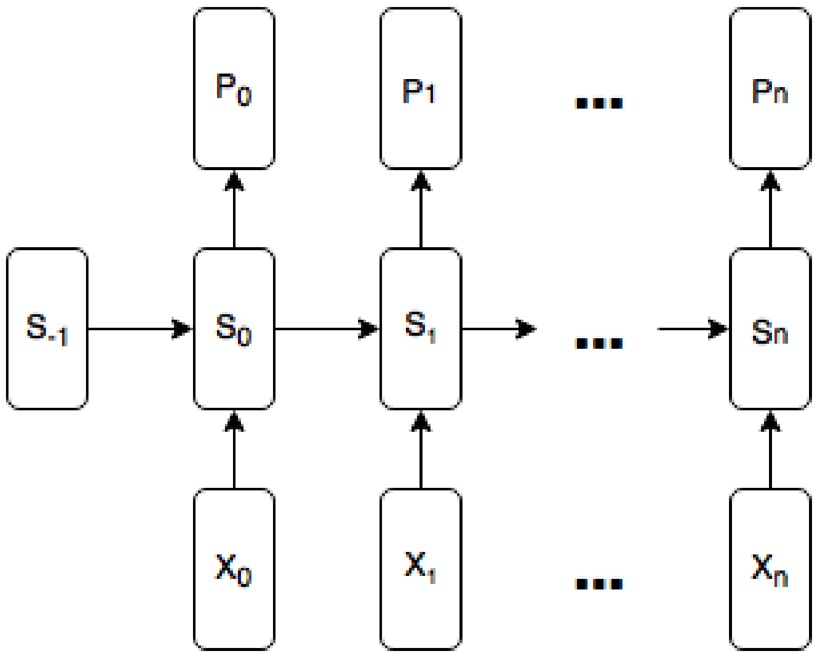
Research Question: Could the basic Recurrent Neural Network help to achieve better accuracy for MSD classification task?

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.

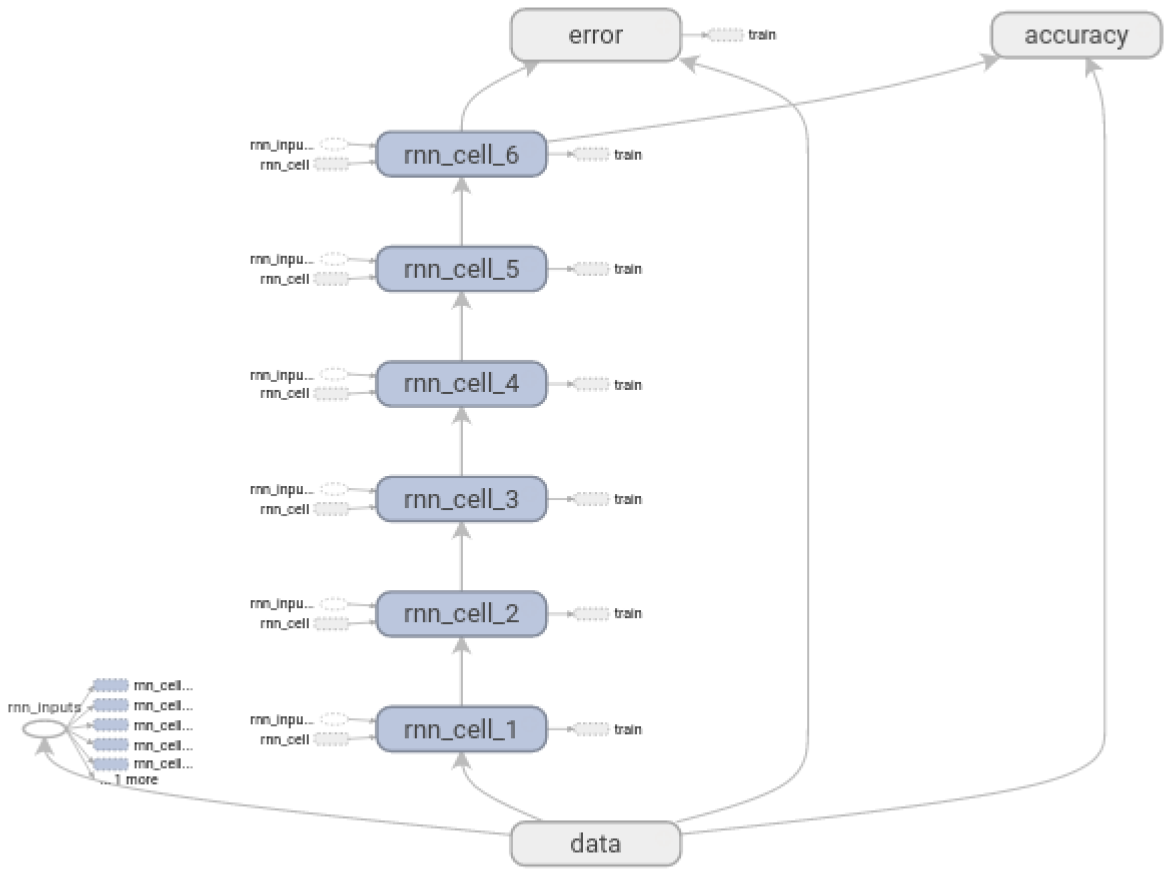
Note that this has the implication that we are going to ask from our neural network to classify each part of the song. However we expect that since the final state of each part is feeded as the initial state of the next part that the last part will contain all the necessary information and we are going 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 <u>not</u> include batch normalization, nor dropout, nor L2 regularization.

