ML_Programming Assignment3

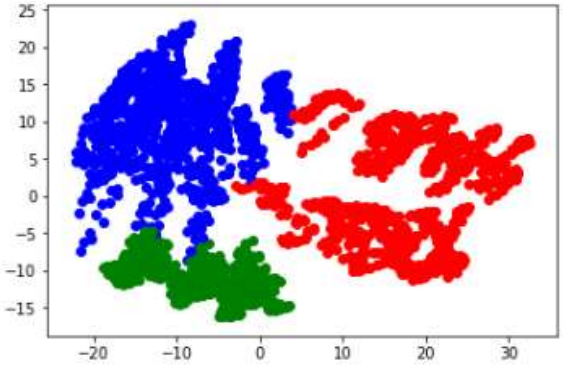
In this problem, you need to build a neural network in order to classify images into three classes. Particularly, instead of calling on any existing functions in MATLAB or Python that do the tricks for you, you need to code on the error backpropagation algorithm by yourself in order to realize the learning process.

---

1. **Data Preparation**

   I divide the data into training part and testing part, with the ratio of 9:1.

   Feed data to PCA with deducted_dim=2.



   After PCA, three classes are in two dimensions, and are seperated with each other. (Remember to normalize the data fisrt, otherwise three clusters will still overlap with each other)

2. **Training**

   The training part is composed of feed forward and back propogate.

   The stochastic gradient descent is the central idea of building model.

   First, give weights of each layers.

   Second, calculating the predictions of output, we will get probability of three classes because we use softmax as our final layer'a activation function.

   Third, having predicted probabilities and real_target, we get softmax cross entropy loss. Propogate error back through the network and adjusting weights of each layers. Return to process One Again.

   ＊Activation Function and its derivative.

   Sigmoid:

   Derivative of Sigmoid function

$$y = \frac{1}{1+e^{-x}}$$

$$\frac{dy}{dx} = -\frac{1}{(1+e^{-x})^2}(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2}$$

$$= \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) = y(1-y)$$

Relu:

$$\text{ReLU activation} \qquad R = max(0, Z) \qquad R'(Z) = \begin{Bmatrix} 0 & Z < 0 \\ 1 & Z > 0 \end{Bmatrix}$$

＊Error Back Process

In order to calculate the derivative(gradient) and pass it back to the previous layer during backpropagation, we should figure out how each later process.

In updating process, we use chain rule fo from end to start to get delta terms, then we can get new adjusted weights.

With softmax:

If $i = j$,

$$\begin{aligned} \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} &= \frac{e^{a_i} \sum_{k=1}^{N} e^{a_k} - e^{a_j} e^{a_i}}{\left(\sum_{k=1}^{N} e^{a_k}\right)^2} \\ &= \frac{e^{a_i} \left(\sum_{k=1}^{N} e^{a_k} - e^{a_j}\right)}{\left(\sum_{k=1}^{N} e^{a_k}\right)^2} \\ &= \frac{e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \times \frac{\left(\sum_{k=1}^{N} e^{a_k} - e^{a_j}\right)}{\sum_{k=1}^{N} e^{a_k}} \\ &= p_i(1 - p_j) \end{aligned}$$

For $i \neq j$,

$$\begin{aligned} \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} &= \frac{0 - e^{a_j} e^{a_i}}{\left(\sum_{k=1}^{N} e^{a_k}\right)^2} \\ &= \frac{-e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \times \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}} \\ &= -p_j . p_i \end{aligned}$$

So the derivative of the softmax function is given as,

$$\frac{\partial p_j}{\partial a_j} = \begin{cases} p_i(1 - p_j) & if \quad i = j \\ -p_j . p_i & if \quad i \neq j \end{cases}$$

Or using Kronecker delta $\delta ij = \begin{cases} 1 & if \quad i = j \\ 0 & if \quad i \neq j \end{cases}$

$$\frac{\partial p_j}{\partial a_j} = p_i(\delta_{ij} - p_j)$$

Derivative of Cross Entropy with Softmax:

$$L = -\sum_i y_i log(p_i)$$

$$\frac{\partial L}{\partial o_i} = -\sum_k y_k \frac{\partial log(p_k)}{\partial o_i}$$

$$= -\sum_k y_k \frac{\partial log(p_k)}{\partial p_k} \times \frac{\partial p_k}{\partial o_i}$$

$$= -\sum y_k \frac{1}{p_k} \times \frac{\partial p_k}{\partial o_i}$$

From derivative of softmax we derived earlier,

$$\frac{\partial L}{\partial o_i} = -y_i(1 - p_i) - \sum_{k \neq i} y_k \frac{1}{p_k}(-p_k . p_i)$$

$$= -y_i(1 - p_i) + \sum_{k \neq 1} y_k . p_i$$

$$= -y_i + y_i p_i + \sum_{k \neq 1} y_k . p_i$$

$$= p_i \left(y_i + \sum_{k \neq 1} y_k \right) - y_i$$

$$= p_i \left(y_i + \sum_{k \neq 1} y_k \right) - y_i$$

y is a one hot encoded vector for the labels, so $\sum_k y_k = 1$, and $y_i + \sum_{k \neq 1} y_k = 1$. So we have,

$$\frac{\partial L}{\partial o_i} = p_i - y_i$$

＊Choosing parameters

Number of hidden units:

Part A. Two Layer with Sigmoid activation functon

	1	**3**	**5**	**10**
Test Accuracy	29.99%	98.66%	99.33%	99.33%

I choose 5 as my hidden units size.

