EECS 349 Machine Learning

Project 8

Ding Xiang

Nov. 21, 2017

Problem 1

A. No.

One-node perceptron has the following form.

$$\sigma = \begin{cases} 1, & if \sum_{i=0}^{n} w_i x_i > 0 \\ 0, & else \end{cases}$$

So it can only represent linear decision surface.

B. The back propagation of error training method may not work in some cases.

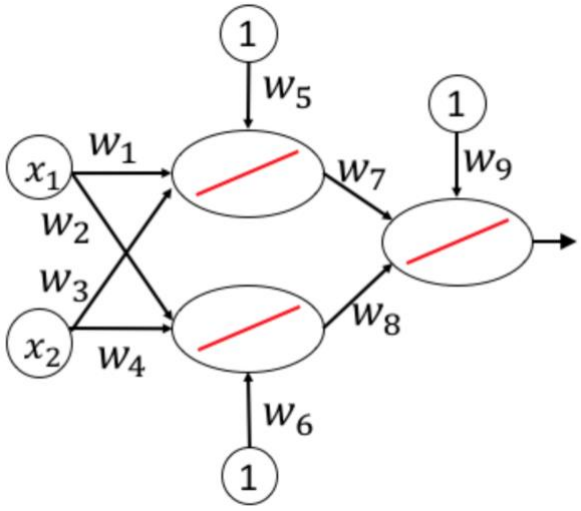
It's equivalent to say that

$$Output = \begin{cases} 1, & if\ w^T x \geq 0 \\ -1, & else \end{cases}$$

So the output becomes a special kind of step function, which doesn't have gradient at some point. This will lead to problems while running back propagation of error training method. For example it cannot learn non-linear decision surface in this case.

Problem 2

A. No, because combining linear functions only gives us linear functions.
Take the following figure as an example.



The final result is
output
$= (x_1 w_1 + x_2 w_3 + w_5) w_7 + (x_1 w_2 + x_2 w_4 + w_6) w_8 + w_9$
$= x_1 (w_1 w_7 + w_2 w_8) + x_2 (w_3 w_7 + w_4 w_8) + (w_5 w_7 + w_6 w_8 + w_9)$
So it's just a linear function of x_1 and x_2, which cannot represent non-linear decision surfaces.

Problem 3

A. Feature map is "a set of nodes that share connection weights." (By lecture notes). In another word, feature map is the output activations for a given filter.

B. Since it's common to look for multiple features,

where each feature map will specialize on one thing. Also even with many feature maps, we still have fewer weights than that of a fully connected net.

C. It's a layer after doing a kind of down sampling by the following rule.

$$f(x) = \max(x_1, x_2, \dots, x_n)$$

where n is specified size of each pool for the previous layer.

D. The reason why people use max pool layer is that it can help reduce the number of weights and keep the property of the previous layer to some extent. So they train faster and they need fewer training examples.

Problem 4

1. The softmax function $\sigma(\cdot)$ is defined as below.

$$\sigma(\boldsymbol{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \qquad for\ j = 1, 2, \dots, K$$

where $\boldsymbol{z}$ is a K-dimensional vector.
So the softmax function squashes the outputs of each unit to be between 0 and 1. And it's obvious that the total sum of each unit of softmax function equals to 1.
One reason for using a softmax function instead of normalizing a distribution is that from the

perspective of information theory, we usually want to minimize the cross-entropy loss of the training model, i.e.

$$H(\boldsymbol{p}, \boldsymbol{q}) = -\sum_{j} p_j \log\, q_j$$

(where $\boldsymbol{p}$ is the "true" distribution and $\boldsymbol{q}$ is the estimated distribution)
then softmax function,

$$q_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

is the optimal solution minimizing cross-entropy loss between the estimated distribution and the true distribution.

Sources:

1. https://en.wikipedia.org/wiki/Softmax_function
2. http://cs231n.github.io/linear-classify/#softmax
3. https://www.cs.toronto.edu/~urtasun/courses/CSC2515/05nnets-2515.pdf

Problem 5

A. Including input and output layers, there are totally 4 layers. The architecture is: the input layer (1st layer) fully connected to the 2nd layer, which has 32 nodes, and the 2nd layer's nodes are fully connected to the 3rd layer's 32 nodes, then the 3rd layer's nodes are fully connected to (the output layer) the 4th layer's 2 nodes.

So, there are 2 hidden layers. The 2nd and 3rd layers have RELU activation function, which is

$$f(x) = \max(x, 0)$$

The 4th layer has SoftMax activation function, which is the same as in the problem 4. Specifically,

$$\text{softmax}[i, j] = \exp(\text{logits}[i, j]) / \text{sum}(\exp(\text{logits}[i]))$$

(Source: http://tflearn.org/activations/#softmax)

This network trains for 10 epochs.

