Acquired Intelligence & Adaptive Behaviour: Quantitative Investigation of How the Number of Layers Affects the Performance of a Feedforward Neural Network

Name: Ufkun Ozalp

March 5, 2022

1 Abstract

The number of layers and the number of nodes in each hidden layer are the two key hyper-parameters that influence the performance of artificial neural networks. Depending on the complexity and the type of the problem, increasing the number of hidden layer might or might not improve the accuracy and therefore the performance of a feedforward neural network. In this paper, the impact of the number of layers of a feedforward neural network will be investigated within the scope of XOR gate problem.

2 Introduction

A hidden layer is positioned between the algorithm's input and output in neural networks, and it applies weights to the inputs and directs them through an activation function as the output. In a nutshell, the hidden layers conduct nonlinear changes on the network's inputs. The function of the neural network determines the hidden layers, and the layers themselves may vary depending on their associated weights. Hidden layers are required to represent non-linear separable functions or decisions. In other words, hidden layers are necessary to solve the problems where linear classifiers cannot solve such as XOR gate. It is expected that, a neural network with 1 hidden layer is sufficient to solve XOR gate problem. It is assumed that increasing the number of hidden layers would not improve the total epochs. Moreover it might even have negative impact on the amount of time required to solve the problem. In order to examine the hypothesis, multi-layer neural network structure will be built using Python and different scenarios will be tested.

3 Method

XOR gate problem will be implemented using pytorch to investigate the impact of the number of layers on the performance of a neural network. Artificial neural networks can be implemented in two steps: (1) building the neural network and (2) learning the weights of the neural network. Neural networks involve some input x, some weights W, optionally, a bias b, an activation function f, and an output $\hat{y}$. A pseudocode is given below to show the illustrate the prediction output of a multi-layer perceptron:

$$h = f(W_1 \cdot x + b_1)$$

$$\hat{y} = W_2 \cdot h + b_2$$

As activation functions, sigmoid, reLU or tanh functions can be used.

In order to implement a multi-layer neural network, optimizers must be used. Optimizers will collect gradients as computations are done and then perform optimisation algorithms (such as stochastic gradient descent) to update the

parameters (weights and biases). The loss function provides a measure of "how wrong" a prediction was.

Calculating gradients and updating weights and biases consist of four steps, detailed below (these must be performed in order).

Clear gradients: This is an artefact of the way pytorch works. Before calculating new gradients, gradients must be cleared out. This can be done by calling $optimizer.zero\_grad()$

Predict output: See the pseudocode above.

Calculate loss: Illustrates how the prediction is wrong. This can be done with $loss\_fn(prediction, true)$

Calculate gradients: Illustrates the gradients of the loss with respect to given weights and biases. This can be achieved with $loss.backward()$

Update weights and biases: Once the gradients are known, they must be used to update the weights and biases. This can be done with $optimizer.step()$

For each test, values in weights and bias tensors are randomly generated. tanh activation function is used. Each hidden layer has the same amount of hidden neurons. Such value is stored in *hidden_size* variable (equals 8 by default, therefore each hidden layer has 8 neurons). Learning rate is set to 0.03. In order to distinguish the impact of the number of hidden layers, 2 scenarios are tested: (1) a neural network with 1 hidden layer and (2) a neural network with 3 hidden layers.

4 Results

The impact of hidden layer number in a neural network can be vary. Also accuracy and performance of a neural network depend on other parameters as well such as type of the problem, number of hidden neurons in each hidden layer, activation function, learning rate, etc. For the XOR gate problem, while making the number of hidden neurons in each hidden layer, activation function and learning rate constant, increasing the number of layers led to negative impact on the number of epochs. With 1 hidden-layer neural network, XOR gate problem can be solved between 200-600 epochs (See appendix for a sample output). However, with 3-hidden-layer neural network, the same problem solved in more than 1500 epochs in the sample output. Since weights and bias tensors are being created with random numbers in each test, results of multilayered neural networks vary. Some tests could not solve the problem within 10000 epochs. Other than the number of layers, it is also observed that tanh activation function performed better than sigmoid or relu activation functions in the test.

5 Discussion

Contribution of the number of layers in a neural network is debatable since it depends on the type of the problem. Based on the test results, providing one hidden layer for solving the XOR gate problem is more than sufficient and it is the most efficient choice. Increasing the number of layers increases the complexity and therefore increases the required amount of time to solve the problem which is an expected result. Therefore the test results match up with the hypothesis.