Part B. Three Layer with Sigmoid activation functon

	1	**3**	**5**	**10**
Test Accuracy	29.99%	98.33%	99.33%	98.99%

I choose h1=5, h2=5 as my hidden units size.

Part C. Three Layer with Relu activation functon

	1	**3**	**5**	**10**
Test Accuracy	35%	97%	98%	98%

I choose h1=5, h2=5 as my hidden units size.

Numbers of Epochs:

Part A. Two Layer with Sigmoid activation functon

	10	**100**	**1000**	**3000**
Test Accuracy	99.33%	98.33%	98.66%	99.33%

I choose 3000 as my number of epochs.

Part B. Three Layer with Sigmoid activation functon

	10	**100**	**1000**	**3000**

Test Accuracy	29.99%	97.99%	99.33%	98.99%

I choose 3000 as my number of epochs.

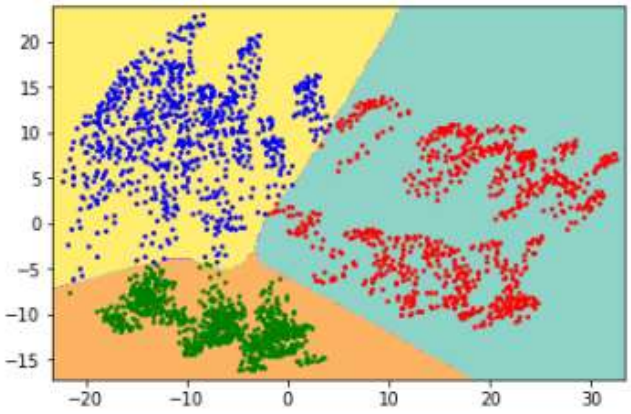
Part C. Three Layer with Relu activation functon

	10	**100**	**1000**	**3000**
Test Accuracy	84%	95%	96%	98%

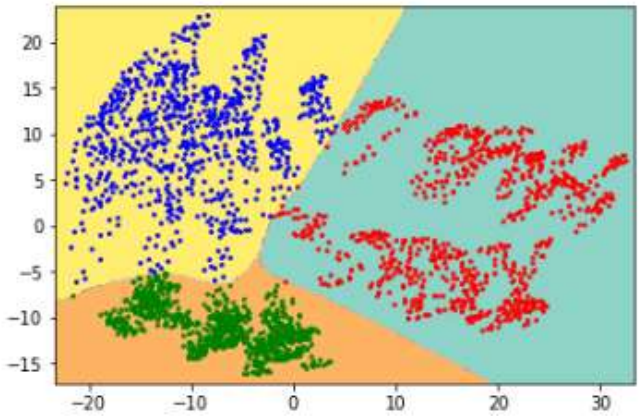
I choose 3000 as my number of epochs.

3. **Testing Result**

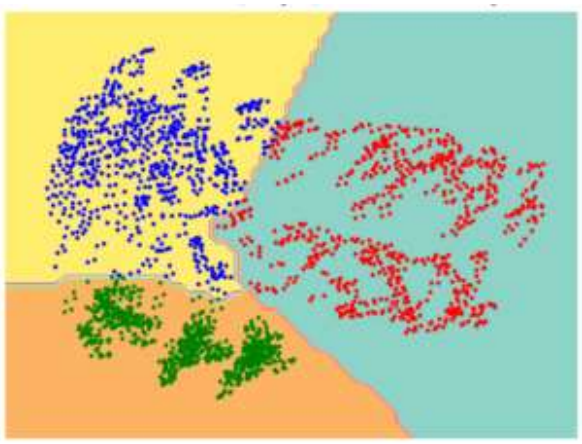
Part A. Accuracy: 99.33%



Part B. Accuracy: 98.99%



Part C. Accuracy: 98%



4. **Problems I have encountered**

＊Relu activation function will put <0 part to zeros, while it may make the divider becomes zeros.

Solution: Assign smaller learning rate. When the learning rate is 1e-3, the loss becomes NAN quickly. After I change to 1e-8, the gradient is smaller and loss can be counted.

＊Division in Python 2 has some unknown bugs so that I get weird results, Python 3 fixs these problems. Therefore, I writes 'from __future__ import division' in the front of the scripts.

＊In the backpropogation process, what we feed to Prime function is different. When we choose Sigmoid as activation function, we feed the activated layer; while if we choose Relu, we feed the non-activated layer instead.