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

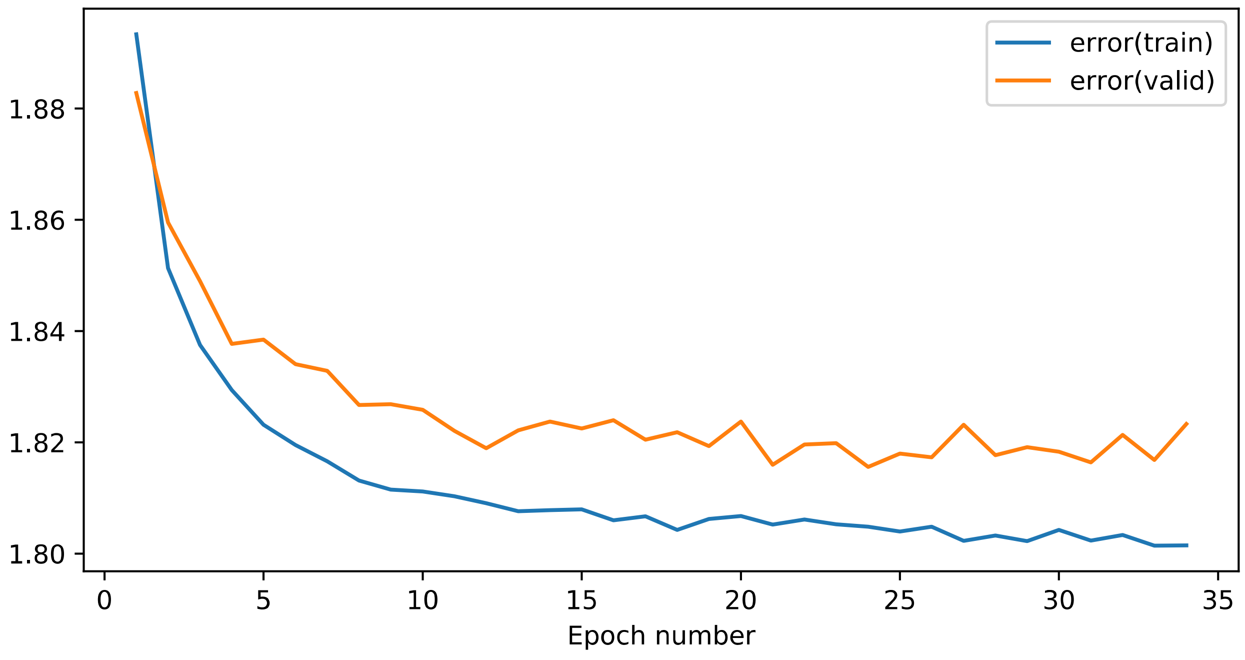


Graph 3: Basic Recurrent Neural Network unfolded to each RNN cell with shared weights – RNN steps: 6

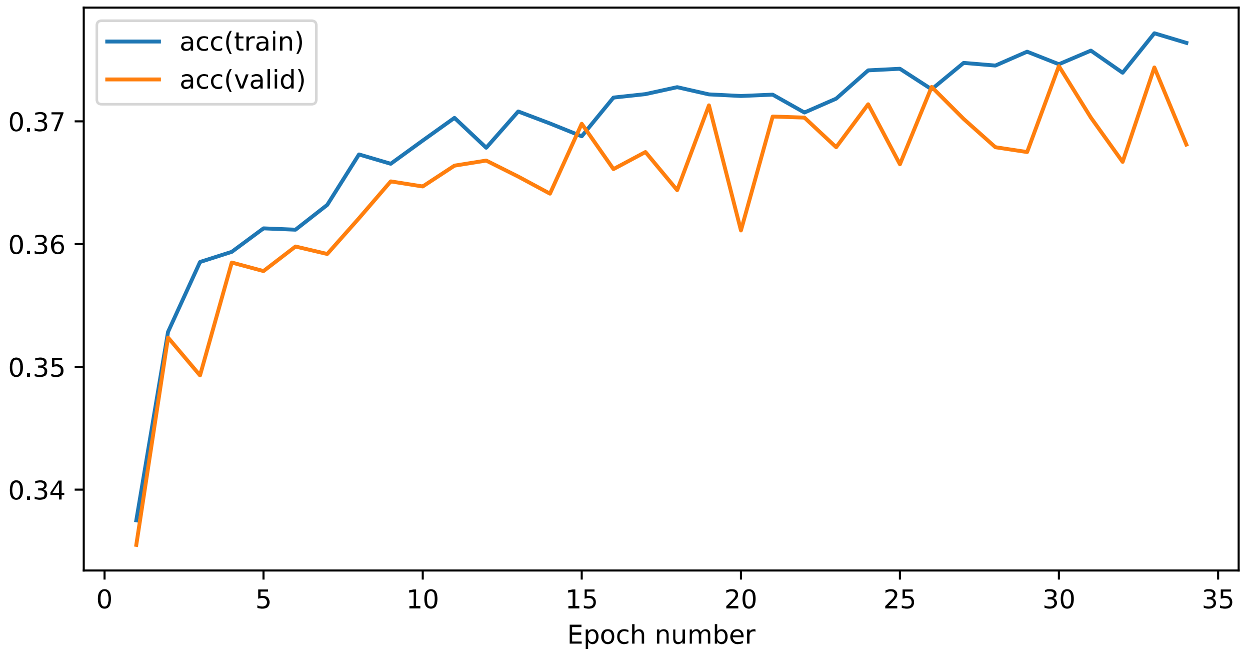
Note that this is a simplistic model where the state size is exactly equal with the number of classes included in our classification task, here ten(10), that is why we are able to apply a softmax directly to the outputs of the final state. This is of course very restrictive but this is only a first basic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

