Zhiping Xiu

1):

A): No, what perceptron trying to do is looking for a hyper-plain to correctly divide all the input data-points. And one-layer perceptron cannot handle the situation that input datas are not linear separable.

B): No, multiple layers of cascaded linear units still produce only linear functions.

C): Yes, sigmoid function produce an outputs that is non-linear to the inputs, thus can capture non-linear functions. Besides, it fits similar to smoothed version of threshold function (perceptron).

D): With the introduction of discontinuous threshold (sign), we lost the continuity between layers, which caused same thing as using threshold perceptron unit: the back propagation will not 'penetrate' the layers.

2):

A):
By wikipedia definition: (https://en.wikipedia.org/wiki/Boltzmann_machine)

A Boltzmann machine, like a Hopfield network, is a network of units with an "energy" defined for the network. It also has binary units, but unlike Hopfield nets, Boltzmann machine units are stochastic. The global energy, E, in a Boltzmann machine is identical in form to that of a Hopfield network

Reference from: (https://en.wikipedia.org/wiki/Restricted_Boltzmann_machine)
A **Restricted** Boltzmann machine (RBM), is a variant of general Boltzmann machine, instead of fully connect every node in the network of Boltzmann machine, RBM group its node to two groups, the node in different group are fully connect, the node in the same group, however, have no connection.

B):
DBN can be seen as a stack of RBMs. When training, we use the input data X to train the 'outer' RBM, and use the trained RBM to produce X', and use X' as input data to train the second RBM (another hidden layer fully connected to the hidden layer of 'outer' RBM). We repeat doing this until the final layer (output layer) is reached.

3):

A): According to the paper, Kernel Machine is described as a group of shallow learning method. Who generally consist of **one** layer of **fixed** kernel functions, whose role is to match the incoming pattern with templates extracted from a training set, followed by a linear combination of the matching scores. (reference from Scaling Learning Algorithms towards AI)

One example of it is SVMs.

> 1: A general limitation for shallow architecture, is its representative power, with only one layer (or sometime two layers), the representative power is limited because to reach an ideal result, we need to put exponential number of nodes, which is impractical for complex learning problem, or require significant computational power.

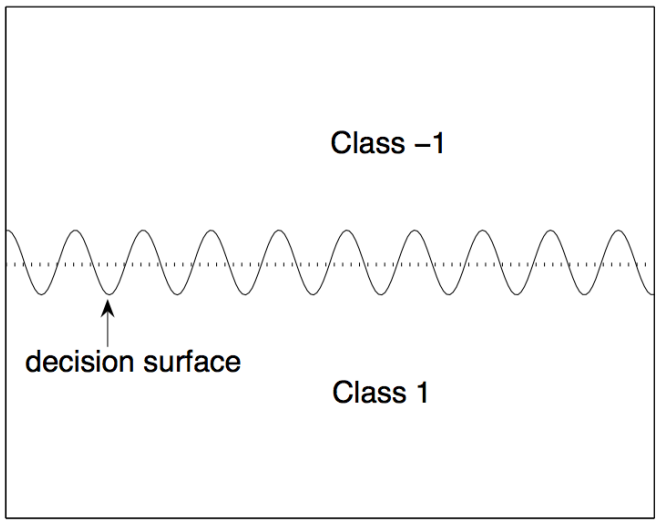


Figure 2: The dotted line crosses the decision surface 19 times: one thus needs at least 10 Gaussians to learn it with an affine combination of Gaussians with same width.

2: This exist in 'local' kernel machines, this kind of algorithm have a difficult to represent a function that locally varies a lot (need more templates for training, and more 'node' for representation).

A high curvature implies the necessity of a large number of training examples in order to cover all the desired twists and turns with locally constant or locally linear pieces.

