Jared Yu
Module 8 Assignment

1. Show a perceptron that calculates $\bar{x}_1 x_2 \bar{x}_3$.

Ans: (References: [1], [2])

The approach for this problem and every other problem are similar. First, the PyTorch package in the Python programming language was used to do the forward and backward propagation steps over a set number of epochs in order to train a set of weights that could solve each problem. The inputs to each of the perceptrons (or neural networks) are a 3-dimensional input representing binary values for x_1, x_2, and x_3, while the target output is a 1-dimensional array consisting of the corresponding truth values based on the truth table. The truth table for Problem 1 can be seen below in Table 1.

Table 1 Truth table for Problem 1.

x_1	x_2	x_3	$\bar{x}_1$	$\bar{x}_3$	$\bar{x}_1 x_2 \bar{x}_3$
1	1	1	0	0	0
1	1	0	0	1	0
1	0	1	0	0	0
1	0	0	0	1	0
0	1	1	1	0	0
0	1	0	1	1	1
0	0	1	1	0	0
0	0	0	1	1	0

Then the perceptron graph created in PyTorch for Problem 1 can be seen below in Figure 1. The way that it works is that the x_1, x_2, and x_3 inputs from Table 1 are summed together along with $x_0 = 1$ (the bias term) after being multiplied by their corresponding weights $w_0, \cdots, w_3$. After going through the summation, $\sum$, which is calculating $x_0 w_0 + \cdots + x_3 w_3$, it then passes through the activation function, f, which here is the sigmoid function $f(x) = \frac{1}{1+\exp(-x)}$. This gives the final output $y = f(x)$. The trick here is to find the suitable set of weights, such that this graph will give the desired outputs in the furthest right column of Table 1 after inserting the corresponding x_1, x_2, and x_3.

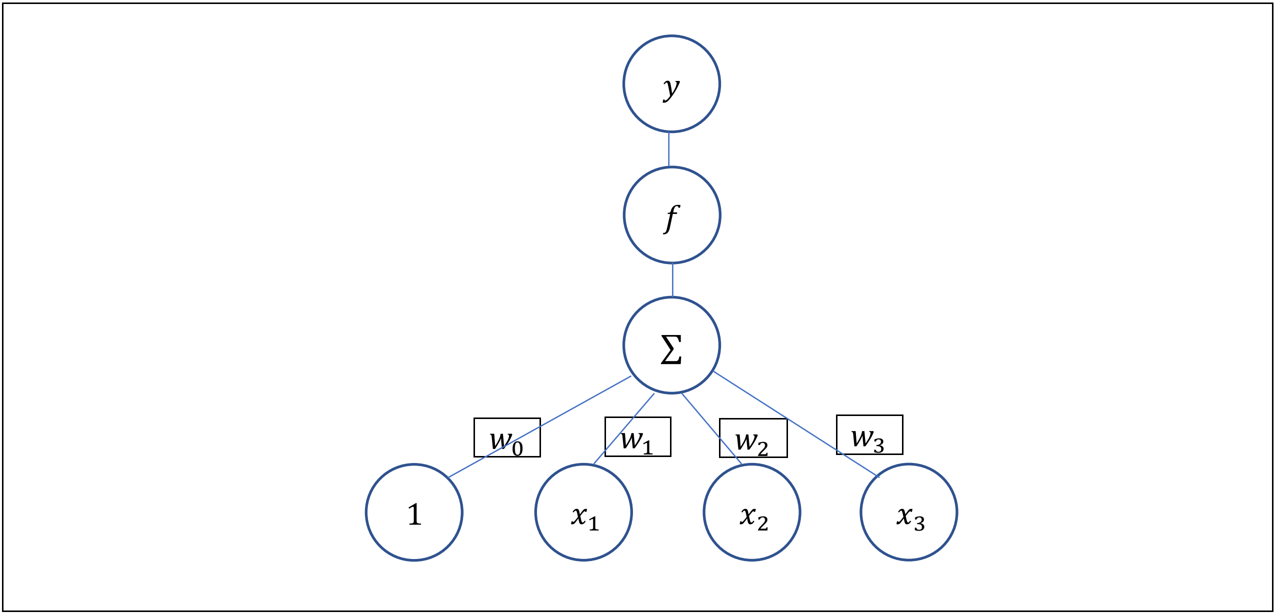

Figure 1 The above figure shows the perceptron in Problem 1.

After training for 10,000 epochs, the trained weights can be seen below in Table 2.

Table 2 The trained weights for Problem 1.

w_0	w_1	w_2	w_3
≈ -3.8801	≈ -7.6102	≈ 7.3863	≈ -7.6146

We can check that these weights are correct by first setting a rule. The sigmoid function itself doesn't automatically generate a binary output of 1 or 0. Instead, it typically outputs a probability between that range. Therefore, after having trained the weights using the above architecture seen in Figure 1, we can set a rule that if the output of $f(x) \geq 0.5$, then the perceptron has decided that the output is 1. Otherwise, if $f(x) < 0.5$, then the perceptron has decided that the output is 0.

