Student ID: 0860832
Name: Thong The Nguyen (阮世聰)

Lab 01 Report

1. Introduction

This report describes the implementation of backpropagation algorithm and experimental result after trying various configurations.

Backpropagation algorithm is used for updating weights of neural networks by repeatingly calculating the gradient of loss function with respect to the weights of the network. Then updating weights by changing the weights proportionally with the opposite direction of that gradient. The algorithm uses chain rule to calculate the derivative efficiently between layers. Backpropagation is widely used in Machine Learning.

2. Experimental Setup

Sigmoid functions

Easy implementation, applying sigmoid function for each element of numpy array

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

Implementation for derivative of sigmoid. For efficiency, we assume the input of this function as output of sigmoid function (y = sigmoid(x)), so that we can use immediate data in the network without recomputing sigmoid.

```
def der_sigmoid(y):
    return y * (1 - y)
```

Neural network

Network has 4 layers: 1 input layer, 2 hidden layers, 1 output layer

Weight of network:

```
self.w1 = np.random.rand(hidden_size, 2)
self.w2 = np.random.rand(hidden_size, hidden_size)
self.w3 = np.random.rand(1, hidden_size)
```

- with *hidden_size* is a param, specify the number of nodes in each hidden layer
- Initializing network's weight by uniform distribution in range [0, 1)
- $w1_{ij}$: weight that connects j^{th} node of input layer with i^{th} of hidden layer 1
- $w2_{ij}$: weight that connects j^{th} node of hidden layer 1 with i^{th} of hidden layer 2
- $w3_{ij}$: weight that connects j^{th} node of hidden layer 2 with i^{th} of output layer

Training

```
n = inputs.shape[0]

for epochs in range(1, self.num_step + 1):
    for idx in range(0, n, self.batch_size):
        # operation in each training step:
        #   1. forward passing
        #   2. compute loss
        #   3. propagate gradient backward to the front
        self.output = self.forward(inputs[idx: idx + self.batch_size, :])
        self.grad_loss = (self.output - labels[idx: idx + self.batch_size, :].T) / self.batch_size
        self.backward()

    if epochs % self.print_interval == 0:
        print('Epochs {}: '.format(epochs), end='')
        error = self.test(inputs, labels, False)
        self.log.append((epochs, error))

return self.log
```

- For each input data point, we calculate the output of the network, then measure loss and finally use backpropagation to update weights of the network.
- Loss function is Mean Square Error, *grad_loss* is derivative of loss function with respect to outputs of the last layer.

Forward pass

```
def forward(self, inputs):
    self.a0 = inputs.T

    self.z1 = self.w1 @ self.a0
    self.a1 = sigmoid(self.z1)

    self.z2 = self.w2 @ self.a1
    self.a2 = sigmoid(self.z2)

    self.z3 = self.w3 @ self.a2
    self.a3 = sigmoid(self.z3)

    return self.a3
```

- *z1*, *z2*, *z3*: values after multiply the weight with output from previous layer

- $a0$, $a1$, $a2$, $a3$: values after applying sigmoid activation on each element of corresponding z.

Back-propagation

```python
def backward(self):
    dldz3 = 2 * self.error * der_sigmoid(self.a3)
    dldw3 = self.a2 @ dldz3.T

    dldz2 = (self.w3.T @ dldz3) * der_sigmoid(self.a2)
    dldw2 = self.a1 @ dldz2.T

    dldz1 = (self.w2.T @ dldz2) * der_sigmoid(self.a1)
    dldw1 = self.a0 @ dldz1.T

    self.w3 -= self.learning_rate * dldw3.T
    self.w2 -= self.learning_rate * dldw2.T
    self.w1 -= self.learning_rate * dldw1.T
```

- Variables
    - $dldz3$: derivative of loss with respect to $z3$
    - $dldz2$: derivative of loss with respect to $z2$
    - $dldz1$: derivative of loss with respect to $z1$
    - $dldw3$: derivative of loss with respect to $w3$
    - $dldw2$: derivative of loss with respect to $w2$
    - $dldw1$: derivative of loss with respect to $w1$
- For each layer, we compute the derivative of loss function with respect to the output before applying activation of that layer. Then we multiply that derivative with the previous input activation to get the weight gradient.
- We update weights by the opposite direction of the gradient, so that the loss function will be minimized.

