Jeremy Midvidy, jam658
EECS 349, Fall 2017
HW#8, PDF

Problem 1:

A.

A “single layer” or one-node perceptron is only capable of learning linearly separable data. Thus, a one-node perceptron cannot represent a nonlinear decision surface because that surface cannot be correctly represented by the architecture of the one-node perceptron (hence, its decision making in training will never converge)

B.

Back-propagation when a sigmoid function is manipulated with the sign equation means that error and end-decision of the network will, on the training step, switch to a 1 or -1 classification (with a slight horizontal shift of .5 on the input set). This means it would only be able to classify inputs as 1 or -1, and this also means that however the network nodes train after the first training step would be the final weights of the network regardless of how many more training steps are programmed.

Problem 2:

Yes, it is possible to represent non-linear decision surfaces with only linear activation units. This makes sense logically because you can configure the nodes and weights of a network to represent how a non-linear surface would make decisions. Each level in the network represents a transformation in the data the way a nonlinear classifier would classify data, and then, you can combine different transformations in different layers of the network to build the non-linear decision surface.

In this problem, the given network is as follows:

$W5 = W1X1 + W3X2$
$W6 = W4X2 + W2X1$

$W7 = 1 + W5$
$W8 = 1 + W6$

where $OUT = W9 = 1 + W7 + W8$

And, with substitution:
$OUT = 3 + X1(W1 + W2) + X2(W3 + W4)$

With this resulting function, it is clear that with two inputs [x1, x2], the network transforms them by multiplying them by a summation of weights. This is a nonlinear transformation which is only formed by using linear activation functions (in each node of the network to calculate the resulting 'input' at the next layer).

So thus, both logically and mathematically (for this given network) we can use a multi-layer perceptron with only linear activation units to represent a nonlinear decision function.

Problem 3:

A.

A feature map is a function which maps a data vector into a feature space. It helps to present the learning algorithm with more data that is then better able to be regressed upon or classified because it has more usable data than it did with just its original data vector.

B.

It's common to use multiple feature maps in a single layer to produce different feature maps for different relations, correlations, and other useful classification data in order to perform more comprehensive analysis.

C.

The max pool layer is a "kind of downsizing" whereby all of the information in a layer is represented more simply as an agreed upon form (layer N is represented as just the darkest color in layer N+1). This makes computation less expensive.

D.

Max pool layers are used because it simplifies the computational expenditure when doing machine learning with neural networks. It truncates many layers and also pools the contents of one layer into a simpler, representative top layer that can simply computation without affecting learning results.

Problem 4:

Softmax equation:

$$\sigma(Zj) = \frac{e^{Zj}}{\sum_{k=1}^{k} e^{Zk}} \quad \text{for j = 1,2,..,k}$$

Softmax is a good for reducing the influence of extreme data points on the data operations without removing all extreme data points. It tempers (hence “soft” in the name) by applying a nonlinear transformation (raising all data points to e), and then using that to normalize. This is better than regular normalizing because it reduces the influence of outliers and extreme points (graphically, the shape of e becomes less different as input increases).

With a really large data set, softmax ensures a relatively limited range with reduced influence of outliers more than standard normalization would, so it has clear benefits for data science.

Problem 5:

A.

The given titanic_predictor.py file as a neural network with these features:

- An input layer that accepts 6 input nodes
- 3 hidden layers
  - The first two hidden layers have 32 nodes and use activation function ‘relu’
  - The third hidden layer has 2 output nodes, and uses activation function ‘softmax’
- The network trains for 10 epochs

B.

The first non-zero accuracy value occurs at training step two where:

Ts2 | acc = .5625

The final accuracy value occurs at

Ts820 | acc = .7792

C.

After increasing the n_epoch variable from 10 to 20, the final accuracy was:

Ts1640 | acc = .7625

And from 20 to 100:

Ts8200 | acc = 0.7897

This shows that with more epochs, the network will not necessarily be more accurate. The network's architecture is designed to maximize prediction capability relatively quickly, since final accuracy when n_epochs=10 was very close to when n_epochs=100. This shows that the network reaches its optimal weights within a few epochs, and increasing the number of epochs beyond that does not change the weights much, and thus, will not change the accuracy.

D.

With three different n_epochs, survival rates are as follows:

n_epochs	Result
10	DiCaprio Surviving Rate: 0.178896 Winslet Surviving Rate: 0.854827
20	DiCaprio Surviving Rate: 0.172367 Winslet Surviving Rate: 0.853268
100	DiCaprio Surviving Rate: 0.177838 Winslet Surviving Rate: 0.999106

Problem 6:

A.

