# Feed Forward Network

Introduction to neural networks - AS 04

By: Nadav Porat

# Notes

- The Backpropagation algorithm I used was derived from page #120 in “Introduction to neural computation” by Jhon Hertz.
- I added the biases as separate neurons valued (+1) in each layer (excluding the output layer).
- The class receives the sizes of the layers as a list: $[N_0, \dots, N_n]$ and optionally a step size (default is $\eta = 0.3$), then creates a FeedForward Network based on those.

# Notes



of the $i$th unit in the $m$th layer. $V_i^0$ will be a synonym for $\xi_i$, the $i$th input. Note that superscript $m$'s label layers, not patterns. We let $w_{ij}^m$ mean the connection from $V_j^{m-1}$ to $V_i^m$. Then the back-propagation procedure is:

1. Initialize the weights to small random values.
2. Choose a pattern $\xi_k^\mu$ and apply it to the input layer ($m = 0$) so that

$$V_k^0 = \xi_k^\mu \qquad \text{for all } k. \tag{6.15}$$

3. Propagate the signal forwards through the network using

$$V_i^m = g(h_i^m) = g\Big(\sum_j w_{ij}^m V_j^{m-1}\Big) \tag{6.16}$$

for each $i$ and $m$ until the final outputs $V_i^M$ have all been calculated.

4. Compute the deltas for the output layer

$$\delta_i^M = g'(h_i^M)[\zeta_i^\mu - V_i^M] \tag{6.17}$$

by comparing the actual outputs $V_i^M$ with the desired ones $\zeta_i^\mu$ for the pattern $\mu$ being considered.

5. Compute the deltas for the preceding layers by propagating the errors backwards

$$\delta_i^{m-1} = g'(h_i^{m-1}) \sum_j w_{ji}^m \delta_j^m \tag{6.18}$$

for $m = M, M-1, \ldots, 2$ until a delta has been calculated for every unit.

6. Use

$$\Delta w_{ij}^m = \eta \delta_i^m V_j^{m-1} \tag{6.19}$$

to update all connections according to $w_{ij}^{\text{new}} = w_{ij}^{\text{old}} + \Delta w_{ij}$.

7. Go back to step 2 and repeat for the next pattern.

It is straightforward to generalize back-propagation to other kinds of networks where connections jump over one or more layers, such as the direct input-to-output connections in Fig. 6.5(b). This produces the same kind of error propagation scheme as long as the network is *feed-forward*, without any backward or lateral connections.