3. Experimental Result

Setting up experiments with 2, 5 and 10 neurons for each hidden layer with Linear data mode. Configuration are:

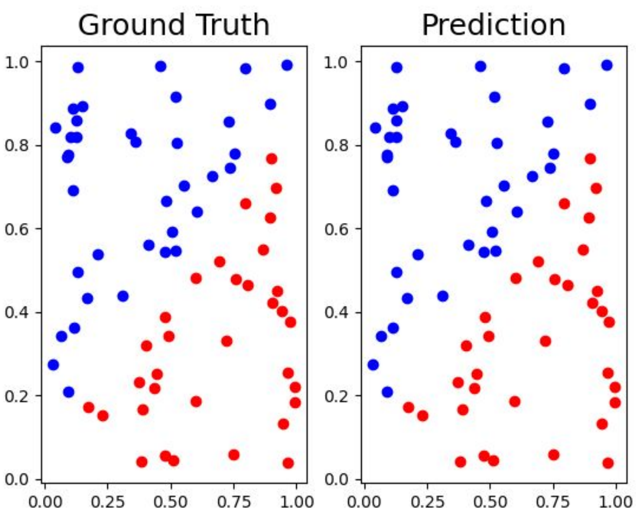
- Learning rate: 0.01
- Batch size: 1
- Epochs: 30000

For 2 neurons case

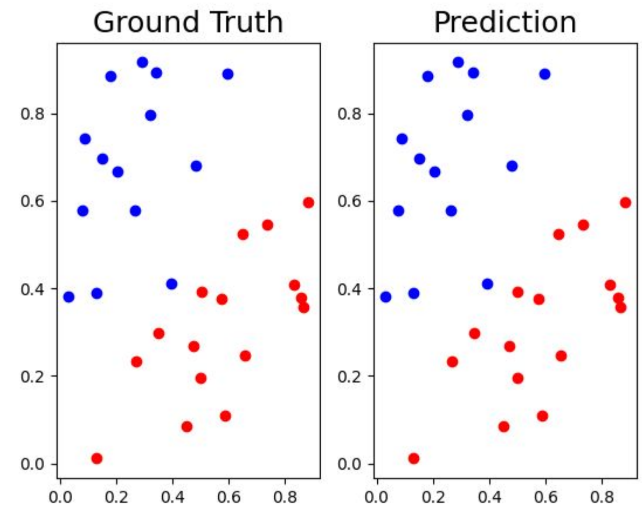
```
Epochs 100: loss: 0.498148028958550
Epochs 200: loss: 0.498157587256099
Epochs 300: loss: 0.498107483639680
Epochs 400: loss: 0.498037743726145
Epochs 500: loss: 0.497936406870668
Epochs 600: loss: 0.497780928223413
Epochs 700: loss: 0.497526550448727
Epochs 800: loss: 0.497075776680708
Epochs 900: loss: 0.496189040193490
Epochs 1000: loss: 0.494174698777935
Epochs 1100: loss: 0.488546366822672
Epochs 1200: loss: 0.467714286398852
Epochs 1300: loss: 0.385924186151563
Epochs 1400: loss: 0.261417286484927
Epochs 1500: loss: 0.183729010098401
Epochs 1600: loss: 0.139532028295363
Epochs 1700: loss: 0.111999905696715
Epochs 1800: loss: 0.093417480813354
Epochs 1900: loss: 0.080103783614775
Epochs 2000: loss: 0.070129483561636
Epochs 2100: loss: 0.062396145040452
Epochs 2200: loss: 0.056235220641333
Epochs 2300: loss: 0.051217544133403
Epochs 2400: loss: 0.047055540006358
Epochs 2500: loss: 0.043549622129766
Epochs 2600: loss: 0.040557157756303
Epochs 2700: loss: 0.037973655117389
Epochs 2800: loss: 0.035720912693170
Epochs 2900: loss: 0.033739305708666
Epochs 3000: loss: 0.031982622331586
```

```
Testing: loss: 0.010349418419387
Prediction:
0.999592723354337
0.000214192533982
0.999816514065708
0.999808165876027
0.014559158042012
0.002955428428603
0.000205144754782
0.999819884561266
0.999804722242267
0.000352587419289
0.000230912059846
0.000211896875501
0.004788270683717
0.000328730523590
0.000203712558624
0.999738891040104
0.999820506720734
0.000207321463099
0.735114151419946
0.000211376019315
0.000872322557875
0.999735936964249
0.016284639135378
0.999802101137778
0.999812813045507
0.000455202635936
0.000461720284977
0.999822737789262
0.999770434006250
0.999600481171470
```