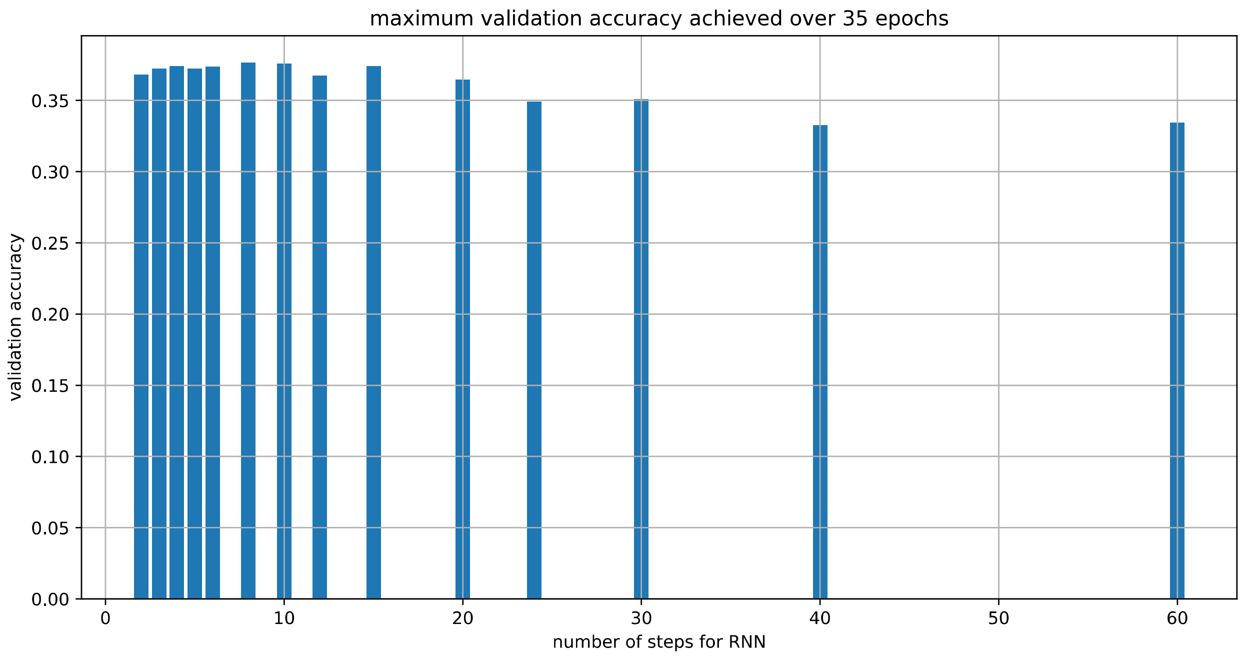
Since the 120 segments can be divided exactly by only a fixed set of number of steps we may as well search all of the following cases in order to get a hint for the optimal number of steps for our current dataset.

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

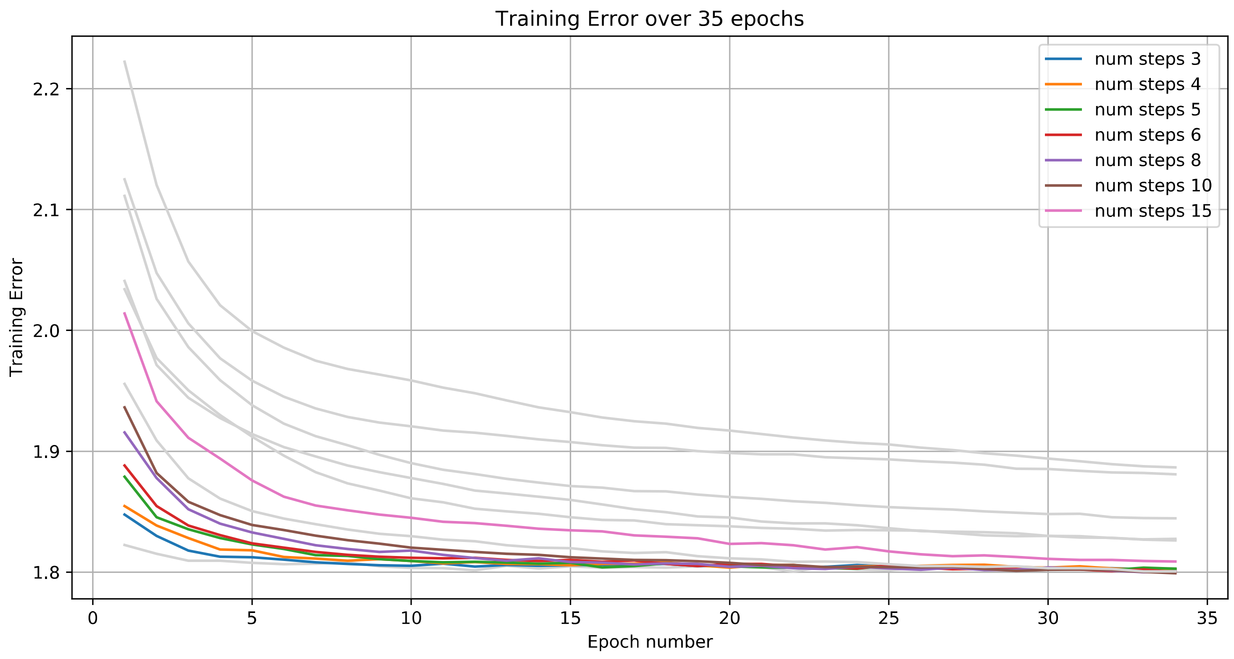


Plot 25: Maximum Validation Accuracy achieved within 35 epochs for various RNN steps for Basic Recurrent Neural Network without readout layer with State size: 10

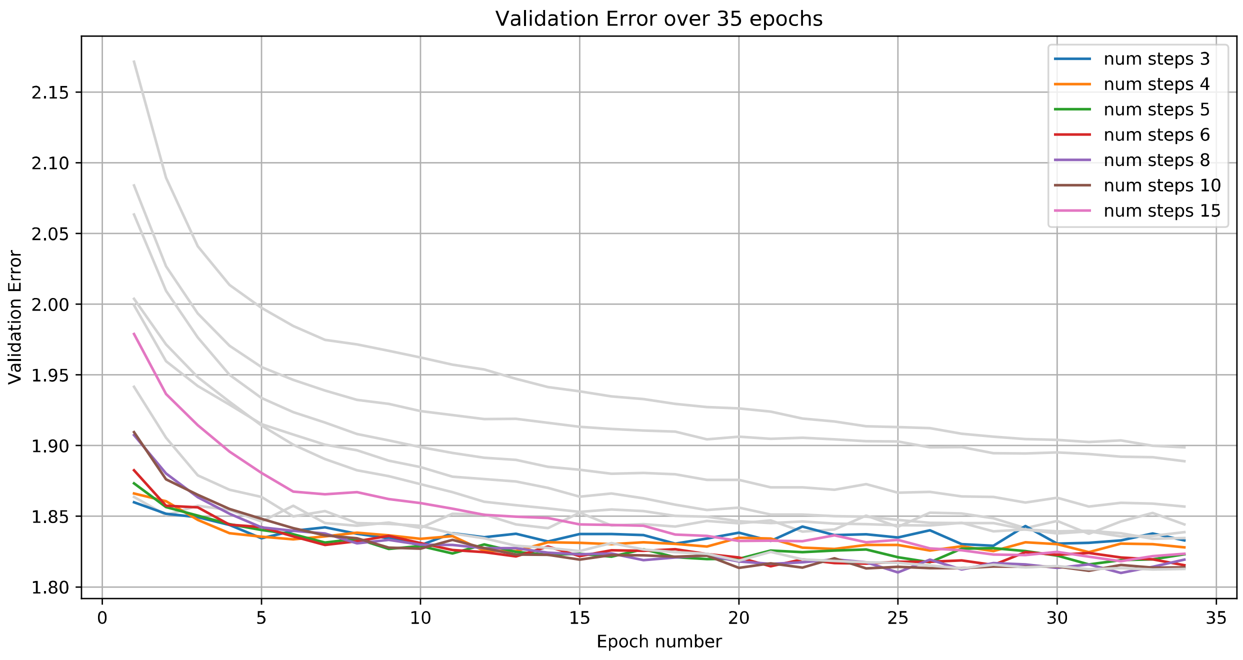
The seven number of steps as hyperparameter which correspond to the highest validation accuracy are ordered from higher to lower:

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

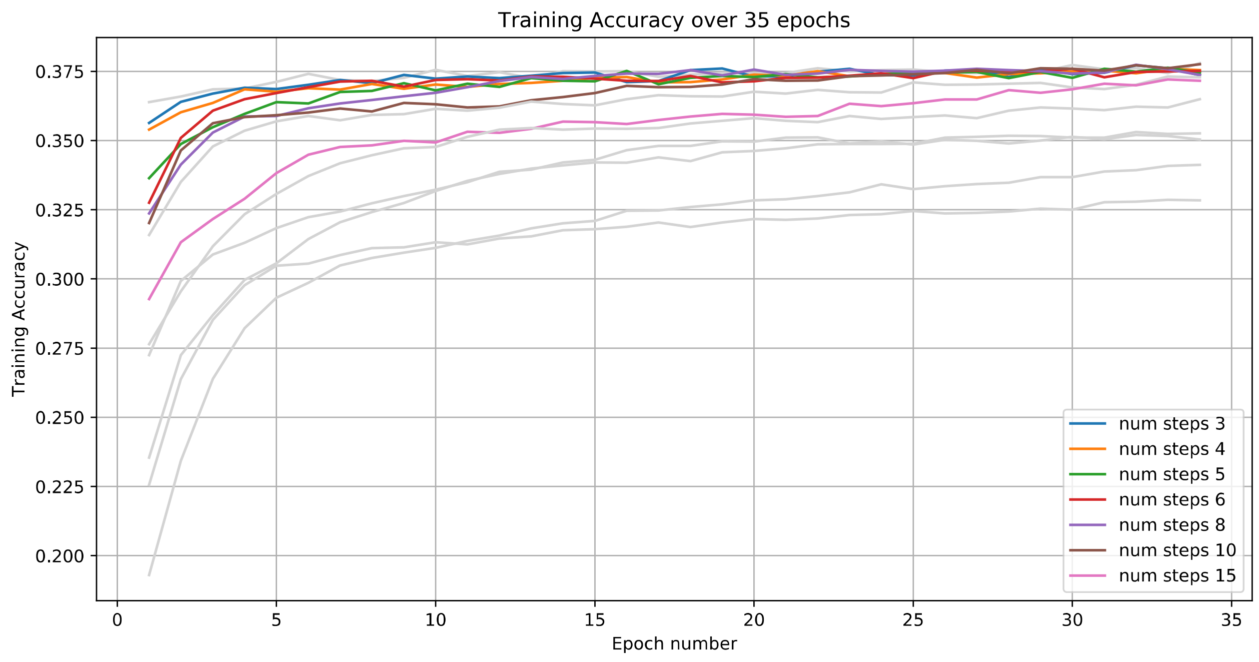
Below we are plotting the Training and Validation Error as well as the Training and Validation Accuracy of the seven best number of steps to examine and compare their behavior.



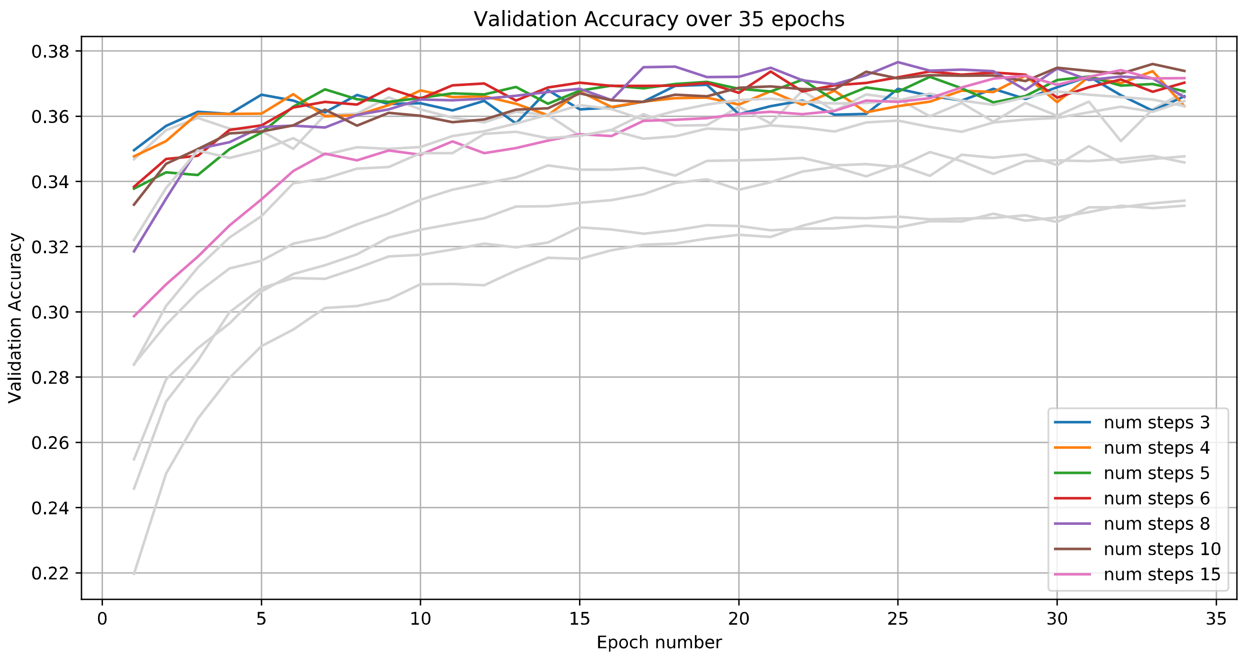
Plot 26: Training Error for various RNN steps for Basic Recurrent Neural Network without readout layer and with State size: 10



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.