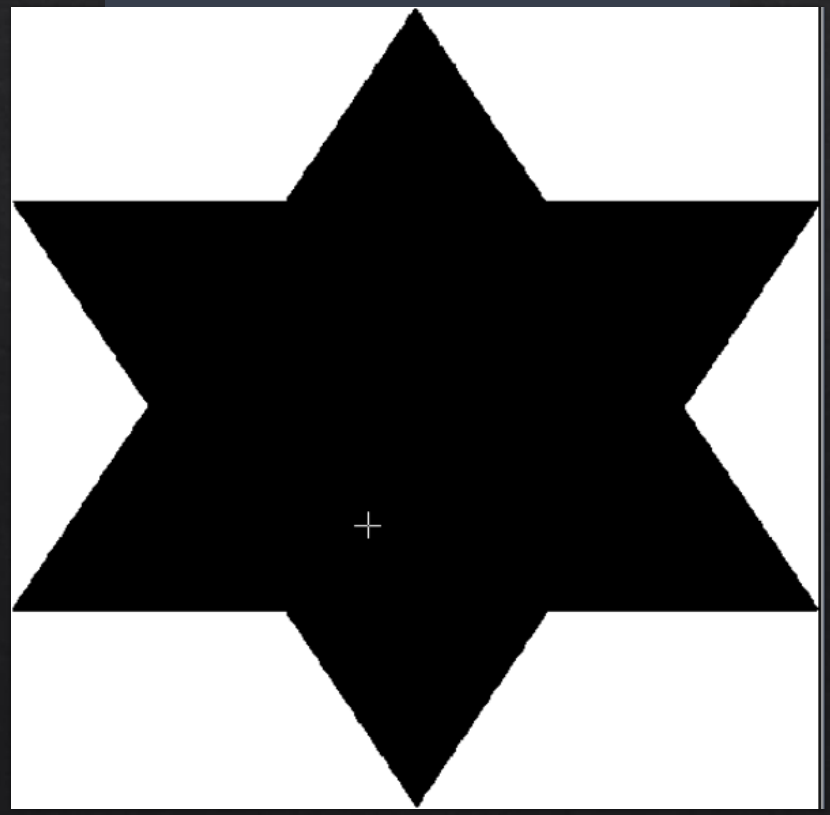
## 6.2 Variations on Back-Propagation

Back-propagation has been much studied in the past few years, and many extensions and modifications have been considered. The basic algorithm given above is exceedingly slow to converge in a multi-layer network, and many variations have

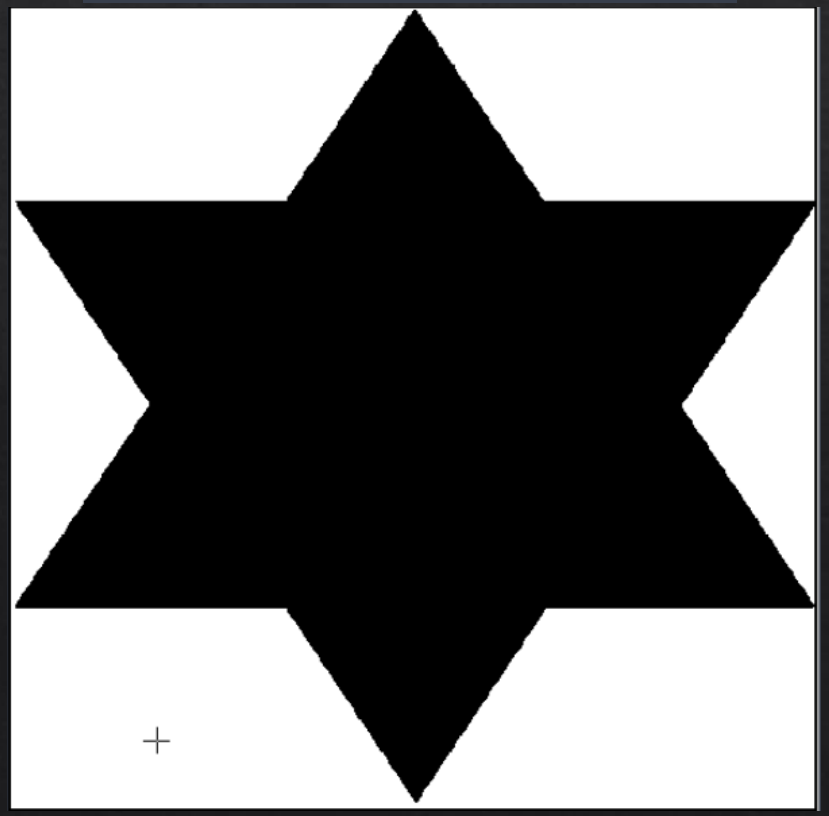
# Perceptron Vs Multi-Layers

- The multi-layered net used is built form the following layers [2, 6, 1], the perceptron is [2, 1].
- The networks are trained to produce (+1) if the point is in the star and (0) if it is outside.