Visualizing training and testing of 2 neurons case



Training



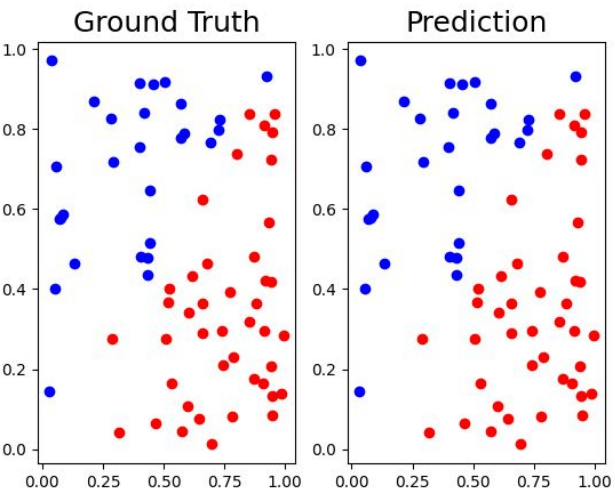
Testing

For 5 neurons case

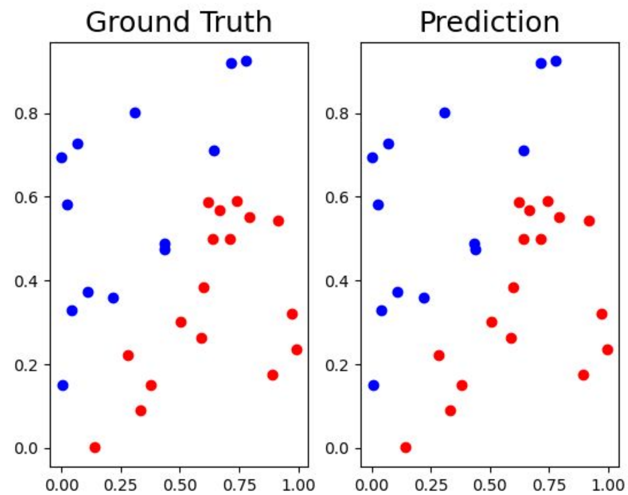
```
Epochs 100: loss: 0.480304907847044
Epochs 200: loss: 0.479798075121852
Epochs 300: loss: 0.479042099448203
Epochs 400: loss: 0.477733269243228
Epochs 500: loss: 0.474968881429127
Epochs 600: loss: 0.467145481376536
Epochs 700: loss: 0.433541289911109
Epochs 800: loss: 0.312339824757573
Epochs 900: loss: 0.207204141648969
Epochs 1000: loss: 0.156955217270923
Epochs 1100: loss: 0.128976791565272
Epochs 1200: loss: 0.111018377323002
Epochs 1300: loss: 0.098436100838926
Epochs 1400: loss: 0.089092906631746
Epochs 1500: loss: 0.081857147399896
Epochs 1600: loss: 0.076069062684495
Epochs 1700: loss: 0.071316887953583
Epochs 1800: loss: 0.067330444867081
Epochs 1900: loss: 0.063925571859470
Epochs 2000: loss: 0.060972894932261
```

```
Testing: loss: 0.000123882438227
Prediction:
0.000000015215218
0.999999831063216
0.999999374665016
0.000000015436225
0.000000039791804
0.999999311787177
0.000001249886859
0.000045048997829
0.000000015831775
0.000000178155244
0.999982243234982
0.999999781486925
0.000000021138716
0.000000136554205
0.999930772244974
0.000000023847018
0.003007231145082
0.999999835809095
0.000000036561327
0.999999299732326
0.000000053193430
0.000000045916255
0.999999841839433
0.999999840650603
0.999999763700019
0.000000036110260
0.999999680488601
0.000000173090512
0.999428317581081
0.000000046558506
```