B):

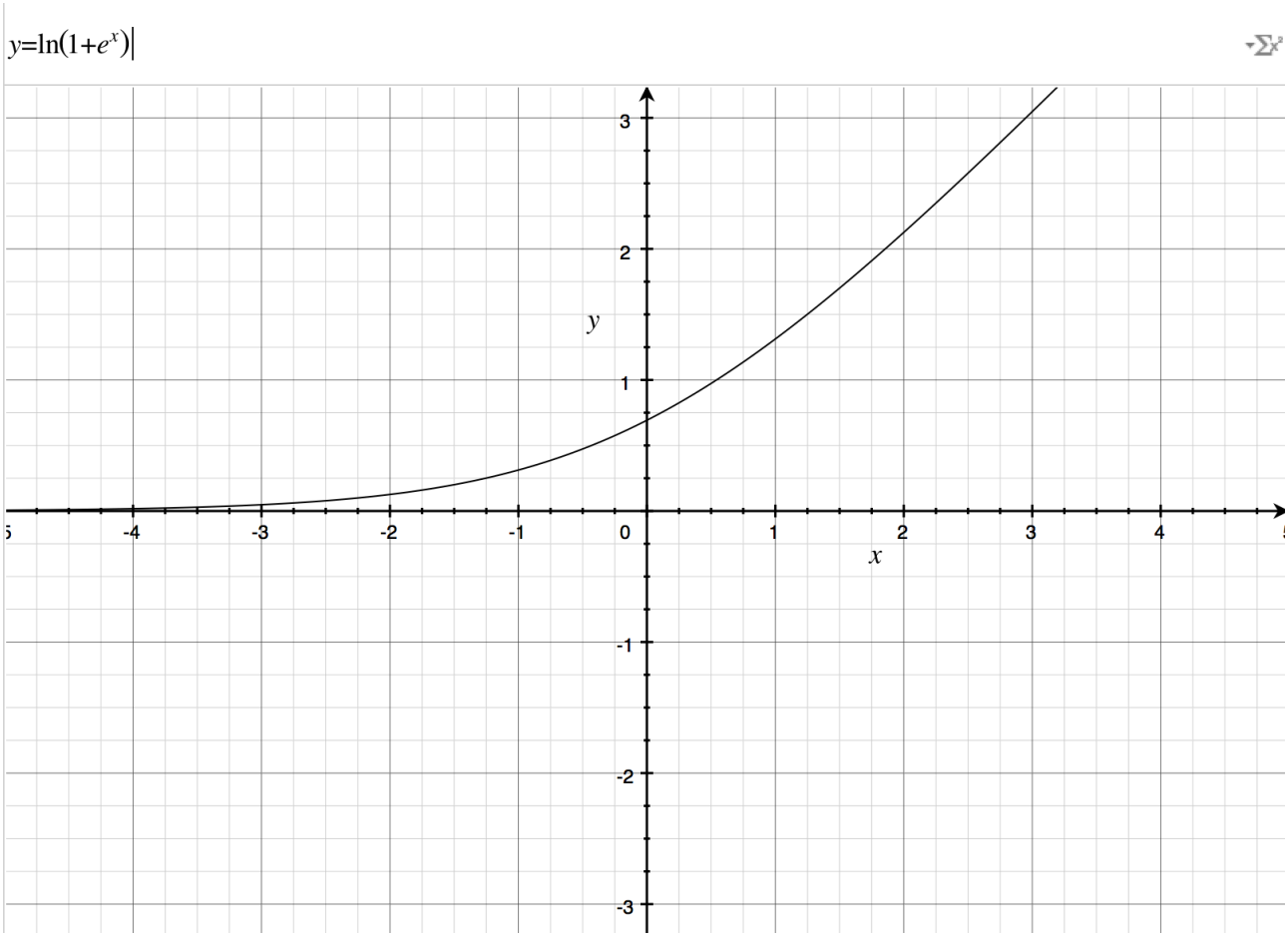
1. In section 3.2, the article talked about the trade off between depth and breadth, instead of adding more node to a one layer learning, we can hugely reduce the node (analogy for computational power) required by adding a few nodes in a new layer, and map the output of first layer as the input of the second (and succeeding) layer.

2. By introducing more layer, deep architecture can map to larger representations with fewer elementary operations. This is illustrated by 3 examples in page 15.

Reference to Scaling Learning Algorithms towards AI.

