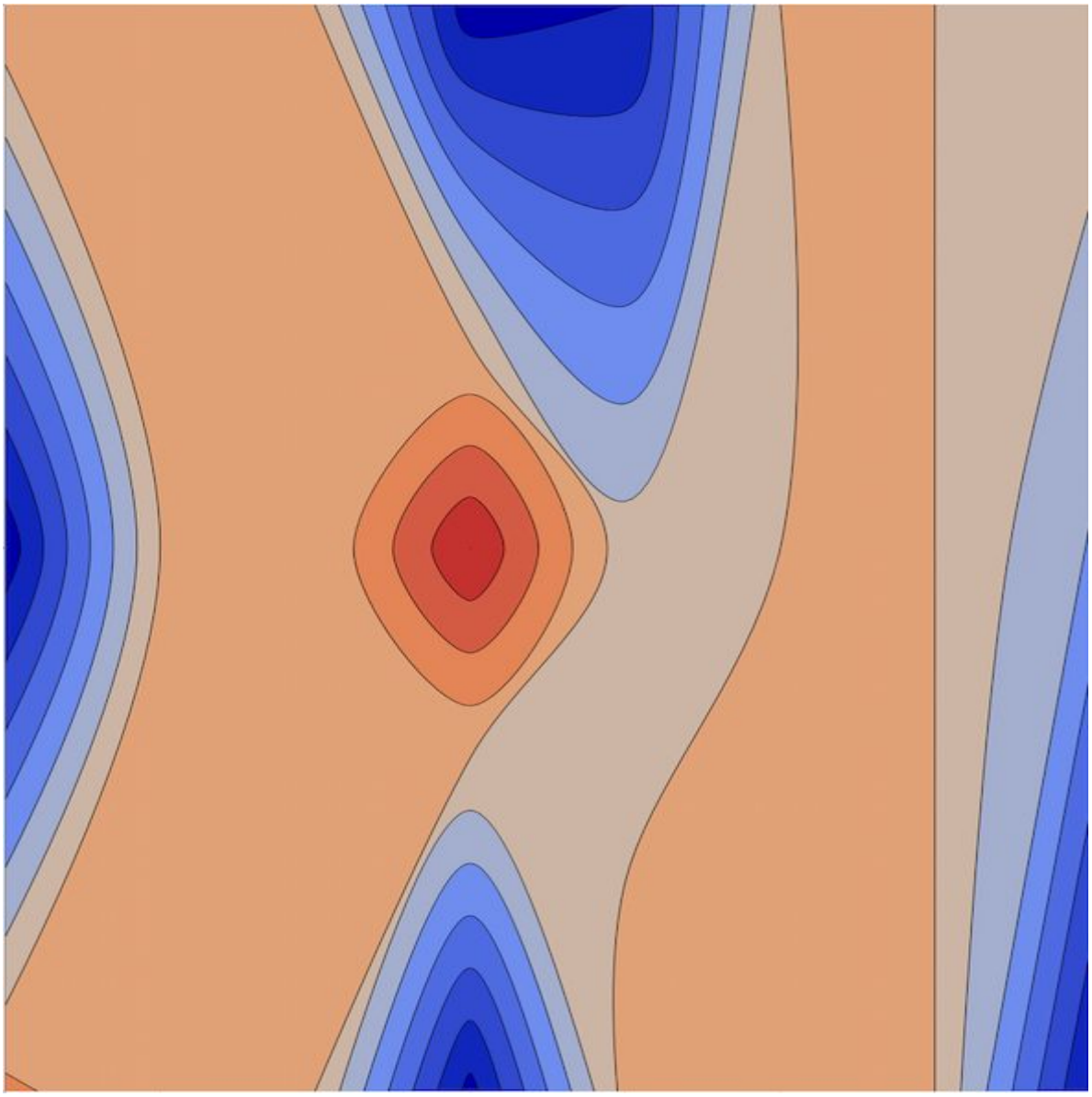
Clinton Cario
BMI 203
Final Assignment
Mar 06, 2015



Layer 2 weights as a contour plot for the 8x3x8 autoencoder after training

Autoencoder

The autoencoder was implemented in julia using an artificial neural network (Clint's ANN; CANN), and then trained over ~80,000 iterations. CANN was then compared to FANN, a C implementation with a julia wrapper library using default weights. CANN was then optimized using several julia specific techniques (CANNO). A comparison is shown here:

	FANN	CANN	CANNO (optimized)
Iterations (K)	80	80	80
Runtime (s)	36	1000	340
Error (MSE)	48.99	36.54	36.54
Memory (Mb)	3.301	348.316	174.041

The output of the layer activations and weights after training with CANNO (alpha = 40.0 for 3 million iterations; mean half squared error < 1 x 10^{7}) is also shown:

```
-----------------------
Layer 2 Θ:
-----------------------
[ 5.2189  -6.9153  -3.0416   6.2935  -4.7070   2.9786  5.1281  -5.3965
  4.4769  -0.2674  -7.6975  -4.9669  -0.2291  -5.7866  6.1851   7.3822
 -6.3695   5.9339  -0.5555  -5.1746  -7.6412   6.4792  6.6525  -0.1900 ]
-----------------------
Layer 2 biases:
-----------------------
[0.5595      0.0973      0.1344]


-----------------------
Layer 3 Θ:
-----------------------
[     8.2386     15.9530    -16.5489
    -18.2664     -3.7811     25.6600
    -17.0759    -35.2274     -0.7702
     11.8102    -16.4572    -15.9678
    -16.8370      1.6875    -33.4893
      8.2058    -17.3427     16.0201
      7.7290     15.0981     15.2428
    -16.0669     28.2116     -0.2012 ]
-----------------------
```

```
Layer 3 biases:
-----------------------
[ -16.2182 -16.3877 8.9669 -3.7270 6.8328 -16.1290 -30.5600 -20.5539 ]


-----------------------
Output for 8 bitstrings
(Input is 8x8 ident):
-----------------------
 [0.99957 0.00000 0.00000 0.00026 0.00024 0.00000 0.00038 0.00018 ]
 [0.00000 0.99942 0.00035 0.00000 0.00000 0.00031 0.00022 0.00038 ]
 [0.00000 0.00049 0.99934 0.00011 0.00044 0.00011 0.00000 0.00000 ]
 [0.00034 0.00000 0.00023 0.99961 0.00004 0.00035 0.00000 0.00000 ]
 [0.00018 0.00000 0.00043 0.00001 0.99935 0.00000 0.00000 0.00049 ]
 [0.00000 0.00020 0.00020 0.00026 0.00000 0.99958 0.00042 0.00000 ]
 [0.00018 0.00000 0.00000 0.00000 0.00000 0.00010 0.99941 0.00018 ]
 [0.00026 0.00039 0.00000 0.00000 0.00037 0.00000 0.00034 0.99940 ]
```

While the FANN implementation is much faster and more memory efficient, CANNO was used for the remainder of the project since the code could be tweaked to evaluate methods specific to this assignment.

Rap1 Binding Site prediction

Network Inputs, Encodings, and Evaluation

The CANNO neural network used for the autoencoder in the first part of the assignment was applied to the rap1 binding site problem. The data files provided with the assignment were processed as follows:

Positive sequences

- All 137 sequences from *rap1-leib-positives.txt* were read into memory as an array
- Sequences were encoded to binary floats, 2 bits per base (described below)

Negative sequences

- All sequences from *yeast-upstream-1k-negative.fa* were read into memory as an array
    - '>' header lines were ignored
- A random starting position between 1bp and >17bp from the sequence end was chosen, and then 17 consecutive base pairs from that position were selected.

- Five times as many sequences were used as than for the positive examples for the final classification (discussed below).
- Sequences were also encoded to binary floats, 2 bits/base

Encoding

Each nucleotide was encoded as a two bit float before being fed as input to the neural network. As there were 17bp for both positive and negative training sequences, this meant 34 binary inputs into the neural network. The encoding scheme was as follows:

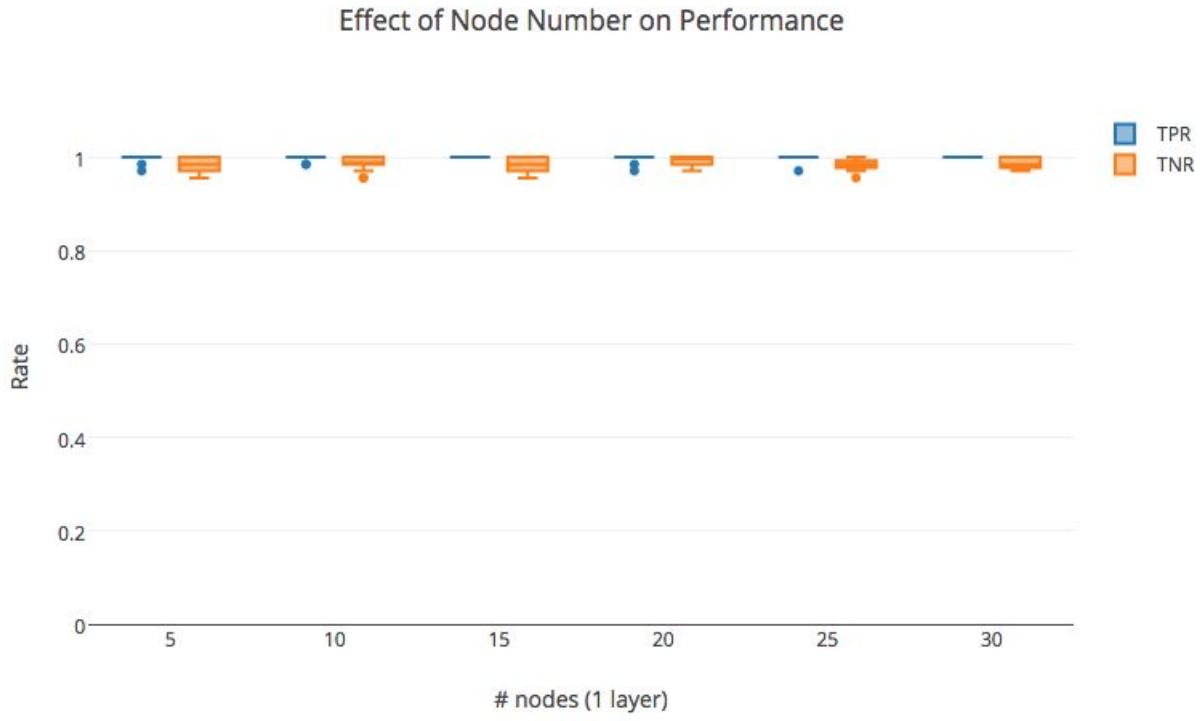
- `A ⇒ [0, 0]`
- `C ⇒ [0, 1]`
- `G ⇒ [1, 0]`
- `T ⇒ [1, 1]`