To clarify this approach, we can set a function $g(x_1, x_2, x_3)$ as follows:

$$g(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } f(\Sigma_{i=0}^{3} w_i x_i) \geq 0.5 \\ 0 & \text{if } f(\Sigma_{i=0}^{3} w_i x_i) < 0.5, \end{cases}$$

where $f(x) = \frac{1}{1+\exp(-x)}$ is the sigmoid function. The tested outputs can be seen below in Table 3. Looking at the table, it's clear that after the inputs go through the perceptron that the results match what is desired.

Table 3 The range of outputs for the trained perceptron.

x_1	x_2	x_3	$f(\Sigma_{i=0}^{3} w_i x_i)$	$g(x_1, x_2, x_3)$	$\bar{x}_1 x_2 \bar{x}_3$
1	1	1	8.1418×10^{-6}	0	0
1	1	0	0.0162	0	0
1	0	1	5.0454×10^{-9}	0	0
1	0	0	1.0229×10^{-5}	0	0
0	1	1	0.0162	0	0
0	1	0	0.9709	1	1
0	0	1	1.0185×10^{-5}	0	0
0	0	0	0.0202	0	0

2. Show a perceptron that calculates $\bar{x}_2 x_3$.

Ans: (References: [1], [2])
Problem 2 is highly similar to Problem 1, except that the desired output is different. The truth table for Problem 2 can be seen below in Table 4. The method for solving it is the same as what is seen in Problem 1, therefore the perceptron architecture will not be redrawn. The solution for problem 2 involves the same combination of $f(\Sigma_{i=0}^{3} w_i x_i)$ and $g(x_1, x_2, x_3)$.

Table 4 Truth table for Problem 2.

x_1	x_2	x_3	$\bar{x}_2$	$\bar{x}_2 x_3$
1	1	1	0	0
1	1	0	0	0
1	0	1	1	1
1	0	0	1	0
0	1	1	0	0
0	1	0	0	0
0	0	1	1	1
0	0	0	1	0

After training for 10,000 epochs, the trained weights for Problem 2 can be seen below in Table 5.

Table 5 The trained weights for Problem 2.

w_0	w_1	w_2	w_3
≈ -4.1868	≈ -0.1176	≈ -8.5621	≈ 8.3490

Then, to test that these weights are correct, the following Table 6 shows the results from reinserting the inputs x_1, x_2, and x_3. It clearly shows that the trained weights are capable of solving Problem 2.

Table 6 The range of outputs for the trained perceptron.

x_1	x_2	x_3	$f(\Sigma_{i=0}^{3} w_i x_i)$	$g(x_1, x_2, x_3)$	$\bar{x}_2 x_3$
1	1	1	0.0108	0	0
1	1	0	2.5829×10^{-6}	0	0
1	0	1	0.9828	1	1
1	0	0	0.0133	0	0
0	1	1	0.0121	0	0
0	1	0	2.9052×10^{-6}	0	0
0	0	1	0.9847	1	1
0	0	0	0.0150	0	0

3. Show a neural network that calculates $\bar{x}_1\bar{x}_2\bar{x}_3 + x_1x_2x_3$.

Ans: (References: [1], [2])
For problems 3-5, the situation is slightly different from what was seen previously in problems 1-2. The difference is that rather than the network architecture being that of a perceptron, here it is that of a multilayer perceptron, otherwise known as a neural network. Here, there is the possibility of one to many "hidden layers" between the input and the output. This concept will be illustrated below in Figure 2. Since it involves hidden layers with multiple hidden units, rather than using some graph to visualize it, the math is directly shown instead.

$$\boxed{\begin{aligned}
\underbrace{\mathbf{a}_1}_{1\times5} &= \underbrace{\mathbf{x}^\top}_{1\times3} \times \underbrace{\mathbf{W}_1}_{3\times5} + \underbrace{\mathbf{b}_1}_{1\times5} \quad (1)\\
\underbrace{\mathbf{z}}_{1\times5} &= sigmoid(\mathbf{a}_1) \quad (2)\\
\underbrace{\mathbf{a}_2}_{1\times1} &= \underbrace{\mathbf{z}}_{1\times5} \times \underbrace{\mathbf{W}_2}_{5\times1} + \underbrace{\mathbf{b}_2}_{1\times1} \quad (3)\\
\underbrace{y}_{1\times1} &= sigmoid(\mathbf{a}_2) \quad (4)
\end{aligned}}$$

Figure 2 The above figure shows the forward propagation of the neural network in Problem 3.