Visualizing training and testing of 5 neurons case



Training

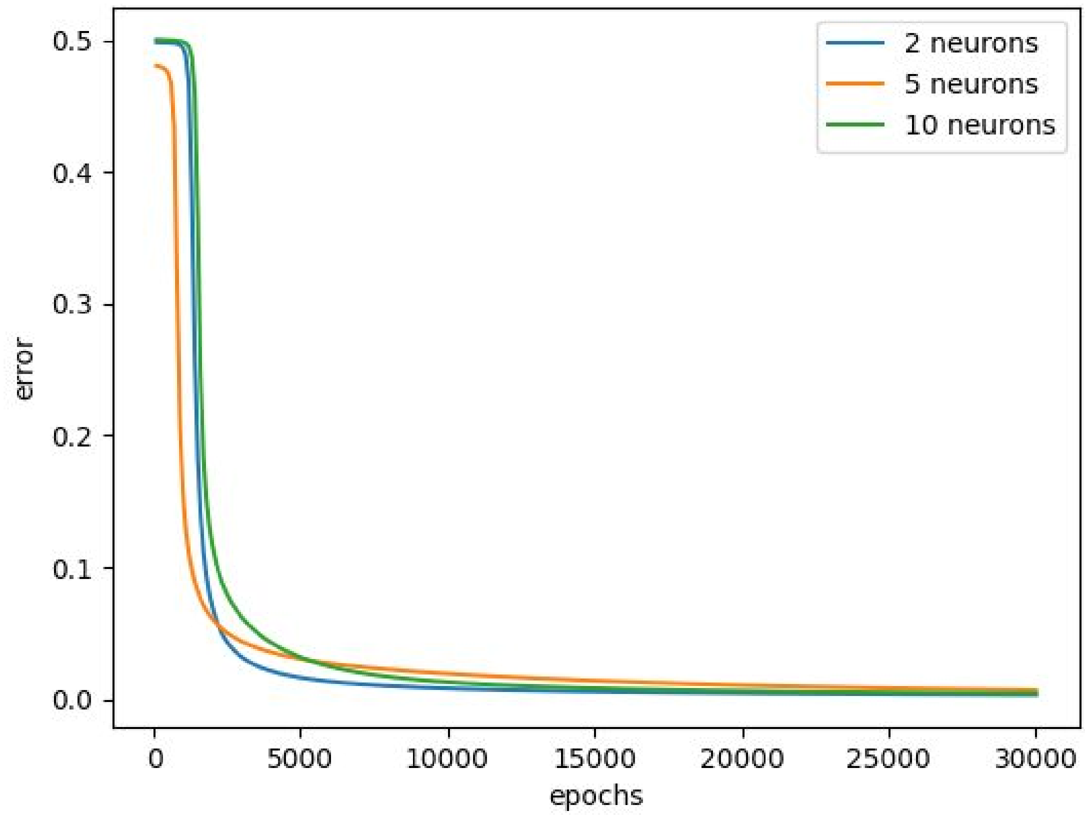


Testing

For 10 neurons case

```
Epochs 100: loss: 0.500287366221310
Epochs 200: loss: 0.500208985262084
Epochs 300: loss: 0.500128292978963
Epochs 400: loss: 0.500039911951186
Epochs 500: loss: 0.499937105179121
Epochs 600: loss: 0.499810375339164
Epochs 700: loss: 0.499644921701746
Epochs 800: loss: 0.499415408137220
Epochs 900: loss: 0.499073818505270
Epochs 1000: loss: 0.498516939387158
Epochs 1100: loss: 0.497481501314093
Epochs 1200: loss: 0.495114398246574
Epochs 1300: loss: 0.487831122322461
Epochs 1400: loss: 0.460405051317115
Epochs 1500: loss: 0.364052265690707
Epochs 1600: loss: 0.245764624358942
Epochs 1700: loss: 0.184286047564045
Epochs 1800: loss: 0.151615455271226
Epochs 1900: loss: 0.131198830798985
Epochs 2000: loss: 0.116878067277170
```

```
Testing: loss: 0.001963293037106
Prediction:
0.000001826784386
0.000010072194808
0.999983702238637
0.999997543700025
0.000001848066861
0.000002983278363
0.999993838227588
0.999997480250569
0.023716560206306
0.999988778215353
0.000002387949513
0.999757500993736
0.999997561031048
0.000017312988391
0.000001360219896
0.000041193614941
0.999990712219231
0.000200218316921
0.000003242763855
0.000018265248339
0.034233114136363
0.000041369689950
0.000003997816557
0.000019589901690
0.999996939951937
0.000001691841409
0.000002232817483
0.999996679939446
0.999997418599799
0.999722321355484
```

