Roman Grigorii

HW 2 write-up

Problem 1.

My learning aim is to find the robot states which attribute to the largest amount of noise in the data and use this information to ignore some of the measurements that are taken when the robot Is in, or close to, these set of states. My assumption initially was that if the robot is turning, for example, it's position measurements during this time will not be as accurate as they would be when the robot is moving slower and especially not as accurate as they would be when it is sitting still. My speculation is confirmed by some of the initial analysis of the data, the accuracy of measurements does indeed depend on the angular velocity, however I found that it is for times when the angular velocity is 0 that there is most error! (data from ds1).

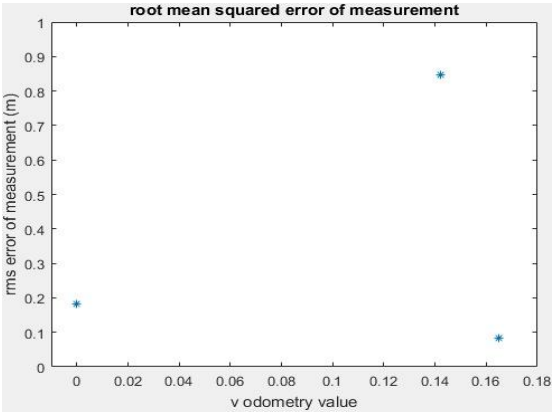

Figure 2 rms error as function of odometer translational velocity values

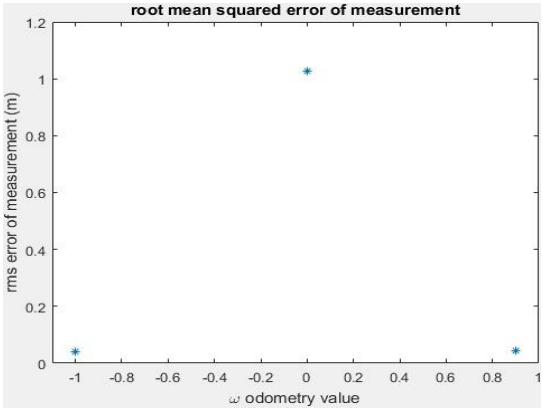

Figure 1 rms error as function of odometer angular velocity values

From Figure 1 and 2 one can see that the error in Position estimation is a function of odometer readings. A further analysis into the nature of errors has shown that the errors in measurements some odometer readings are not perfectly Gaussian in nature. Figures 3 and 4 show that large discrepancies in position measurements occur for very specific odometer values. Figure 4 points to the fact that when the robot is completely stationary, it tends to make very large errors in measurement.

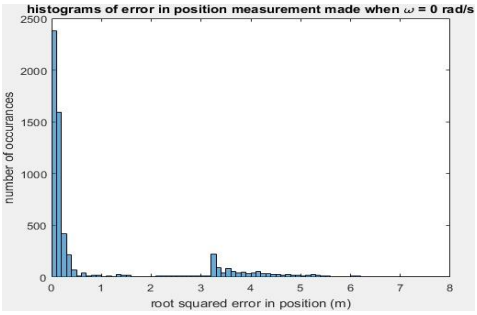

Figure 3 histogram of error in position measurements made when ω=0

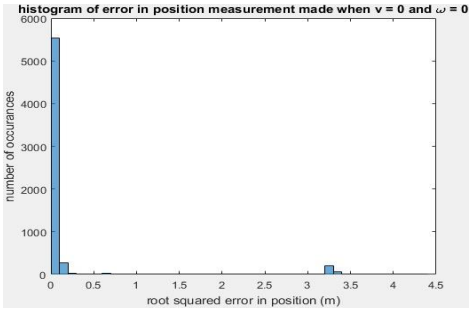

Figure 4 histogram of error in position measurements made when ω=0 and v = 0

I will train a NN model that will estimate a positon error (through classification) based on the odometer readings. Because NN models can have multi-dimensional inputs, I can feed both angular and translational velocity readings to get an approximated error I should expect with those values.

Problem 2.

