Homework Review

- HW1
- HW2
- HW3
Homework 1

- **Q1:** draw a neuron (with weights and bias for a step activation function) to implement a NOR gate
Q1: NOR gate
Homework 1

• **Q2:** Use multiple neurons to build a neural network that realizes a half adder using NOR only gates (each NOR gate is a neuron).

  A. Label weights and threshold on the diagram
  B. How many layers exist in your network?
Q2: half adder:
First let’s define what half adder is: a circuit of logic gates that can add two bits.
The output of half adder is a sum bit and a carry bit.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Truth table

Schematic

Realization
Step 1: for Sum bit implement XOR using NOR gates

• Sum = $\overline{A \cdot B} + A \cdot \overline{B}$

\[
= \left(\overline{A \cdot B} + A \cdot \overline{B}\right)
= \left((\overline{A} \cdot \overline{B}) \cdot (AB)\right)
= \left((A + \overline{B}) \cdot (\overline{A} + B)\right)
= \left(\overline{A \cdot B} + A \cdot \overline{B}\right)
= \left((\overline{A} + \overline{B}) + (A + B)\right)
= \left(((A + A) + (B + B)) + (A + B)\right)
\]
Step 2: for carry bit implement AND gate using NOR gates

• Carry = A.B

  \[
  = \overline{A \cdot B}
  = (\overline{A} + \overline{B})
  = \overline{(A + A) + (B + B)}
  
  \]

Half adder using NOR logic
Step 3: Transform each NOR gate into a neuron like the one you drew in Q1

Half adder using NOR logic

3 layers for sum
2 layers for carry
Homework 1

• **Q3:** Add a fourth layer to the network that recognizes handwritten numbers. Transform the outputs (coming from layer 3) into binary format. The resulting network will look like this:

**Hint:** set weights in the network and don’t worry about bias
Q3: ANN to recognize handwritten numbers
Example: recognizing handwritten 6

• Suppose the image is made up of 28 x 28 pixels
• Each pixel holds one value = intensity
• We can build a neural network to tell us whether or not the number shown is 6

# neurons in input layer = 28 x 28 = 784
# neurons in output layer = 1 (yes or no)
# neurons in hidden layer(s) depends on the design
a network for all 10 digits
The modified network should look like
Q3: in the third (hidden) layer, **only one** neuron will fire. That’s the neuron that has the value detected by the network

- If the input was an image that represents the digit 0 then the first neuron in the hidden layer will output 1. All other neurons in the third layer will output 0
- If the image represents the digit 9 then the last neuron in the hidden layer will output 1. All other neurons in the third layer will output 0
- The same for any given digit: only the neuron that represents this digit will be 1 and all others will be 0
Example: if the detected digit was 3
**Solution:** We need to set the weights coming from each digit to change the neurons in the output layer in a manner consistent with this digit’s binary representation.

The weights of the desired digit are the only ones active at this point because all others are not fired from the hidden layer.
Example: Digit 3
Assume bias = 0

\[ x_3 \times w_{30} = 1 \times 0 = 0 \]
\[ x_3 \times w_{31} = 1 \times 0 = 0 \]
\[ x_3 \times w_{32} = 1 \times 1 = 1 \]
\[ x_3 \times w_{33} = 1 \times 1 = 1 \]
Homework 2: Programming assignment

- Implement the perceptron class to use the Least Mean Squares (LMS) cost function and the steepest descent adjustment algorithm.