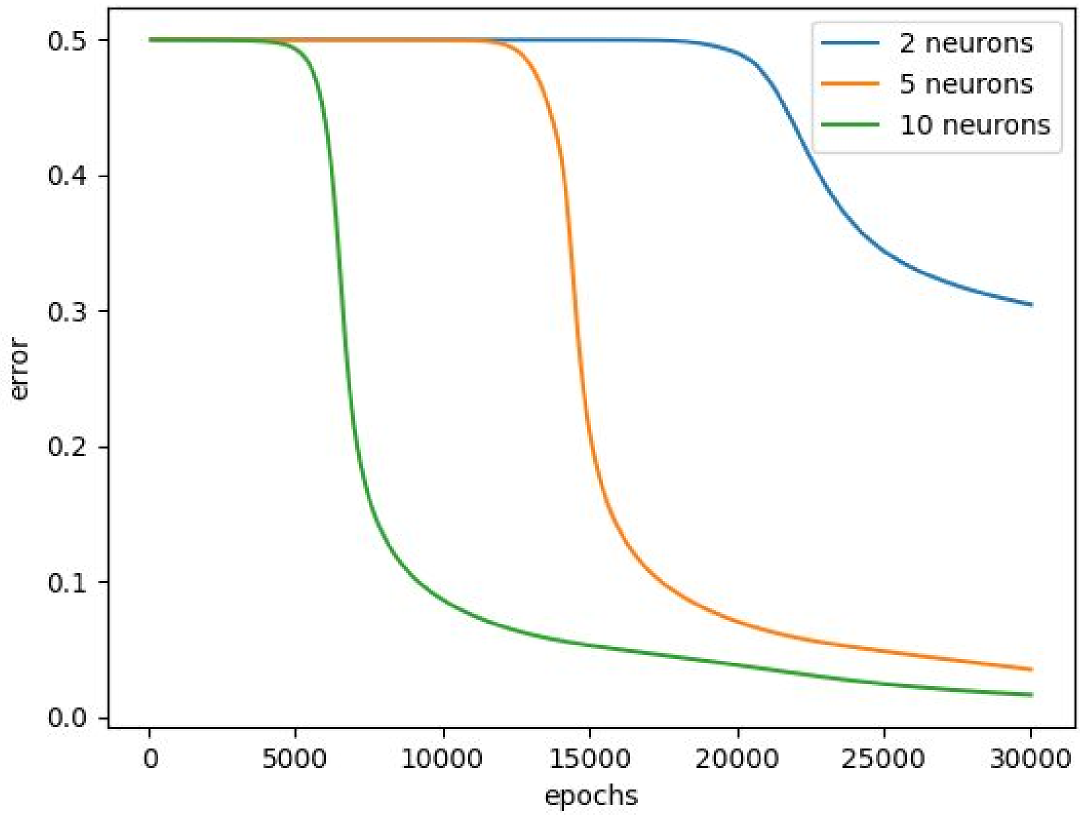


Compare error with number of epochs for each configuration

Both 3 cases all achieve nearly 100% accuracy after 30000 epochs. It seems like 2 and 5 neurons converged faster than 10 neurons, but it is not very noticeable, maybe of the linear separation property of the dataset.