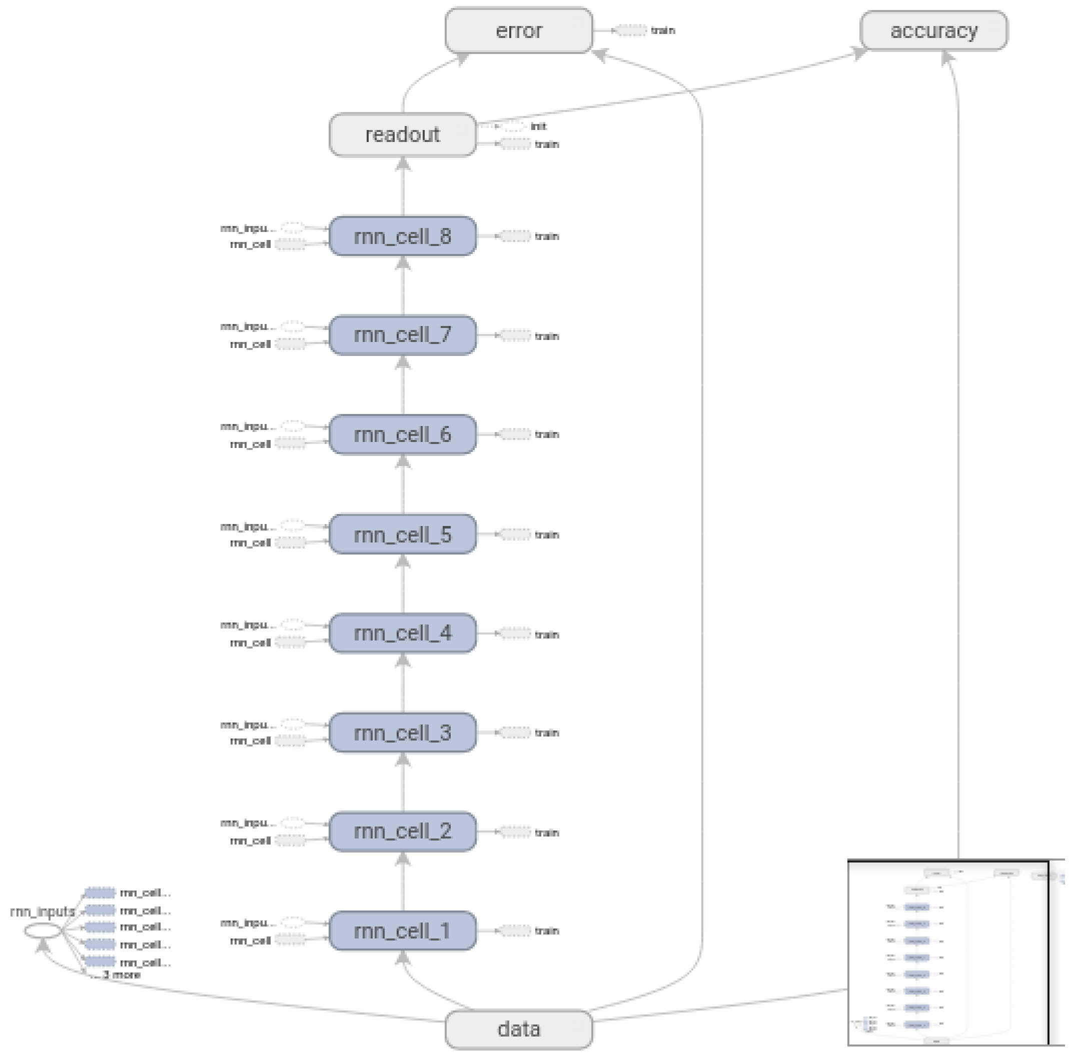
Exploration of best combination of state size and RNN number of steps with Bayesian Optimization

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.