The math is displaying the forward propagation steps of the neural network. It can be seen that in Equation (1) within Figure 2, the first activation is being calculated. This takes in the x_1, x_2, and x_3 as input and matrix multiplies it with the first weight matrix, $\mathbf{W}_1$, this represents the weights within the first layer that has dimensions 3×5. This indicates that in the first hidden layer, there are a total of 5 hidden units. Additionally, it is added with $\mathbf{b}_1$, which is the bias term within the first layer. In Equation (2) within Figure 2, the result of the activation (a vector) is put through the sigmoid activation function to calculate $\mathbf{z}$, where each element of the vector has the sigmoid function applied to it. In Equation (3) within Figure 2, the output of the first activation function then is matrix multiplied with the second weight matrix, $\mathbf{W}_2$. This is then added to the bias term, $\mathbf{b}_2$ to create the second activation output, $\mathbf{a}_2$. This final activation is then sent again through a sigmoid activation function to give the output y.

The truth table for Problem 3 can be seen below in Table 7.

Table 7 Truth table for Problem 3.

x_1	x_2	x_3	$\bar{x}_1$	$\bar{x}_2$	$\bar{x}_3$	$\bar{x}_1\bar{x}_2\bar{x}_3$	$x_1x_2x_3$	$\bar{x}_1\bar{x}_2\bar{x}_3 + x_1x_2x_3$
1	1	1	0	0	0	0	1	1
1	1	0	0	0	1	0	0	0
1	0	1	0	1	0	0	0	0
1	0	0	0	1	1	0	0	0
0	1	1	1	0	0	0	0	0
0	1	0	1	0	1	0	0	0
0	0	1	1	1	0	0	0	0
0	0	0	1	1	1	1	0	1

The weights and biases explained in Figure 2 can be seen below in Table 8. They were calculated after 100,000 epochs of forward and backward propagation passes.

Table 8 The below table shows the weights and biases calculated for Problem 3.

$\mathbf{W}_1 =$
```
[-4.8960,  3.0015,  3.1421],
[ 3.0452,  2.3374,  2.1497],
[ 4.6246,  5.7488,  5.4122],
[-0.6025, -1.4961,  4.4929],
[ 3.8875,  2.1354, -2.6413]
```

$\mathbf{b}_1 =$
```
[ 0.9898, -6.1660, -2.6799,  0.4270, -1.5372]
```

$\mathbf{W}_2 =$
```
[  4.0389,   7.2320, -12.4403,   3.7324,   5.1744]
```

$\mathbf{b}_2 =$
```
[-1.1351]
```

The same approach from previously is applied here. After having calculated the weights, they are tested by having all the possible inputs passed through the forward propagation stage. Again, since the final activation function/output function is a sigmoid function, the result is a probability between 0 and 1. The same rule is applied where if $f(x) \geq 0.5$, then the decision is that the model has decided 1 and 0 if $f(x) < 0.5$. The only difference now is that there is a hidden layer within this model. So, the result is not simply $f(\Sigma_{i=0}^{3} w_i x_i)$, but $f(\mathbf{a}_2) = f(f(\mathbf{a}_1) \times \mathbf{W}_2 + \mathbf{b}_2)$, where f is the sigmoid function and the other terms have been previously been explained. The results can be seen below in Table 9. They clearly show how the forward propagation steps using the trained weights leads to the correct results based on the decision rule of the g function.

Table 9 The range of outputs for the trained perceptron.

x_1	x_2	x_3	$f(\mathbf{a}_2)$	$g(x_1, x_2, x_3)$	$\bar{x}_1\bar{x}_2\bar{x}_3 + x_1x_2x_3$
1	1	1	0.9878	1	1
1	1	0	0.0118	0	0
1	0	1	0.0121	0	0
1	0	0	0.0055	0	0
0	1	1	0.0149	0	0
0	1	0	0.0098	0	0
0	0	1	0.0072	0	0
0	0	0	0.9852	1	1

4. Show a neural network that calculates $\bar{p}_1 + x_1x_2x_3$.

Ans: (References: [1], [2])
The truth table for Problem 4 can be seen below in Table 10.

Table 10 Truth table for Problem 4.

x_1	x_2	x_3	p_1	$\bar{p}_1$	$x_1x_2x_3$	$\bar{p}_1 + x_1x_2x_3$
1	1	1	1	0	1	1
1	1	0	1	0	0	1
1	0	1	1	0	0	1
1	0	0	1	0	0	1
0	1	1	0	1	0	0
0	1	0	0	1	0	0
0	0	1	0	1	0	0
0	0	0	0	1	0	0

Problem 4 follows the same format as problem 3. The same architecture in Figure 2 is also used, where there are 5 units in the first hidden layer. A difference here is that the training only took 10,000 epochs rather than 100,00 like before.

Table 11 The below table shows the weights and biases calculated for Problem 4.