B. It's the 2nd training step where system shows non-zero accuracy.

```
Training Step: 1  | time: 0.120s
| Adam | epoch: 001 | loss: 0.00000 - acc: 0.0000 -- iter: 0016/1309
--
Training Step: 2  | total loss: 0.62551 | time: 0.122s
| Adam | epoch: 001 | loss: 0.62551 - acc: 0.3937 -- iter: 0032/1309
--
```

The final accuracy value on the training data is 0.7945.

```
Training Step: 820  | total loss: 0.46822 | time: 0.233s
| Adam | epoch: 010 | loss: 0.46822 - acc: 0.7945 -- iter: 1309/1309
--┤
DiCaprio Surviving Rate: 0.124982
Winslet Surviving Rate: 0.929768

Process finished with exit code 0
```

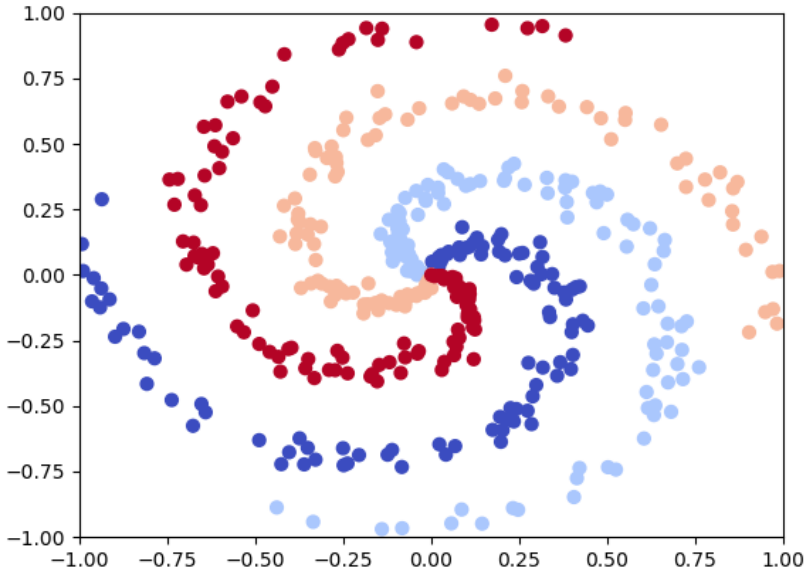
C. After increase the epochs to 20, the accuracy becomes 0.8032. Further more, if increase to 100, the accuracy becomes 0.8041. This tells us the network can predict the correct outcome (survived or not) up to about 80% of the total passengers in the training data. Mathematically speaking the gradient method finds a local minimum but this is not the perfect accuracy we want. We may want around 100% accuracy of a network architecture learned in the training data. In another

word the current network architecture has some learning limit, which is ok if we accept accuracy of 80%, but definitely not perfect if we demand higher accuracy.

D. In the 100 epoches, DiCaprio Surviving Rate is 0.17096 and Winslet Surviving Rate is 0.962549.

Problem 6

A. The original plot of data looks as below.



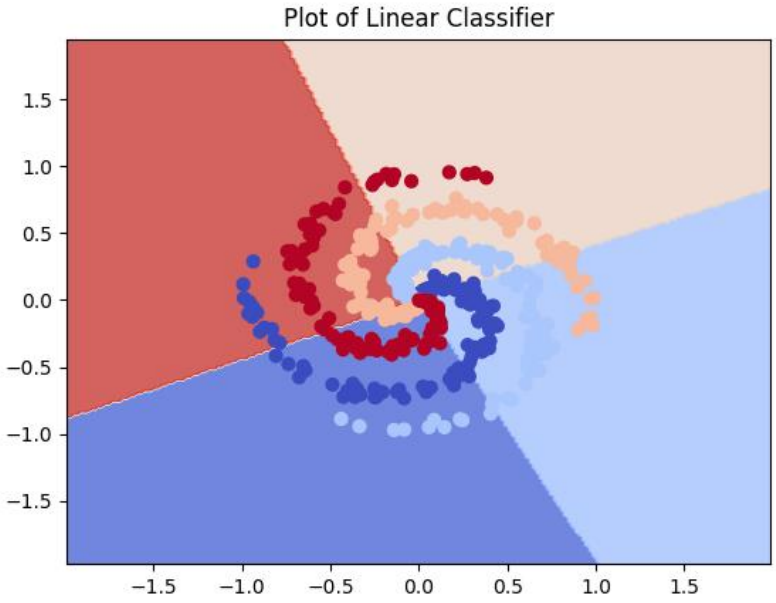
For the linear classifier, please see code "*spiral_classifier.py*".

