Inference of XOR Deep Learning Algorithm

Vignesh Desmond Sriram R

02 February 2020

1 XOR Code

1.1 Basic elements

The code for implementing XOR gate via Neural Network has different parts, each with their own important function. The basic elements here are the class declaration *__init__* with objects *weights_input_to_hidden* and *weights_hidden_to_output* used to hold input, hidden and output layer parameters. Module *numpy* is also imported for array related operations.

1.1.1 Activation Function

Activation function decides whether a neuron should fire or not based on its weighted inputs (as I see it). There are three functions discussed so far: *sigmoid, tanh, and ReLU*. These functions map the input in a range depending on the function. The derivatives of these functions are used to calculate delta, which helps in optimising weights. Plot is shown in Task 3.

1.1.2 Forward Propagation

It is the process of input propagating through the nodes. The output is given by $y=wx$, w being the weight of the path.The weight is initialised randomly and the then outputs are calculated. The code shows the output as the activated function of sum of weighted inputs.

1.1.3 Back Propagation

Its a feedback system used to minimise the loss of the model, loss defined by *output expected output*. The loss is then used to optimise the weights by calculating delta of each layer. The code shows the back propagation of the loss and delta value.

1.1.4 Optimising weights

Weight of each layer is optimised by taking dot product of current weights and weight deltas. For every *epoch* (full iteration of data set), the weights are optimised to minimise loss function even further, known as a gradient descent of loss function.

1.1.5 Training

The model is given possible inputs and set of expected outputs. Then it is set to run for a set number of epochs. The algorithm then runs and optimises along the way. After the training is complete, the model is now ready to predict the outcome for given possible inputs

2 Task 1

2.0.1 Hidden Layers

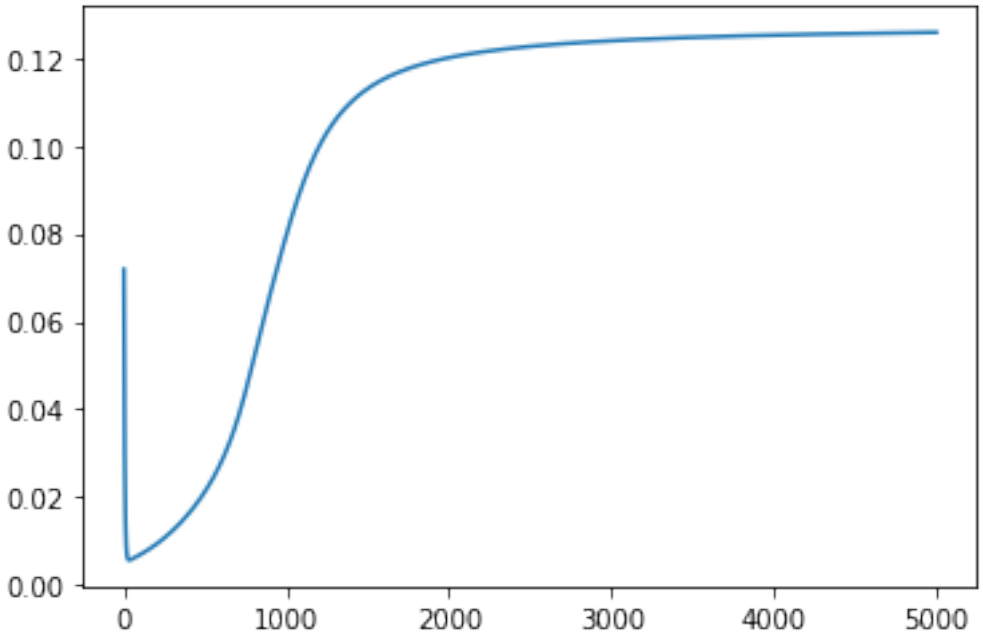

Figure 1: Epoch-Loss function plot when *hidden_size=1*

The *hidden_size* is default to 5 in original code. Changing it to 1, the loss function becomes undesirable as shown.

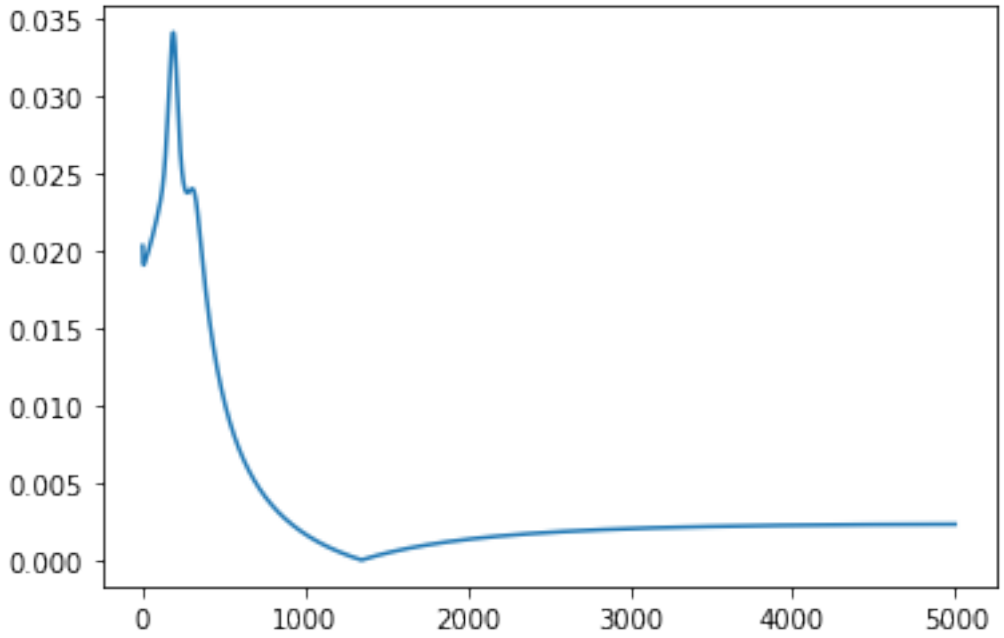

Figure 2: Epoch-Loss function plot when *hidden_size=2*

Changing hidden_size from 1 to 2 shows significant improvement in the loss function as shown. But it is a bit chaotic at the start.