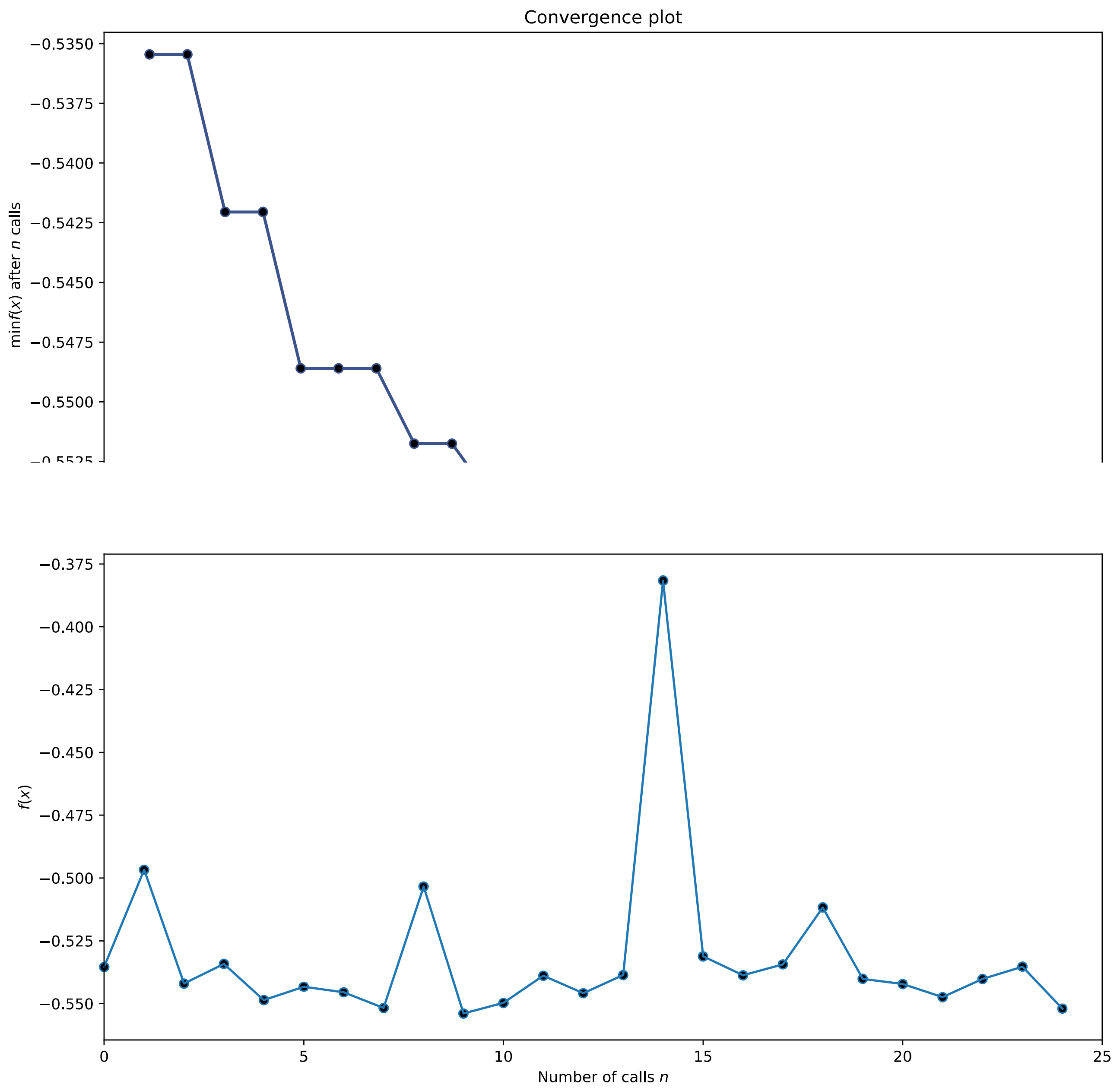
Hyperparameter Kappa, the trade-off between exploration and exploitation is set to 1.9.

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.

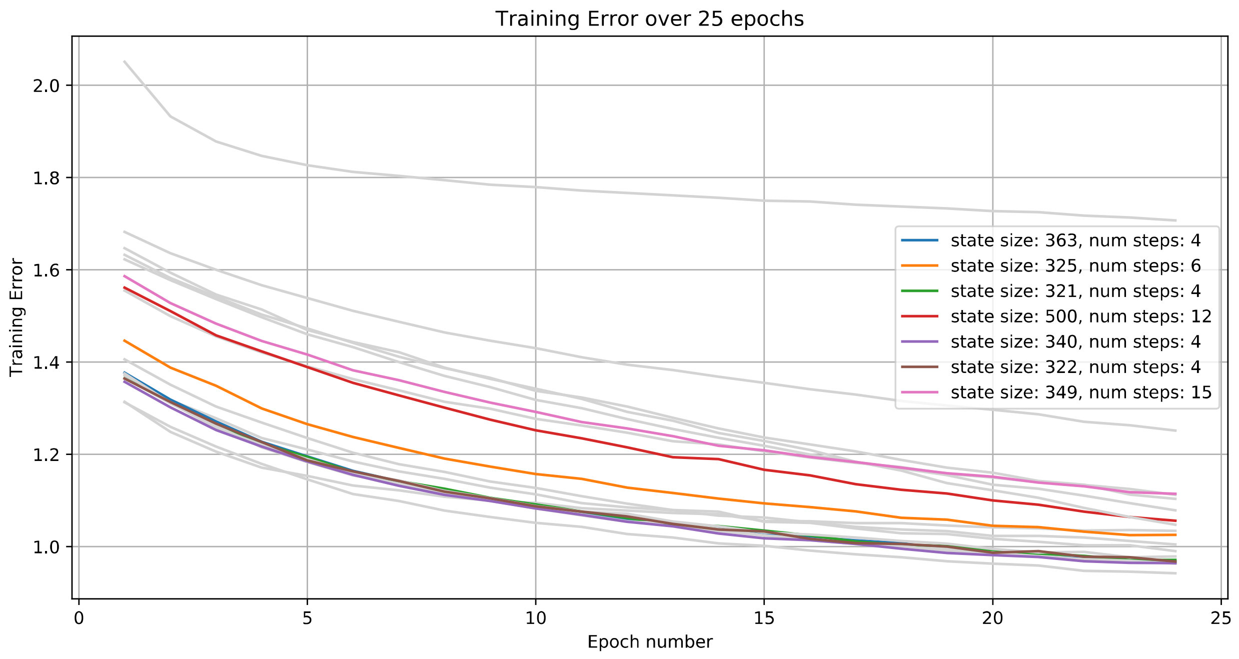
For similar reason each experiment is executed for 25 epochs.