Operation of the algorithm

The core of Neural Network algorithm is not very difficult to understand. The basic idea is that one takes inputs along with their corresponding labels and cross correlates the entries of each by building a map of intermediate activation functions that best reflect a trend in the data. To start breaking down how these equations are formed, it is helpful to look at the following image of a simple Neural Nets model:

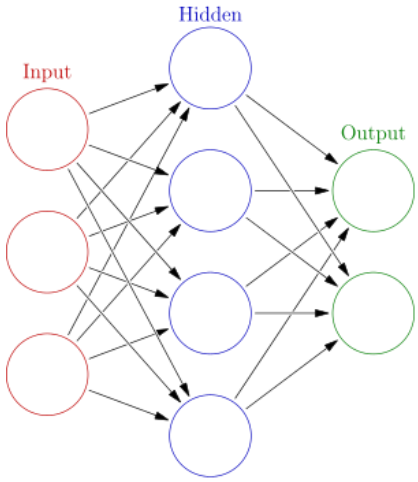

Figure 5 A simple Neural Network maphttps://upload.wikimedia.org/wikipedia/commons/thumb/4/46/Colored_neural_network.svg/300px-Colored_neural_network.svg.png

Three entries in the input level are used to set four activation functions of the single hidden layer. In other words, each of the three inputs is cross correlated in a linear summation and passed through an activation function to set the values of the next layer. There are many activation functions that can be used, but one of the most common ones is sigmoid which is of the form:

$$f = \frac{1}{1 + e^{-x}}$$

In the above equation x refers to a linear combination of the entries in the layer previous to the one that the activation function is setting values on. To understand the math a little better, I break down the mathematics that occur during the population of the nodes in the image above.

The formulation of the activation functions of each layer are as follows:

$f^1 = \{x_1, x_2, x_3\}$: The activation functions in the first layer are set to be the inputs. Note: this reflects the model posed above only which has 3 inputs. The input layer can have only a single value or multiple values.

$f^l = \left(1 + e^{-u_j^l}\right)^{-1}$ where $u_j^l = \sum_i w_{ji}^l f_i^{l-1} + b_j^l$

for $i = 1 \dots length(layer\ l-1)\ \ j = 1 \dots length(layer\ l)$

Every layer l, other than the first, is populated with an activation function evaluated by the above formula, where w_{ji} are the weights that set the significance of the influence of preceding layer on the next and b_j are the offsets between values in one layer and values in the next. In effect, every entry in some layer is composed of linear combination of every entry from the previous layer. The last layer, referred from here on as the output layer, will be composed of forwarded propagated activation functions. In my code, the initial weights and offsets have been set with random values that match the range of values of the output. For example, if the labels are in the binary range, my weights and offsets at each layer are initially set to be random values between 0 and 1.

Once the inputs have been forward propagated to obtain a set of values at the output layer, a cost function (C) is evaluated that associates the desired labels and the actual values obtained at the last layer of the map through the evaluation of the activation function.

$C = \frac{1}{2}\sum_i \left(y_i - f_i^L\right)^2$ for $i = 1 \dots length(layer\ L)$: where f^L is a set of activation values computed from forward propagation and y is the set of outputs, or in other words the labels placed on the input values. The goal of course is to minimize the cost function, which is done by changing the weights and the offset values of all the previous layers. This process begins at the last layer and works back to the first, which is why this procedure is called backpropagation. We work to minimize C with respect to w and b by computing its gradient starting at the last layer and updating the weight and offset values in a way that reduces the cost. The formulation of the gradients is shown below:

$$\frac{\partial C}{\partial b^l} = \frac{\partial C}{\partial u^l}\frac{\partial u^l}{\partial b^l} = \frac{\partial C}{\partial u^l} = \frac{\partial C}{\partial f^l}\frac{\partial f^l}{\partial u^l}\ for\ (layers)\ l = 1 \dots L$$

$$\frac{\partial C}{\partial b^L} = \frac{\partial C}{\partial f^L}\frac{\partial f^L}{\partial u^L} = \frac{1}{2}\sum_i \frac{\partial C}{\partial f_i^L}\left(y_i - f_i^L\right)^2 \frac{\partial f^L}{\partial u^L} = \sum_i (f_i^L - y_i)\frac{\partial f^L}{\partial u^L}\ where\ L\ is\ output\ layer$$

$\frac{\partial f^L}{\partial u^L}$ is actually very easy to compute thanks to the nature of the activation function I have used.

$$\frac{\partial f^L}{\partial u^L} = \frac{\partial}{\partial u^L}\left(1 + e^{-u^L}\right)^{-1} = \frac{\left(1 - \left(1 + e^{-u^L}\right)^{-1}\right)}{1 + e^{-u^L}} = f^L(1 - f^L)$$

$$(1) \qquad \frac{\partial C}{\partial b^L} = f^L(1 - f^L)\sum_i (f_i^L - y_i) \quad where\ i = 1..length(layer\ L)$$

