Deep Learning and Practice

Lab1: Backpropagation

H092093 林佳縈

1. Introduction

In Lab1, the target is to train a neural network with two hidden layers, which is thus able to do binary classification on both linear and non-linear datasets.

The following computational graph demonstrates my neural network structure.

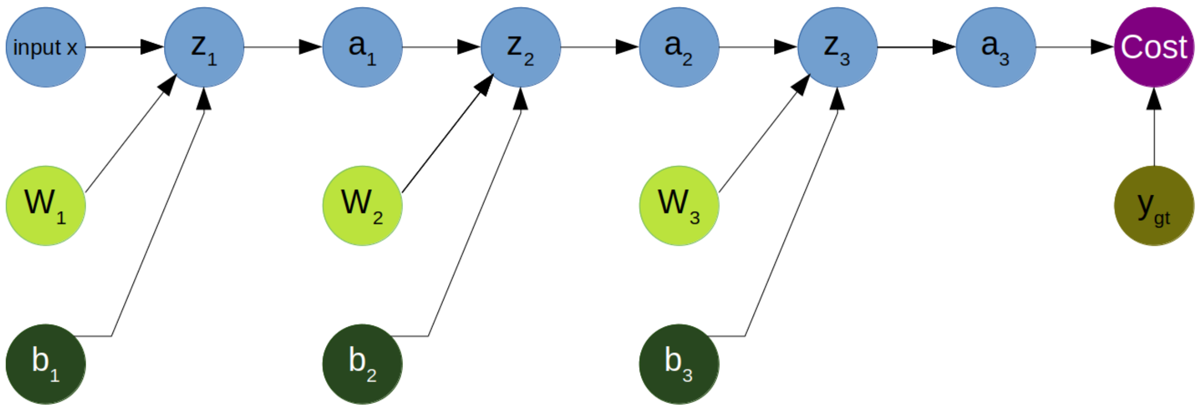


Figure 1-1: Computational Graph for 2-layer hidden neural network.

Here's how the graph works. I'd like to introduce the whole process in 3 sequential steps. Let's take the ***linear classification problem*** for example.

Step1: Obtain temporary output Z

In this step, we'll perform matrix multiplication on input matrix and weight matrix and finally add the bias onto it. The calculation is as followed:

$$Z_1 = (input\ x) * W_1 + b_1$$

$$Z_2 = (a_1) * W_2 + b_2$$

$$Z_3 = (a_2) * W_3 + b_3$$

Z represents the temporary output matrix derived from the multiplication results of each layer's input and weight plus the corresponding bias.

Step2: Obtain activated output a

After obtaining the temporary output Z, to introduce non-linearity into the output of neurons (the Zs), we have to calculate the activated output *a* by placing the previously obtained Z into activation function. In this lab, I chose **sigmoid** to perform activation. For sigmoid functions' further details, I'd introduce in Section II: Experiment Setups.

Step3: Calculate distance between ground-truth data and predicted data

After the final output a_3 is successfully calculated, the last step is to calculate the distance between ground-truth data y_{gt} and a_3, the loss. For loss function choice, I have binary cross-entropy to perform loss computation.

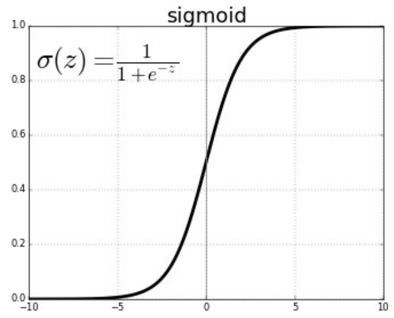
2. Experiment Setups:

A. Sigmoid Functions

The mathematical expression for sigmoid function is as followed:

$$\sigma(x) = 1/1+exp(-x)$$

which could be plot as this graph:



In forward propagation, the sigmoid functions:

```
def sigmoid(self, x):
        return 1./(1. + np.exp(-x))
```

[Tipps] During implementation, at first, I used python's ***math*** package's ***math.exp()***, but failed. The reason is because the package math can only support size-1 arrays, while the training dataset is normally an n-dimensional array.

B. Neural Network

This is my neural network structure. There are:

- 1 input layer: 2 pink neurons (x0, x1)
- 2 hidden layers: 3 green neurons for each (N1, N2)
- 1 output layer: 1 blue neuron (y)

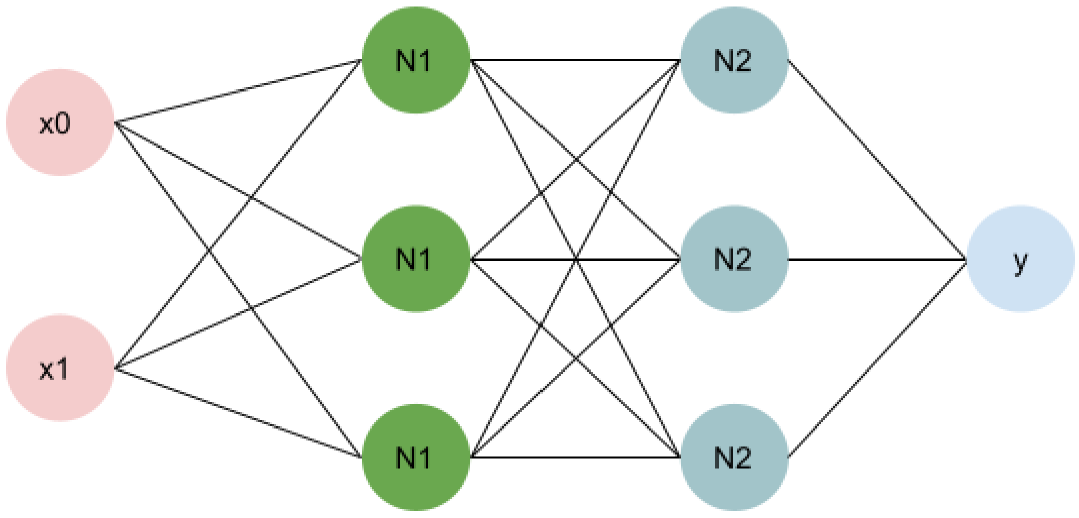


Figure 2-1: Neural Network structure, one input layer, two hidden layers and one output layer.

