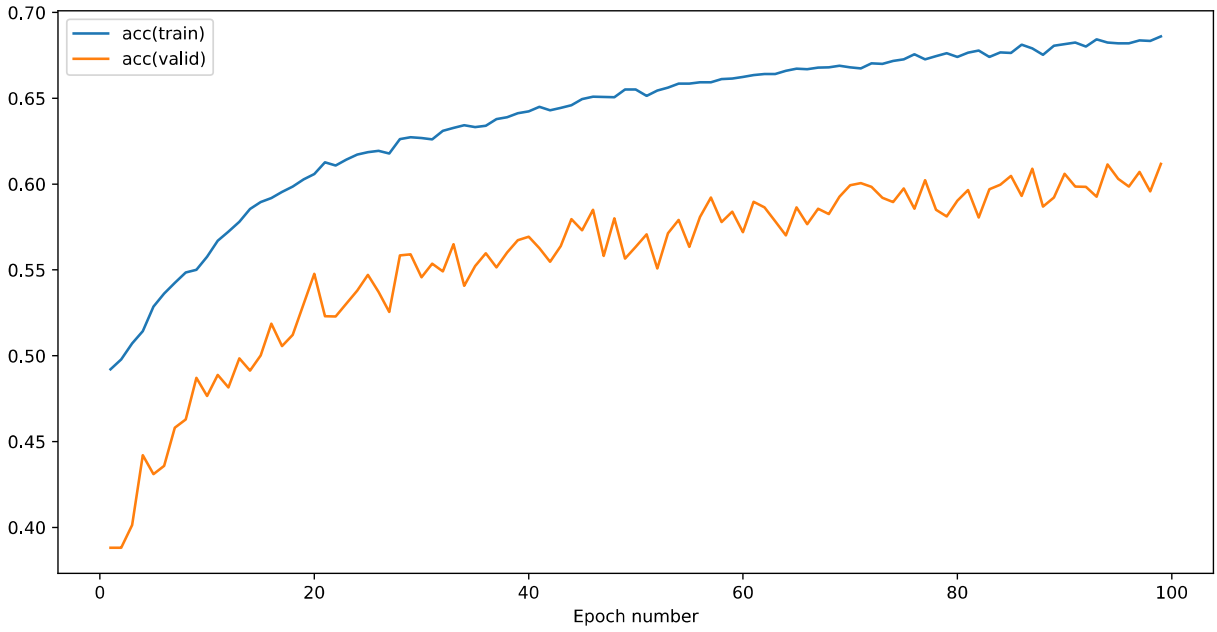
We are executing the experiment for **100 epochs** with the best parameters:

- **State Size**: 341
- **Number of RNN steps**: 4

Results



Plot 65: Training & Validation Error - Basic Recurrent Neural Network with Batch Normalization – Epochs: 100 – State Size: 341 – RNN steps: 4



Plot 66: Training & Validation Accuracy - Basic Recurrent Neural Network with Batch Normalization – Epochs: 100 – State Size: 341 – RNN steps: 4

Conclusions

We are pleased with the results from running training for 100 epochs since we have peaked at 61.17% without getting any clear indications of overfitting.
If we had more computational resources available it would be interesting to see how it performs for two hundred or more epochs.

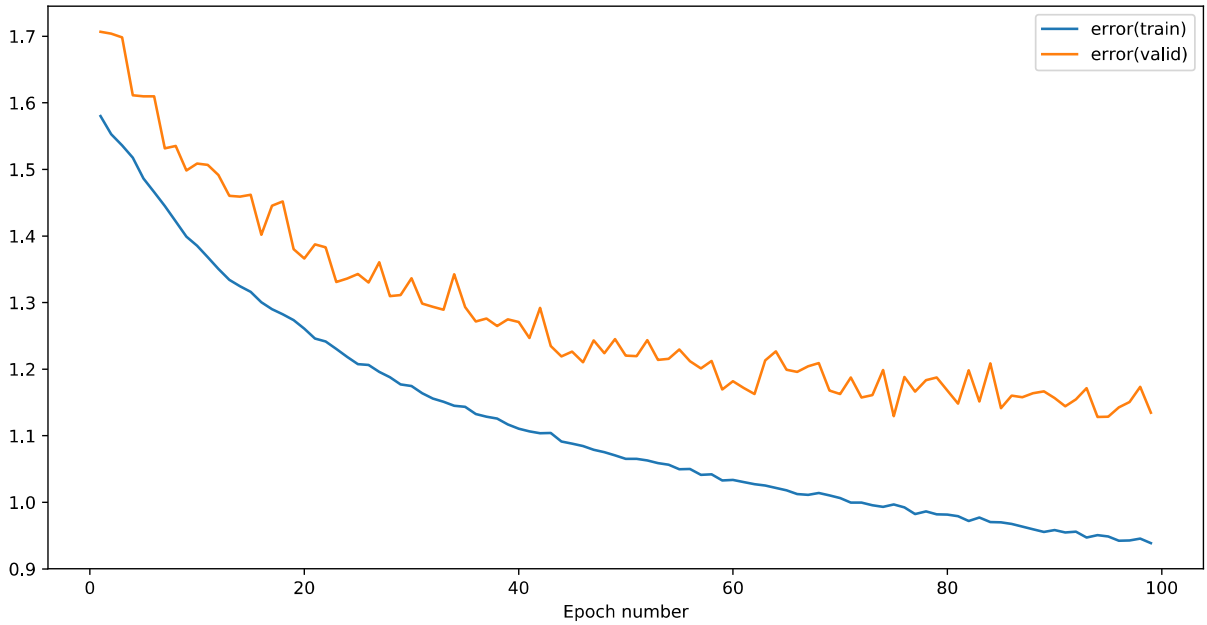
Now it is worthwhile to compare against the 8 rnn steps which had shown promising results before when we were doing the grid search, to see if it performs better or worse.