Note that $\frac{\partial C}{\partial b^L} \neq \frac{\partial C}{\partial b^l}\ for\ l = 1..L-1$ because the formulation of the cost function hinges on the computed difference between projected and desired labels on the last layer. This makes it straightforward to compute at the output layer but doesn't allow for a trivial computation in the intermediate layers. I begin by starting in some layer that is not output layer, and express the gradient of the cost with respect to the offset in terms of the gradient computed at the latter layer. This offers a cleaner solution, and a makes more algorithmic sense because of the chronology of gradient computation (back to front).

$$\frac{\partial C}{\partial b^l} = \frac{\partial C}{\partial u^l} = \frac{\partial C}{\partial f^L}\frac{\partial f^L}{\partial u^l} = \sum_i \left(f_i^L - y_i\right)\frac{\partial f^L}{\partial u^L}\frac{\partial u^L}{\partial u^{L-1}} \dots \frac{\partial u^{l+1}}{\partial u^l}$$

I plug in the following equations:

$$\frac{\partial f^L}{\partial u^L} = f^L(1 - f^L)$$

$$\frac{\partial u^l}{\partial u^{l-1}} = \frac{\partial}{\partial u^{l-1}}\left(\sum_i w_{ji}^L \frac{1}{1+e^{u^{l-1}}} + b_j^l\right) = \sum_i \frac{\partial}{\partial u^{l-1}}\left(w_{ji}^l \frac{1}{1+e^{u^{l-1}}}\right) = \sum_i w_i^l f^{l-1}(1 - f^{l-1})$$

To obtain:

$$\frac{\partial C}{\partial b^l} = \sum_i \left(f_i^L - y_i\right) f^L(1 - f^L)\sum_i w_i^{L-1} f^{L-1}(1 - f^{L-1}) \quad \dots \quad f^{l+1}\left(1 - f^{l+1}\right)\sum_h w_h^l f^l(1 - f^l)$$

Which in rewritten form becomes:

(2) $$\frac{\partial C}{\partial b_k^l} = \frac{\partial C}{\partial b^{l+1}}\sum_h w_{kh}^l f^l(1 - f^l) \; for\; h = 1 \dots length(layer\; l+1)$$

Once we find the gradient at the output layer, we can work back to find the gradient of cost at every single layer working backwards through the NN map. The same tactic applies for computing the gradient with respect to the weights at each of the layers. I will express two equations, as I did for the offsets, that allow for finding the complete set of gradients at every layer. One thing to note is that I included the subscripts in my derivation for the second set of gradients because there are two sets of subscripts on the weights, and it is important to keep track of which subscripts the sum is being taken over.

I begin at the output layer:

$$\frac{\partial C}{\partial w_{ij}^L} = \frac{\partial}{\partial w_{ij}^L}\left(\frac{1}{2}\sum_i \left(y_i - f_i^L\right)^2\right) = \sum_i \left(\left(y_i - f_i^L\right)\frac{\partial f_i^L}{\partial u_i^L}\frac{\partial u_i^L}{\partial w_{ij}^L}\right) = \sum_i \left(\left(y_i - f_i^L\right)f_i^L(1 - f_i^L)\frac{\partial u_i^L}{\partial w_{ij}^L}\right)$$

(3) $$\frac{\partial C}{\partial w_{ij}^L} = \sum_i \left(\left(y_i - f_i^L\right)f_i^L(1 - f_i^L)\right)\sum_j f_j^{L-1}$$

Now I find the general gradient for all other layers.

$$\frac{\partial C}{\partial w_{ko}^l} = \frac{\partial u_k^l}{\partial w_{ko}^l}\frac{\partial u_o^{l+1}}{\partial u_k^l} \dots \frac{\partial u_j^{L-1}}{\partial u_v^{L-2}}\frac{\partial u_i^L}{\partial u_j^{L-1}}\frac{\partial w_{ij}^L}{\partial u_i^L}\frac{\partial C}{\partial w_{ij}^L}$$

$$= \sum_o f_o^{l-1}\sum_o w_{ko} f^l(1 - f^l) \dots \sum_i w_{ji} f^{L-1}(1 - f^{L-1})\left(\sum_j f_j^{L-1}\right)^{-1}\frac{\partial C}{\partial w_{ij}^L}$$

We can once again rewrite this complex equation in terms of the gradient of the cost at the layer following the one in question, as we did for the gradient of offset variable b.