$\mathbf{W}_1 =$
```
[ 3.4203, -0.1661, -0.0407],
[-0.8649, -2.1062, -0.8363],
[-2.8322, -0.2734,  0.0121],
[-1.6227, -0.0411,  0.3832],
[-4.8063,  0.3051,  0.0914]
```
$\mathbf{b}_1 =$
```
[-1.0902,  0.2653,  0.9703, -0.2629,  1.7901]
```
$\mathbf{W}_2 =$
```
[ 4.8701, -0.1386, -2.7994, -1.5249, -6.1054]
```
$\mathbf{b}_2 =$
```
[1.6766]
```

Like before, to check the weights, they are tested by putting all the inputs through the forward propagation stage. As stated previously, the decision rule is that the model decides 1 if $f(\mathbf{a}_2) \geq 0.5$ and 0 if $f(\mathbf{a}_2) < 0.5$. They clearly show how the forward propagation steps using the trained weights leads to the correct results based on the decision rule of the g function.

Table 12 The range of outputs for the trained perceptron.

x_1	x_2	x_3	$f(\mathbf{a}_2)$	$g(x_1, x_2, x_3)$	$\bar{x}_1\bar{x}_2\bar{x}_3 + x_1x_2x_3$
1	1	1	0.9936	1	1
1	1	0	0.9943	1	1
1	0	1	0.9940	1	1
1	0	0	0.9945	1	1
0	1	1	0.0043	0	0
0	1	0	0.0054	0	0
0	0	1	0.0048	0	0
0	0	0	0.0061	0	0

5. Show a neural network that calculates $p_2(x_1, x_2) \cdot p_3(x_1, x_2, x_3)$.

Ans: (References: [1], [2])
The truth table for Problem 5 can be seen below in Table 13.

Table 13 Truth table for Problem 5.

x_1	x_2	x_3	$p_2(x_1, x_2)$	$p_3(x_1, x_2, x_3)$	$p_2(x_1, x_2) \cdot p_3(x_1, x_2, x_3)$
1	1	1	0	1	0
1	1	0	0	0	0
1	0	1	1	0	0
1	0	0	1	1	1
0	1	1	1	0	0
0	1	0	1	1	1
0	0	1	0	1	0
0	0	0	0	0	0

The approach in problem 3 and 4 are done again here in problem 5. The same architecture is used where there are 5 units in the hidden layer, and it was trained using 100,000 epochs. The weights and biases can be seen below in Table 14.

Table 14 The below table shows the weights and biases calculated for Problem 5.

```
W1 =[-2.0187, -1.7206, -1.4887],
    [ 4.2645,  3.9943,  2.4229],
    [ 1.0179, -2.0601, -0.1133],
    [-5.3538, -5.5443,  4.5531],
    [ 3.4594,  3.7288,  0.5516]
b1 =[ 2.8620, -6.2845, -0.8694,  2.0805, -0.9889]
W2 =[ 3.5864, -8.9740,  1.3763, -9.0327,  3.5677]
b2 =[-0.9460]
```

Once again, the trained weights and biases are tested to check whether they can solve the problem. The results like before are shown below in Table 15. They clearly show how the forward propagation steps using the trained weights leads to the correct results based on the decision rule of the g function.

Table 15 The range of outputs for the trained perceptron.

x_1	x_2	x_3	$f(\mathbf{a}_2)$	$g(x_1,x_2,x_3)$	$\bar{x}_1\bar{x}_2\bar{x}_3 + x_1x_2x_3$
1	1	1	0.0027	0	0
1	1	0	0.0174	0	0
1	0	1	0.0003	0	0
1	0	0	0.9854	1	1
0	1	1	0.0006	0	0
0	1	0	0.9836	1	1
0	0	1	0.0040	0	0
0	0	0	0.0144	0	0

6. Generate two-dimensional samples for each of two Gaussians, $p(\mathbf{x}|\omega_i) \sim N(\boldsymbol{\mu}_i, \Sigma_i)$ with
$$\boldsymbol{\mu}_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \boldsymbol{\mu}_2 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \Sigma_1 = \begin{pmatrix} 0.2 & 0 \\ 0 & 0.2 \end{pmatrix} = \Sigma_2.$$
Produce 100 samples from each distribution, but to guarantee linear separability, produce a replacement for a Class 1 vector whenever $x_1 - x_2 < 1$ and produce a replacement for a Class 2 vector whenever $x_1 - x_2 > 1$.

   Use these vectors to design a linear classifier using the perceptron algorithm. After convergence, draw the decision boundary.

Ans: (Reference: [3])

The same steps done in problems 1 and 2 were done again in 6. After generating the data, a model was created using the same architecture. The only difference is that there are 2 inputs instead of 3. The same classification rule applies, where the model decides 1 if $f(x) \geq 0.5$ and 0 if $f(x) < 0.5$. Below in Table 16 it shows the weights and bias calculated in the perceptron after 100,000 epochs.

