Deep Learning Course (980)

Assignment 1: Conceptual Exercises

Vignesh V
301383459

---

1) Least Squared Error & Cross Entropy Error

1.) Least Squared Error

$$\text{Error}(E) = \frac{1}{2m} \sum_{i=1}^{m} (Wx_i - y_i)^2.$$

Taking derivative

$$\frac{\partial E}{\partial w} = \frac{\partial}{\partial w}\left[\frac{1}{2}(Wx - y)^2\right].$$

$$= (\cancel{2})\left(\frac{1}{\cancel{2}}\right)\left[Wx - y\right]\left(\frac{\partial}{\partial w}(Wx - y)\right)$$

$$= (Wx - y)\frac{\partial}{\partial w}(Wx - y)$$

$$\boxed{\frac{\partial E}{\partial w} = (Wx - y)(x).} = \underline{(\sigma\hat{y} - y)(x)}.$$

2.) Cross - Entropy Error.

$$CE = -y\log(\hat{y}) - (1-y)\log(1-\hat{y}).$$

where y = target vector $\hat{y}$ = output vector.

Taking derivative

$$\frac{\partial E}{\partial w} = \frac{-y}{\cancel{\sigma \hat{y}}} \cdot \cancel{\sigma \hat{y}}(1-\sigma \hat{y}) \cdot \frac{\partial}{\partial w}(y) - \frac{(1-y)(x)}{\cancel{(1-\sigma \hat{y})}}\left(-\sigma \hat{y}\,\cancel{(1-\sigma \hat{y})}\right)\frac{\partial}{\partial w}(y)$$

After cancellation, we get

$$= -y(1-\sigma \hat{y}) \cdot \frac{\partial}{\partial w}(y) - (1-y)(x)(-\sigma \hat{y})\left(\frac{\partial}{\partial w}(y)\right)$$

$$= -xy(1-\sigma \hat{y}) + (1-y)(\sigma \hat{y} \cdot x).$$ we took the derivative of y where $\boxed{y = wx}$ is x.

$$= x\left[-y(1-\sigma \hat{y}) + (1-y)\sigma \hat{y}\right].$$

$$= x\left[-y + \cancel{\sigma \hat{y}(y)} + \sigma y - \cancel{\sigma \hat{y}(y)}\right]$$

$$= x\left[-y + \sigma \hat{y}\right].$$

Therefore, $$\frac{\partial E}{\partial w} = x(-y + \sigma \hat{y}).$$

Incorporating gradients in Backpropagation Algorithm

Step 1: From the above derivative that we have derived, we now got the gradient for the output layer.
Step 2: From the output gradient, using backpropagation we can obtain the gradient for the previous layers for each hidden node.
Step 3: Perform the gradient descent updates for each weight to obtain the updated weight vector. In the below expression we can see that the gradient value is needed to get these updated weights.

$$w_{ij} \leftarrow w_{ij} + \alpha \times a_i \times \Delta[j]$$

Hence, we can conclude that it is possible to update the weights through back propagation through these gradients.

2. BROADCASTING

② BROADCASTING

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} 4 & 7 \\ 8 & 15 \end{pmatrix} + \begin{pmatrix} 4 & 5 \end{pmatrix}$$

Assuming broadcasting $= \begin{pmatrix} (4+4) & (7+5) \\ (8+4) & (15+5) \end{pmatrix}$

$$= \begin{pmatrix} 8 & 12 \\ 12 & 20 \end{pmatrix}$$

3. TRACE BACKPROPAGATION

③ TRACE BACKPROPAGATION

Given $W_{ac} = 0.1$ $\quad W_{cd} = 0.1$

$W_{bc} = 0.1$ $\quad W_{od} = 0.1$

$W_{oc} = 0.1$

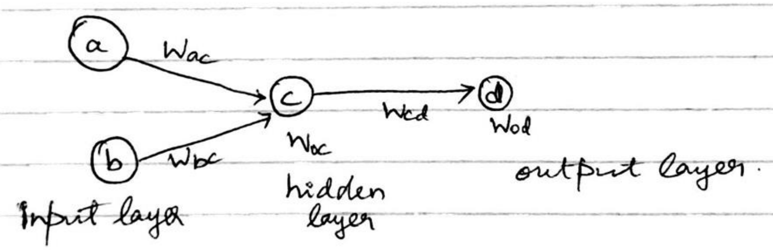
$a =$ input1 $\quad b =$ input2 $\quad$ hidden unit $= c$

Assumption $\quad \eta = 0.2$ $\quad$ output $= d$.