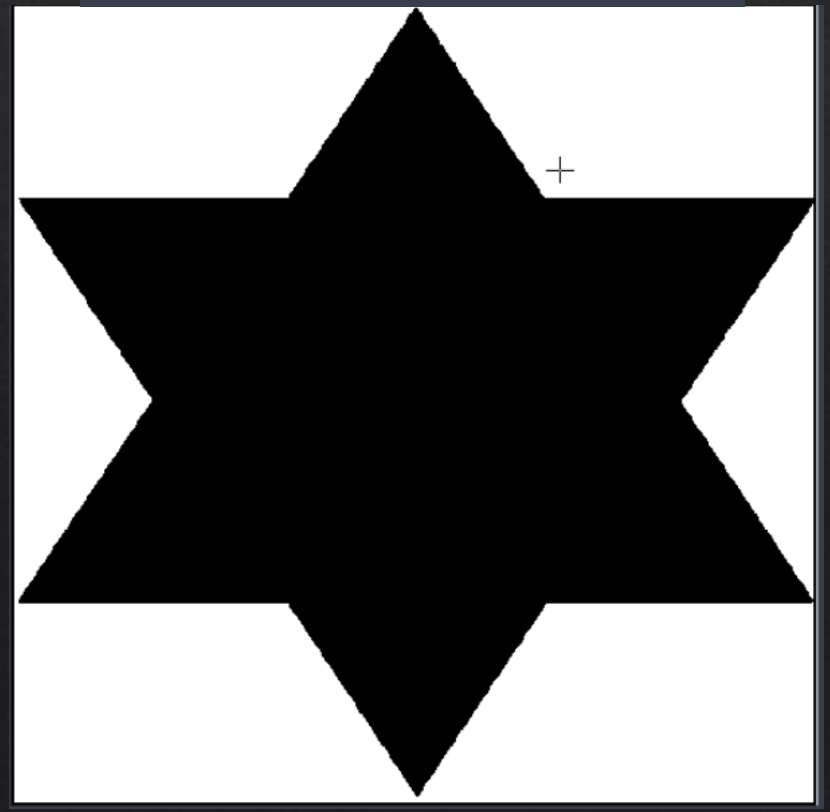
```
Training Iterations: 100000
Precptron answer: 0.6621355864282868
Multi-layered net answer: 0.9969087457324958
```



```
Training Iterations: 100000
Precptron answer: 0.6638671902337844
Multi-layered net answer: 0.08947038581709348
```

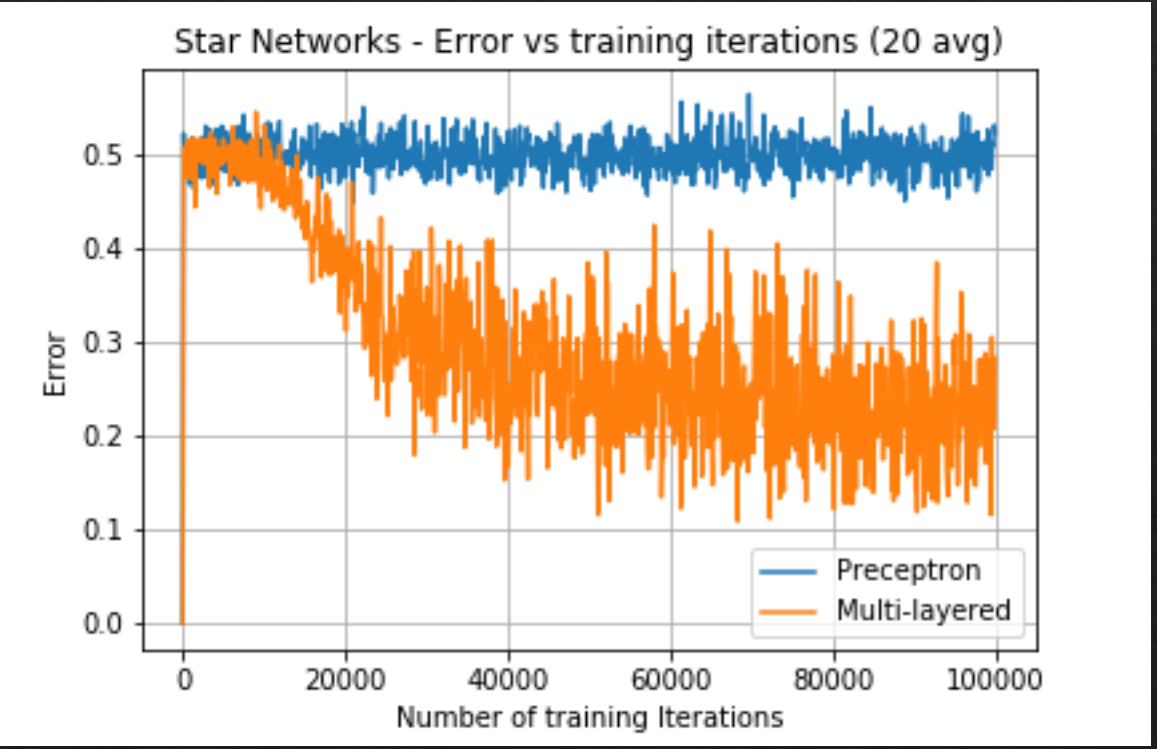


```
Training Iterations: 100000
Precptron answer: 0.6499682232885629
Multi-layered net answer: 0.617654587035361
```