4):

A): ReLU: rectified linear unit's formula and shape (generated using Grapher). (reference: https://en.wikipedia.org/wiki/Rectifier_(neural_networks))



B): Softmax activation function from Wikipedia (https://en.wikipedia.org/wiki/Softmax_function):

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

The reason why using softmax activation function instead of normalizing a distribution in a standard way, by its definition, is: the output of the softmax function is used to represent a **categorical distribution**, which according to its definition (https://en.wikipedia.org/wiki/Categorical_distribution), is a probability distribution that describes the possible results of a random event that can take on one of K possible outcomes, with the probability of each outcome separately specified, and **should be adds up to 1**. this can only be reached by this way.

5):

A):

1: Including input and output layers, titanic_predictor contain 4 layers
2: 2 layers (in the middle between input and output layers) is hidden layer.
3: Input layer has 6 nodes, two hidden layers has 32 nodes each, output layer has 2 nodes.
4: Hidden layers used ReLU activation function, output layer used softmax activation function.
5: This network trains for 10 epoch.

B):

1: The first non-zero accuracy is 0.3375
2: The final accuracy is 0.7883

C):

1: The final accuracy for 20 epoch is: 0.7815, the final accuracy for 100 epoch is: 0.8308.
2: It doesn't add too much, so I believe the presentative power of the net work has its limit, this might be caused by the nature of its architecture.

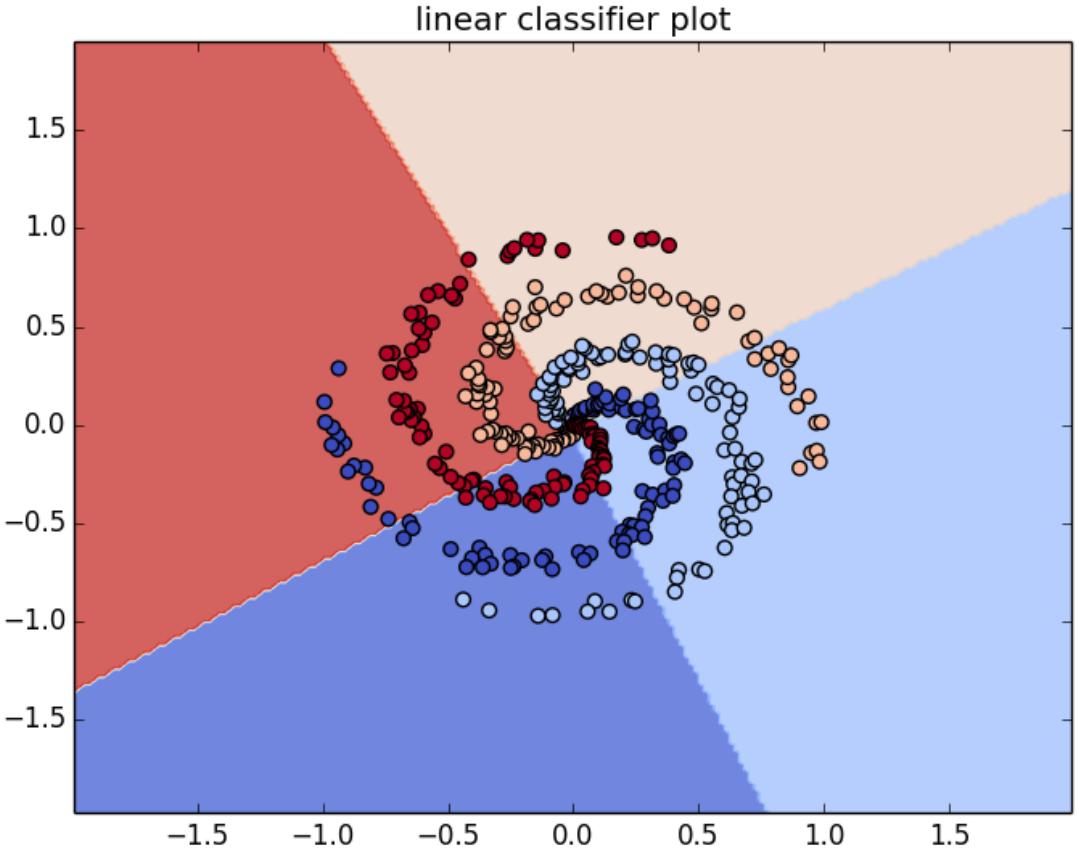
D):