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

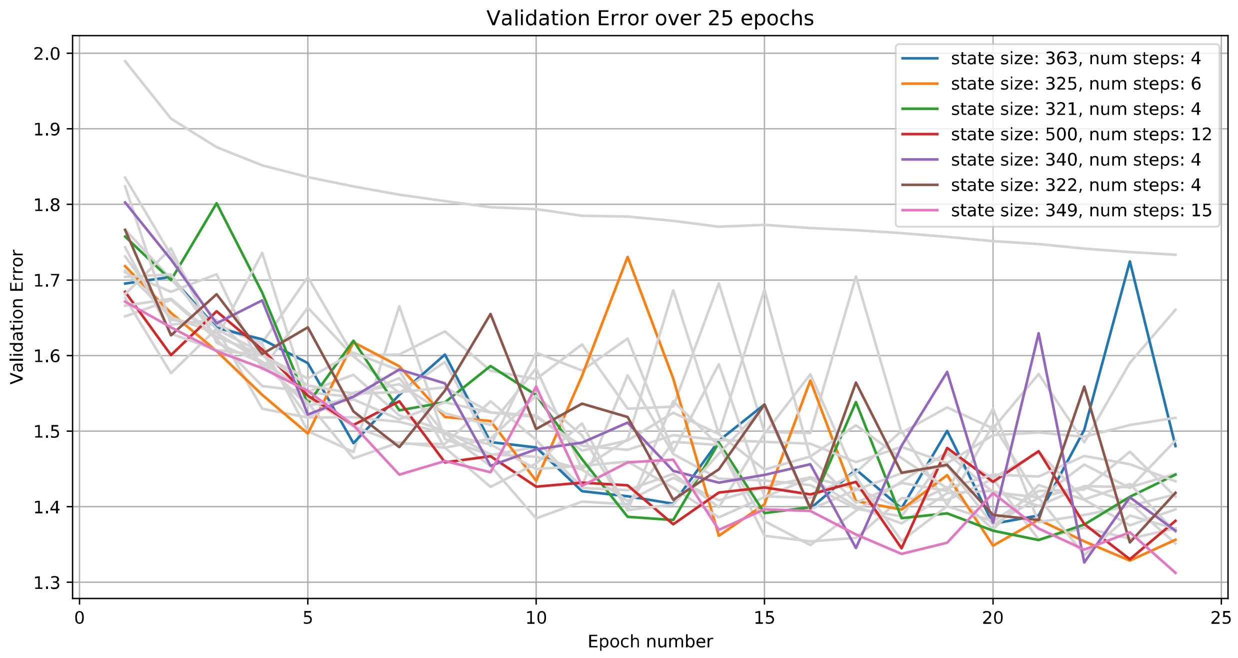
Results



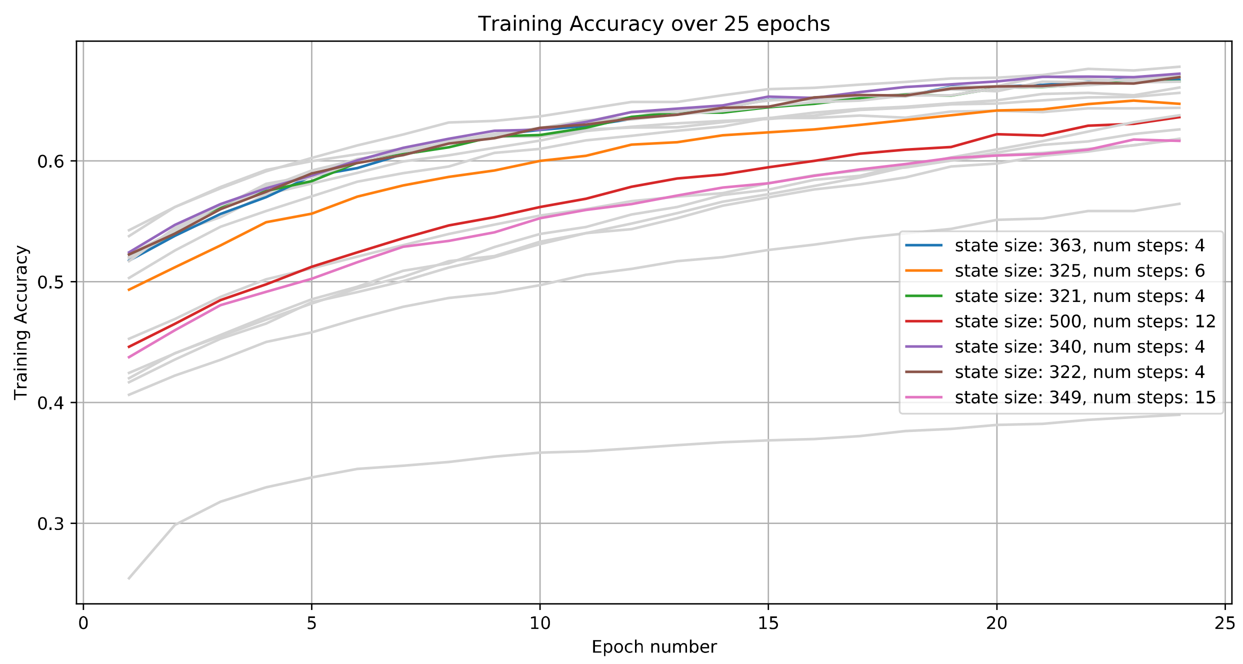
Plot 31: Objective Function output of Bayesian Optimization trying to find the best RNN steps and best State Size for Basic Recurrent Neural Network – Objective Function returns the average of the top two validation accuracies for training of 25 epochs



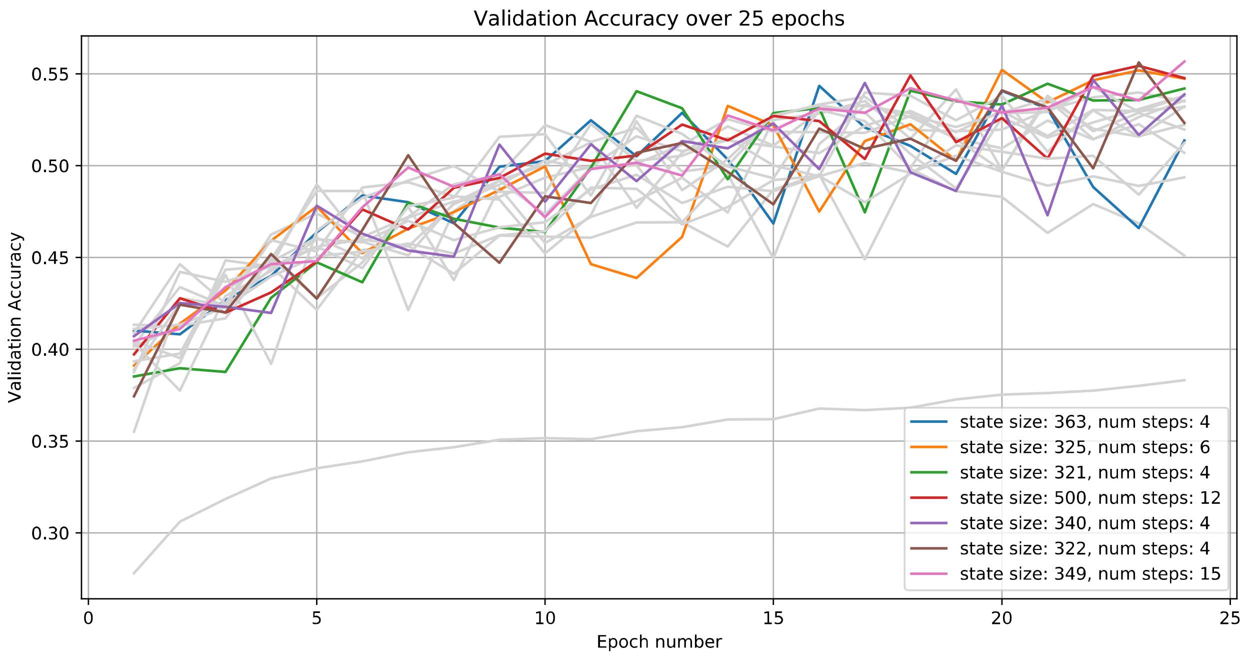
Plot 32: Training Error – Bayesian Optimization trying to find the best RNN steps and best State Size for Basic Recurrent Neural Network – Objective Function returns the average of the top two validation accuracies for training of 25 epochs