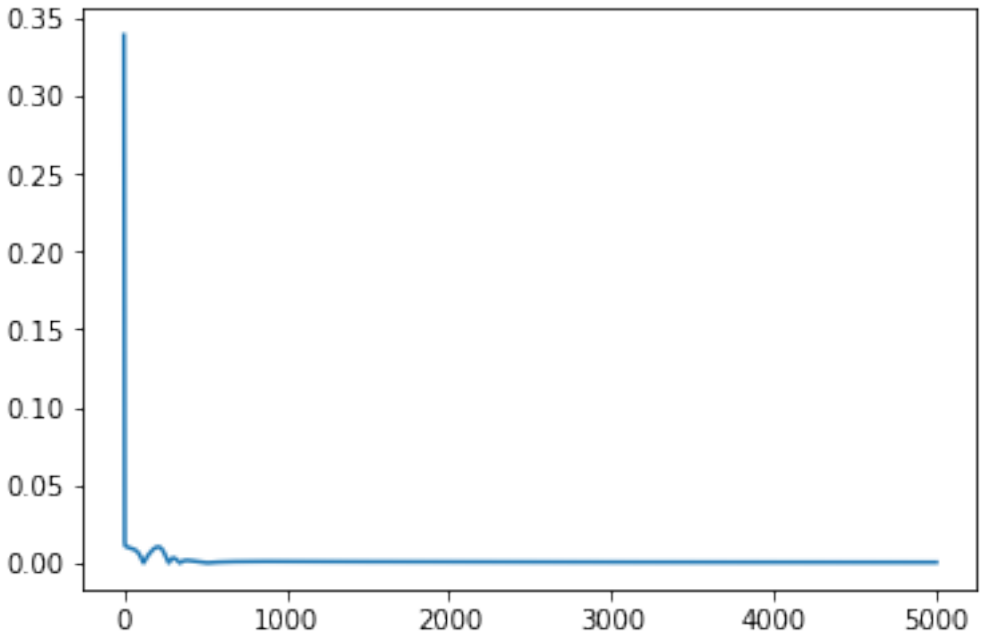

Figure 3: Epoch-Loss function plot when *hidden_size=5*

Increasing hidden_size further gives better results, from 3 to 8 (even 9). I found the optimal ones to be 4 to 6. One is shown below.

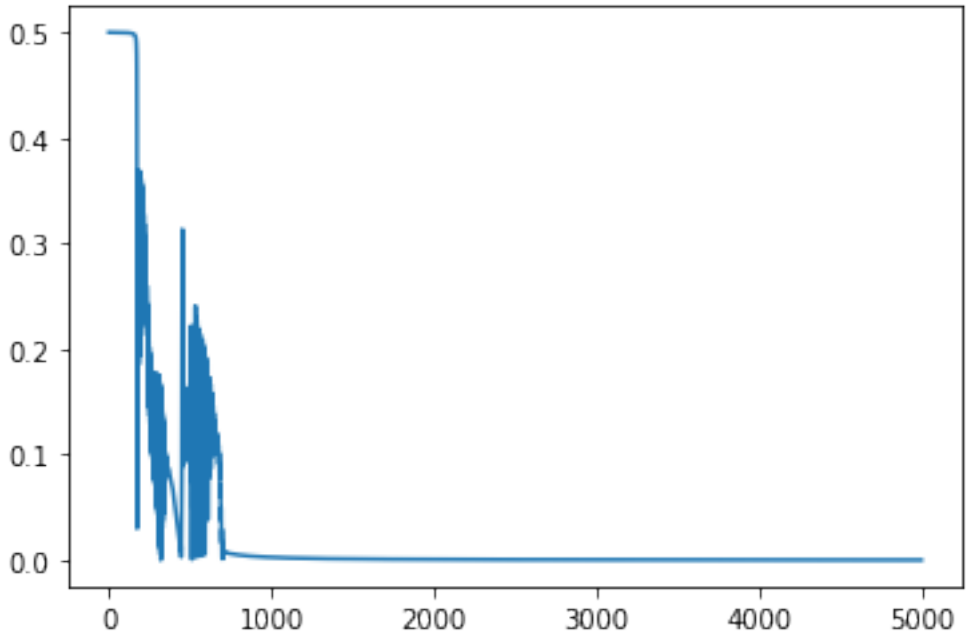

Figure 4: Epoch-Loss function plot when *hidden_size=25*

If hidden size is increased too much, the loss function becomes distorted, especially at the lower epochs. 25 for hidden size was probably overkill.

This shows that lesser number of hidden layers reduces the efficiency of the loss reducing function (at least that's what I infer) and too many layers might mean the model is trying to find weights for all those layers, resulting in some sort of loss fluctuation.