6 Conclusion

To sum, since it is not linearly separable, in order to solve the XOR gate problem, a multi-layer neural network must be used. However, increasing the number of hidden layers also increase the total number of epochs. Therefore using one hidden layer is more efficient than having more hidden layers to solve the XOR gate problem.

References

[1] C. Buckley, Lab Sheet 1. [Online]. Available: https://colab.research.google.com/drive/1SWh_PbrNfN_jmgg_s-K43IoIrEBkx2wX?usp=sharing. [Accessed: 05-Mar-2022].

[2] J. Brownlee, "How to configure the number of layers and nodes in a neural network," Machine Learning Mastery, 06-Aug-2019. [Online]. Available: https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/. [Accessed: 05-Mar-2022].

[3] J. Brownlee, "How to configure the number of layers and nodes in a neural network," Machine Learning Mastery, 06-Aug-2019. [Online]. Available: https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/. [Accessed: 05-Mar-2022].

[4] M. Uzair and N. Jamil, "Effects of hidden layers on the efficiency of Neural Networks," 2020 IEEE 23rd International Multitopic Conference (INMIC), 2020.

Appendix

XOR gate problem solved using tanh activation function with 1 hidden layer, 8 hidden nodes and 0.03 learning rate:

```
# Provides extra neural network functions
import torch.nn as nn
# Provides optimizers
import torch.optim as optim

# number of epochs
num_epochs = 10000

# Define our data for XOR problem
input_data = torch.tensor([[0., 0.], [0., 1.], [1., 0.], [1., 1.]])
output_data = torch.tensor([[0.], [1.], [1.], [0.]])

input_size = 2
hidden_size = 8

# Define weights & biases for first layer (todo)
W_1 = torch.rand(input_size, hidden_size)
b_1 = torch.rand(1, 1)
# Define weights & biases for second layer (todo)
W_2 = torch.rand(hidden_size, 1)
b_2 = torch.rand(1, 1)

x = nn.Parameter(x)
W_1 = nn.Parameter(W_1)
W_2 = nn.Parameter(W_2)
b_1 = nn.Parameter(b_1)
b_2 = nn.Parameter(b_2)

# Setup our loss function
loss_fn = nn.MSELoss()

# Setup our optimizer
optimizer = optim.SGD([W_1, W_2, b_1, b_2], lr=0.03)

# Define our predict function (todo)
def predict(x, W_1, W_2, b_1, b_2):
    h = torch.tanh(torch.mm(x, W_1) + b_1)
    output = torch.mm(h, W_2) + b_2
    return output

# Training loop
for epoch in range(num_epochs):
    for i in range(input_data.size(0)):
        # Get example `i` (and unsqueeze to [1, 2] and [1, 1])
        x = input_data[i].unsqueeze(0)
        y = output_data[i].unsqueeze(0)


        # Clear gradients (todo)
        optimizer.zero_grad()
        # Predict outputs (todo)
        y_hat = predict(x, W_1, W_2, b_1, b_2)
```

```
        # Calculate loss (todo)
        loss = loss_fn(y_hat, y)
        # Calculate gradients (todo)
        loss.backward()
        # Update weights (todo)
        optimizer.step()

    # Test our network
    if epoch % 100 == 0:
        print(f"Testing network @ epoch {epoch}")
        for i in range(input_data.size(0)):
            # Make a prediction
            x = input_data[i].unsqueeze(0)
            y = output_data[i].unsqueeze(0)
            y_hat = predict(x, W_1, W_2, b_1, b_2)
            # Print result
            print("Input:{} Target: {} Predicted:[{}] Error:[{}]".
    format(
                x.data.numpy(),
                y.data.numpy(),
                np.round(y_hat.data.numpy(), 4),
                np.round(y.data.numpy() - y_hat.data.numpy(), 4)
            ))
```