Table 16 The below table shows the weights and bias calculated for Problem 6.

w_1	w_2	b
≈ -26.9488	≈ 28.1079	≈ 28.9409

Then, the points were plotted onto a scatter plot which can be seen below in Figure 3. The blue points indicate Class 1 and red points indicate Class 2. The decision boundary is drawn as a straight line between the two groups of points. The reference [3] was used to determine the correct slope and y-intercept for the decision boundary line.

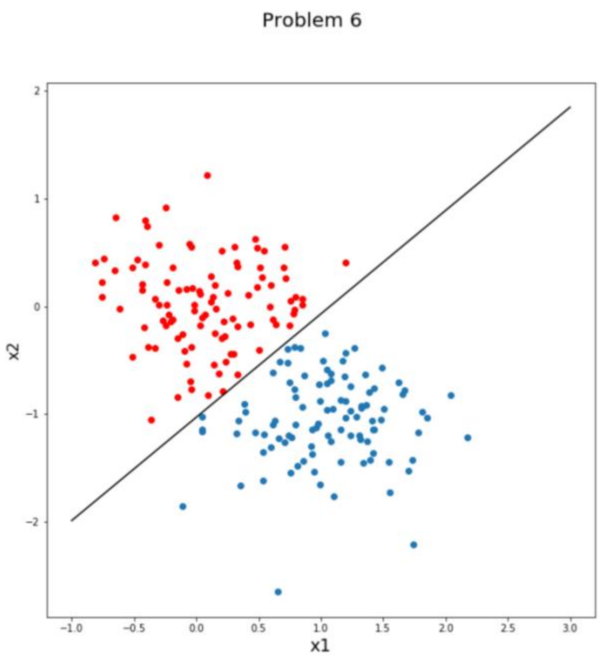

Figure 3 The above figure shows a scatter plot of the two classes and the resulting decision boundary. The blue points show Class 1 and the red points show Class 2.

Reference:

[1] https://courses.cs.washington.edu/courses/cse446/18wi/sections/section8/XOR-Pytorch.html

[2] https://piazza.com/class/kc0jkwru805u1?cid=157

[3] https://medium.com/@thomascountz/calculate-the-decision-boundary-of-a-single-perceptron-visualizing-linear-separability-c4d77099ef38