2.0.2 Epochs

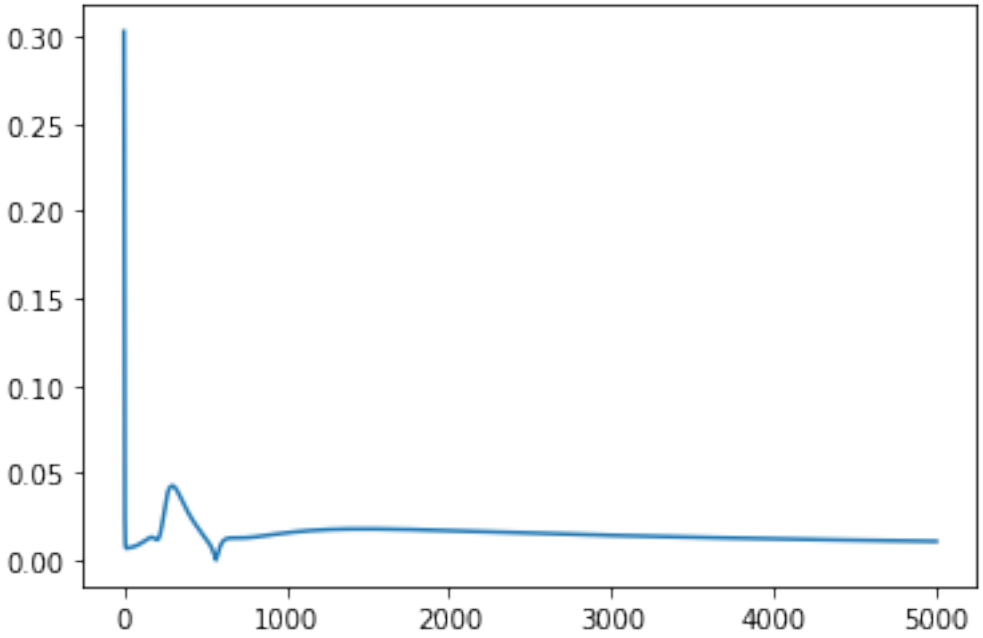

Figure 5: Epoch-Loss function plot when *num_epochs=5000*

The epoch variable *num_epochs* is set at 5000 by default, for a given *hidden_size=5*. The loss is as follows.

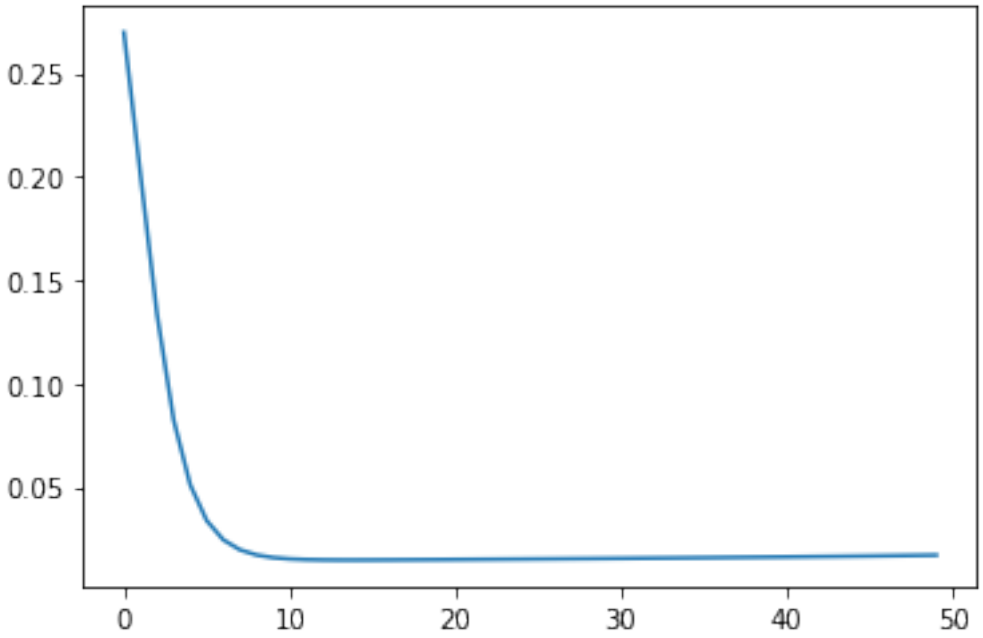

Figure 6: Epoch-Loss function plot when *num_epochs=50*

When num_epochs is reduced to 50, the loss function is much smoother and surprisingly looks better than num_epochs=5000, but the actual reason may be different, to which I'm not aware. Possible reason might be the low sample count of xor function