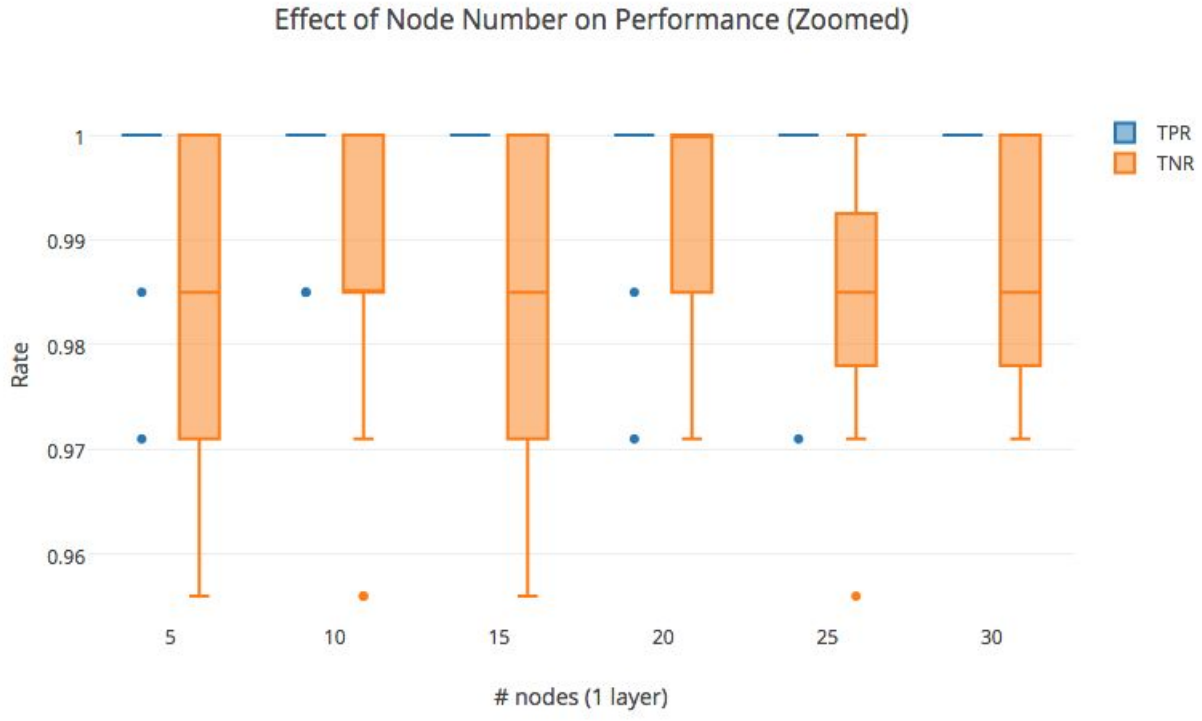
Performance Evaluation

Utility functions were written to randomly divide both positive and negative datasets into training and test cases of equal size. The network was then initially trained with 20,000 iterations on the training data. Test data was then used to measure the True Positive (TPR) and True Negative (TNR) rates. This process was repeated 20 times. Several parameter choices and architecture designs were evaluated.

Layer Architecture

A neural network having one hidden layer with {5, 10, 15, 20, 25, or 30} nodes was evaluated as shown below. Surprisingly, the number of nodes per layer had only a minor effect on performance, with an optimal number being around 20. The number of layers did not matter as long as the total number of nodes was held constant. The system also seems to bias towards False Positives (ie. a poorer TNR), most likely a reflection of the training data used (see above).


Effect of Node Number on Performance
TPR
TNR
Rate
1
0.8
0.6
0.4
0.2
0
5
10
15
20
25
30
nodes (1 layer)

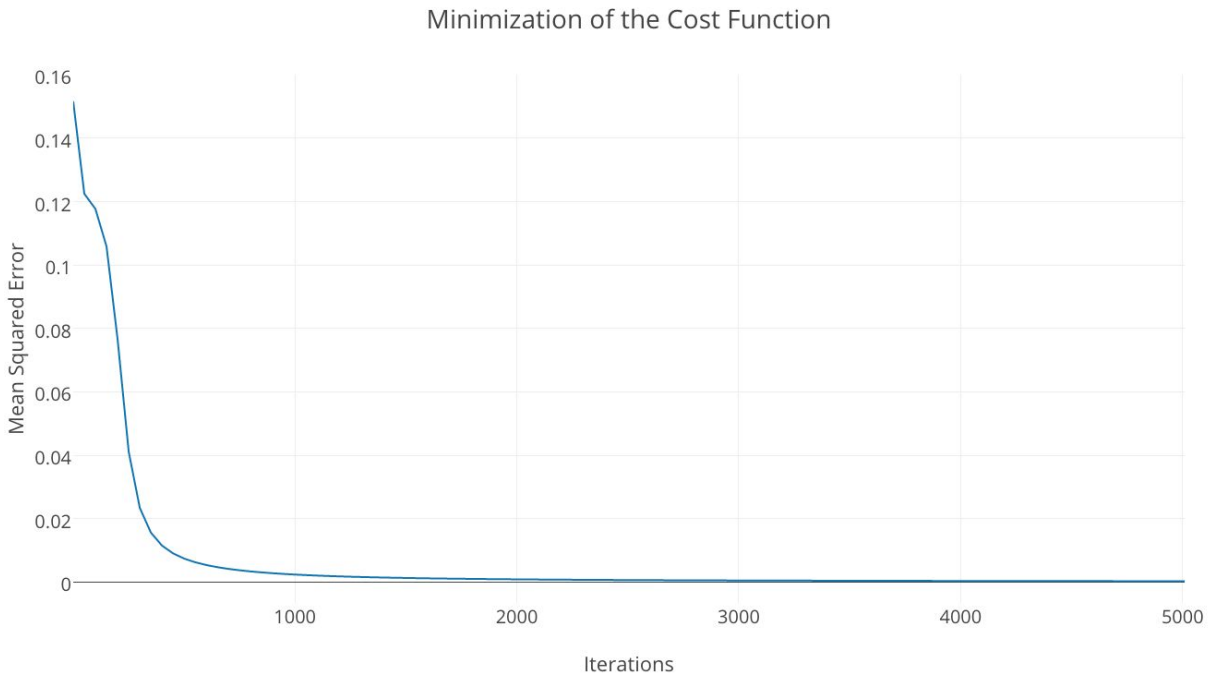

Effect of Node Number on Performance (Zoomed)
TPR
TNR
Rate
1
0.99
0.98
0.97
0.96
5
10
15
20
25
30
nodes (1 layer)

Alpha (training rate) Level

Next evaluated was the level of alpha on performance. Using the same training and testing data and 1 hidden layer with 15 nodes, neural networks with training rates of {0.03, 0.3, 3, or 30} were tested over 50,000 iterations. A training value between 0.3 and 3 performed best, with higher rates often not converging, and lower rates often taking longer to converge.