B. The results are shown below.

```
Training Step: 399  | total loss: 1.32887 | time: 0.006s
| Adam | epoch: 100 | loss: 1.32887 - acc: 0.2962 -- iter: 300/400
--
Training Step: 400  | total loss: 1.32508 | time: 0.008s
| Adam | epoch: 100 | loss: 1.32508 - acc: 0.3036 -- iter: 400/400
--

Process finished with exit code 0
```

Here I specify epochs to be 100 with batch size of 100, which seems enough to train a linear 2 layer network. The plot of classification results is shown below.



The accuracy of the classifier after training is 0.3036. So it's not good enough to correctly classify all of data points with just one layer. From the physical structure we can see that 2 dimensional linear network will end put with linear boundaries, which obvious cannot classify the no-linear distributed data points (let alone they are spiraling).

C. See code "*spiral_classifier.py*".

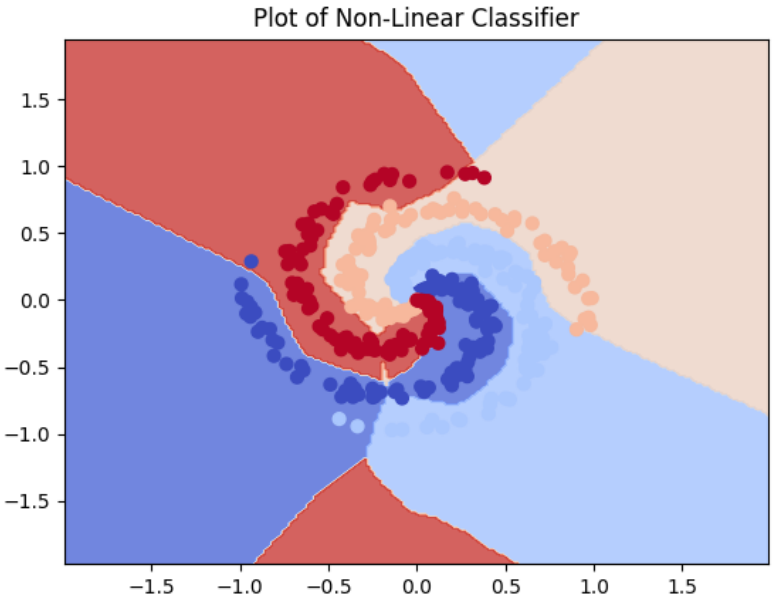
D. I specify the hidden layer to be a 64-node layer with activation function ReLU. I set the epochs to be 2000 with batch size of 100. After training the

two-layer network classifier, it got the following results.

```
Training Step: 7998  | total loss: 0.78984 | time: 0.014s
| Adam | epoch: 2000 | loss: 0.78984 - acc: 0.7091 -- iter: 200/400
Training Step: 7999  | total loss: 0.77291 | time: 0.019s
| Adam | epoch: 2000 | loss: 0.77291 - acc: 0.7172 -- iter: 300/400
Training Step: 8000  | total loss: 0.75283 | time: 0.022s
| Adam | epoch: 2000 | loss: 0.75283 - acc: 0.7255 -- iter: 400/400

Process finished with exit code 0
```

So the final accuracy is 0.7255.
After trying several different values of epoch in my model, I found that as the value of epoch increases the accuracy of the classifier also increases.
The plot of classifier is given below.



E. Since the input layer and output layer are given, the only layer that we can specify is the hidden layer. As I said above, the hidden layer has 64 nodes, uses activation function ReLU. And I set the epochs to be 2000 with batch size of 100.

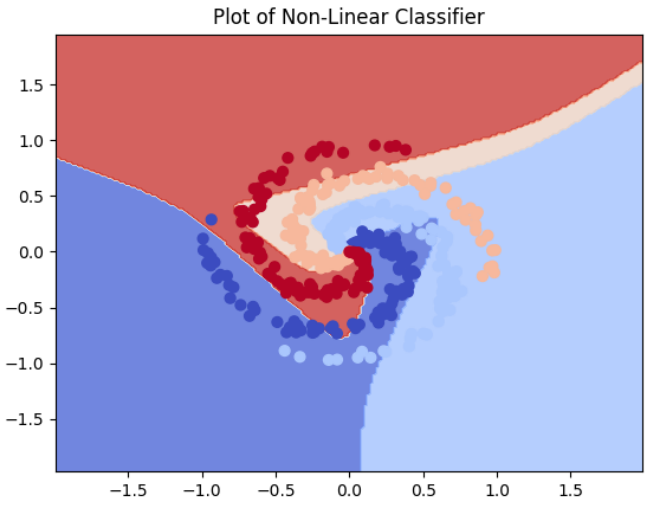
The reason I chose 64 nodes is I want the network is complex enough to describe the relationship of the spiral and the classes. But I don't want it to be too large, because first it's not that necessary; second it may increase the convergent time.