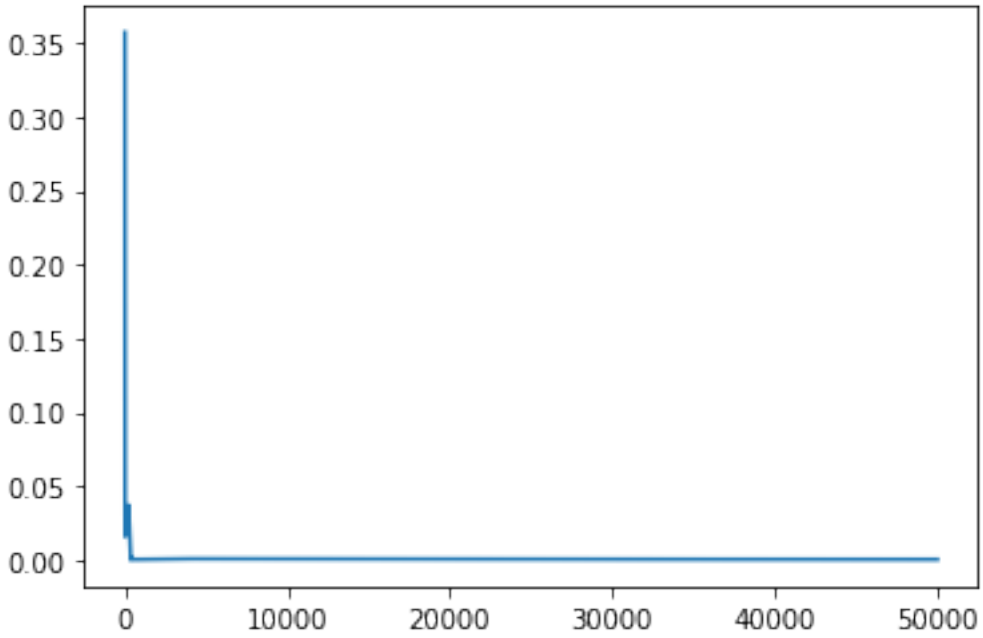

Figure 7: Epoch-Loss function plot when *num_epochs=50000*

Value of num_epochs doesn't seem to change much in this example, except a bump in the middle. For 50000, the bump is extremely sharp and short.

Honestly, num_epochs doesn't seem to impact the loss function that much. Maybe for an example requiring a lot of samples, epochs would play an important role. Or maybe I'm missing something.

3 Task 2

3.1 Tanh

Using the *tanh* activation function instead of sigmoid for *hidden_size=3* and *num_epochs=5000* , the following loss function is obtained.

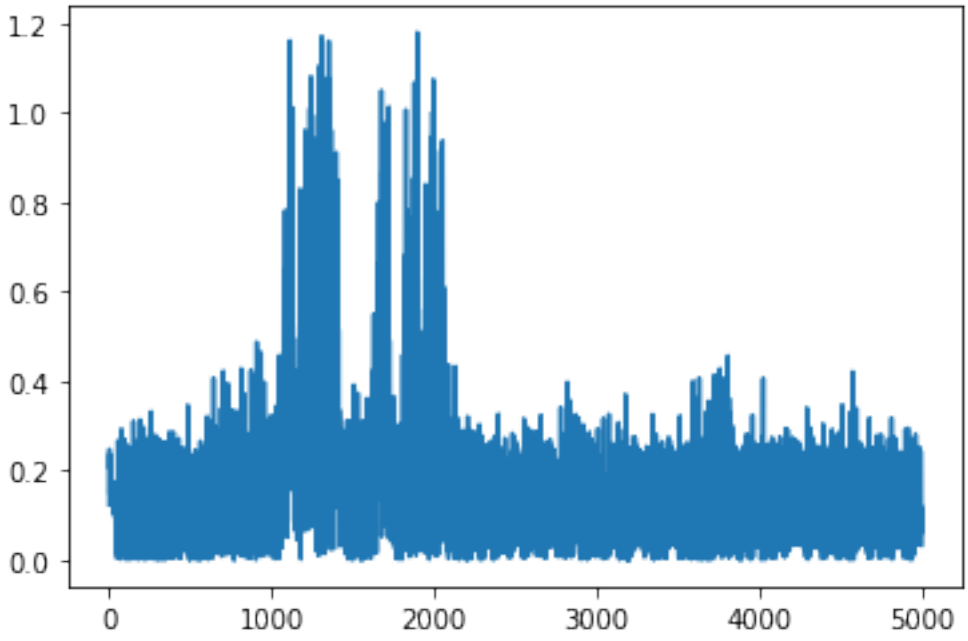

Figure 8: Epoch-Loss function plot with activation function *tanh*

It can be see that the loss function is somewhat oscillating. Initially, I was confused, but after looking up online I found some interesting points. We know that tanh is a steep function, hence it's gradient descent is high. A very high gradient can cause the weights to overshoot the minimum loss value and trying to trace back, leading to the oscillation. This can be controlled by reducing the learning rate *(new weight=existing weight - learning rate * gradient)*. So, I kept learning rate as 0.2 and this is the result.