Activation function = sigmoid.

3.1 Computing the quantities for node x.

Graph as per the given network



① For any node x

$$a_x = g(in_x) = g\left(\sum_i W_i \times a_i\right) = \sigma(W_{ax} \times a).$$

② $\Delta[x] = g'(in_x)\,\Delta y$ considering output $= y$.

$$\Delta[x] = (y_x - a_x)\, a_x\, (1 - a_x).$$

③ $W_{xy} = W_{xy} + \alpha\left[\Delta W_{xy}\right]$ $\quad \Delta W_{xy} = \Delta y \times a_x.$

where $\Delta y = a_y(1 - a_y)(y - a_y).$

3.2 Datapoints calculation.

2. At datapoint x_1, we derive the below

1) $a_c = \sigma(W_{ac}(a) + W_{bc}(b) + W_{oc})$

Here we know that at datapoint x_1

$a = 1 \quad b = 0 \quad d = 1.$

$\therefore a_c = \sigma(0.1 + 0 + 0.1) = \sigma(0.2)$

Have taken sigmoid function here to calculate the value $\sigma(x) = \frac{1}{1+e^{-x}}$

After calculating $\sigma(0.2) \Rightarrow \boxed{a_c = 0.54983}$

2) $a_d = \sigma($weight of i/p of $d \times$ activation$_c + W_{od})$

$= \sigma(W_{cd} \times a_c + W_{od})$

$= \sigma(0.1 \times 0.54983 + 0.1)$

$= \sigma(0.154983) = 0.53867$

$\boxed{a_d = 0.53867}$

(3) $\Delta d = (y - a_d)$ (ie) $\Delta d = g'(in_d)\, \Delta y$

Since we consider output $y = 1$.

$$\therefore \Delta d = a_d(1 - a_d)(y - a_d)$$

$$= (0.53867)[1 - 0.53867][1 - 0.53867]$$

$$= 0.24850 \times 0.46133$$

$$\boxed{\Delta d = 0.114642}$$

(4) $\Delta c = a_c(1 - a_c)\, W_{cd} \times \Delta d.$

$$= (0.54983)[1 - 0.54983](0.1)(0.114642)$$

$$\boxed{\Delta c = 0.00283}$$

(5) $W_{oc} = W_{oc} + \eta(\Delta W_{oc})$ we consider (+)ve over here by multiplying both the (-ve) factors for step size & gradient descent (ie)

$$W_{oc} = W_{oc} - \eta(-\Delta W_{oc}) = W_{oc} + \eta(\Delta W_{oc}).$$

$\therefore W_{0c} = W_{0c} + \eta(\Delta W_{0c})$

$\Delta W_{0c} = \Delta c \times 1 = 0.00283$

$W_{0c} = (0.1) + (0.2)(0.00283)$

$\boxed{W_{0c} = 0.10056}$

(6) $W_{bc} = W_{bc} + \eta(\Delta W_{bc})$.

$\Delta W_{bc} = \Delta c \times b$. we know that $b = 0$

for datapoint x_1.

$\therefore \boxed{W_{bc} = 0.1}$

(7) $W_{cd} = W_{cd} + \eta(\Delta W_{cd})$

$\Delta W_{cd} = \Delta d \times a_c = (0.114642 \times 0.54983)$

$W_{cd} = (0.1) + (0.2)(0.114642 \times 0.54983)$

$\boxed{W_{cd} = 0.112606}$

8.) $W_{od} = W_{od} + \eta(\Delta W_{od})$

$\Delta W_{od} = \Delta d \times 1 = 0.114642.$

$W_{od} = (0.1) + (0.2)(0.114642)$

$\boxed{W_{od} = 0.12292}$

9.) $W_{ac} = W_{ac} + \eta(\Delta W_{ac})$

$\Delta W_{ac} = \Delta c \times a = 0.00283.$

$W_{ac} = (0.1) + (0.2)(0.00283) \Rightarrow \boxed{W_{ac} = 0.10056}$

At the end of x_1 datapoint the table looks like below :-

At Datapoint x_2, we take the values from the above table

1.) $a_c = \sigma(W_{ac}(a) + W_{bc}(b) + W_{oc})$ since $a = 0$ in this scenario