For how I determine the hidden layer neurons, I've actually tried many different numbers of neurons possibilities. First I set the hidden layers' neurons (1,1), but found the performance is really awful; later on, I set (20,7), the learning result is pretty well, but it seems that to obtain good performance on the given datasets, it's no need to use that many of neurons. So I calculate the mean value of the input and output neurons to have it as my hidden layer neurons quantity. In Lab1's case, it would be (3,3)

hidden layer neurons = mean(input neurons/output neurons)

C. Backpropagation

According to figure 1-1, here is how I calculate backpropagation. Since the most convenient characteristic during implementing backpropagation is the calculated value can be reused.

$$\frac{dC}{da_3}$$

$$\frac{dC}{dZ_3} = \frac{dC}{da_3}\,\frac{da_3}{dZ_3}$$

$$\frac{dC}{da_2} = \frac{dC}{dZ_3}\,\frac{dZ_3}{da_2}$$

$$\frac{dC}{dW_3} = \frac{dC}{dZ_3}\,\frac{dZ_3}{dW_3}$$

$$\frac{dC}{db_3} = \frac{dC}{dZ_3}\,\frac{dZ_3}{db_3}$$

From figure 2-2 on the left-hand side, we can see that once the derivative of C over a_3 is calculated, the derivative of C over Z_3 can thus be derived by only calculating the derivative of a_3 over Z_3. Therefore, the red-highlighted parts of figure 2-2 are the parts requiring no re-calculation, which makes the back propagation relatively easy.

Figure 2-2: Backprop

During implementation, I calculated them as below (take the last layer for example):

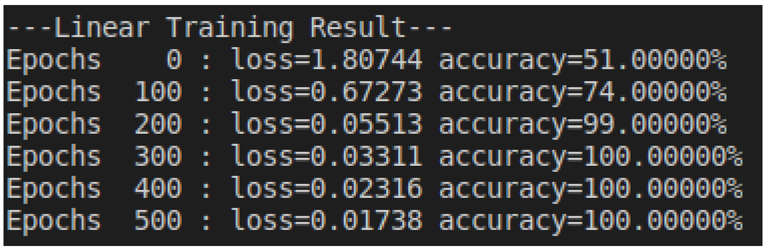
```
self.grada[3] = -(np.divide(y, pred_y+self.eps) - np.divide(1-y,
1-pred_y+self.eps))

self.gradZ[3] = np.multiply(self.grada[3], self.derivative_sigmoid(self.a[3]))

self.gradW[3] = np.matmul(self.gradZ[3], self.a[2].T)*(1/batch_size)

self.gradb[3] = np.sum(self.gradZ[3], axis=1, keepdims=True)*(1/batch_size)
```

[Tipps] Since there's sometimes divide-by-zero error appears, so I add a tiny value, denoted as eps (ε, value=1e-8) to avoid this problem.

3. Results of my testing

A. Screenshot and comparison figure

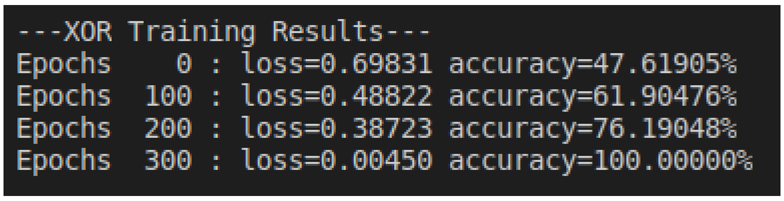
- ***Training Session: print loss value (print after every 100 epochs)***
    - Linear:



```
---Linear Training Result---
Epochs    0 : loss=1.80744 accuracy=51.00000%
Epochs  100 : loss=0.67273 accuracy=74.00000%
Epochs  200 : loss=0.05513 accuracy=99.00000%
Epochs  300 : loss=0.03311 accuracy=100.00000%
Epochs  400 : loss=0.02316 accuracy=100.00000%
Epochs  500 : loss=0.01738 accuracy=100.00000%
```

Figure 3A-1: linear training result

    - XOR:



```
---XOR Training Results---
Epochs    0 : loss=0.69831 accuracy=47.61905%
Epochs  100 : loss=0.48822 accuracy=61.90476%
Epochs  200 : loss=0.38723 accuracy=76.19048%
Epochs  300 : loss=0.00450 accuracy=100.00000%
```

Figure 3A-2: XOR training result

Since I add some conditions for early stopping on both the Linear and XOR training process, so the screenshots are with different stopping epochs numbers. The early stopping conditions are:

- **for linear:** if the loss is less than linear_threshold(0.015) up to 3 times, terminate.
- **for XOR:** if the loss is less than xor_threshold(0.001) up to 3 times, terminate.

- ***Comparison Figure***
  - Linear

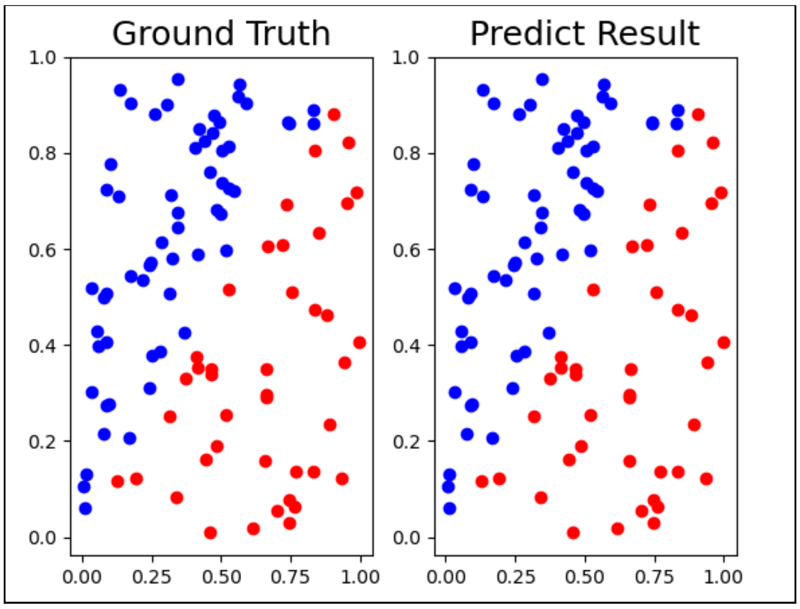


Figure 3A-3: Linear result plots

  - XOR:

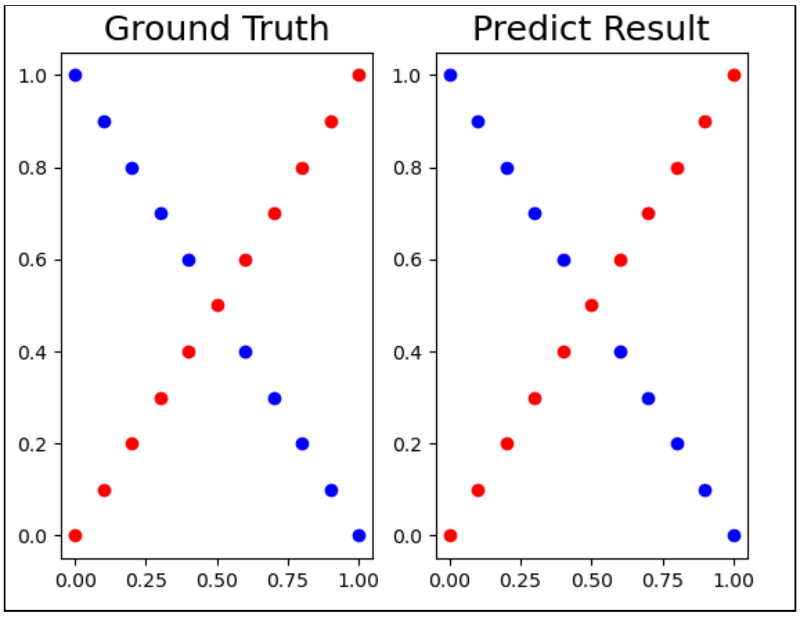
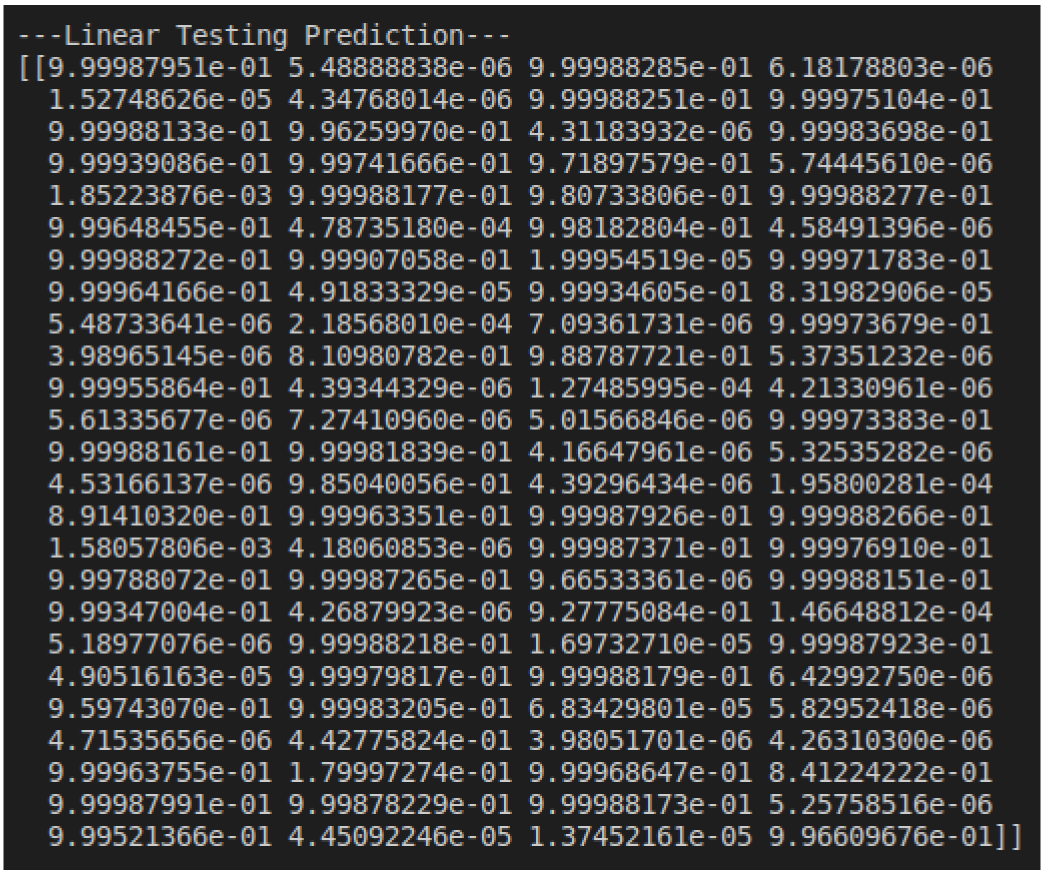


Figure 3A-4: XOR result plots

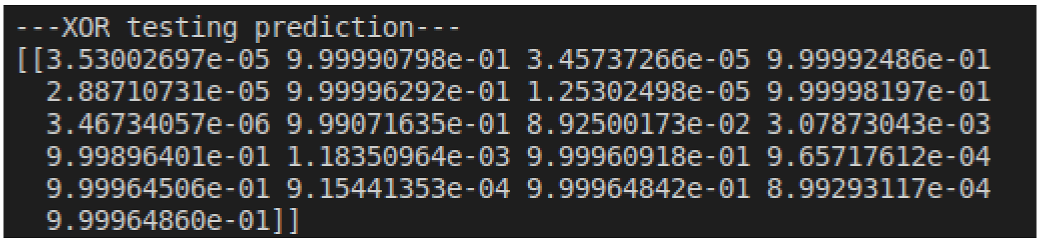
- ***Testing : show the prediction***
    - Linear



```
---Linear Testing Prediction---
[[9.99987951e-01 5.48888838e-06 9.99988285e-01 6.18178803e-06
  1.52748626e-05 4.34768014e-06 9.99988251e-01 9.99975104e-01
  9.99988133e-01 9.96259970e-01 4.31183932e-06 9.99983698e-01
  9.99939086e-01 9.99741666e-01 9.71897579e-01 5.74445610e-06
  1.85223876e-03 9.99988177e-01 9.80733806e-01 9.99988277e-01
  9.99648455e-01 4.78735180e-04 9.98182804e-01 4.58491396e-06
  9.99988272e-01 9.99907058e-01 1.99954519e-05 9.99971783e-01
  9.99964166e-01 4.91833329e-05 9.99934605e-01 8.31982906e-05
  5.48733641e-06 2.18568010e-04 7.09361731e-06 9.99973679e-01
  3.98965145e-06 8.10980782e-01 9.88787721e-01 5.37351232e-06
  9.99955864e-01 4.39344329e-06 1.27485995e-04 4.21330961e-06
  5.61335677e-06 7.27410960e-06 5.01566846e-06 9.99973383e-01
  9.99988161e-01 9.99981839e-01 4.16647961e-06 5.32535282e-06
  4.53166137e-06 9.85040056e-01 4.39296434e-06 1.95800281e-04
  8.91410320e-01 9.99963351e-01 9.99987926e-01 9.99988266e-01
  1.58057806e-03 4.18060853e-06 9.99987371e-01 9.99976910e-01
  9.99788072e-01 9.99987265e-01 9.66533361e-06 9.99988151e-01
  9.99347004e-01 4.26879923e-06 9.27775084e-01 1.46648812e-04
  5.18977076e-06 9.99988218e-01 1.69732710e-05 9.99987923e-01
  4.90516163e-05 9.99979817e-01 9.99988179e-01 6.42992750e-06
  9.59743070e-01 9.99983205e-01 6.83429801e-05 5.82952418e-06
  4.71535656e-06 4.42775824e-01 3.98051701e-06 4.26310300e-06
  9.99963755e-01 1.79997274e-01 9.99968647e-01 8.41224222e-01
  9.99987991e-01 9.99878229e-01 9.99988173e-01 5.25758516e-06
  9.99521366e-01 4.45092246e-05 1.37452161e-05 9.96609676e-01]]
```