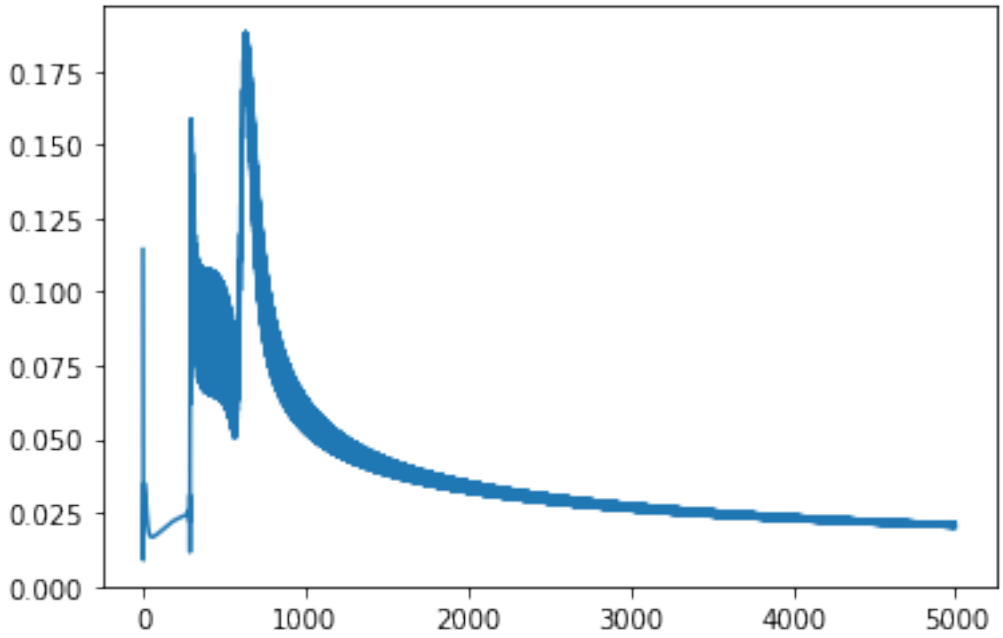

Figure 9: Epoch-Loss function plot with activation function *tanh* and *learning rate=0.2*

Looks better than the last one. So this example was very surprisingly informative.

3.1.1 ReLU

Using the *ReLU* activation function for *hidden_size=5* and *num_epochs=5000* the following loss function is obtained.

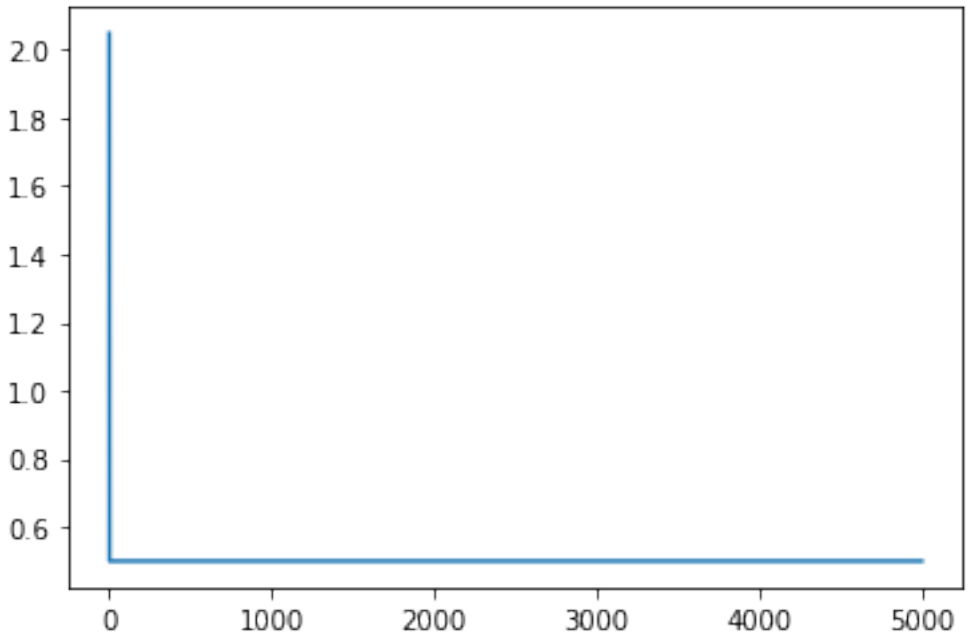

Figure 10: Epoch-Loss function plot with activation function *ReLU*

The loss function very surprisingly settles at 0.6. Same result was obtained for different hidden sizes and epochs, which led me to think about the learning rate again.

Even after changing learning rate from 1 to 0.1, I got the same results. So I went lower, and made the learning rate 0.01, which gave the following results.

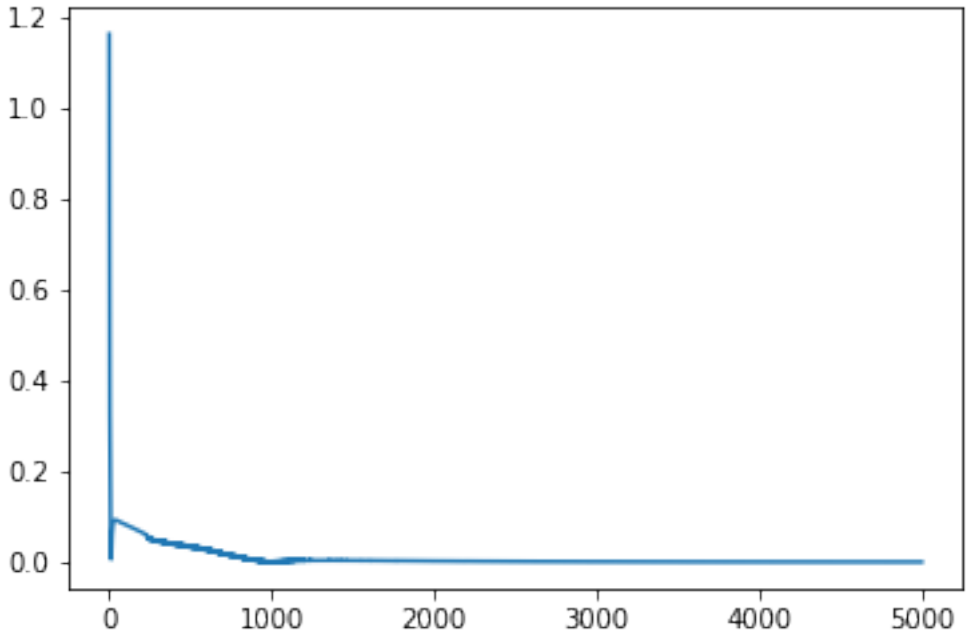

Figure 11: Epoch-Loss function plot with activation function *ReLU* and *learning rate=0.01*

The loss function is now decreasing as intended.