$a_c = \sigma(0 + 0.1 + 0.10056)$

$\boxed{a_c = 0.54997}$

2.) $a_d = \sigma(W_{cd} \times a_c + W_{od})$

$= \sigma(0.112606 \times 0.54997 + 0.12292)$

$= \sigma(0.1848499218)$

$\boxed{a_d = 0.54608}$

3.) $\Delta c = a_c(1 - a_c) W_{cd} \times \Delta d.$

$= (0.54997)(1 - 0.54997)(0.112602)(-0.135360)$

$\boxed{\Delta c = -0.00377}$

4.) $\Delta d = a_d(1 - a_d)(y - a_d) = (0.54608)[1 - 0.54608][0 - 0.54608]$

$\Delta d = -0.2982033604 \times 0.45392$

$\boxed{\Delta d = -0.135360}$

5.) $W_{ac} = W_{ac} + \eta \Delta W_{ac}$

$\Delta W_{ac} = \Delta c \times a = 0.$ $\therefore$ $\boxed{W_{ac} = 0.10056}$

6.) $W_{bc} = W_{bc} + \eta \Delta W_{bc}$

$\Delta W_{bc} = \Delta c \times b = -0.00377$

$W_{bc} = (0.1) + (0.2)(-0.00377)$

$W_{bc} = 0.1 - 0.000754$

$\boxed{W_{bc} = 0.09623}$

7.) $W_{cd} = W_{cd} + \eta \Delta W_{cd}$

$\Delta W_{cd} = \Delta d \times a_c = -0.135360 \times 0.54997$

$\boxed{\Delta W_{cd} = -0.07444}$

$\therefore W_{cd} = (0.112606) + (0.2)(-0.07444)$

$W_{cd} = 0.112606 - 0.014888$

$\boxed{W_{cd} = 0.09771}$

8.) $W_{od} = W_{od} + \eta (\Delta W_{od})$

$\Delta W_{od} = \Delta d \times 1 = -0.135360$

$W_{od} = 0.12292 + (0.2)(-0.135360) = 0.12292 - 0.027072$

$\boxed{W_{od} = 0.095848}$

9.) $W_{0c} = W_{0c} + \eta(\Delta W_{0c})$

$\Delta W_{0c} = \Delta c \times 1 = -0.00377.$

$W_{0c} = 0.10056 + (0.2)(-0.00377)$

$W_{0c} = 0.10056 - 0.000754$

$\boxed{W_{0c} = 0.099806}$

The final output table after 2 iterations looks as below: -

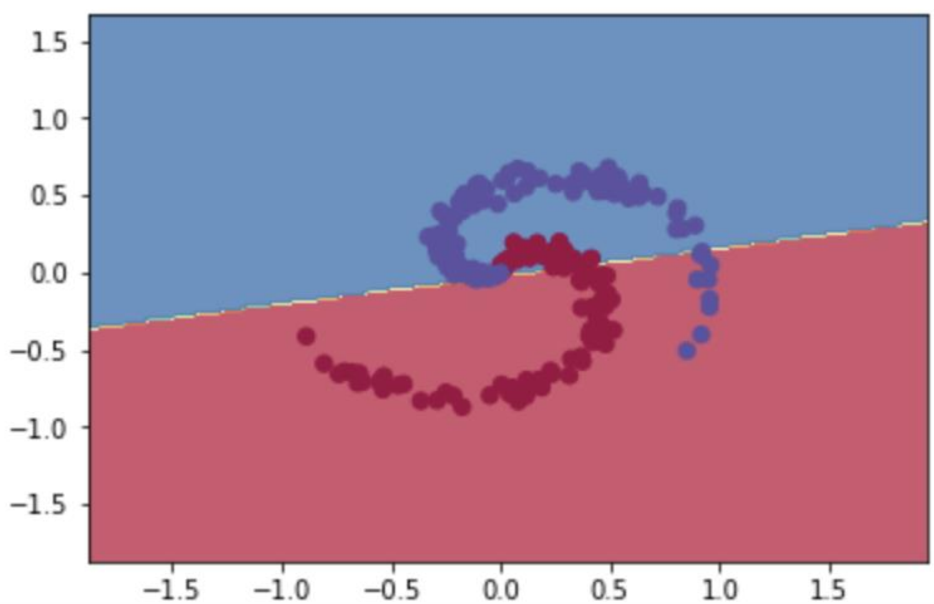
Datapoint	ac	delta_c	ad	delta_d	w0c	wac	wbc	wcd	w0d
x1	0.54983	0.00283	0.53867	0.114642	0.10056	0.10056	0.1	0.112606	0.12292
x2	0.54997	-0.00377	0.54608	-0.13536	0.099806	0.10056	0.09623	0.09771	0.095848