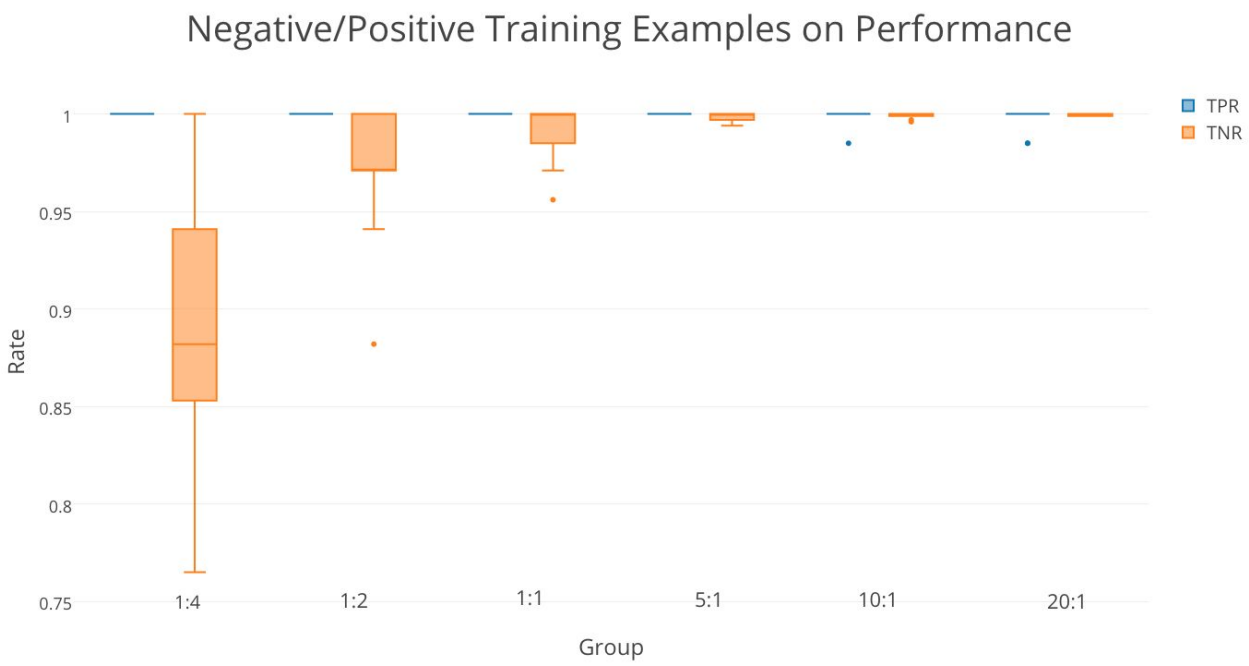
Convergence

Training of the neural network was terminated after a set number of iterations or the cost function (average half squared error) reached a set lower threshold (0.001 for testing). A function terminating after little cost change between iterations (a “diminished return” function) could have also been used. In most cases, a stable cost value was reached after around 2,000 iterations.



Cross Validation

Training a neural network on unbalanced or all available data can cause it to overfit noise within the dataset. It is often recommended to withhold some of the available training data for testing purposes since the model is blinded and not specifically fit to it. Likewise, unbalanced data (eg. many more negative examples than positive) can also bias the model towards dominate trends within the training examples. The amount of negative training data far outways positive, so the proportion of negative to positive data's impact on performance was tested for ratios of {1:4, 1:2, 1:1, 5:1, 10:1, and 20:1}.



From these results, a ratio of 5:1 negative to positive data had the highest TNR without affecting the TPR.

Similarly, the optimal ratio of data withheld for testing vs. training was not known, so several were considered {1:10, 1:4, 1:3, 1:2, 2:3, 3:4, or 9:10}. The network given the most training data (9/10 of total data or 1:10 testing to training) seemed to perform best, though this effect could result from the limited number of examples left to actually test on (ie. reduced power from small sample sizes). Regardless, given the low ratio of training data to the number of sequences in the final prediction set, it probably is best to use as much training data as possible.