Figure 3A-5: Linear prediction value

- XOR



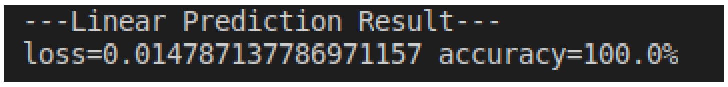
```
---XOR testing prediction---
[[3.53002697e-05 9.99990798e-01 3.45737266e-05 9.99992486e-01
  2.88710731e-05 9.99996292e-01 1.25302498e-05 9.99998197e-01
  3.46734057e-06 9.99071635e-01 8.92500173e-02 3.07873043e-03
  9.99896401e-01 1.18350964e-03 9.99960918e-01 9.65717612e-04
  9.99964506e-01 9.15441353e-04 9.99964842e-01 8.99293117e-04
  9.99964860e-01]]
```

Figure 3A-6: XOR prediction value

B. Show the accuracy of my prediction

- Linear:

The accuracy tested on linear datasets is normally within 99% to 100%.

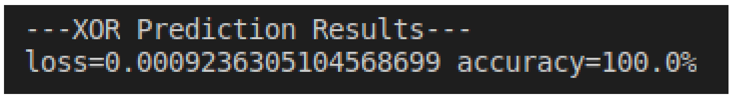


```
---Linear Prediction Result---
loss=0.014787137786971157 accuracy=100.0%
```

Figure 3B-1: Linear prediction result

- XOR:

  For XOR, normally the accuracy is within the range of 95 to 100.



```
---XOR Prediction Results---
loss=0.0009236305104568699 accuracy=100.0%
```

Figure 3B-2: XOR Prediction result

C. Learning curve (loss, epoch curve)

- **loss - epoch:**

  With x-axis standing for numbers of epochs and y-axis the loss value, we can tell from the graph that loss is eventually decreased as iterated over epochs.

  <u>Linear</u>

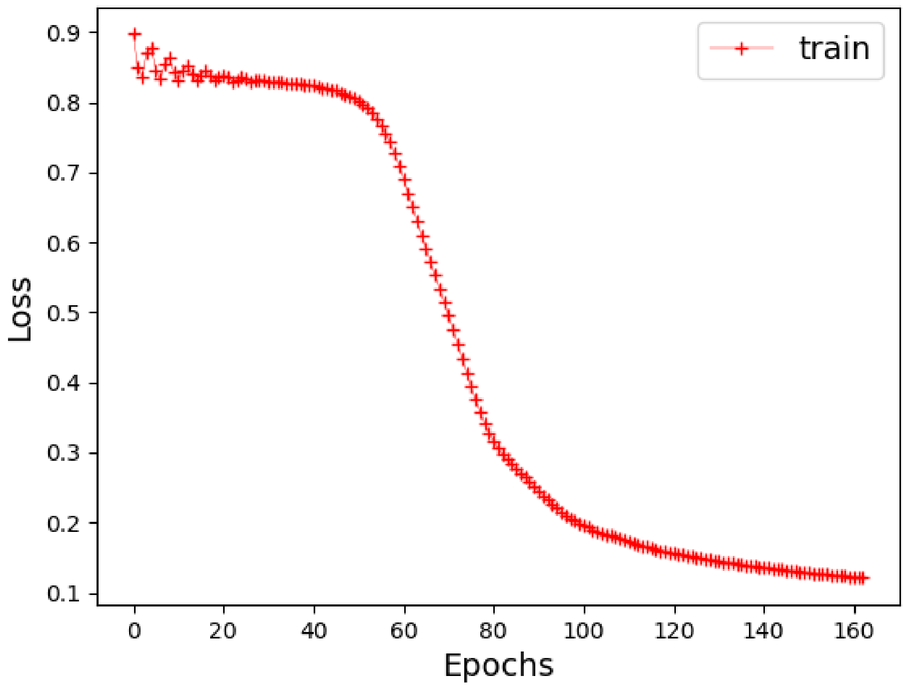


Figure 3C-1: Linear loss vs. epoch

XOR

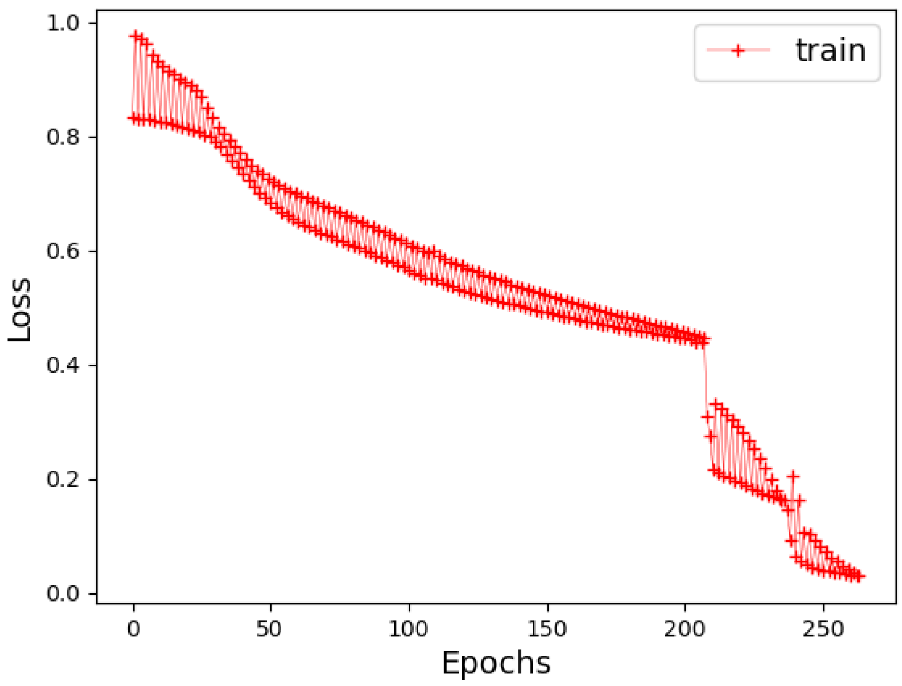
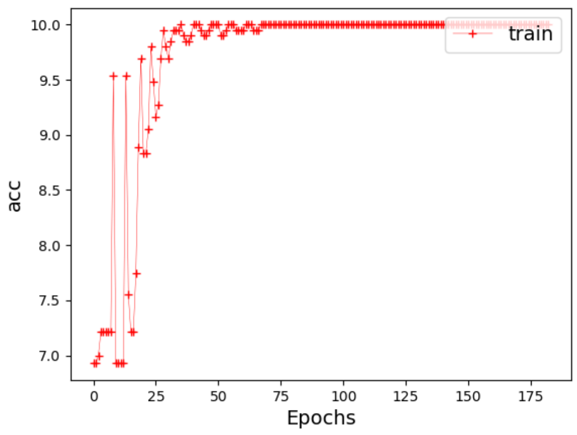