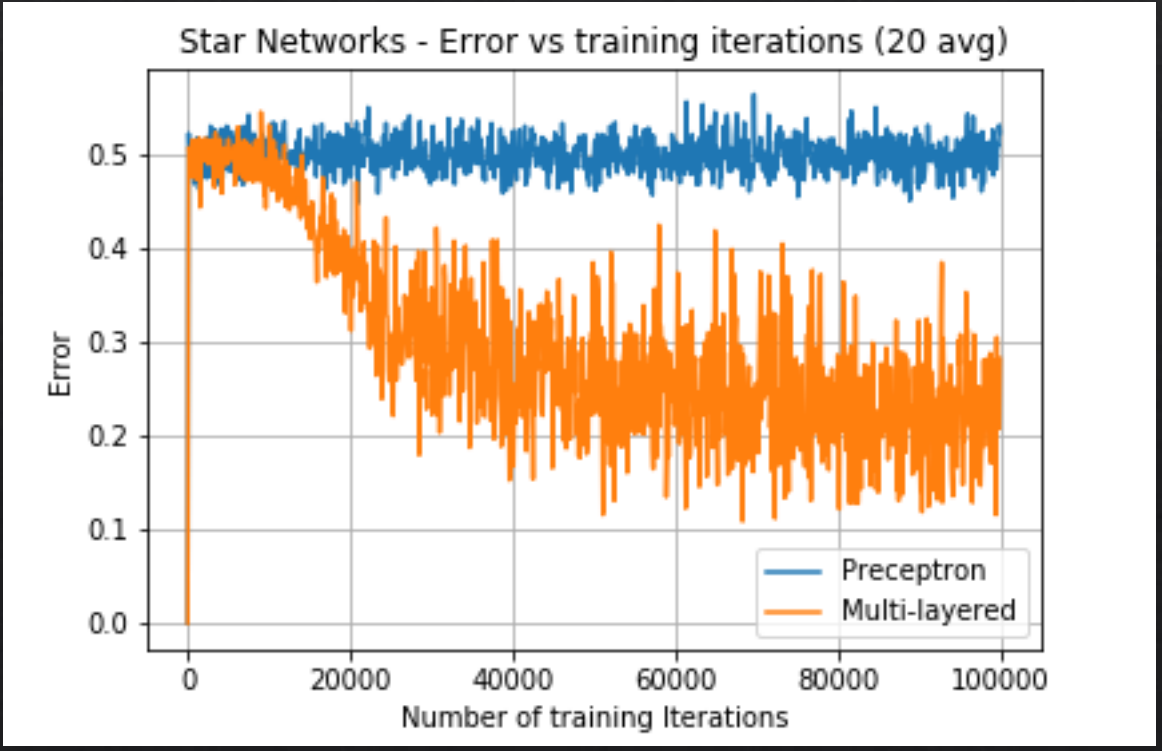
# Perceptron Vs Multi-Layers

In the graph below we can see the Perceptron and the Multi-Layered net errors as functions of their training/learning.
Between 100 training-iterations leaps, a random point was selected on the star picture and then served as an input to both the perceptron and the layered networks. Then graphed the error: $|output - expected|$. Finally, I **averaged 20 graphs** into the following one:



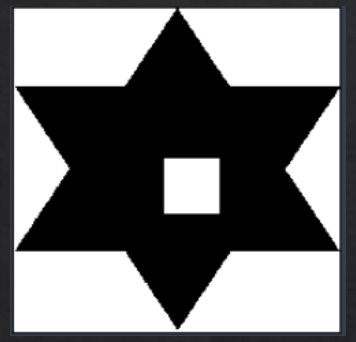
# Perceptron Vs Multi-Layers

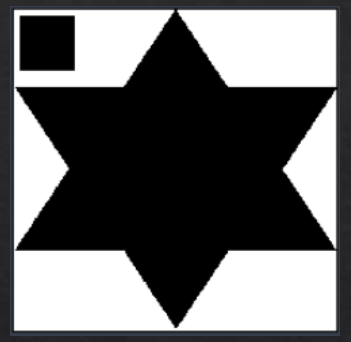
We can see that the Perceptron is basically unchanging at 50% error, and the Multi-Layered network is “learning” and stabilize at ~25% error.