Sample output of the code:

```
Testing network @ epoch 0
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[-2.5895]]] Error:[[[2.5895]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[-1.4035]]] Error:[[[2.4035]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[-1.2186]]] Error:[[[2.2186]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.3276]]] Error:[[[-0.3276]]]
Testing network @ epoch 100
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.2061]]] Error:[[[-0.2061]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.7994]]] Error:[[[0.2006]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.7193]]] Error:[[[0.2807]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.2238]]] Error:[[[-0.2238]]]
Testing network @ epoch 200
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.0221]]] Error:[[[-0.0221]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.9809]]] Error:[[[0.0191]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.9729]]] Error:[[[0.0271]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.0193]]] Error:[[[-0.0193]]]
Testing network @ epoch 300
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.0013]]] Error:[[[-0.0013]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.9989]]] Error:[[[0.0011]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.9984]]] Error:[[[0.0016]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.0011]]] Error:[[[-0.0011]]]
Testing network @ epoch 400
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[1.e-04]]] Error:[[[-1.e-04]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.9999]]] Error:[[[1.e-04]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.9999]]] Error:[[[1.e-04]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[1.e-04]]] Error:[[[-1.e-04]]]
Testing network @ epoch 500
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.]]] Error:[[[-0.]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[1.]]] Error:[[[0.]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[1.]]] Error:[[[0.]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.]]] Error:[[[-0.]]]
```

XOR gate problem solved using tanh activation function with 3 hidden layers, 8 hidden nodes and 0.03 learning rate:

```
# Provides extra neural network functions
import torch.nn as nn
# Provides optimizers
import torch.optim as optim

# number of epochs
num_epochs = 10000

# Define our data for XOR problem
input_data = torch.tensor([[0., 0.], [0., 1.], [1., 0.], [1., 1.]])
output_data = torch.tensor([[0.], [1.], [1.], [0.]])

input_size = 2
hidden_size = 8

# Define weights & biases for first layer (todo)
W_1 = torch.rand(input_size, hidden_size)
b_1 = torch.rand(1, 1)
# Define weights & biases for second layer (todo)
W_2 = torch.rand(hidden_size, hidden_size)
b_2 = torch.rand(1, 1)
# Define weights & biases for the third layer (todo)
W_3 = torch.rand(hidden_size, hidden_size)
b_3 = torch.rand(1, 1)
# Define weights & biases for the fourth layer (todo)
W_4 = torch.rand(hidden_size, 1)
b_4 = torch.rand(1, 1)


x = nn.Parameter(x)
W_1 = nn.Parameter(W_1)
b_1 = nn.Parameter(b_1)
W_2 = nn.Parameter(W_2)
b_2 = nn.Parameter(b_2)
W_3 = nn.Parameter(W_3)
b_3 = nn.Parameter(b_3)
W_4 = nn.Parameter(W_4)
b_4 = nn.Parameter(b_4)

# Setup our loss function
loss_fn = nn.MSELoss()

# Setup our optimizer
optimizer = optim.SGD([W_1, W_2, W_3, W_4, b_1, b_2, b_3, b_4], lr
    =0.03)

# Define our predict function (todo)
def predict(x, W_1, W_2, W_3, W_4, b_1, b_2, b_3, b_4):
    z1 = torch.mm(x, W_1) + b_1
    h1 = torch.tanh(z1)
    z2 = torch.mm(h1, W_2) + b_2
    h2 = torch.tanh(z2)
    z3 = torch.mm(h2, W_3) + b_2
    h3 = torch.tanh(z3)
    output = torch.mm(h3, W_4) + b_4
```

```
    return output

# Training loop
for epoch in range(num_epochs):
    for i in range(input_data.size(0)):
        # Get example `i` (and unsqueeze to [1, 2] and [1, 1])
        x = input_data[i].unsqueeze(0)
        y = output_data[i].unsqueeze(0)


        # Clear gradients (todo)
        optimizer.zero_grad()
        # Predict outputs (todo)
        y_hat = predict(x, W_1, W_2, W_3, W_4, b_1, b_2, b_3, b_4)
        # Calculate loss (todo)
        loss = loss_fn(y_hat, y)
        # Calculate gradients (todo)
        loss.backward()
        # Update weights (todo)
        optimizer.step()

    # Test our network
    if epoch % 100 == 0:
        print(f"Testing network @ epoch {epoch}")
        for i in range(input_data.size(0)):
            # Make a prediction
            x = input_data[i].unsqueeze(0)
            y = output_data[i].unsqueeze(0)
            y_hat = predict(x, W_1, W_2, W_3, W_4, b_1, b_2, b_3,
    b_4)
            # Print result
            print("Input:{} Target: {} Predicted:[{}] Error:[{}]".
    format(
                x.data.numpy(),
                y.data.numpy(),
                np.round(y_hat.data.numpy(), 4),
                np.round(y.data.numpy() - y_hat.data.numpy(), 4)
            ))
```

