ECSE 421 Final Project Report

Implementation of a neural network for number recognition

Ben Steel – 260804961 and Nikita Rudin - 260809135, McGill University, April 2018

***Abstract*—This report explains the implementation of a Neural Network on the MyRio Board using Labview. The Neural network is used to recognize numbers drawn in the air while holding the board. It uses only accelerometer data as inputs.**

I. Introduction

This project was part the Course ECSE 421 Introduction to Embedded systems. In previous labs we learned how to train and use a neural network. The purpose of this final project was to apply that knowledge to a new problem. We were asked to generate are own data and not use any external resources. This report retraces the implementation, as such it contains the following sections:

1. Adapting the previously developed Neural Network.
2. Generating the training data
3. Training different version of the network
4. Testing the recognition performance
5. Possible improvements

II. Making a general Neural Network

A. *Reconfigurable network*

The neural network we had previously used in earlier labs was in the 'Position Net' configuration, with 5 inputs, a single hidden layer of 7 nodes, and 3 outputs. As such the function used to perform the neural network was hard coded with this configuration. The first step was to make the network generic, to allow a flexible number of inputs, hidden layers, hidden layer sizes, and outputs.

This was fairly easily achieved by putting the nodes in for loops and using Labview's autoindexing function to output the sigmoid output of each node into a vector.

B. *Reconfigurable training*

The next step was to make the training function generic, again to allow a fully configurable neural network. The weight generation function was easily adapted to create an array of weight matrices for any number of hidden layers. The back-propagation algorithm implementation was also adapted to allow for any number of hidden layers, using for loops and shift registers to pass the delta function between loops.

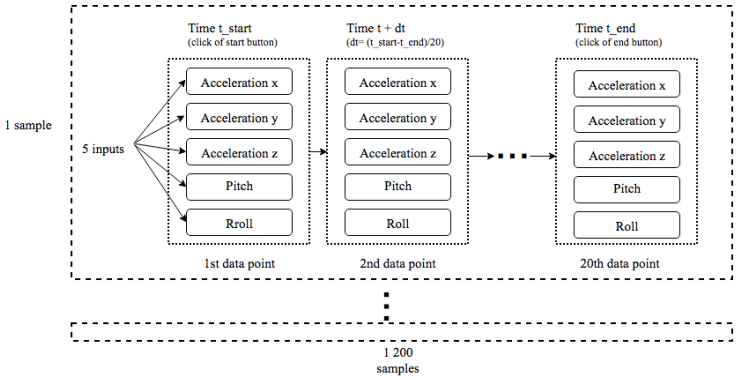
III. Generating the training data

Before generating the data needed for the training we had to decide how our recognition would work and what would be the inputs to the system. In the previous labs we had developed a system that could output the 3 accelerations as well as the pitch and roll of the board. We decided to record all these five outputs in order to have more flexibility in our design choices later on.

An important decision was made about the sampling rate of these outputs. We decided to make this rate flexible. The user tells the system when to begin recording and when to end recording. During that time we record all outputs t a sampling period of 5ms. When the recording is complete we subsample 20 evenly distributed data points. The data is then reshaped as a 1 dimensional, length 100 vector (5x20 cells), and then exported to a csv file and the process repeated with another number.

We have recorded a total of 1200 samples (120 for each number). 1000 were used for the training and the other 200 for validation. The data samples were recorded across in chunks of 100 samples at a time, across multiple days to create more variability in the data.

This relatively large data set allowed us to encompass more possible ways to draw the numbers and results in a final recognition that relies less on a specific pattern. It is worth mentioning that we drew the numbers vertically (in the air) with the board mostly parallel to the table. The complete sampling procedure is shown in the figure below:



This main data set was then copied and modified to create different version of inputs to the training system. We have tried the following modifications:

A. *Reduction of inputs*

Even though we had recorded all three acceleration, pitch and roll, it seemed to us like the pitch and roll should not intervene in the recognition process as the board remains mainly flat. We have therefore created a data set without pitch and roll. The number of inputs was therefore reduced to 60 (3x20).

We then took the same idea further and removed the x-axis acceleration from the dataset. When the board is kept flat and moved vertically, the contribution of the x-axis acceleration should have negligible effect on the recognition of the number drawn. This new dataset now contained 40 inputs per data point. As discussed below this provided the best results.

B. *Data points reduction*

We also tried reducing the amount of data points from 20 to 10 for each sample. After trial and error this didn't seem to improve the performance.

C. *Normalizing the inputs*

We had the issue that all our inputs were very small. This should normally be fixed by larger weights, but since the sigmoid function is nearly constant for very small or very large inputs, we saw that the outputs were stuck in these "tails" of the sigmoid. We therefore decided to divide all of our data by the overall average (this was only possible for the data set that contained only accelerations). This resulted in multiplying all inputs by 10 000. This proved to be a good solution.

IV. Training different versions of the network

Training of the network required a lot of trial and error. We have experimented with the following variables: versions of the dataset, number of hidden layers, size of hidden layers, batch size, learning rate and finally weights initialization.