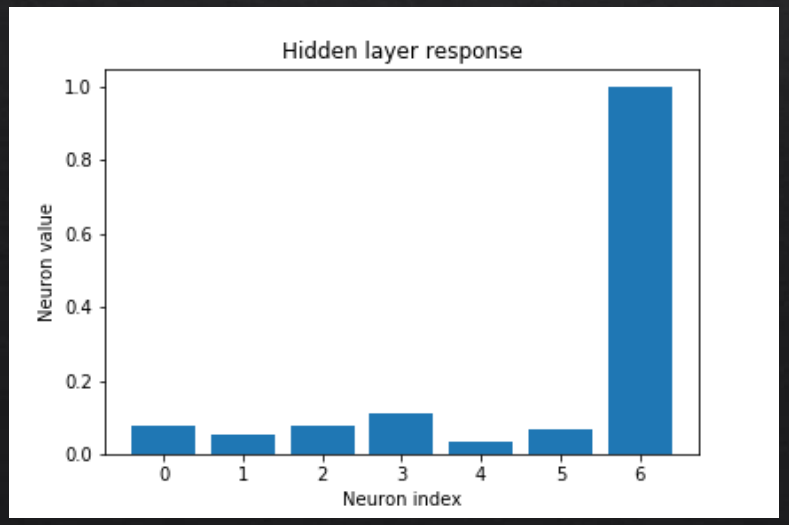


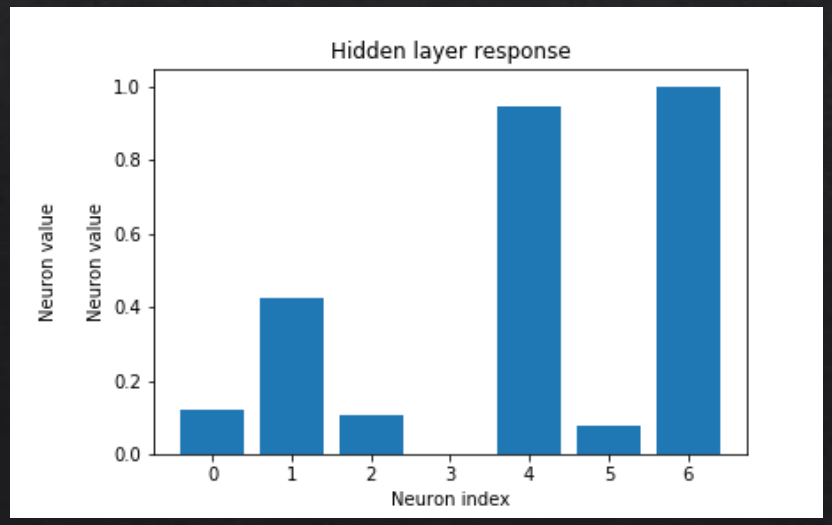
# Hidden layer analysis

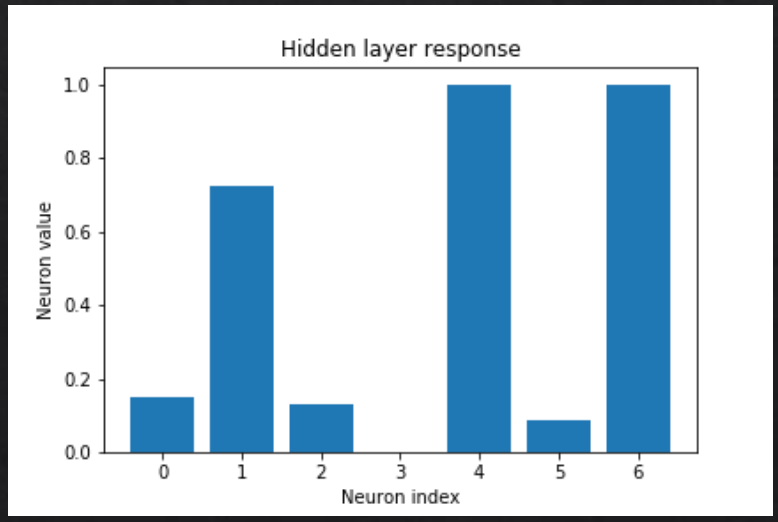
I selected squares on the star picture and propagated their points on the Multi-Layered net. Then I averaged the results of the hidden layer in nearby squares (notice that the two on the right are similar):