Code Appendix:

```
# https://courses.cs.washington.edu/courses/cse446/18wi/sections/section8/XOR-Pytorch.html

import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

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

X = torch.Tensor([[1, 1, 1],
                  [1, 1, 0],
                  [1, 0, 1],
                  [1, 0, 0],
                  [0, 1, 1],
                  [0, 1, 0],
                  [0, 0, 1],
                  [0, 0, 0]])

Y = torch.Tensor([0,0,0,0,0,1,0,0]).view(-1,1)

class Problem1(nn.Module):
    def __init__(self, input_dim = 3):
        super(Problem1, self).__init__()
        self.lin1 = nn.Linear(input_dim, 1)

    def forward(self, x):
        x = self.lin1(x)
        x = F.sigmoid(x)
        return x

model1 = Problem1()

def weights_init(model):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            # initialize the weight tensor, here we use a normal distribution
            m.weight.data.normal_(0, 1)

weights_init(model1)

loss_func = nn.MSELoss()

optimizer = optim.SGD(model1.parameters(), lr=0.02, momentum=0.9)

epochs = int(1e4 + 1.)
steps = X.size(0)
for i in range(epochs):
    for j in range(steps):
        data_point = np.random.randint(X.size(0))
        x_var = Variable(X[data_point], requires_grad=False)
        y_var = Variable(Y[data_point], requires_grad=False)

        optimizer.zero_grad()
        y_hat = model1(x_var)
        loss = loss_func.forward(y_hat, y_var)
        loss.backward()
        optimizer.step()

    if (i % 1e3 == 0):
        print("Epoch: {0}, Loss: {1}, ".format(i, loss.data.numpy()))

for i in range(len(Y)):
    print(model1(X[i,]))

Y

model_params = list(model1.parameters())

model_weights = model_params[0].data.numpy()
model_bias = model_params[1].data.numpy()

model_weights

model_bias

sigmoid(X[0,] @ model_weights.T + model_bias)

# https://www.programmersought.com/article/479977857/
model_params = list(model1.parameters())

vis_graph = make_dot(model1(X[1,]), params=dict(model1.named_parameters()))
vis_graph.view()

# x1, x2, x3...!x2...!x2x3
# 1, 1, 1; 0; 0
# 1, 1, 0; 0; 0
# 1, 0, 1; 1; 1
# 1, 0, 0; 1; 0
# 0, 1, 1; 0; 0
# 0, 1, 0; 0; 0
# 0, 0, 1; 1; 1
# 0, 0, 0; 1; 0

X = torch.Tensor([[1, 1, 1],
                  [1, 1, 0],
                  [1, 0, 1],
                  [1, 0, 0],
                  [0, 1, 1],
                  [0, 1, 0],
                  [0, 0, 1],
                  [0, 0, 0]])

Y = torch.Tensor([0,0,1,0,0,0,1,0]).view(-1,1)

class Problem2(nn.Module):
    def __init__(self, input_dim = 3):
        super(Problem2, self).__init__()
        self.lin1 = nn.Linear(input_dim, 1)

    def forward(self, x):
        x = self.lin1(x)
        x = F.sigmoid(x)
        return x

model2 = Problem2()
```

```
def weights_init(model):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            # initialize the weight tensor, here we use a normal distribution
            m.weight.data.normal_(0, 1)

weights_init(model2)

loss_func = nn.MSELoss()

optimizer = optim.SGD(model2.parameters(), lr=0.02, momentum=0.9)

epochs = int(1e4 + 1.)
steps = X.size(0)
for i in range(epochs):
    for j in range(steps):
        data_point = np.random.randint(X.size(0))
        x_var = Variable(X[data_point], requires_grad=False)
        y_var = Variable(Y[data_point], requires_grad=False)

        optimizer.zero_grad()
        y_hat = model2(x_var)
        loss = loss_func.forward(y_hat, y_var)
        loss.backward()
        optimizer.step()

    if (i % 1e3 == 0):
        print("Epoch: {0}, Loss: {1}, ".format(i, loss.data.numpy()))

for i in range(len(Y)):
    print(model2(X[i,]))

Y

model_params = list(model2.parameters())

model_weights = model_params[0].data.numpy()
model_bias = model_params[1].data.numpy()

model_weights

model_bias

sigmoid(X[0,] @ model_weights.T + model_bias)

# x1, x2, x3...ans
# 1, 1, 1; 1
# 1, 1, 0; 0
# 1, 0, 1; 0
# 1, 0, 0; 0
# 0, 1, 1; 0
# 0, 1, 0; 0
# 0, 0, 1; 0
# 0, 0, 0; 1

X = torch.Tensor([[1, 1, 1],
                  [1, 1, 0],
                  [1, 0, 1],
                  [1, 0, 0],
                  [0, 1, 1],
                  [0, 1, 0],
                  [0, 0, 1],
                  [0, 0, 0]])

Y = torch.Tensor([1,0,0,0,0,0,0,1]).view(-1,1)

class Problem3(nn.Module):
    def __init__(self, input_dim = 3, output_dim=1, hidden_dim=5):
        super(Problem3, self).__init__()
        self.lin1 = nn.Linear(input_dim, hidden_dim)
        self.lin2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.lin1(x)
        x = F.sigmoid(x)
        x = self.lin2(x)
        x = F.sigmoid(x)
        return x

model3 = Problem3()

def weights_init(model):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            # initialize the weight tensor, here we use a normal distribution