Most unsuccessful configurations lead to the error not going below 1 and all outputs going to zero. This can be understood by the fact that when all outputs are zero the error will always be 1 because of the one hot encoding, and the network has reached a local minimum from which many configurations couldn't get out of with a low learning rate.

A. *Dataset*

Our best results were obtained with smallest number of inputs, that is y and z accelerations. This is most probably due to the fact that a simpler system results in an easier and faster training.

B. *Hidden layers*

This choice was totally arbitrary and changed a lot during the trial and error period. In the end we chose to have 2 hidden layer of 15 nodes each. However we experimented with 1 and 3 hidden layers, and with hidden layer sizes of up to 300. These larger neural network sizes could have produced more flexibility, but led to very large training iteration times, resulting in incredibly slow convergence times.

C. *Batch size*

Even though we have tried batch size ranging from 1 to 20, the time needed for each iteration increases drastically with the batch size. We chose once again to go with the simplest solution and kept the batch size at 1.

Since it can be adjusted without stopping the training, we have tried increasing the batch size towards the end of the training in order to get less noisy weights updates.

D. *Learning rate*

The learning rate was adjusted by looking at the weights update. If it is too large the weights never converge because each step is too large and is not representative of the gradient at that position, and the weights solution will never reach the bottom of the minimum. If it is too small the training requires many more steps before convergence and as such much more time. We wanted the weights of the output layer to only change their 2nd or 3rd digit at each step. This was achieved with a learning rate of 0.1. However we found it very useful to use a very large learning rate in the event that the network had reached a clear local minimum, to bump the weights solution into another, potential global minimum.

Since it can also be adjusted live during training, we tried decreasing the learning rate at the end to get closer to the minimum of the error.

E. *Weights initialization*

We have already discussed the issue of having very small all very large input to the sigmoid function. For the 1st layer this was solved by adjust the values of the inputs. We had the opposite problem for other layers. With weights between -1 and 1 the weighted sum at each node could be too large and therefore we were once again stuck in the "tail" of the sigmoid function. There is a formula recommended by [1] for the initialization of weights:

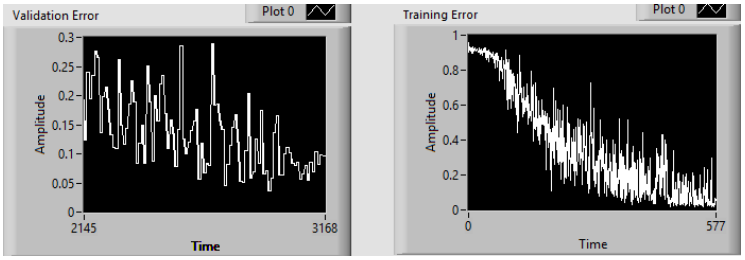
$$W_0 \in [-\sqrt{\frac{2}{N}}; \sqrt{\frac{2}{N}}]$$

Where N is the number of inputs to the particular node. N is respectively 40, 15 and 15 for our 3 layers.

Once we used this initialization together with the normalized data, we never encountered the issue of stuck outputs.

F. *Best training results*

Using the described configuration, our network was left to train for about 15min. After around 5770 training iteration (the plot values are averaged over 10 loops) the validation error dropped to 0.07.



V. Recognition Performance

Using the provided weights our network is able to consistently recognize all 10 numbers. We need to mention that all of our training data was generated by the same person. As such there is a special drawing pattern which is best recognized by the system. For the person who generated the data the rate of success is around 90%, while for the other member of the group that rate is closer to 60 %. Knowing this it would of course make sense to involve more people in the generation of training data to increase the variability.

The output of the network is an array of 10 numbers between 0 and 1. The index of the maximum of that array is interpreted as the recognized number. We also interpret the value of the maximum as the confidence of the recognition. This is of course a simplification and "confidence" may not be the best term, but this value gives a feeling of how much more weight is given to this number compared to others. Usually this value is between 80% and 95%. In the rare cases when the system is mistaken, the confidence value drops to 30% - 60%.

VI. Possible improvements

A. *More Variation in the training data*

As mentioned before, an important improvement would be to introduce more variation in the training data. This can be done by having different people generating this data and by drawing the numbers with more varying patterns.

B. *Train on a PC*

The final network must of course run on the MyRio board, but nothing forces our training to run on the same device. It would be much faster to train on a powerful computer. We have tried to implement our code directly on the lab computer, but we didn't manage to make it work without the board. More work on this side can allow much faster training.

C. *Use another loss function*

A potential alternate choice in the solution space would be to choose another loss function. The cross-entropy loss function, combined with a 'softmax' output (Where all outputs are normalised by the sum of the other outputs, resulting in a probability distribution output), seems to be best practice in industry for a multi-class classification neural network.

VII. Conclusion

In conclusion, a neural network was successfully implemented. The final system is able to accurately recognize all 10 numbers with a great accuracy. This was achieved by a long trial and error period. Multiple configurations were tested. This was made possible by our design of a flexible neural network and training algorithm. Throughout the final labs and this project, we learned the basics of neural networks and ha the chance to apply this knowledge to a practical example. It is great to see that the relatively simple concept we learned, could be applied to the nontrivial case of number recognition described in this report.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification" eprint arXiv:1502.01852, 02/2015