1:
For 100 epoch:
DiCaprio Surviving Rate: 0.14286917448
Winslet Surviving Rate: 0.974280536175
For 20 epoch:
DiCaprio Surviving Rate: 0.123581960797
Winslet Surviving Rate: 0.833312213421
For 10 epoch:
DiCaprio Surviving Rate: 0.167561039329
Winslet Surviving Rate: 0.845157563686

6):

A):
Coded according to the request, please check the code.

B):
I choose to use 1000 epoch, because I want to see if the accuracy can change with the iteration, the result shown it changes very little amount, since the representative power of two layers network is limited.
I also batch_size equal to length of the input data (none stochastic gradient decent)
The other chooses are made according to the requirement specified in 6-A.
The plot is as follow: The final accuracy is fluctuate around 30%.



C):
Coded according to the request, please check the code.

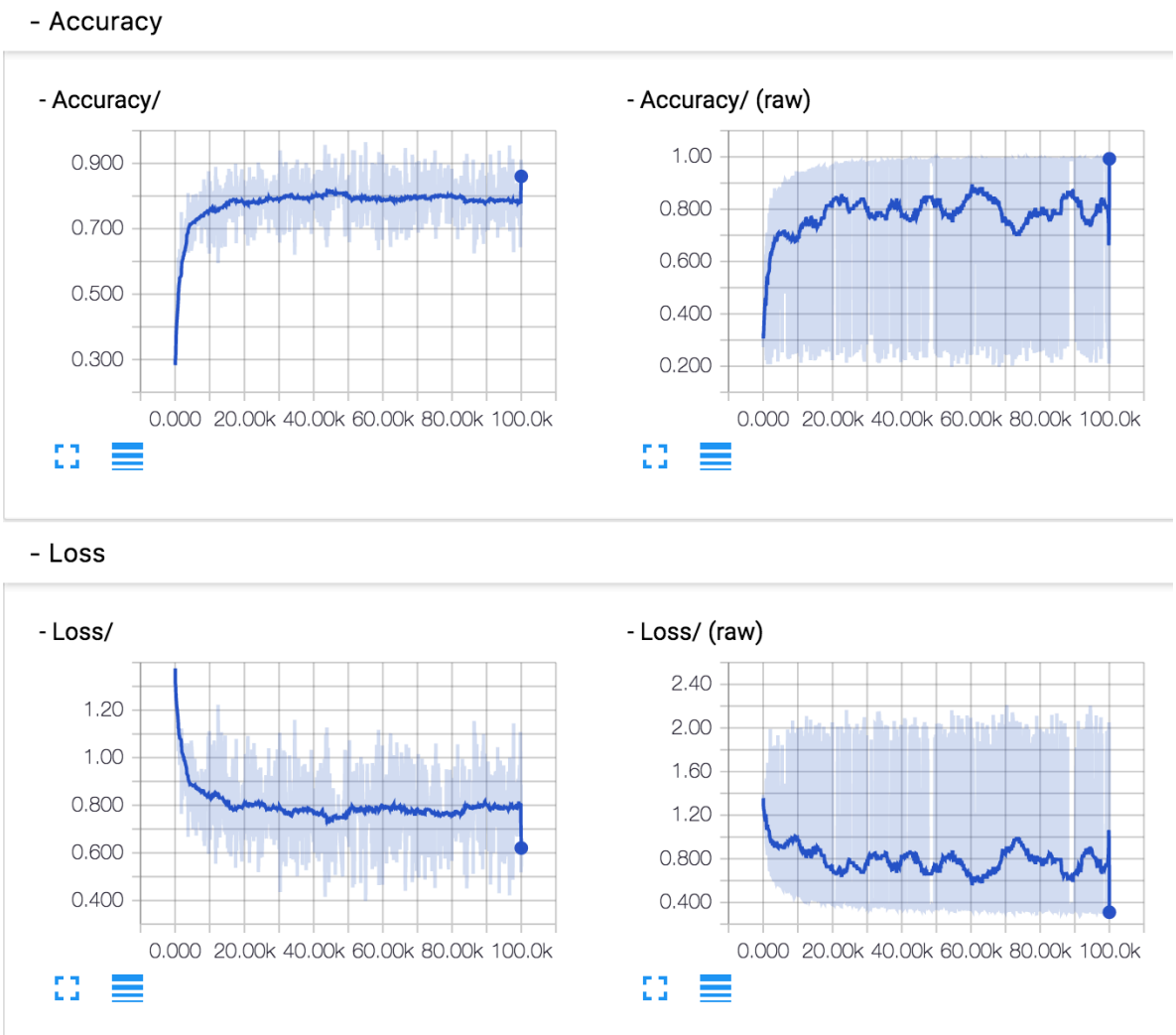
The decisions made about the code is described in 6-E.