            m.weight.data.normal_(0, 1)
```

```
weights_init(model3)

loss_func = nn.MSELoss()

optimizer = optim.SGD(model3.parameters(), lr=0.02, momentum=0.9)

epochs = int(1e4 + 1.)
steps = X.size(0)
for i in range(epochs):
    for j in range(steps):
        data_point = np.random.randint(X.size(0))
        x_var = Variable(X[data_point], requires_grad=False)
        y_var = Variable(Y[data_point], requires_grad=False)

        optimizer.zero_grad()
        y_hat = model3(x_var)
        loss = loss_func.forward(y_hat, y_var)
        loss.backward()
        optimizer.step()

    if (i % 5e3 == 0):
        print("Epoch: {0}, Loss: {1}, ".format(i, loss.data.numpy()))

for i in range(len(Y)):
    print(model3(X[i,]))
#     print(model3(X[i,]), F.sigmoid(F.sigmoid(X[i,] @ W1.T + b1) @ W2.T + b2))

Y

model_params = list(model3.parameters())

model_params

# W1; (10x3) b1; (10,1); W2; (1,10); b2; (1,1)

model_params[0].shape, model_params[1].shape, model_params[2].shape, model_params[3].shape

W1 = model_params[0].data.numpy()
b1 = model_params[1].data.numpy()
W2 = model_params[2].data.numpy()
b2 = model_params[3].data.numpy()

F.sigmoid(F.sigmoid(X[0,] @ W1.T + b1) @ W2.T + b2)

# x1, x2, x3...ans
# 1, 1, 1; 1
# 1, 1, 0; 1
# 1, 0, 1; 1
# 1, 0, 0; 1
# 0, 1, 1; 0
# 0, 1, 0; 0
# 0, 0, 1; 0
# 0, 0, 0; 0

X = torch.Tensor([[1, 1, 1],
                  [1, 1, 0],
                  [1, 0, 1],
                  [1, 0, 0],
                  [0, 1, 1],
                  [0, 1, 0],
                  [0, 0, 1],
                  [0, 0, 0]])

Y = torch.Tensor([1,1,1,1,0,0,0,0]).view(-1,1)

class Problem4(nn.Module):
    def __init__(self, input_dim = 3, output_dim=1, hidden_dim=5):
        super(Problem4, self).__init__()
        self.lin1 = nn.Linear(input_dim, hidden_dim)
        self.lin2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.lin1(x)
        x = F.sigmoid(x)
        x = self.lin2(x)
        x = F.sigmoid(x)
        return x

model4 = Problem4()

def weights_init(model):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            # initialize the weight tensor, here we use a normal distribution
            m.weight.data.normal_(0, 1)

weights_init(model4)
```

```
loss_func = nn.MSELoss()

optimizer = optim.SGD(model4.parameters(), lr=0.02, momentum=0.9)

epochs = int(1e4 + 1.)
steps = X.size(0)
for i in range(epochs):
    for j in range(steps):
        data_point = np.random.randint(X.size(0))
        x_var = Variable(X[data_point], requires_grad=False)
        y_var = Variable(Y[data_point], requires_grad=False)

        optimizer.zero_grad()
        y_hat = model4(x_var)
        loss = loss_func.forward(y_hat, y_var)
        loss.backward()
        optimizer.step()

    if (i % 5e3 == 0):
        print("Epoch: {0}, Loss: {1}, ".format(i, loss.data.numpy()))

for i in range(len(Y)):
    print(model4(X[i,]))

Y

model_params = list(model4.parameters())

model_params

# W1; (10x3) b1; (10,1); W2; (1,10); b2; (1,1)

model_params[0].shape, model_params[1].shape, model_params[2].shape, model_params[3].shape

W1 = model_params[0].data.numpy()
b1 = model_params[1].data.numpy()
W2 = model_params[2].data.numpy()
b2 = model_params[3].data.numpy()

F.sigmoid(F.sigmoid(X[0,] @ W1.T + b1) @ W2.T + b2)

# x1, x2, x3...ans
# 1, 1, 1; 0
# 1, 1, 0; 0
# 1, 0, 1; 0
# 1, 0, 0; 1
# 0, 1, 1; 0
# 0, 1, 0; 1
# 0, 0, 1; 0
# 0, 0, 0; 0

X = torch.Tensor([[1, 1, 1],
                  [1, 1, 0],
                  [1, 0, 1],
                  [1, 0, 0],
                  [0, 1, 1],
                  [0, 1, 0],
                  [0, 0, 1],
                  [0, 0, 0]])

Y = torch.Tensor([0,0,0,1,0,1,0,0]).view(-1,1)

class Problem5(nn.Module):
    def __init__(self, input_dim = 3, output_dim=1, hidden_dim=5):
        super(Problem5, self).__init__()
        self.lin1 = nn.Linear(input_dim, hidden_dim)
        self.lin2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.lin1(x)
        x = F.sigmoid(x)
        x = self.lin2(x)
        x = F.sigmoid(x)
        return x

model5 = Problem5()

def weights_init(model):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            # initialize the weight tensor, here we use a normal distribution
            m.weight.data.normal_(0, 1)

weights_init(model5)

loss_func = nn.MSELoss()

optimizer = optim.SGD(model5.parameters(), lr=0.02, momentum=0.9)
```

```
epochs = int(1e4 + 1.)
steps = X.size(0)
for i in range(epochs):
    for j in range(steps):
        data_point = np.random.randint(X.size(0))
        x_var = Variable(X[data_point], requires_grad=False)
        y_var = Variable(Y[data_point], requires_grad=False)

        optimizer.zero_grad()