Figure 3C-2: XOR loss vs. epoch

D. Others

accuracy - epoch



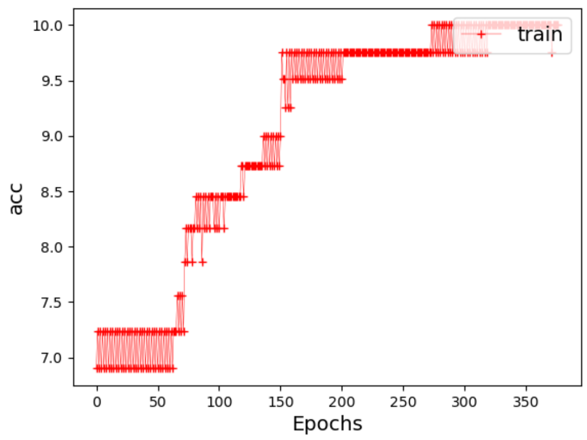


Figure 3C-3: linear acc vs. epoch **Figure 3C-4: XOR acc vs. epoch**

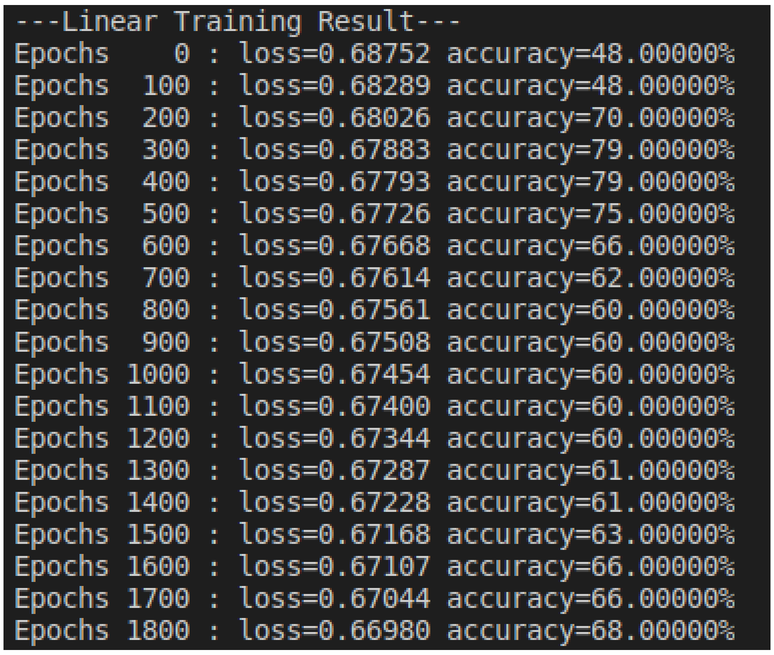
4. Discussion

A. Different Learning Rate

By trying different learning rates, I separated the experiment into two different trials: tiny learning rate and huge learning rate. Also, I've added a learning rate scheduler during training, but for this part, I'll leave it to section D: Others to do a more thorough introduction.

Trial 1: set learning rates a tiny value (0.001) linear case

Experiment setting: lr = 0.001, epochs=100000, LR_scheduler=off



```
---Linear Training Result---
Epochs    0 : loss=0.68752 accuracy=48.00000%
Epochs  100 : loss=0.68289 accuracy=48.00000%
Epochs  200 : loss=0.68026 accuracy=70.00000%
Epochs  300 : loss=0.67883 accuracy=79.00000%
Epochs  400 : loss=0.67793 accuracy=79.00000%
Epochs  500 : loss=0.67726 accuracy=75.00000%
Epochs  600 : loss=0.67668 accuracy=66.00000%
Epochs  700 : loss=0.67614 accuracy=62.00000%
Epochs  800 : loss=0.67561 accuracy=60.00000%
Epochs  900 : loss=0.67508 accuracy=60.00000%
Epochs 1000 : loss=0.67454 accuracy=60.00000%
Epochs 1100 : loss=0.67400 accuracy=60.00000%
Epochs 1200 : loss=0.67344 accuracy=60.00000%
Epochs 1300 : loss=0.67287 accuracy=61.00000%
Epochs 1400 : loss=0.67228 accuracy=61.00000%
Epochs 1500 : loss=0.67168 accuracy=63.00000%
Epochs 1600 : loss=0.67107 accuracy=66.00000%
Epochs 1700 : loss=0.67044 accuracy=66.00000%
Epochs 1800 : loss=0.66980 accuracy=68.00000%
```

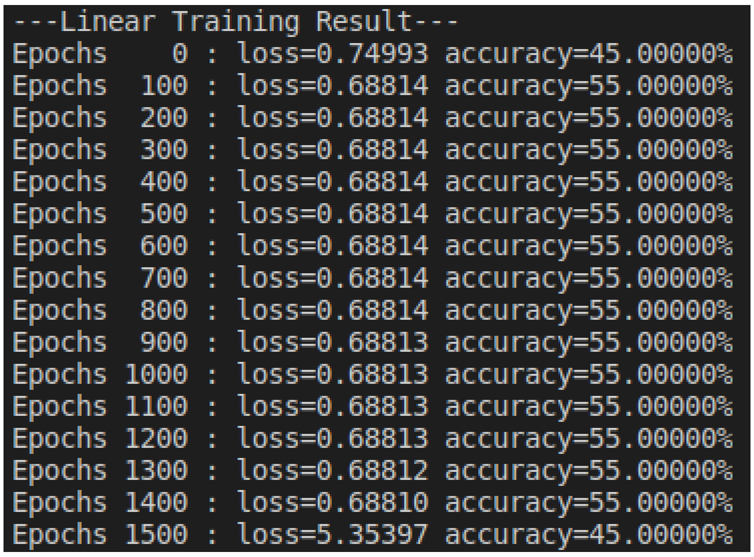
Figure 4A-1: tiny learning rate result

Setting a tiny learning rate, we can yes, there seems to be progress on the accuracy and the loss is also reduced, yet, at a fairly slow pace. Also, in some cases, since I've added a learning rate scheduler, which makes the learning rate decrease as epoch number goes high, which in this case will add negative effects on accuracy progress and the decrease of loss.