ASSIGNMENT ONE

1. Install Tensorflow & Jupyter Notebook. Implement Linear Regression.

- TensorFlow was installed and all the code implementation is done via Jupyter notebook. The code is implemented and saved under "**Assignment_1_301383459**" folder as ipynb file.
- When tried to run the code which was already implemented, the received accuracy was around 61 %. This is because **bias** was not been added in the layer. A bias value allows us to fit the data much better.
- After **adding the bias**, the data seems to fit better with an accuracy of around 75%. This is a success for us because, we can see that there is an increase in accuracy only after we added bias. Bias is one of the reasons to change the accuracy.

```
Epoch: 97 loss: 0.140 acc: 0.750
Epoch: 98 loss: 0.140 acc: 0.750
Epoch: 99 loss: 0.140 acc: 0.750
Epoch: 100 loss: 0.140 acc: 0.750
```

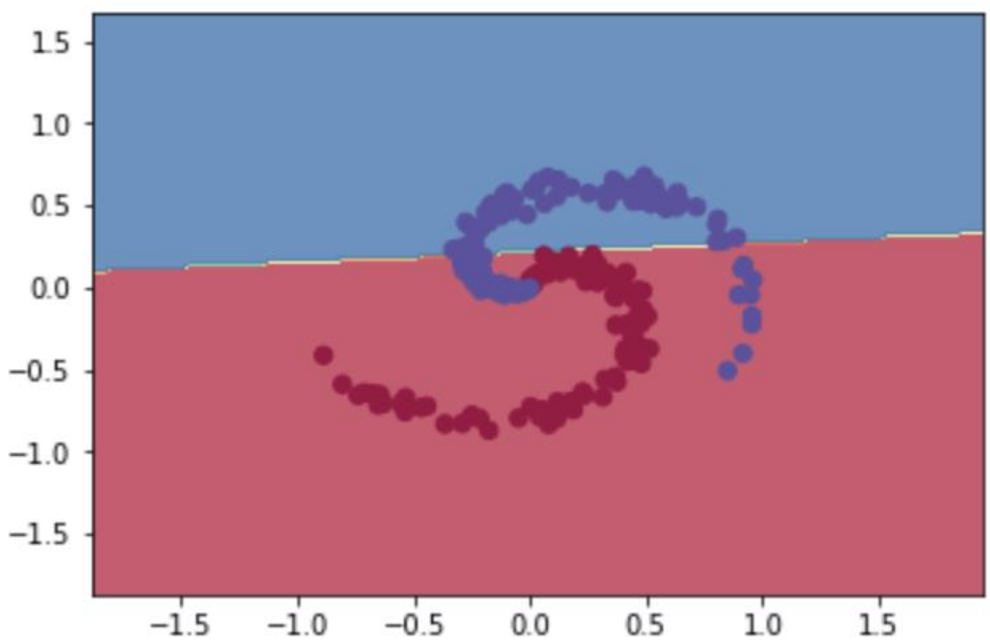


2. Implement Logistic Regression.

- After the layer is initialised, have used **sigmoid** as the activation function and the correct loss function is "**cross entropy**". tf.nn.sigmoid_cross_entropy_with_logits() function does the magic for us by passing the logits and labels.

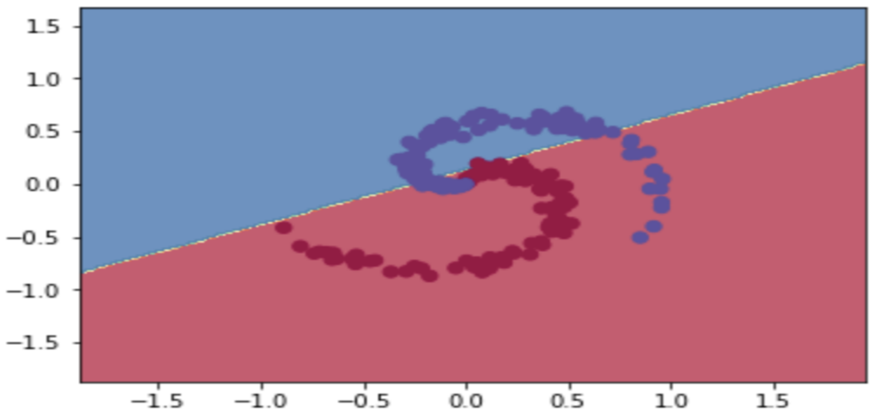
- The below curve is better compared to our previous linear regression and also there is an increase in accuracy. This proves us that using the correct loss function helped us improve the accuracy.

```
Epoch: 0 loss: 0.667 accuracy = 0.750
Epoch: 500 loss: 0.641 accuracy = 0.780
```



- **Fine tuning improved the accuracy**. I tried changing the no of epochs to 10000 and changing the learning rate to 0.9 provided better results with increase in accuracy.