        y_hat = model5(x_var)
        loss = loss_func.forward(y_hat, y_var)
        loss.backward()
        optimizer.step()

    if (i % 5e3 == 0):
        print("Epoch: {0}, Loss: {1}, ".format(i, loss.data.numpy()))

for i in range(len(Y)):
    print(model5(X[i,]))

Y

model_params = list(model5.parameters())

model_params

model_params[0].shape, model_params[1].shape, model_params[2].shape, model_params[3].shape

W1 = model_params[0].data.numpy()
b1 = model_params[1].data.numpy()
W2 = model_params[2].data.numpy()
b2 = model_params[3].data.numpy()

F.sigmoid(F.sigmoid(X[0,] @ W1.T + b1) @ W2.T + b2)

# https://github.com/szagoruyko/functional-zoo/blob/master/visualize.py

# https://www.programmersought.com/article/479977857/
model_params = list(model5.parameters())

vis_graph = make_dot(model5(X[1,]), params=dict(model5.named_parameters()))

vis_graph.view()

import pandas as pd

pd.set_option('display.max_rows', 500)

np.random.seed(6)

sample11 = np.random.normal(loc=1, scale=np.sqrt(0.2), size=(100,1))
sample12 = np.random.normal(loc=-1, scale=np.sqrt(0.2), size=(100,1))

sample21 = np.random.normal(loc=0, scale=np.sqrt(0.2), size=(100,1))
sample22 = np.random.normal(loc=0, scale=np.sqrt(0.2), size=(100,1))

sample1 = pd.DataFrame({
    'x11':[i[0] for i in sample11],
    'x12':[i[0] for i in sample12]
    })

sample2 = pd.DataFrame({
    'x21':[i[0] for i in sample21],
    'x22':[i[0] for i in sample22]
    })

sum(sample1.x11 - sample1.x12 < 1)

sum(sample2.x21 - sample2.x22 > 1)

bad_indices1 = sample1[sample1.x11 - sample1.x12 < 1].index
sample1.iloc[bad_indices1,1] = sample1.iloc[bad_indices1,1] - 0.3
sum(sample1.x11 - sample1.x12 < 1)

bad_indices2 = sample2[sample2.x21 - sample2.x22 > 1].index
sample2.iloc[bad_indices2, 1] = sample2.iloc[bad_indices2, 1] + 0.5
sum(sample2.x21 - sample2.x22 > 1)

class_list = [0] * 100
class_list.extend([1] * 100)

sample1.columns = ['x1', 'x2']
sample2.columns = ['x1', 'x2']
sample_data = pd.concat([sample1, sample2], axis=0, ignore_index=True)

sample_data['Class'] = class_list
```

```
sample_data.head()

X = torch.Tensor(np.array(sample_data.loc[:,['x1','x2']]))
Y = torch.Tensor(np.array(sample_data.Class)).view(-1,1)

class Problem6(nn.Module):
    def __init__(self, input_dim = 2):
        super(Problem6, self).__init__()
        self.lin1 = nn.Linear(input_dim, 1)

    def forward(self, x):
        x = self.lin1(x)
        x = F.sigmoid(x)
        return x

model6 = Problem6()

def weights_init(model):
    for m in model.modules():
        if isinstance(m, nn.Linear):
            # initialize the weight tensor, here we use a normal distribution
            m.weight.data.normal_(0, 1)

weights_init(model6)

loss_func = nn.MSELoss()

optimizer = optim.SGD(model6.parameters(), lr=0.02, momentum=0.9)

epochs = int(1e4 + 1.)
steps = X.size(0)
for i in range(epochs):
    for j in range(steps):
        data_point = np.random.randint(X.size(0))
        x_var = Variable(X[data_point], requires_grad=False)
        y_var = Variable(Y[data_point], requires_grad=False)

        optimizer.zero_grad()
        y_hat = model6(x_var)
        loss = loss_func.forward(y_hat, y_var)
        loss.backward()
        optimizer.step()

    if (i % 1e3 == 0):
        print("Epoch: {0}, Loss: {1}, ".format(i, loss.data.numpy()))

for i in range(len(Y)):
    print(model6(X[i,]))

model6(X).data.numpy() > 0.5

Y

model_params = list(model6.parameters())

model_weights = model_params[0].data.numpy()
model_bias = model_params[1].data.numpy()

w1, w2, b

w1 = model_weights[0][0]
w2 = model_weights[0][1]
b = model_bias[0]

# Reference: https://medium.com/@thomascountz/calculate-the-decision-boundary-of-a-single-perceptron-visualizing-
linear-separability-c4d77099ef38
slope = -(b / w2) / (b / w1)
y_int = -b / w2
x = np.linspace(-1,3)

from matplotlib.pyplot import figure
figure(num=None, figsize=(10, 10), dpi=80, facecolor='w', edgecolor='k')

plt.figure(figsize=(10,10))
plt.suptitle('Problem 6', fontsize=20)
plt.xlabel('x1', fontsize=18)
plt.ylabel('x2', fontsize=16)
plt.scatter(sample_data.iloc[0:100,0], sample_data.iloc[0:100,1])
plt.scatter(sample_data.iloc[100:,0], sample_data.iloc[100:,1], c='red')
plt.plot(x, slope * x + y_int, c='black')
plt.savefig('p6.jpg')

plt.scatter(X.numpy()[[0,-1], 0], X.numpy()[[0, -1], 1], s=50)
plt.scatter(X.numpy()[[1,2], 0], X.numpy()[[1, 2], 1], c='red', s=50)

x_1 = np.arange(-0.1, 1.1, 0.1)
y_1 = ((x_1 * model_weights[0,0]) + model_bias[0]) / (-model_weights[0,1])
plt.plot(x_1, y_1)
```

```
x_2 = np.arange(-0.1, 1.1, 0.1)
y_2 = ((x_2 * model_weights[1,0]) + model_bias[1]) / (-model_weights[1,1])
plt.plot(x_2, y_2)
plt.legend(["neuron_1", "neuron_2"], loc=8)
plt.show()
```