Trial 2: set learning rates a huge value (10.0) linear case

Experiment setting: lr = 10.0, epochs=1000, LR_scheduler=off

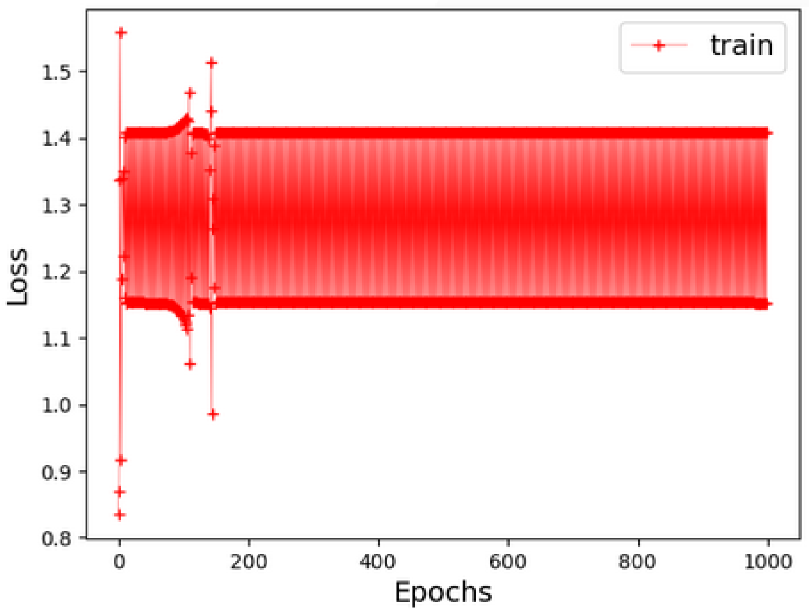
If the learning rate is set too large, the model can't do good gradient descent, which leads to no improvements on accuracy progress and loss decrease.



```
---Linear Training Result---
Epochs    0 : loss=0.74993 accuracy=45.00000%
Epochs  100 : loss=0.68814 accuracy=55.00000%
Epochs  200 : loss=0.68814 accuracy=55.00000%
Epochs  300 : loss=0.68814 accuracy=55.00000%
Epochs  400 : loss=0.68814 accuracy=55.00000%
Epochs  500 : loss=0.68814 accuracy=55.00000%
Epochs  600 : loss=0.68814 accuracy=55.00000%
Epochs  700 : loss=0.68814 accuracy=55.00000%
Epochs  800 : loss=0.68814 accuracy=55.00000%
Epochs  900 : loss=0.68813 accuracy=55.00000%
Epochs 1000 : loss=0.68813 accuracy=55.00000%
Epochs 1100 : loss=0.68813 accuracy=55.00000%
Epochs 1200 : loss=0.68813 accuracy=55.00000%
Epochs 1300 : loss=0.68812 accuracy=55.00000%
Epochs 1400 : loss=0.68810 accuracy=55.00000%
Epochs 1500 : loss=5.35397 accuracy=45.00000%
```

Figure 4A-2: large learning rate result

On the other hand, for the XOR case, if the learning rate is set too big, we can see through the loss-epoch graph that the loss is bouncing relatively severely.



In a nutshell, after these trials, the final decision on the optimal learning rate is 0.8.

B. Different number of hidden layer neurons

Same as the previous section, to do experiments on different numbers of hidden layer, I have the following settings:

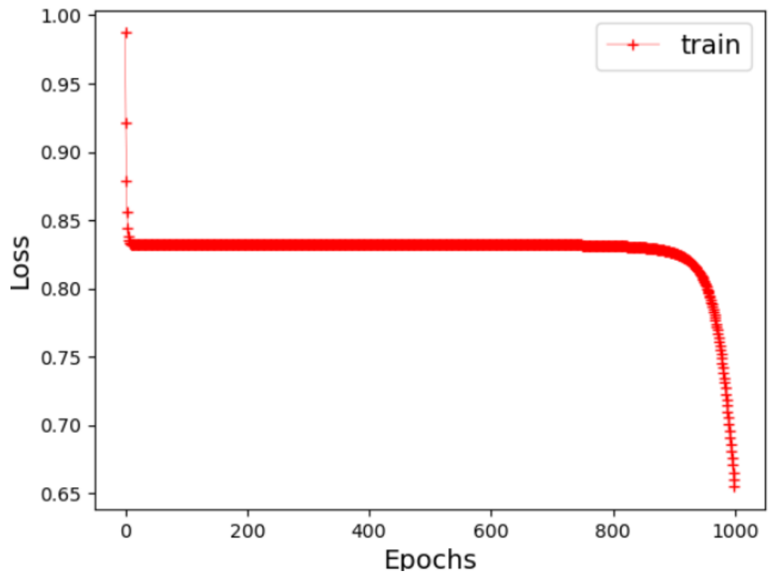
Arguments: (1st-layer-neurons, 2nd-layer-neurons)

- *small amount of neurons: (1,1)*

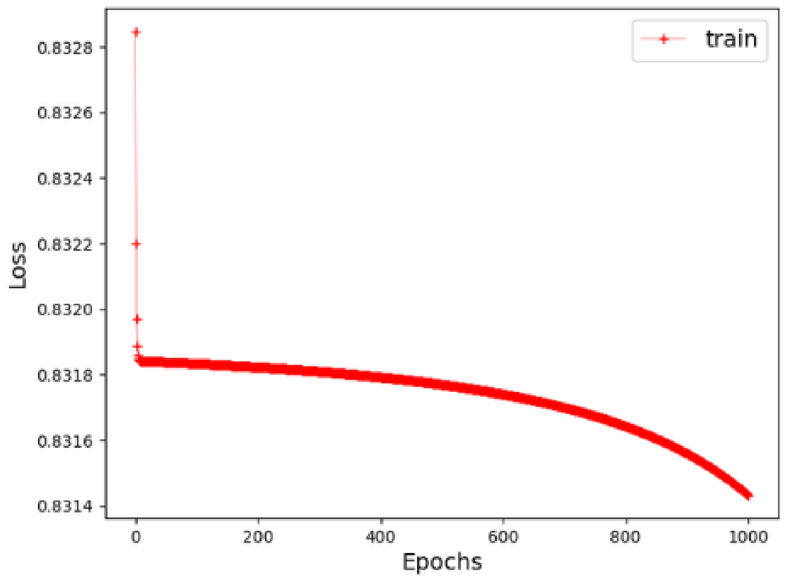
  Experiment setting: lr=0.8, LR_scheduler=on, momentum=off, adagrad=off

  After several trials, we can tell that hidden units do matter during the training process. If set to (1,1), models seem to learn nothing with epoch=1000, otherwise, we have to set epoch to 10000, which is relatively time consuming.