Decreasing the learning rate further i.e., 0.01, we get a slow descending loss function as shown

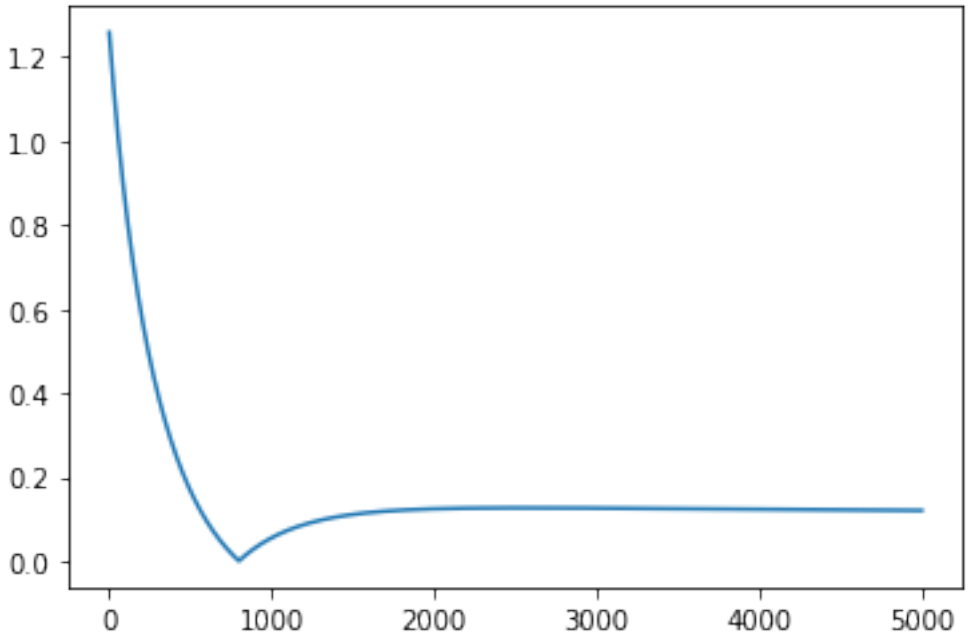

Figure 12: Epoch-Loss function plot with activation function $ReLU$ and *learning rate=0.0001*

4 Task 3

4.1 F=!((A.B)+C)+D

The truth table for the given function s shown below

A	B	**C**	**D**	**F**
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Table 1: Truth table output for F=!((A.B)+C)+D

After changing training values the model with some of the input values, changing *input_size=5* and keeping num_epochs at 5000, we get the following loss function.

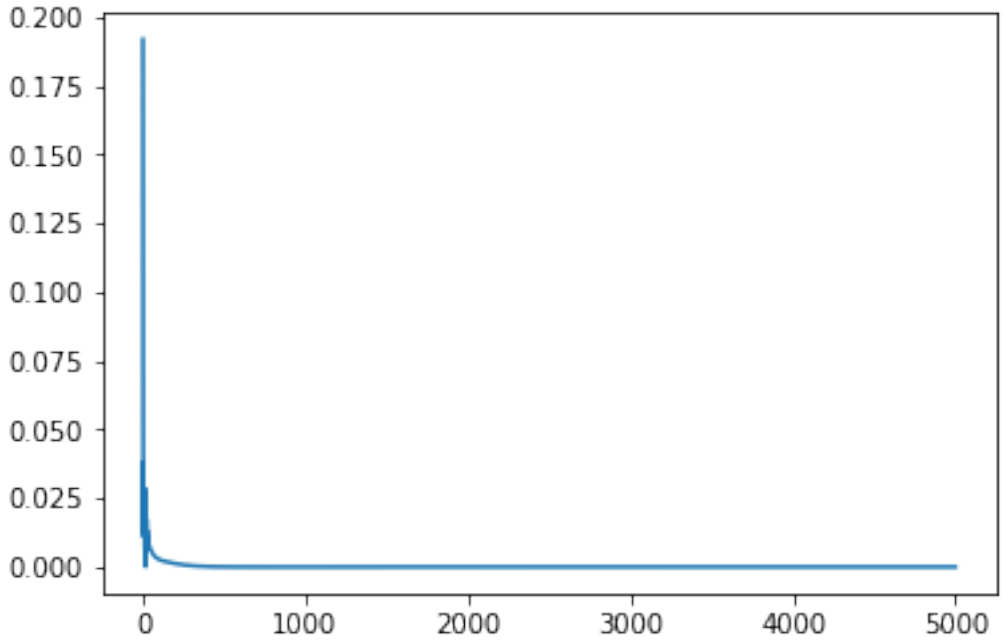

Figure 13: Epoch-Loss function plot with *hidden_size=5* and *num_epochs=5000*

Loss function here looks similar to xor loss function.

Changing the input size has minimal impact on the loss function for a given num_epochs (besides 1or some huge number). But decreasing num_epochs does introduce zigzag patterns in the loss function. One such is shown below.