(4) $$\frac{\partial C}{\partial w_{ko}^l} = \sum_o f_o^{l-1}\sum_k w_{ko} f^l(1 - f^l)\frac{\partial C}{\partial w_{nk}^{l+1}}$$

Equations 1-4 constitute the basis of the mathematics behind the formulation of backpropagation. At every layer during the back propagation the gradient are found, and the weights along with the offsets are updated using steepest descent method.

$$w^l \to w^l - \gamma \frac{\partial C}{\partial w^l}$$

$$b^l \to b^l - \gamma \frac{\partial C}{\partial b^l}$$

There are a lot of ways to set γ to a value that will steer the algorithm away from reaching an unstable point as well as ensuring the descent occurs rapidly enough. In my algorithm I have set

$\gamma = \frac{number\ of\ layers}{number\ of\ inputs}$. The suggestion to use the number of layers and number of inputs was given in a few papers I have read. It turned out that depending on the problem I typically needed to scale γ by multiplying it by another number typically $\ll 1$. In other words, this parameter was tunable.

Having written equations for backpropagation, the overall shape of my algorithm takes the following form:

Let:

I = input set

O = output set.

N = number of backpropagation epochs

K = number of backpropagation loops per input

$w_{ij}^l = rand(\max(label), \min(label))$ where rand gives a random value that falls between two given values $for\ l = 2 \dots L, i = 1 \dots size\ of\ current\ layer\ j = 1 \dots size\ of\ next\ layer$

$b_i^l = rand(\max(label), \min(label))$

```
for n = 1 to N
        for k = 1 to K
                forward propagate input I(k)
                for L to 1
                        compute gradients using equations 1-4
                        apply computed gradient to current weights and offsets using steepest descent
                end
        end
in a stochastic NN approach I slightly offset the computed weights and offsets here.
end
```

Why I chose NN algorithm to achieve my goal

1. NN algorithms are known to produce great results in noisy environment.
2. NN algorithms can take inputs of multiple dimensions and be able to correlate them to a smaller or larger output dimension space.

3. NN algorithms cross correlate the inputs which allows for hidden trends that are not easily observable first hand.
4. NN build a model from which parameters can be extracted and used to classify data very quickly at run time. Unlike some algorithms, having built a model the data on which is has been trained can be discarded from robot memory.

Due to this reasons NN algorithm is a perfect candidate to use on my data set. To reiterate, I hope for my model to be able to predict the validity of the measurements the robot makes based on its states. Because I am setting multiple states as inputs, I have inherent noise in the data, I do not possess full information of what attributes to poor measurements (although I have an idea) I think NN algorithm will be able to do the job best.

Demonstration of NN operation on test sets

Test set 1: Binary valued inputs that have been XORed in some fashion to produce the output.

Example Training set:

Test input						Test Output
1	0	0	0	0	0	1
0	1	0	0	0	0	1
1	1	0	0	0	0	0
0	0	1	0	0	0	1
1	0	1	0	0	0	0
1	1	1	0	0	0	1
0	0	0	1	0	0	1
0	1	0	1	0	0	0
1	1	0	1	0	0	1
0	0	1	1	0	0	0
0	1	1	1	0	0	1
1	1	1	1	0	0	0
0	0	0	0	1	0	0
1	0	0	0	1	0	1
0	1	0	0	1	0	1
1	1	0	0	1	0	0
0	0	1	0	1	0	1
1	0	1	0	1	0	0
0	1	1	0	1	0	0
1	1	1	0	1	0	1
0	0	0	1	1	0	1
1	0	0	1	1	0	0
0	1	0	1	1	0	0
0	0	1	1	1	0	0
1	0	1	1	1	0	1
0	1	1	1	1	0	1
1	1	1	1	1	0	0
0	0	0	0	0	1	0
1	0	0	0	0	1	1
0	1	0	0	0	1	1
0	0	1	0	0	1	1
1	0	1	0	0	1	0
0	1	1	0	0	1	0
1	1	1	0	0	1	1
0	0	0	1	0	1	1
1	0	0	1	0	1	0
0	1	0	1	0	1	0
1	1	0	1	0	1	1
1	0	1	1	0	1	1
1	1	1	1	0	1	0
0	0	0	0	1	1	0
0	1	0	0	1	1	1
1	1	0	0	1	1	0
0	0	1	0	1	1	1
1	0	1	0	1	1	0
0	1	1	0	1	1	0
1	1	1	0	1	1	1
0	0	0	1	1	1	1
1	0	0	1	1	1	0
0	1	0	1	1	1	0
1	1	0	1	1	1	1
1	0	1	1	1	1	1
0	1	1	1	1	1	1
1	1	1	1	1	1	0