For choosing the activation function, it's obvious that "Linear" activation is not enough for this classifier, since the spiral shape is non-linear. So we need to choose a non-linear activation function. Theoretically, non-linear activation functions should suffice for this work; the differences would be their convergent speeds in gradient descent algorithm. For example, if I use SoftMax with epoch 5000 and batch size of 400, the results are shown below.

```
Training Step: 4999  | total loss: 0.90086 | time: 0.006s
| SGD | epoch: 4999 | loss: 0.90086 - acc: 0.6634 -- iter: 400/400
Training Step: 5000  | total loss: 0.87968 | time: 0.007s
| SGD | epoch: 5000 | loss: 0.87968 - acc: 0.6718 -- iter: 400/400

Process finished with exit code 0
```

The accuracy is 0.6718. The classification plot is

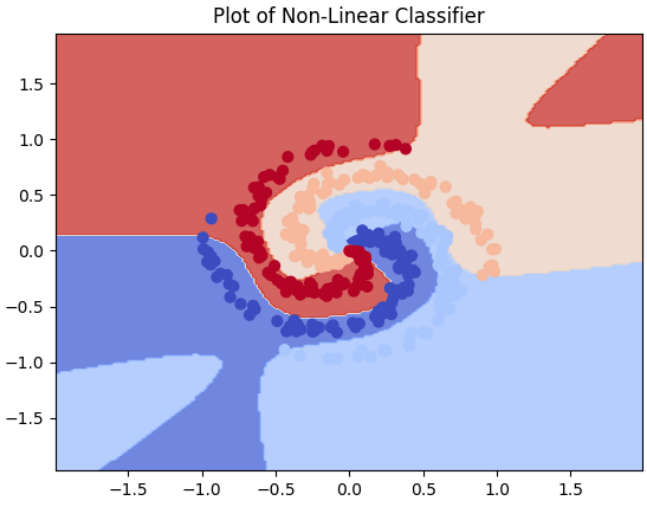


If I change to Sigmoid with epoch 5000 and batch size of 400, the results are shown below.

```
| SGD | epoch: 4998 | loss: 1.05614 - acc: 0.6599 -- iter: 400/400
Training Step: 4999  | total loss: 1.00887 | time: 0.008s
| SGD | epoch: 4999 | loss: 1.00887 - acc: 0.6794 -- iter: 400/400
Training Step: 5000  | total loss: 1.10904 | time: 0.006s
| SGD | epoch: 5000 | loss: 1.10904 - acc: 0.6348 -- iter: 400/400

Process finished with exit code 0
```

The accuracy is 0.6348. The classification plot is

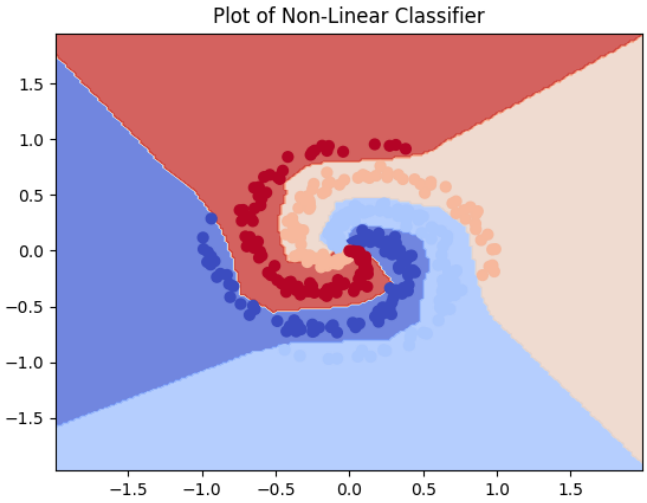


Instead, if we use to ReLU with epoch 5000 and batch size of 400, the results are shown below.

```
Training Step: 4998  | total loss: 0.90201 | time: 0.003s
| SGD | epoch: 4998 | loss: 0.90201 - acc: 0.7427 -- iter: 400/400
Training Step: 4999  | total loss: 0.87419 | time: 0.002s
| SGD | epoch: 4999 | loss: 0.87419 - acc: 0.7539 -- iter: 400/400
Training Step: 5000  | total loss: 1.02181 | time: 0.002s
| SGD | epoch: 5000 | loss: 1.02181 - acc: 0.7035 -- iter: 400/400

Process finished with exit code 0
```

The accuracy is 0.7035. The classification plot is



Just based on the performance, ReLU has a better accuracy with a faster training speed.

For the reason of choosing epochs, in my model, usually the more epochs we train the network the higher accuracy it would be. But we don't want it training for too long. So it depends on our expected accuracy. In this problem, our expected accuracy is to be better than the accuracy of the linear classifier, which is 0.3036. So 2000 epochs seems and turns out to be enough to achieve this goal.