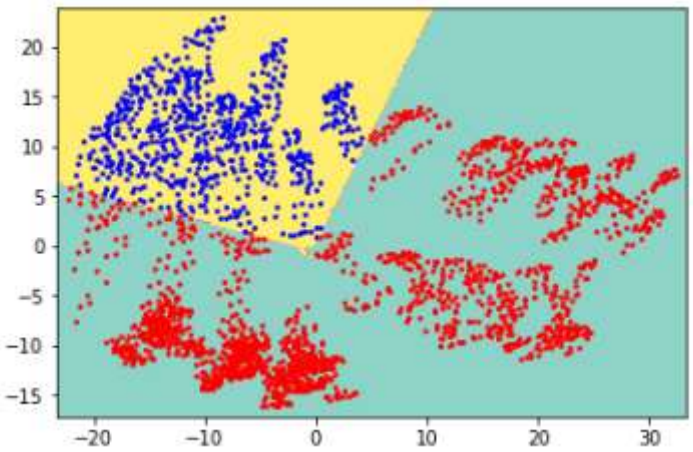
5. **Discussion**

＊Methods of Normalization:

(因為這部份太重要了，我決定用中文解釋)

我原本的 Normalization 是將 x = x-mean 後再除以 std，這種方式在標準化以後數值不會介於 0~1 之間(最大值甚至有到三位數)，當激活函數時 Sigmoid 不會產生問題，因為該函數的輸出 bound 在 0~1 之間。

然而，當我們用 relu，因為 x = (x - mean) / std 後的數值還是很大，丟進 NN 會導致梯度爆炸，整個 module train 不起來，一開始我以為只是 learning rate 有問題，所以我將 LR 調小，狀況雖有所改善但分類結果還是很奇怪，從 jupyter notebook 可以看到只會分出兩類，這就是 module 沒 train 起來導致的 (acc 只有 66%，也就是 2/3，有一類完全消失)

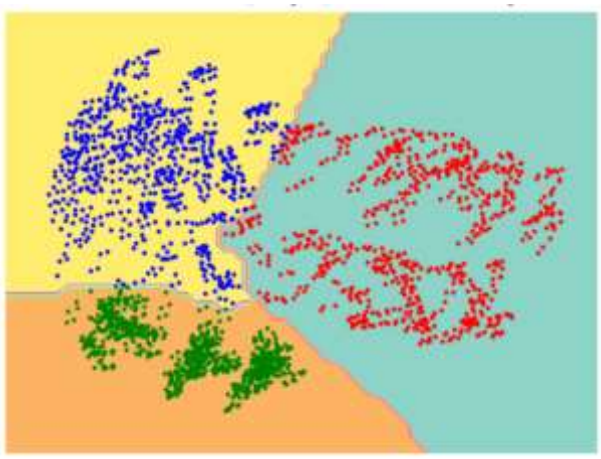


解決方式，使用另一種 normalize 方法，看根據 data 的最大最小值重新 rescale，把所有 data bound 在 0~1 之間，方法如下所示，結果為 accuracy=98%，這才是較正常且符合預期的結果!!!!!!!!

```
def normalize_preliminary(data):
    dimension = data.shape[1]
    x_max = data.max(axis=0)
    x_min = data.min(axis=0)
    return x_max, x_min

def normalize_dataset(data, x_max, x_min, scaling):
    normalised_data = (data - x_min) / (x_max - x_min) * scaling
    return normalised_data
```

Accuracy: 98%



＊Parameters choosing

As we can see from the chart in testing results aboved, the accuracy doesn't necessarily increase if we gives higher parameters. The model has limit and cannot reach 100% accuracy, but it is ideal enough that I get 98~99% accuracy.

The parameters cannot be too small either, because the model will be too simplistic to simulate data. But we don't need to use an over-complicated model in that it doesn't guarantee increasing accuracy, and it will also makes the model more complex, too.

＊Two-layer VS. Three-layer

Both have good performance in this dataset. Sometimes two-layer even behaves better than three-layer. I think the reason is that the dataset teacher gave to us is friendly, so the accuracy can reach 99% in two kinds of model and have only little differences.

But most of the time, three-layer will result better performance than two-layer does.

＊Sigmoid VS. Relu

Sigmoid activation function: non-linear, not blowing up activation

Relu activation function: linear, not vanishing gradient

Pros of Sigmoid:

Forward and backward propagations consist of a series of matrix operations. The decision boundary in Part C is not that smooth as in Part B. The reason is that Sigmoid makes ouput smoother, so the decision is smoother, too.

Pros of Relu:

Gradient of the ReLu function is either 0 for a<0 or 1 for a>0. That means that we can put as many layers as we like, and the model will not suffer from gradient vanishing.

The other benefit of ReLUs is sparsity. Sparsity arises when a≤0. The more such units that exist in a layer the more sparse the resulting representation. Sigmoids on the other hand are always likely to generate some non-zero values resulting in dense representations. Sparse representations seem to be more beneficial than dense representations.

(ref: https://stats.stackexchange.com/questions/126238/what-are-the-advantages-of-relu-over-sigmoid-function-in-deep-neural-networks)

References:

Plotting: http://scikit-learn.org/stable/auto_examples/linear_model/plot_sgd_iris.html

Softmax cross entropy loss:

https://deepnotes.io/softmax-crossentropy

https://deepnotes.io/softmax-crossentropy

NN from scratch: https://github.com/dennybritz/nn-from-scratch/blob/master/nn-from-scratch.ipynb