Sample output of the code:

```
Testing network @ epoch 0
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.3955]]] Error:[[[-0.3955]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.3957]]] Error:[[[0.6043]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4051]]] Error:[[[0.5949]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4107]]] Error:[[[-0.4107]]]
Testing network @ epoch 100
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.3975]]] Error:[[[-0.3975]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.4107]]] Error:[[[0.5893]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.415]]] Error:[[[0.585]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4196]]] Error:[[[-0.4196]]]
Testing network @ epoch 200
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.398]]] Error:[[[-0.398]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.4141]]] Error:[[[0.5859]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.417]]] Error:[[[0.583]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4217]]] Error:[[[-0.4217]]]
Testing network @ epoch 300
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.3988]]] Error:[[[-0.3988]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.4164]]] Error:[[[0.5836]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4184]]] Error:[[[0.5816]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4232]]] Error:[[[-0.4232]]]
Testing network @ epoch 400
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.3993]]] Error:[[[-0.3993]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.419]]] Error:[[[0.581]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4205]]] Error:[[[0.5795]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4255]]] Error:[[[-0.4255]]]
Testing network @ epoch 500
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.4013]]] Error:[[[-0.4013]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.4245]]] Error:[[[0.5755]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4255]]] Error:[[[0.5745]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4311]]] Error:[[[-0.4311]]]
Testing network @ epoch 600
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.4099]]] Error:[[[-0.4099]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.4366]]] Error:[[[0.5634]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4371]]] Error:[[[0.5629]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4433]]] Error:[[[-0.4433]]]
Testing network @ epoch 700
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.4059]]] Error:[[[-0.4059]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.439]]] Error:[[[0.561]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4387]]] Error:[[[0.5613]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4464]]] Error:[[[-0.4464]]]
Testing network @ epoch 800
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.3909]]] Error:[[[-0.3909]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.4403]]] Error:[[[0.5597]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4386]]] Error:[[[0.5614]]]
```

```
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4497]]] Error:[[[-0.4497]]]
Testing network @ epoch 900
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.3627]]] Error:[[[-0.3627]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.4452]]] Error:[[[0.5548]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4398]]] Error:[[[0.5602]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.457]]] Error:[[[-0.457]]]
Testing network @ epoch 1000
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.3001]]] Error:[[[-0.3001]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.4596]]] Error:[[[0.5404]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4453]]] Error:[[[0.5547]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4727]]] Error:[[[-0.4727]]]
Testing network @ epoch 1100
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.1517]]] Error:[[[-0.1517]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.4775]]] Error:[[[0.5225]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4506]]] Error:[[[0.5494]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4886]]] Error:[[[-0.4886]]]
Testing network @ epoch 1200
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.0644]]] Error:[[[-0.0644]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.5038]]] Error:[[[0.4962]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4831]]] Error:[[[0.5169]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.5101]]] Error:[[[-0.5101]]]
Testing network @ epoch 1300
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[0.0065]]] Error:[[[-0.0065]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.5148]]] Error:[[[0.4852]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.5073]]] Error:[[[0.4927]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.517]]] Error:[[[-0.517]]]
Testing network @ epoch 1400
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[1.e-04]]] Error:[[[-1.e-04]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.5137]]] Error:[[[0.4863]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.5103]]] Error:[[[0.4897]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.5119]]] Error:[[[-0.5119]]]
Testing network @ epoch 1500
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[-0.062]]] Error:[[[0.062]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.5129]]] Error:[[[0.4871]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.5052]]] Error:[[[0.4948]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.5057]]] Error:[[[-0.5057]]]
Testing network @ epoch 1600
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[-0.0213]]] Error:[[[0.0213]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[0.5299]]] Error:[[[0.4701]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.4955]]] Error:[[[0.5045]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[0.4926]]] Error:[[[-0.4926]]]
Testing network @ epoch 1700
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[-0.0013]]] Error:[[[0.0013]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[1.0202]]] Error:[[[-0.0202]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[0.991]]] Error:[[[0.009]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[-0.0011]]] Error:[[[0.0011]]]
```

```
Testing network @ epoch 1800
Input:[[0. 0.]] Target: [[0.]] Predicted:[[[-0.]]] Error:[[[0.]]]
Input:[[0. 1.]] Target: [[1.]] Predicted:[[[1.]]] Error:[[[-0.]]]
Input:[[1. 0.]] Target: [[1.]] Predicted:[[[1.]]] Error:[[[0.]]]
Input:[[1. 1.]] Target: [[0.]] Predicted:[[[-0.]]] Error:[[[0.]]]
```