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

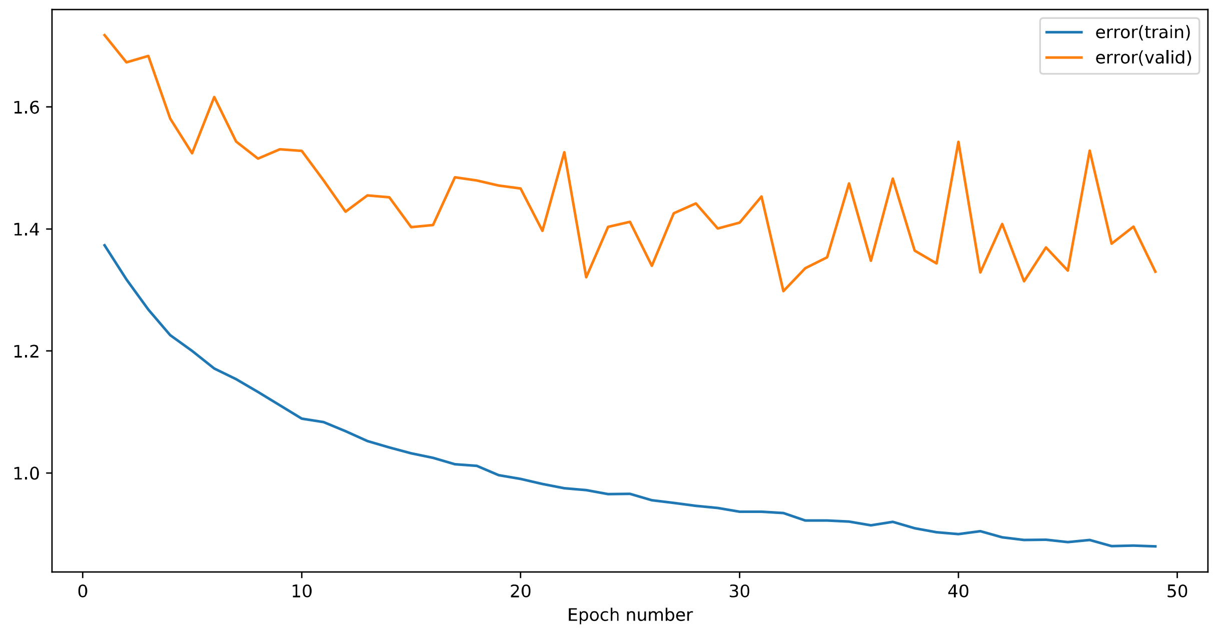
Next step is to take advantage of the best parameters of the Bayesian Optimization and run an experiment for 50 epochs to have the chance and examining the behaviour of our RNN classifier with the best parameters.

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.

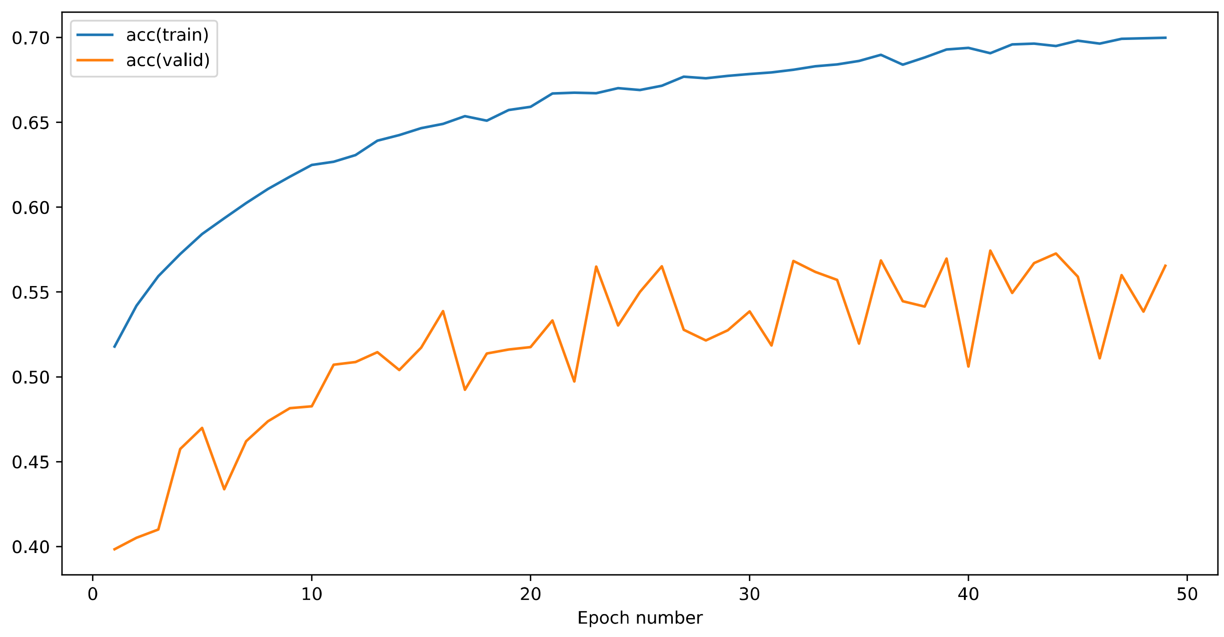
So we execute the experiment according to the following parameters:

- **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.