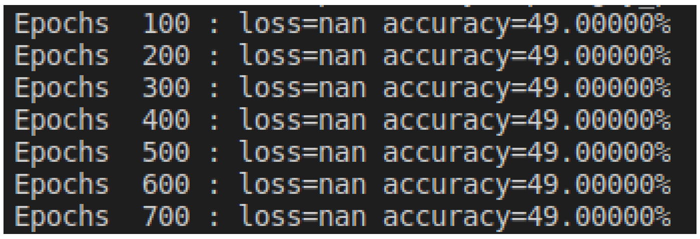
  For linear case, from the graph below, we can tell that the loss remains around 0.8 for quite a long time.



  For XOR case, loss remains around 0.83 for the whole time. (doesn't progress)



- *large amount of neurons at each layer: (100, 100)* there will be divide-by-zero error, which leads to the problem of loss output becoming nan.



```
Epochs  100 : loss=nan accuracy=49.00000%
Epochs  200 : loss=nan accuracy=49.00000%
Epochs  300 : loss=nan accuracy=49.00000%
Epochs  400 : loss=nan accuracy=49.00000%
Epochs  500 : loss=nan accuracy=49.00000%
Epochs  600 : loss=nan accuracy=49.00000%
Epochs  700 : loss=nan accuracy=49.00000%
```

C. Without activation functions

The consequence of not using activation function is that the nonlinear case will fail. Theoretically, there will be 2 parts affected if performing the training process without activation function:

1. Forward propagation: neuron output Z will be directly fed into the next layer.

2. Back propagation: Z's gradient will directly equals to a's gradient. The loss will remain nan. Nonlinear problems (XOR) under this case will be unsolvable.

D. Others

Learning Rate scheduler:

To do gradient descent, one of the most significant things is the learning rate. By only testing fixed learning rate and thus to make the decision seems to be not enough. Therefore, I have the learning rate decreased every 500 epochs by multiplying itself, which carries out the phenomenon of first it walks a big step, then it walks a bit smaller and smaller once it gets closer to the target, the local/global minimum.

Here's how I implement the adaptive learning rate:

```
def update_weight():

       if epoch % steplr_step == 0:

              lr *= lr

       #do weight update

       ...
```

[Tipp] Since linear and XOR are best fit with different step sizes, so for linear, I have the step size of 500, meaning to adapt the learning rate every 500 epoch, whereas for XOR I have the step size of 1000, making the learning rate decrease a bit slower.

5. Extra

A. Implement different optimizers

This part, I implemented 2 different optimizers: **momentum** and **adagrad**.

- ***Momentum:***

  The mathematical expression of momentum is:

$$V_t \longleftarrow \beta V_{t-1} - lr * (dL/dW)$$

$$W \leftarrow W + V_t$$

  In general case, β is set to be 0.9.

For implementation, momentum is added to the part where weight/bias is updated. Previously, without adding momentum, how weight and bias are updated is like this:

```
W -= lr * gradient_W

b -= lr * gradient_b
```

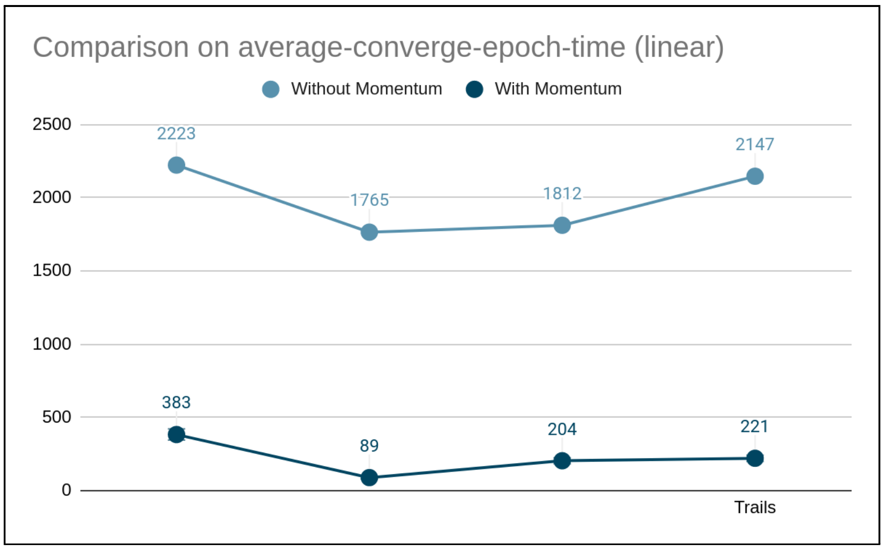
With momentum:

```
vt = beta * vt - lr * gradient_W

W += vt

vt2 = beta * vt2 - lr * gradient_b

b += vt
```

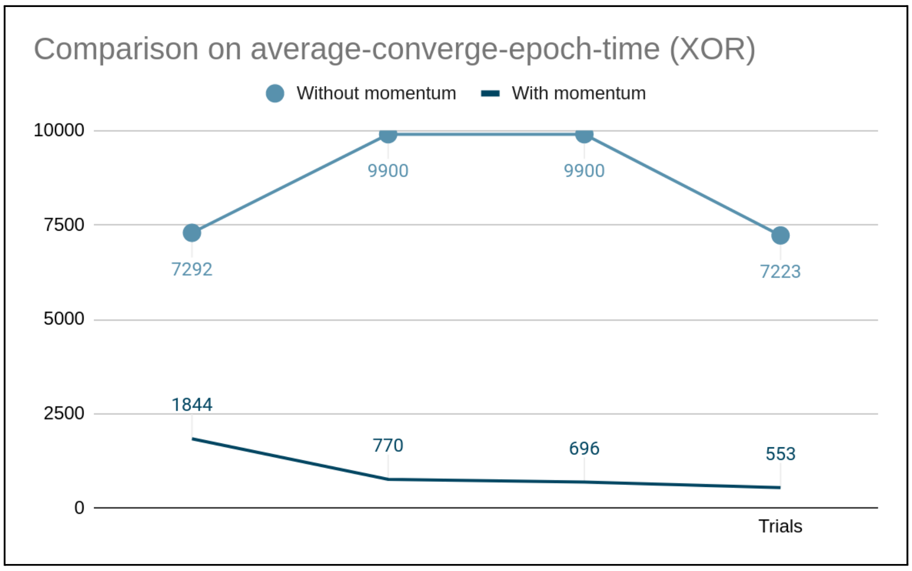
After several trials, I found that by adding momentum can have relatively huge optimization on linear problem convergence. It converges way quicker with less loss and accuracy at 100%. Here is the comparison of with and without using momentum on a linear dataset.