Code

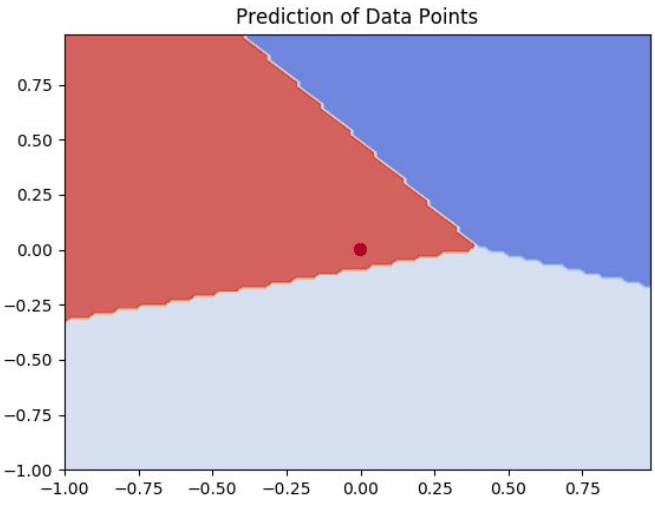
B.

I trained my linear classifier with 40 epochs. I used 40 because it was a good middle ground between too few epochs for training (network won't reach optimal weights) and overfitting.

For testing, I made a small sub-function that tested the predicted of 150 given data points from the inputted data set. The function then went through the 150 data points and compared the predicted classification from the network to the known classification of the data point. At the end, the accuracy of the test was very close to the final accuracy of the network when it trained.

This final accuracy was .2246. This shows that it is not possible to classify all of the data points with just one layer. This makes sense because, with just one layer, the network can only make decisions on each of the 4 output nodes individually, from the input layer. So when it trains, its error (loss) only back-propagates through 1 layer. This training is so weak that even when the network reaches optimal weights, the accuracy is just .2246 because it is essentially guessing ¼ on each classification.

After the network trained, I called `plot_spiral_and_predicted_class()` function, which produced this result:



C.

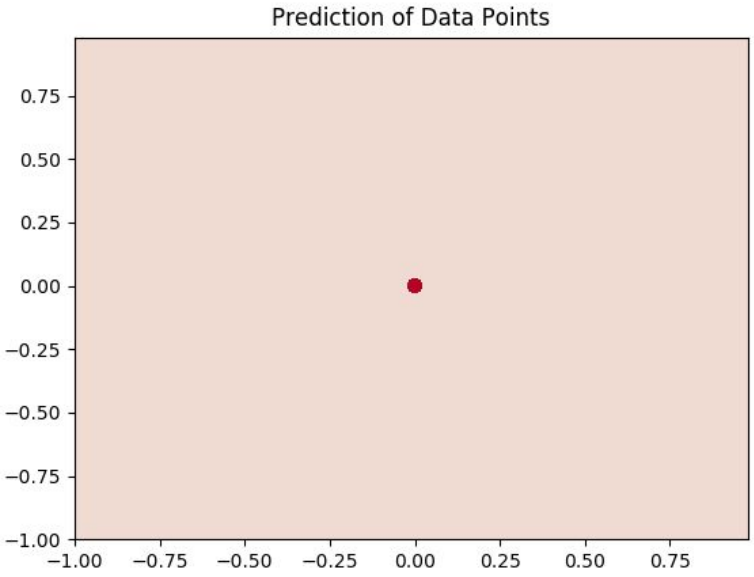
Code

D.

I trained the non-linear-classifier with an added hidden layer of 32 nodes and activation function ‘elu.’ Again, I used 40 epochs. Up to (around) 40 epochs, the classifier got more accurate, but beyond that, when I trained the network with 100, 1000, or 5000 epochs, the networks performed worse. I think this means that the network reaches its optimal weights in about 40 epochs or so.

The final accuracy of the network was .2579. This is better than the linear classifier (up from .2246), but still not that accurate.

I used the `plot_spiral_and_predicted_class()` function to plot the results of the non-linear-classifier, which produced this graph:



E.

I iterated through a number of different parameters. I changed the n_epochs from 40 to 80, 120, 250, 500, 1000, and 5000. With each of these n_epoch parameter, I tried different activation functions. Very quickly I learned that with this data, elu and relu were performing the best. So I alternated between relu and elu for the activation function in the hidden layer. When I found that elu performed the best at 40 epochs, I then tried to find the best number of nodes. I tried 4, 16, 32, 64, and 100. Out of all of these, 32 nodes produced the highest accuracy. So my final parameters for the hidden layer was 32 nodes, activation=’elu’, and for the network to train on 40 epochs.

Like before, I tested with 150 known data points, and compared the known classification to the predicted one. Again, it performed on par with the final accuracy of the non-linear-network, which was .2579