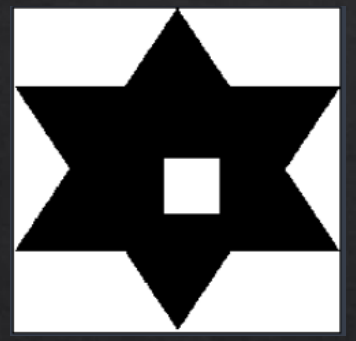


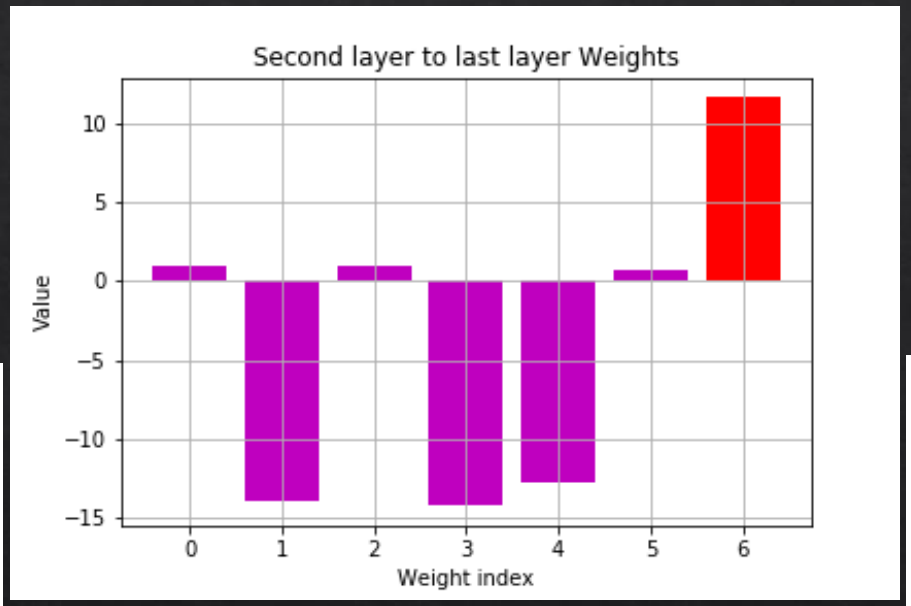



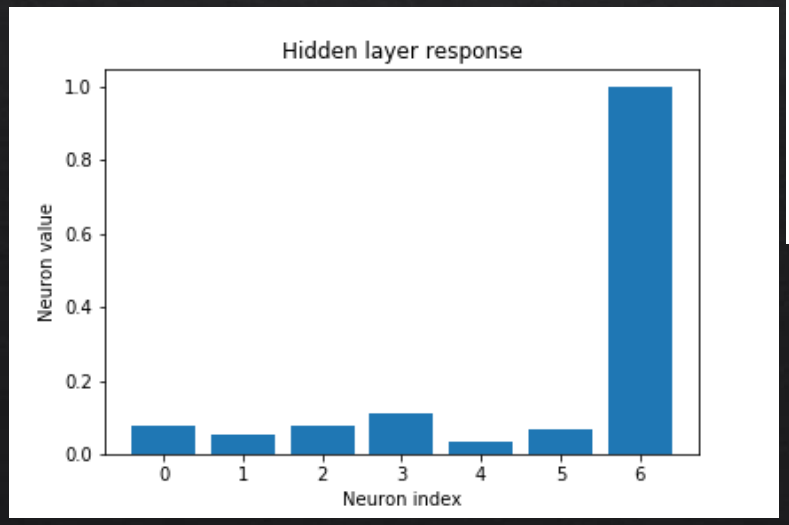



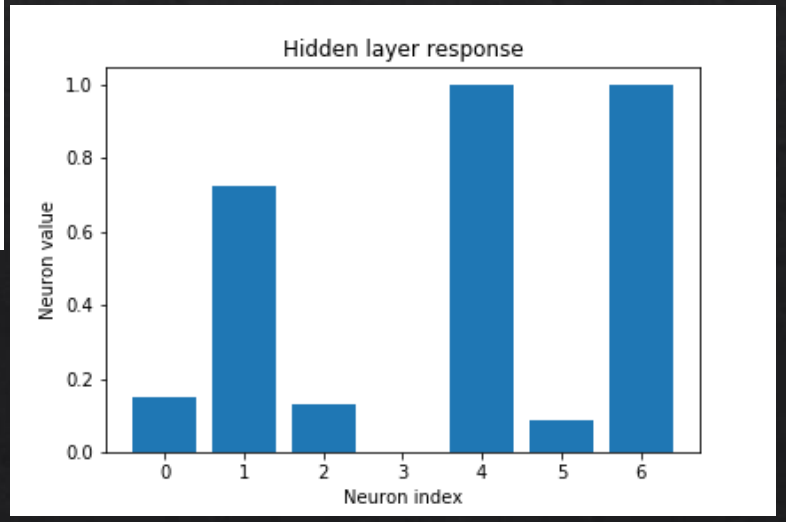
# Hidden layer analysis

If we'll look at the weights, we can see that the bias (in red) controls everything. If all the neurons get a low value (as in the left example), the bias gets dominant and we get a +1. If the other neurons are active, they deduce from the bias and we'll get a 0.