Experiment Setting: lr = 0.8, max_epochs = 10000

The following is the comparison on average-converge-epoch-time for 5 experiments with and without using momentum on linear dataset. With every node representing the used epoch number to reach convergence, we can clearly see how powerful momentum is.



On the linear dataset, adding momentum can converge (acc: 100%) way faster than without this optimizer. On the XOR dataset, momentum also did a great job on performing optimization. Yet, compared to linear, there seems to be still a bit more space for improvements.



- ***Adagrad:***

Another optimizer I've implemented is **Adagrad.**

The mathematical expression of Adagrad is:

$$W \leftarrow W - \eta \frac{1}{\sqrt{n+\epsilon}} \frac{\partial L}{\partial W}$$

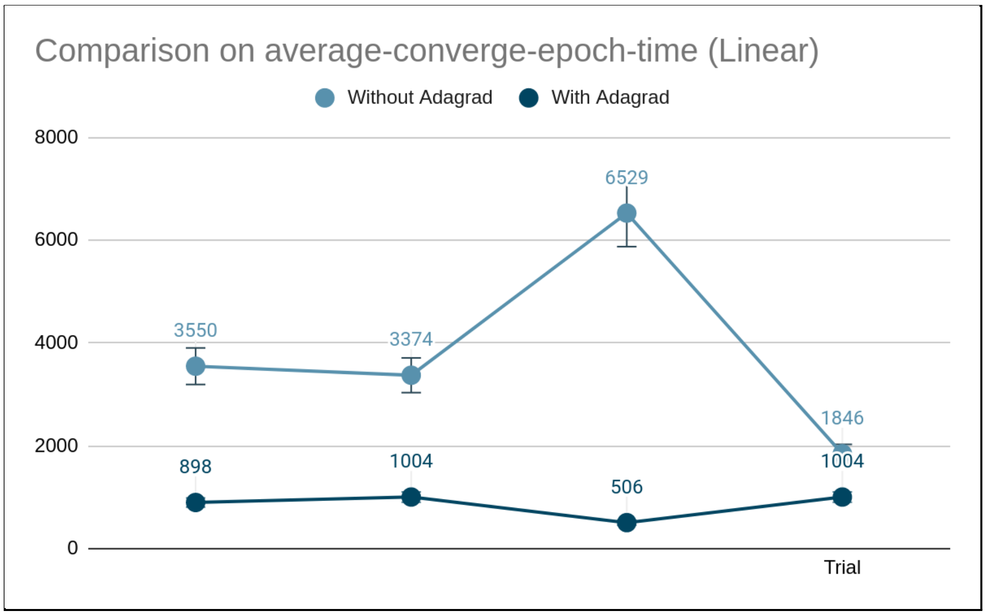
$$n = \sum_{r=1}^{t} (\frac{\partial L_r}{\partial W_r})^2$$

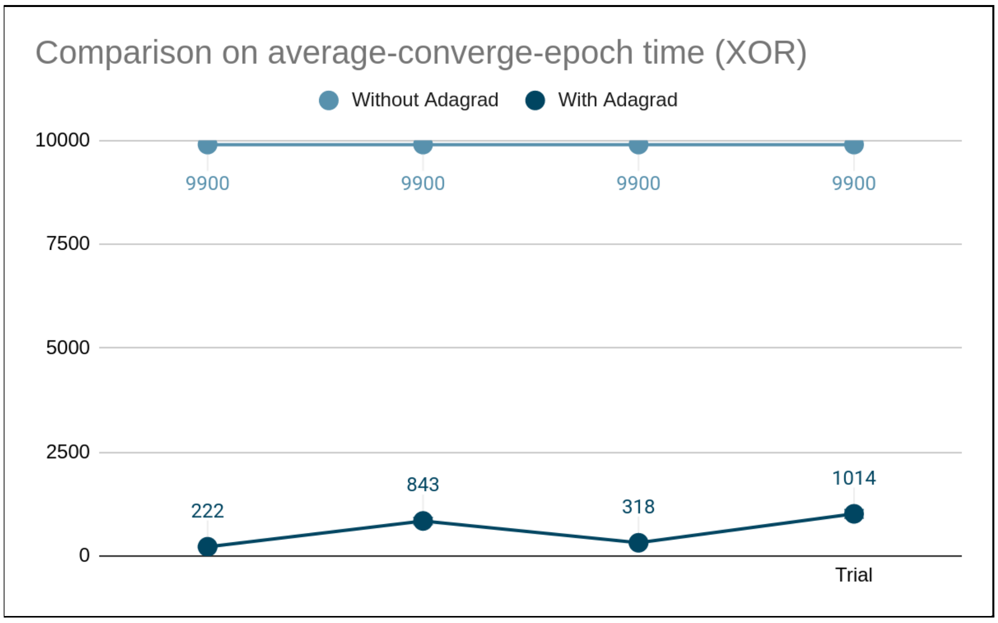
$$W \leftarrow W - \eta \frac{1}{\sqrt{\sum_{r=1}^{t} (\frac{\partial L_r}{\partial W_r})^2 + \epsilon}} \frac{\partial L}{\partial W}$$

Here's how I implement:

```
n = np.sum(self.gradW[i]*self.gradW[i])

self.W[i] = self.W[i] - self.lr * (1/np.sqrt(n+self.eps))*self.gradW[i]

n2 = np.sum(self.gradb[i]*self.gradb[i])

self.b[i] = self.b[i] - self.lr * (1/np.sqrt(n2+self.eps))*self.gradb[i]
```

The following two line graphs down below are the comparison on the use of epoch numbers before reaching convergence on both linear and XOR dataset with and without Adagrad:





From the above two graphs, we can tell that adding Adagrad to the original weight-update function does provide a positive influence on both linear and XOR dataset. The convergence time is widely reduced to nearly 10 times.

Conclusion

Both optimizers that I've implemented posed influential effects on the output result. Yet, after several times of testing, I found that for linear dataset, the optimizer momentum is better than using Adagrad, whereas for XOR non-linear cases, the optimizer Adagrad is better than momentum. So I modified my structure, making it use different optimizers according to different dataset types.