RNN with Batch Normalization Experiment for 100 epochs and with 8 rnn steps

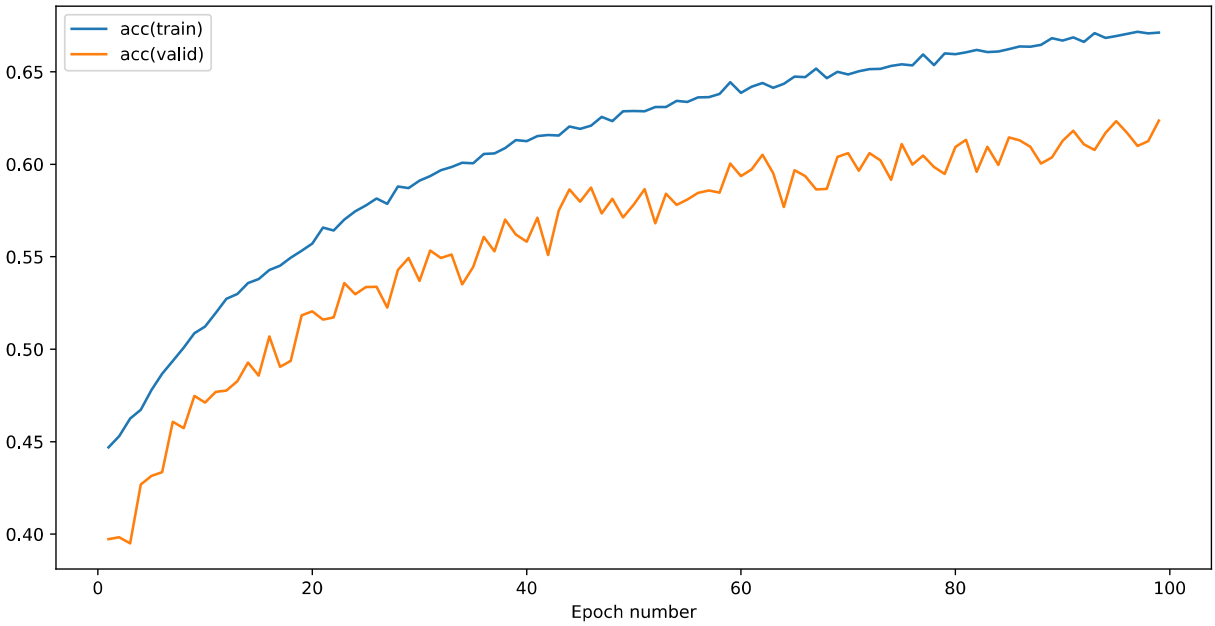
We are executing the experiment for **100 epochs** with parameters:

- **State Size**: 341
- **Number of RNN steps**: 8

Results



Plot 67: Training & Validation Error - Basic Recurrent Neural Network with Batch Normalization – Epochs: 100 – State Size: 341 – RNN steps: 8



Plot 68: Training & Validation Accuracy - Basic Recurrent Neural Network with Batch Normalization – Epochs: 100 – State Size: 341 – RNN steps: 8

Conclusions

We are pleased with the results of this experiment since we peaked at over 62% which is a new record. The difference is not considered substantial enough to give definite results between 4 and 8 RNN steps and in addition the state size was considered fixed and it is a result of another optimization.

Future work (not implemented)

- Investigating Architectures like LSTMs and GRUs
- Investigating which feature, Chrome, Timbre or Loudness is the most prominent in each music class
- Data Augmentation with other ways than Dropout. For example Gaussian Noise
- Experiment with the Variable Length Dataset
- Splitting the Variable Length Dataset into fixed length parts in order to artificially get more instances