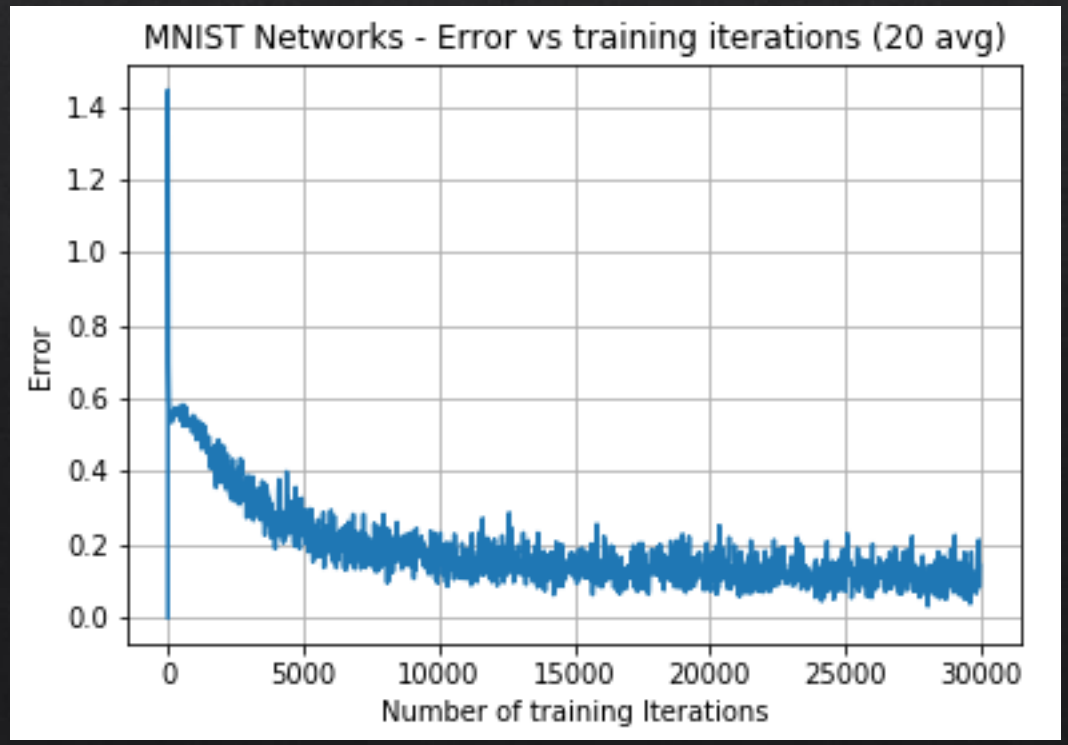


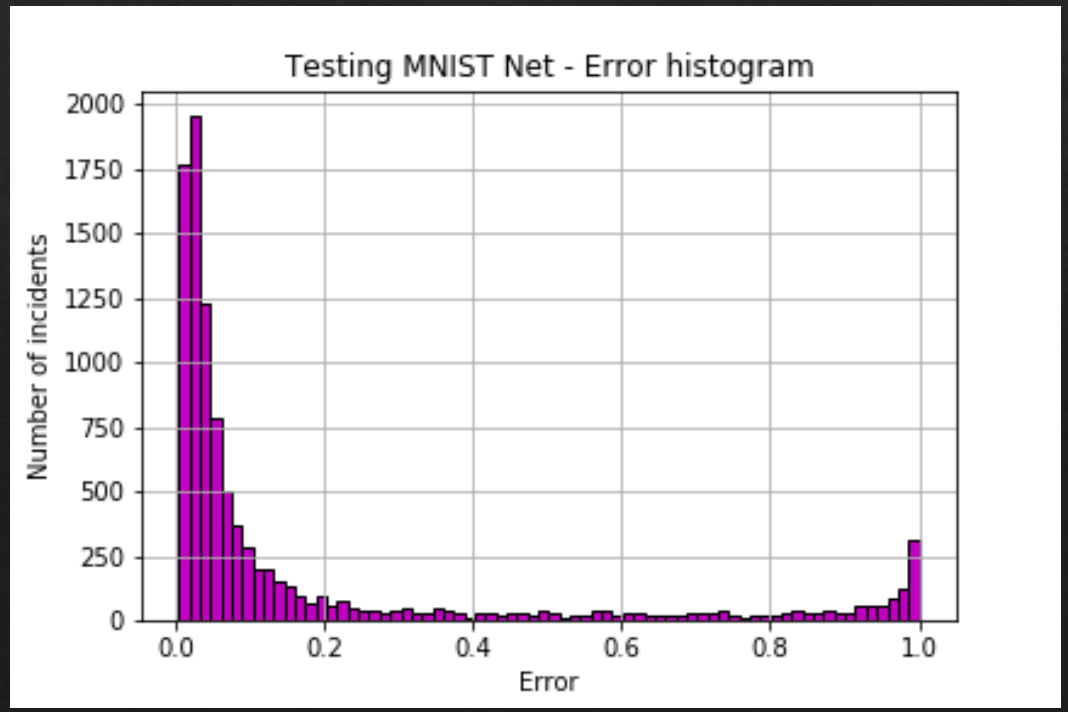
# MNIST Network

- The MNIST network I created is built from the following layer [28*28, 16, 16, 10].
- The output is  a 1x10 array. The value of each index is the probability of the answer being that number.
- The following graph describes the net's error in recognition of a digit from the MNIST test set, as a function of the training iterations the net had gone through. The graph is averaged over 20 runs.



# MNIST Network - Error

- After training the net with 3*10**4 training examples, I tested out the error using the test database. The error is defined as:
  $\delta = \max(|output - result|)$ , result being a 1x10 array with zeros in the wrong indexes and one at the corresponding right index.
- The average error from this histograms data is $\delta_{avg} \approx 1.84$ (fits to the previous slide).



# MNIST Network - Error

- I tested my own recognition of MNIST digits (100 digits), the histogram on the right.
- My own error is $\delta = 2\% = 0.02$, it is far away from the network's error.

As we've seen on the first graph, the network stabilizes at about 18.5% error, so more training wouldn't help it get any better result.
If we do wish to get better results, I believe we need to add another layer and reduce the step $\eta$.