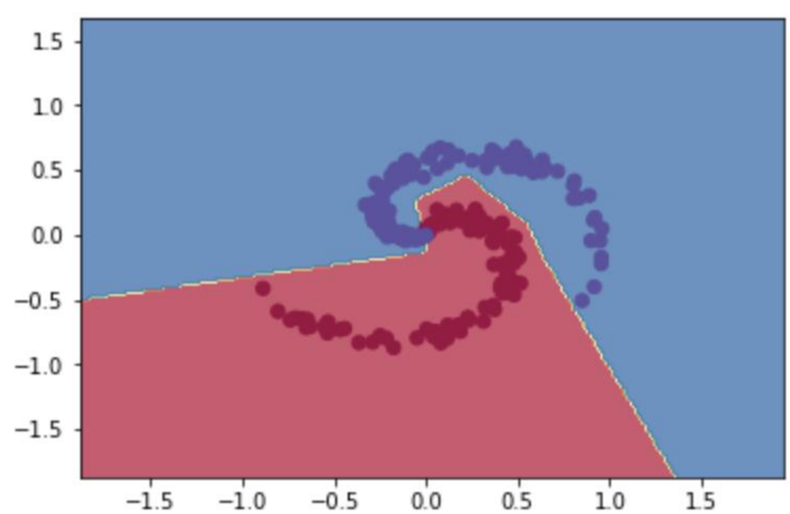
```
Epoch: 6000 loss: 0.587 accuracy = 0.810
Epoch: 6500 loss: 0.587 accuracy = 0.810
Epoch: 7000 loss: 0.586 accuracy = 0.815
Epoch: 7500 loss: 0.586 accuracy = 0.815
Epoch: 8000 loss: 0.585 accuracy = 0.815
Epoch: 8500 loss: 0.585 accuracy = 0.815
Epoch: 9000 loss: 0.585 accuracy = 0.815
Epoch: 9500 loss: 0.584 accuracy = 0.815
```



2. Multi-layer perceptron without Keras

- For the multi-layer perceptron, the weights and bias are initialised for three hidden layers.
- **Relu** activation function is implemented in each of the hidden layer and the output is sigmoid activation with cross entropy loss.
- Cross entropy loss is calculated using the cross-entropy error function
  loss = tf.reduce_mean(-Y*tf.math.log(sigmoid(out)) - (1-Y)* tf.math.log(1-sigmoid(out)))
- To improve the accuracy, Adam Optimizer is used with a learning rate of 0.1.

```
Epoch: 1998 loss: 0.015 acc: 0.995
Epoch: 1999 loss: 0.015 acc: 0.995
Epoch: 2000 loss: 0.015 acc: 0.995
```



- 99.5% accuracy is achieved. However further fine tuning and regularization (maybe) can lead us to 100% accuracy.

4. Multi-layer perceptron with Keras

- Keras implementation is done using the keras **Sequential** and using **Dense** layer function.
- Three Dense hidden layers with **relu** activation are added to the model and the output Dense layer with **sigmoid** activation is implemented in the model.

```
Epoch: 0 loss: 0.021 accuracy = 0.995000004768
Epoch: 1 loss: 0.018 accuracy = 0.995000004768
Epoch: 2 loss: 0.019 accuracy = 0.995000004768
Epoch: 3 loss: 0.025 accuracy = 0.995000004768
Epoch: 4 loss: 0.163 accuracy = 0.990000009537
Epoch: 5 loss: 0.018 accuracy = 0.995000004768
Epoch: 6 loss: 0.023 accuracy = 0.995000004768
Epoch: 7 loss: 0.020 accuracy = 0.995000004768
Epoch: 8 loss: 0.018 accuracy = 0.995000004768
Epoch: 9 loss: 0.020 accuracy = 0.995000004768
```

- 99.5% accuracy is achieved. However further fine tuning and regularization (maybe) can lead us to 100% accuracy.