About the test input:

The test input does not have repeating entries. Each input entry is 6 dimensional, meaning that the size of the first layer in the NN tree is 6. The labels are 1 dimensional entry, so the output layer has a length of one. Historically, machine learning algorithms would fail at picking up on the XOR pattern that existed in the data. The interest in NN in the 70s died down for about a decade before the algorithm gained back its popularity due to being reformulated to account for this drawback. Since then a good first test for the correctness of NN implementation has been to test it on the data that has been processed using the XOR function to compute the labels. I computed by output using the following formula:

$$Output = XOR\left(input\ 6, XOR\left(input\ 5\ , XOR\left(input\ 4\ , XOR\left(input\ 3, XOR\left(input\ 2, XOR\ (input\ 1, input\ 5)\right)\right)\right)\right)\right)$$

Where input 1 corresponds to the leftmost entry in each row, and input 6 corresponds to the rightmost entry.

If my NN algorithm passes this first test it means it can identify patterns of simple logic in the data.

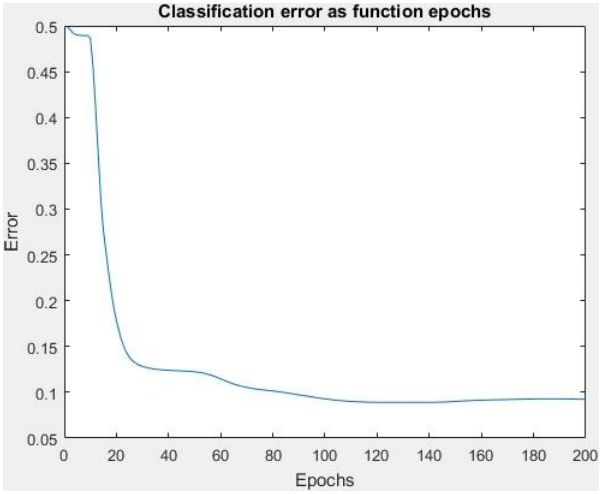

Figure 6 Classification error versus the number of epochs the NN algorithms goes through to settle on its weights and offsets.

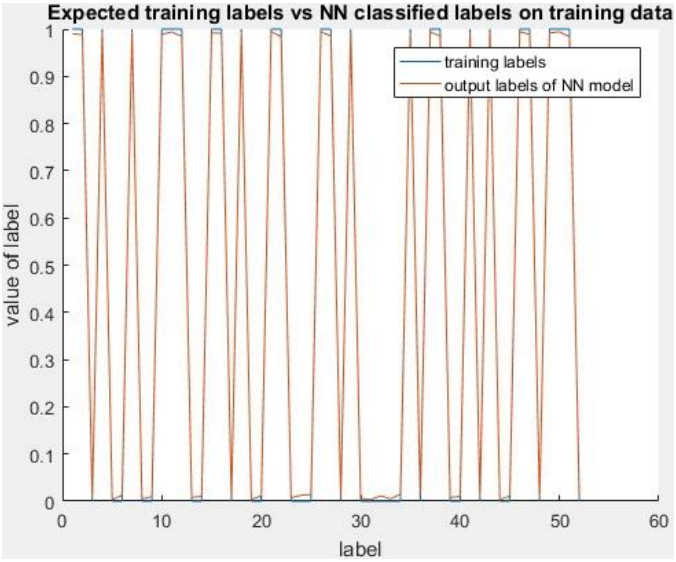

Figure 7 The above plot shows how closely NN algorithm predicts the output on the test labels.

The figures above indicate the operation of the algorithm. When I mention epochs, I am referring to the number of times the model has been back propagated. On Figure 7 we can see that the algorithm makes good classifications to the training data. In this particular problem, I rounded the output labels as the final step which is not part of my NN code. I did this because I did not account for strictly binary outputs in my NN code and decided it would have been outside of the scope of what is important for the assignment, since there are special NN algorithm implementations that do just this. Rounding the output labels has given very promising results – from figure 7 one can see that rounded results bring us to classification accuracy of 100% on training data.

Next I took a look at how well my models perform in classifying the data that was not in the training data set. From Figure 8 one can see that the NN model build performs very well on this validation set. I

build 100 NN models to classify this type of data, and I found that the classification on validation data is with my NN model I at **94%** accuracy, after I have rounded the determined labels to make them binary. This NN model was built with a hidden layer of 4 units in length. It appeared that this number performed very well and also allowed for quick learning.