Setting up experiments with 2, 5 and 10 neurons for each hidden layer with XOR data mode. Configuration are:

- Learning rate: 0.01
- Batch size: 1
- Epochs: 30000



With the XOR dataset, we clearly see the difference of error rate between configuration. The 10 neurons configured achieved the highest accuracy, while the others results are not so great.

```
Testing: loss: 0.299828220672738
Prediction:
0.000007728704655
0.391214545506138
0.000082030351220
0.439205047437993
0.001473097806369
0.488161436800233
0.024174012800695
0.537112828229019
0.182726332199758
0.585097291377219
0.480939981003960
0.631236800189858
0.662769553812461
0.674797907622254
0.715230034162155
0.688373843702526
0.752178811450217
0.595384319351205
0.785477139521124
0.443526377371088
0.815120051950276
0.271011100877551
0.841230663522053
0.135666979026287
0.864023787040180
0.059010003033930
0.883772059988286
0.024048790560171
0.900777414619768
```

```
Testing: loss: 0.024030837235005
Prediction:
0.000000136936570
0.999999390018271
0.000001615662933
0.999999412711546
0.000038119590985
0.999999432450378
0.001019461569173
0.999999435331367
0.015210490098430
0.999999344408944
0.083924690370826
0.999998216143358
0.176391239162938
0.999439076098351
0.190693264721406
0.130457971913053
0.999301131484551
0.062453719371555
0.999997920634974
0.023514353288171
0.999999259660161
0.007939364106244
0.999999355683587
0.002663425481118
0.999999330178485
0.000946876894847
0.999999267391652
0.000369299019808
0.999999176177294
```

Prediction between 2 neurons and 5 neurons configuration

The output sigmoid activation isi the 2 neurons config is not very confident about its decision. But in the case of 5 neurons, we can see that the outputs are almost approximately 0 or 1.

So we can conclude that, with a higher number of neurons in each layer, we can learn more hard patterns in the dataset (XOR compared with Linear), but it takes a little bit longer to converge.

With linearly separable data, the model can get nearly 100% accuracy in the first few epochs, but in a harder pattern, it needs more epochs until convergence.

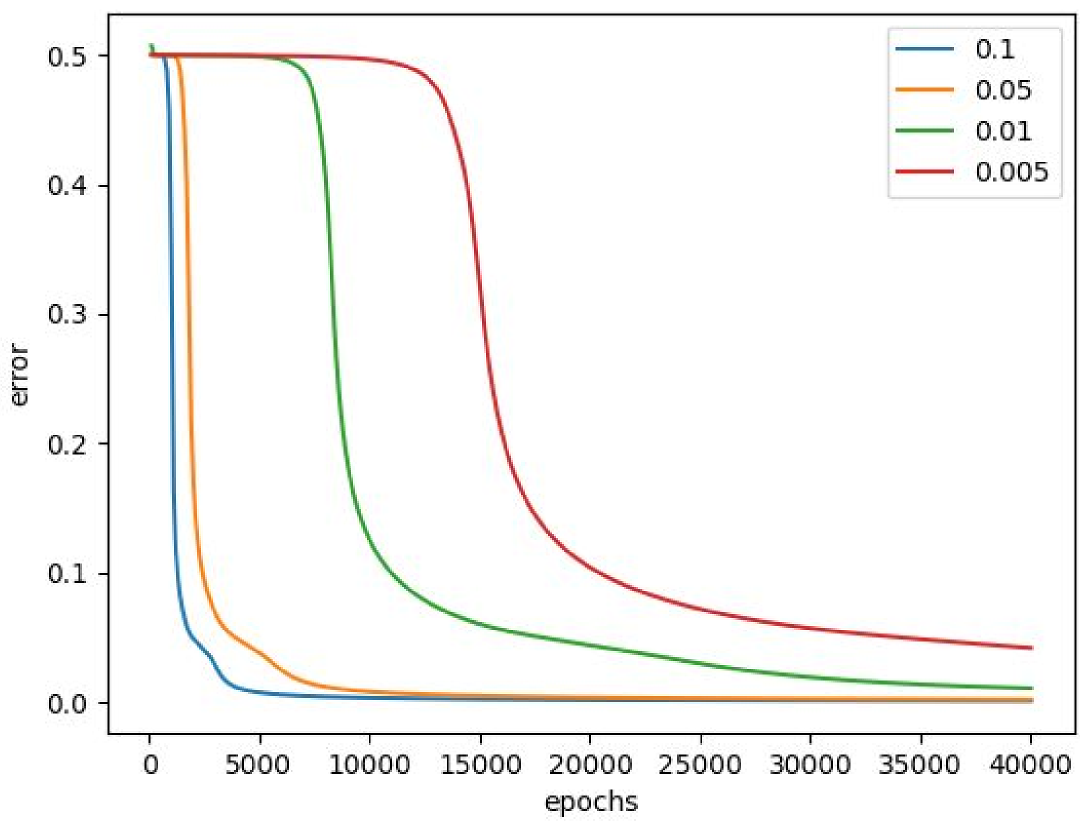
4. Discussion

We test will other parameters to see how the model performs

1. Change learning rate

Setting:

- Data mode: XOR
- Learning rate: [0.1, 0.05, 0.01, 0.005]
- Batch size: 1
- Neuron: 10

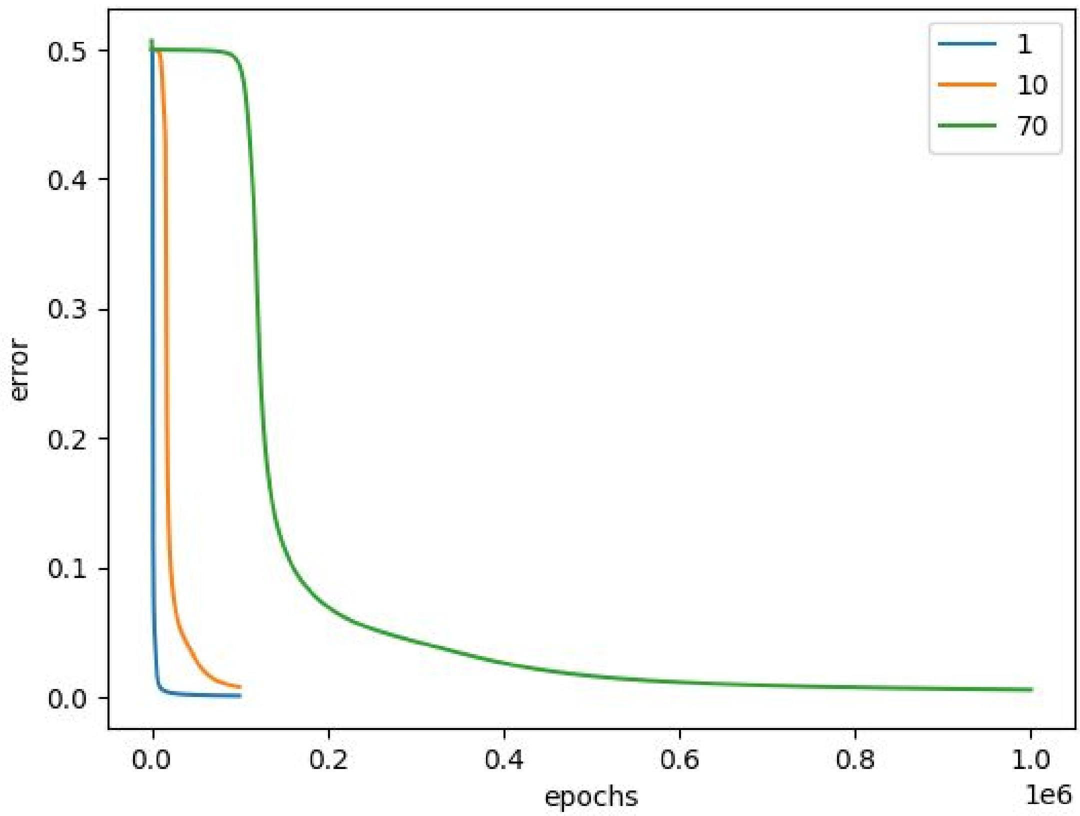


We can see that higher learning rate helps models learn faster, but the learning curve is not very "smooth". Making a model learn too fast may induce more fluctuations.

2. **Change batch size**

Setting:

- Data mode: XOR
- Learning rate: 0.01
- Batch size: [1, 10, 70]
- Neuron: 10



We can see that with larger batch size, the longer time that model will converge. For batch size as a whole size of data set, we need 10^6 epochs to reach the accuracy around 99%. The advantage of larger batch size is reducing the change that model stuck in local optimal points by stochastic gradient descent. However, larger batch sizes require more memory and longer time to converge.