Effect of Training/Testing Size Ratio on Performance
TPR
TNR
Rate
1
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8
Fraction of data withheld for testing


Effect of Training/Testing Size Ratio on Performance (Zoomed)
TPR
TNR
Rate
1
0.98
0.96
0.94
0.92
0.9
0.2
0.4
0.6
0.8
Fraction of data withheld for testing

Parameter selection

Based on the above results, the following parameters were selected for the final prediction:

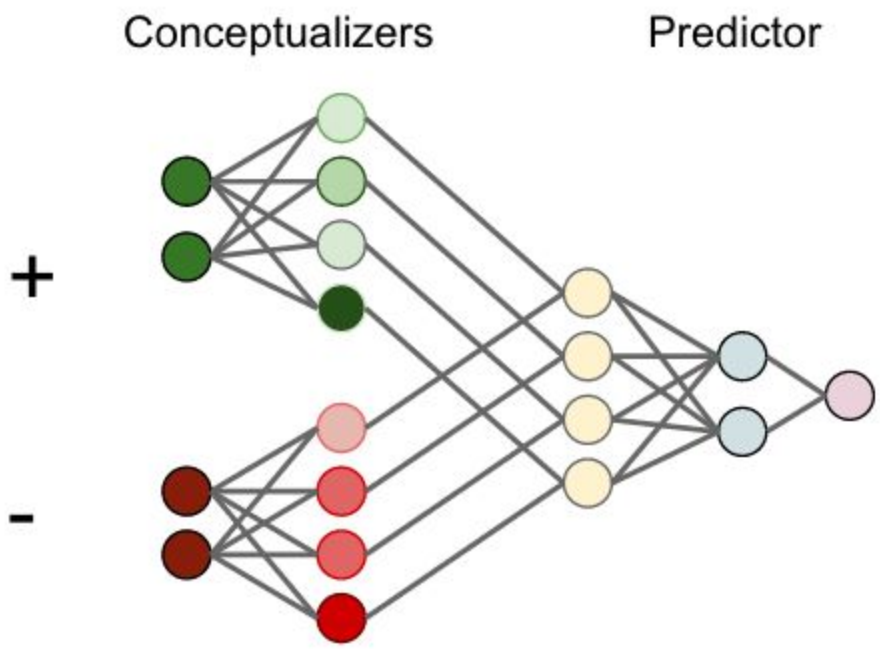
- Architecture: 1 hidden layer with 20 nodes
- Alpha: 0.9
- Convergence iteration: 2,000 + (10 million performed)
- Cross validation: 9/10 data used for training; 5:1 negative:positive data.

The neural network was run, and final mean half cost error rates were reported at 0.0000001. The *rap1-lieb-test.txt* file was then scored using this neural network

Further Extensions

Binding Site "Conception"

In deep learning applications, neural networks are often trained as auto encoders to build 'concepts' of their inputs. These concepts are reflected in the weights of the hidden layers and can be high-order in their data representation, often learning patterns difficult for humans to recognize. Since we have an abundance of negative data with which we don't want to bias our prediction network, it could be possible to use an auto encoder trained on negative data to build a concept of what that data is. If we create a final hidden layer of size equal to the input of our prediction algorithm, we could feed all negative data through the 'conceptualizer' first to distill it for training the predictor. Positive data can be fed directly into the predictor or conceptualized in a similar manner. Interestingly, the conceptualizer can be trained so that some error remains in its predictive capacity, inducing stochasticity into the prediction model which prevents overfitting. A simple schematic is shown below, with only one hidden layer in all networks. Note that the number of input nodes in the conceptualizers would likely equal the number in the predictor (not shown). Red/green colored nodes represent positive and negative data input and concepts, yellow predictor input, blue predictor hidden nodes, and purple the final prediction.



This idea was implemented with just the negative conceptualizer, which was ran with an alpha of 9.0 for 1 million iterations. The negative conceptual data and positive training data was used to train the predictor neural network and trained for 750K iterations with an alpha of 4.0. The final results were no better than the version without the conceptualizers (TPR: 1.0, TNR: 0.985). It is likely that this idea failed to work due to difficulty in auto encoding the negative data conceptualizer, which takes a very long time to train and often plateaus at high error rates. This is realistically caused by the fairly random nature of these sequences.