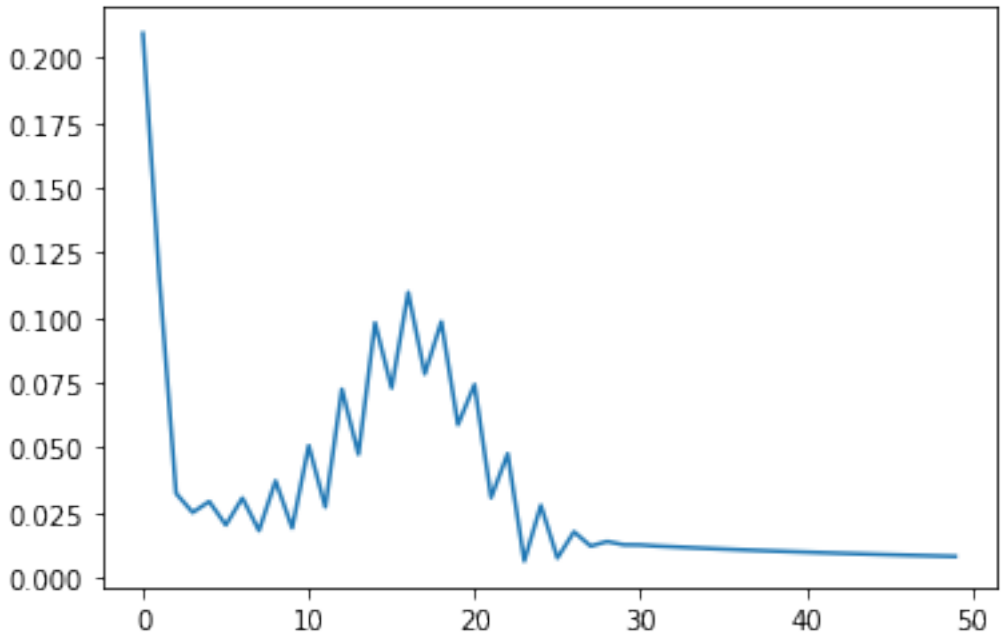

Figure 14: Epoch-Loss function plot with *hidden_size=5* and *num_epochs=50*

Reason for this effect is unknown. I might need some explanation on this part. The effect seems to disappear when num_epochs is in the range of thousands

4.2 F=!(A.B) XOR !(C.D)

The truth table for the given function s shown below

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Table 2: Truth table output for F=!(A.B) XOR !(C.D)

After changing training variables as per truth table and setting *hidden_size=5* and *num_epochs=5000*, we get the following loss function.

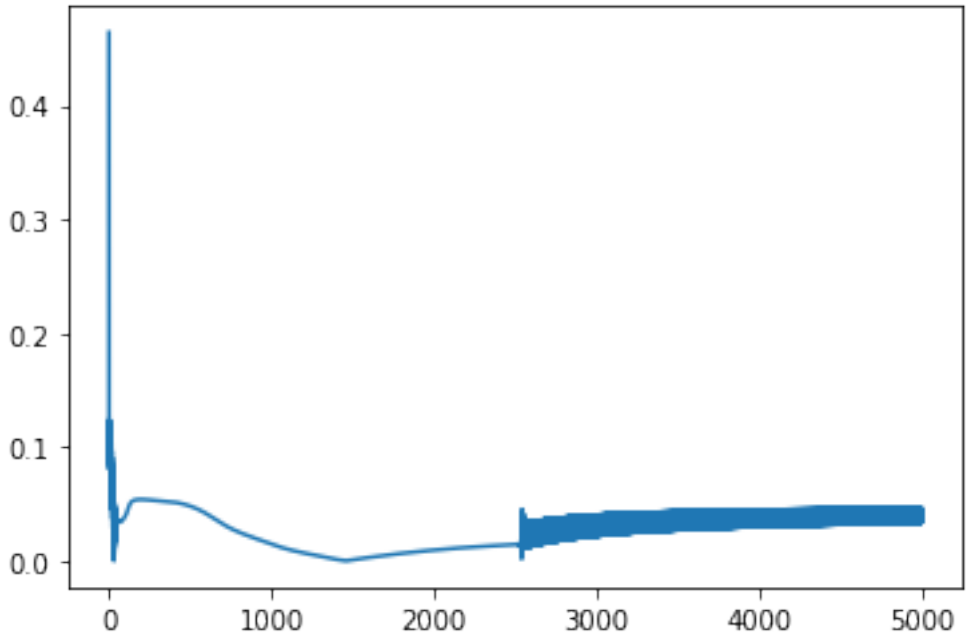

Figure 15: Epoch-Loss function plot with *hidden_size=5* and *num_epochs=5000*

The function is good at start but gets somewhat weird at later epochs. Reason unknown.

When decreasing num_epochs=50 The zigzag patterns appear, similar to the previous problem.