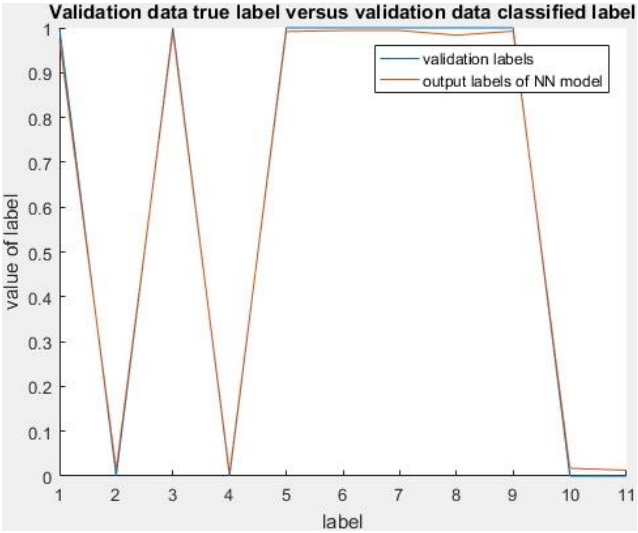

Figure 8 Validation data being classified using the build NN model

I conclude that the first test has successfully passed, and I move on to the second.

Test set 2

My second set of test data is a sinusoidal function f = ½(1+sin(x)) for x = 0 to x = 10, as visualized in

figure 9. The output was calculated by setting the labels to 1 when the input is in the range .5 to .9, and is 0 elsewhere. Figure 9 shows input and output value along with the classification labeling done by the build model on the training data. Note that the classified labels reached the desired with increased number of epochs. The plotted classification result has been generated after only a few epochs (~1 min operation).

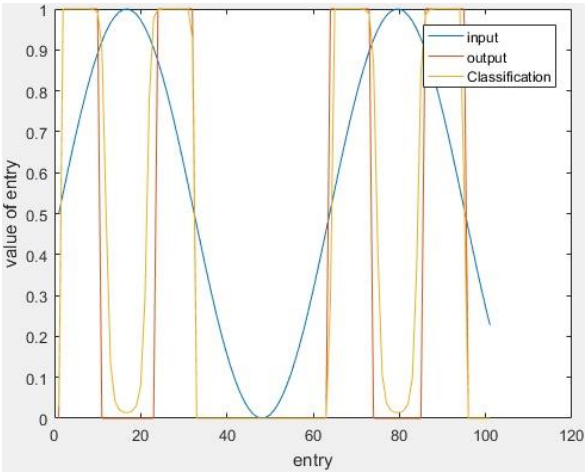

Figure 9 Training input, output, and classification of the built model. Note that no noise has been added to the input data

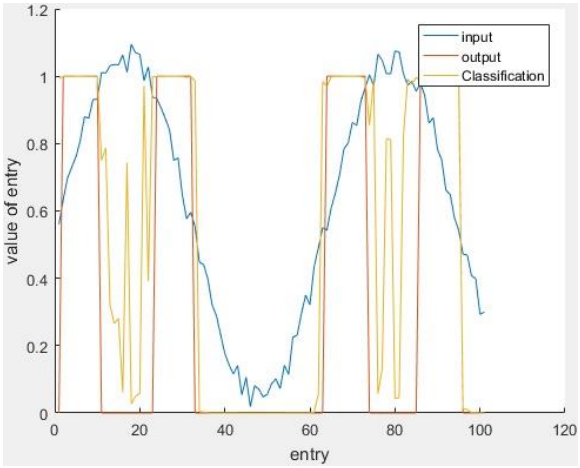

Figure 10 Training input, output, and classification of the built model. Noise has been added to the input.

In Figure 10 one can see how well the NN model classifies input data that is without noise. Figure 10 represents an example of classification with added noise. Both models are built with a single hidden layer of size 10. The algorithm went through 400 back propagations (epochs). When the training data has no noise, the model classifies data with added noise with **93% accuracy**. When the noise is added to the training data, the model classifies validation data with **84% accuracy**. In every instance the added noise was Gaussian distributed with amplitude of .1.

Part A conclusions

The NN algorithm I have implemented is robust to noise and can detect trends of complex logic in the data. I believe that because of the nature of my problem this is a great, if not the best, algorithm to use out of the ones suggested.