D):

A plot for trained result: see at next page.

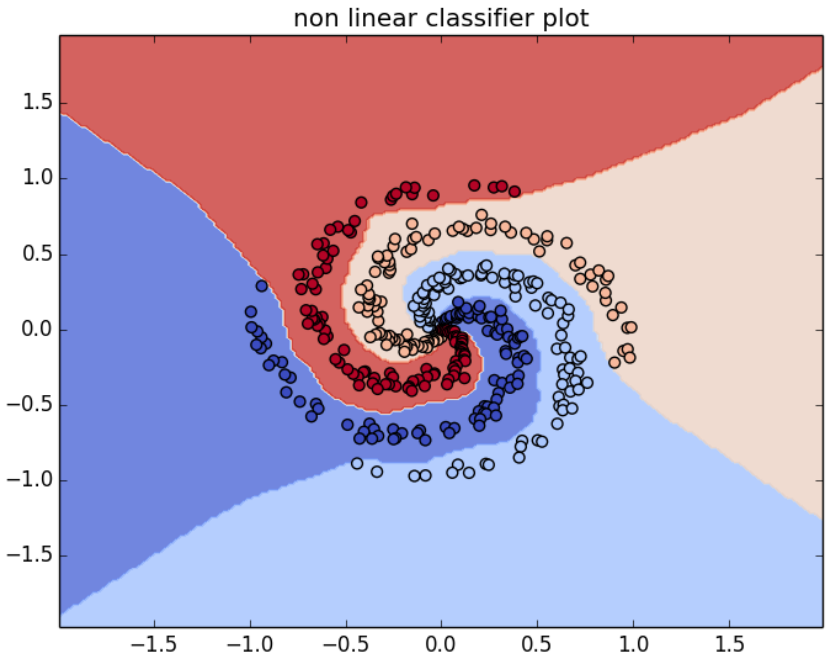
With the increase of the epoch, even-though the losses and accuracy may slightly fluctuate, they show trend towards better results (lower losses and higher accuracy), with a plot shown below:

plot is generated with tensorboard. The final accuracy is around 80% after 100,000 epoch. (I stated a different (Ethan's) method in 6-E, and it can get around 95% in 100 epoch)



Zhiping Xiu

E):I used 256 nodes for my hidden layer. I tested on 64, 128, 256 and 512, eventually selected 256.



I also used tflearn's built in optimizer:

The usage of optimizer is inspired from this note: https://github.com/tflearn/tflearn/blob/master/examples/notebooks/spiral.ipynb
The note used sgd as optimizer when training its data.

After compared few optimizers descried in document: http://tflearn.org/optimizers/ I mainly compared two optimizers, namely sgd and adam, adam is a stochastic optimization method presented based on an article published last year(https://arxiv.org/pdf/1412.6980v8.pdf).

I compared the loss and accuracy changes between two optimization functions. Run them both with 100,000 epoch, and showing adam's output is slightly better.

Improvement:
Then I switch to smaller amount of nodes (8) and batch size (20), and reached a better acc (0.9422), The output digram, however, is lest 'smooth', compared to the above one.