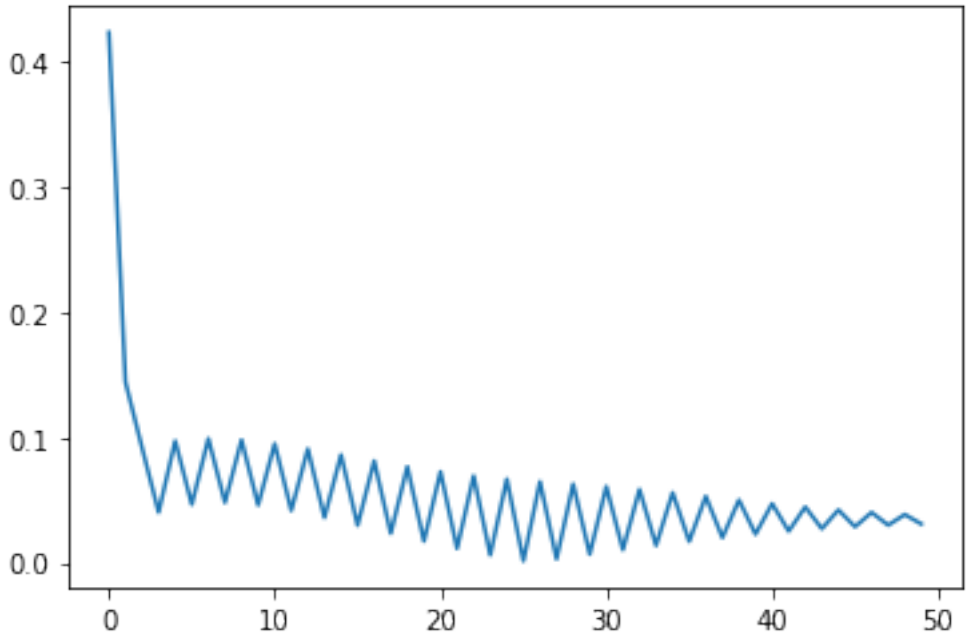

Figure 16: Epoch-Loss function plot with *hidden_size=5* and *num_epochs=50*

I guess that the learning rate is somewhat higher than needed, maybe decreasing it world give better results.

Decreasing the hidden size leads to 2 results in the loss function oscillates wildly. Reason unknown.

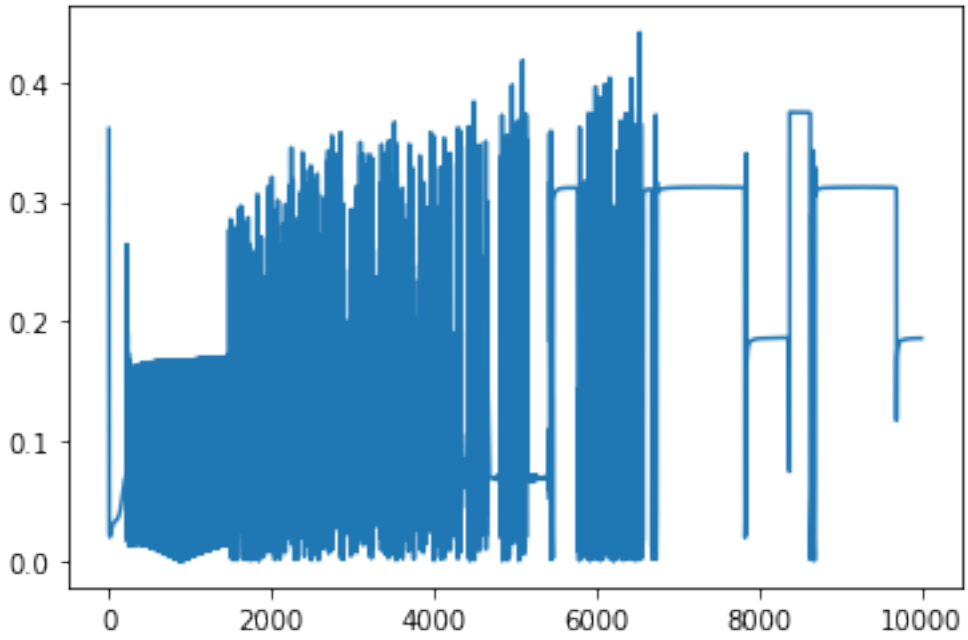

Figure 17: Epoch-Loss function plot with *hidden_size=2* and *num_epochs=10000*

Increasing the hidden size to 20, The loss function looks a bit weird but acceptable nonetheless.

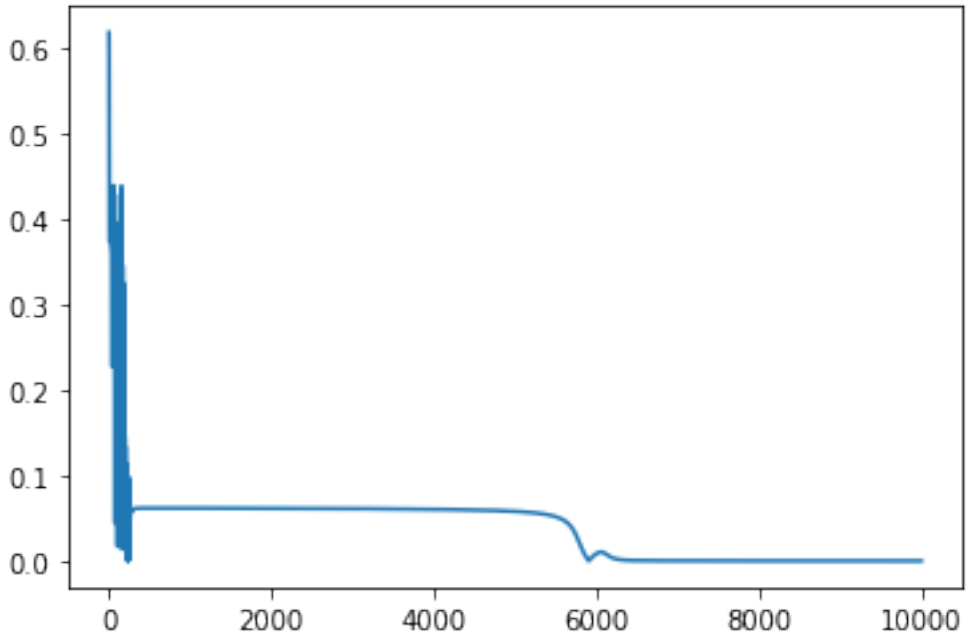

Figure 18: Epoch-Loss function plot with *hidden_size=20* and *num_epochs=10000*