<table>
<thead>
<tr>
<th>TABLE 3.1 Summary of the LMS Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Training Sample:</strong></td>
</tr>
<tr>
<td>Input signal vector = ( x(n) )</td>
</tr>
<tr>
<td>Desired response = ( d(n) )</td>
</tr>
<tr>
<td><strong>User-selected parameter:</strong> ( \eta )</td>
</tr>
<tr>
<td><strong>Initialization.</strong> Set ( \hat{w}(0) = 0 ).</td>
</tr>
<tr>
<td><strong>Computation.</strong> For ( n = 1, 2, \ldots ), compute</td>
</tr>
<tr>
<td>( e(n) = d(n) - \hat{w}^T(n)x(n) )</td>
</tr>
<tr>
<td>( \hat{w}(n + 1) = \hat{w}(n) + \eta x(n)e(n) )</td>
</tr>
</tbody>
</table>
Step 1: take the same training data initialization as in the example from the lecture

```python
import numpy as np
import random

class Perceptron:
    def __init__(self, N):
        # The same training data from the example
        # Random linearly separated data
        xA, yA, xB, yB = [random.uniform(-1, 1) for i in range(4)]
        self.V = np.array([xB * yA - xA * yB, yB - yA, xA - xB])
        print(self.V)
        self.X = self.generate_points(N)

    def generate_points(self, self, N):
        X = []
        for i in range(N):
            x1, x2 = [random.uniform(-1, 1) for i in range(2)]
            x = np.array([1, x1, x2])
            s = int(np.sign(self.V.dot(x)))
            X.append((x, s))
        return X
```
Step 2: define a step activation function

```python
def step_activation(self, x):
    if x < 0:
        return 0
    else:
        return 1
```
Step 3: Define the LMS function

```python
def LMS(self):
    training_data = self.X
    print(training_data)
    wb = np.zeros(3)
    eta = 0.2
    epochs = 20
    for i in range(epochs):
        for x, d in training_data:
            n = np.dot(wb, x)
            a = self.step_activation(n)
            error = d - a
            wb += eta * error * x
```

Step 4: Create a Perceptron object and call LMS

```python
p = Perceptron(30)
p.LMS()
```
import numpy as np
import random

class Perceptron:
    def __init__(self, N):
        xA, yA, xB, yB = [random.uniform(-1, 1) for i in range(4)]
        self.V = np.array([xB * yA - xA * yB, yB - yA, xA - xB])
        print(self.V)
        self.X = self.generate_points(N)

    def generate_points(self, N):
        X = [1]
        for i in range(N):
            x1, x2 = [random.uniform(-1, 1) for i in range(2)]
            x = np.array([1, x1, x2])
            s = int(np.sign(self.V.dot(x)))
            X.append((x, s))
        return X

    def step_activation(self, self, x):
        if x < 0: return 0
        else: return 1

    def LMS(self):
        training_data = self.X
        print(training_data)
        wb = np.zeros(3)
        eta = 0.2
        epochs = 20
        for i in range(epochs):
            for x, d in training_data:
                n = np.dot(wb, x)
                a = self.step_activation(n)
                error = d - a
                wb += eta * error * x

    p = Perceptron(30)
p.LMS()
Homework 3: Backpropagation

• Q1: When does the learning become very slow or stops?
Backpropagation Algorithm

Start with randomly chosen weights $[w^l_{jk}]$

While error is unsatisfactory:

for each input pattern $x$:

feedforward: for each $l = 1, 2, ..., L$ compute $z^l_k$ and $a^l_k$

Compute the error at output layer: $\delta^L_k = (d^L_k - a^L_k)\sigma'(z^L_k)$

Backpropagate the error: for $l = L, L-1, L-2, ... 2$

compute $\delta^l_k = \delta^{l+1}_k w^l_{kj} a^l_{k+1} \sigma'(z^l_j)$

Calculate the gradients: $\frac{\partial E}{\partial w^l_{kj}} = \delta^l_j a^{l-1}_k$ and $\frac{\partial E}{\partial b^l_j} = \delta^l_j$

update weights (and bias): $w(n+1) = w(n) + \Delta w(n)$

where $\Delta w^l_{kj} = \eta \frac{\partial E}{\partial w^l_{kj}}$

Q1: When does $\Delta w(n)$ become small or close to 0?
When does the change in weights become very small?

\[ \Delta w_{kj}^l = \eta \frac{\partial E}{\partial w_{kj}^l} = \eta \delta_j^l a_{k}^{l-1} \]

- If \( \delta \) is very small
- If \( a \) is very small
When does the change in weights become very small?

\[ \Delta w^l_{kj} = \eta \frac{\partial E}{\partial w^l_{kj}} = \eta \delta^l_j a^{l-1}_k \]

If \( a \) is very small

This means that the activation coming from earlier neurons is very small
When does the change in weights become very small?

\[ \Delta w_{kj}^l = \eta \frac{\partial E}{\partial w_{kj}^l} = \eta \delta_j^l a_k^{l-1} \]

• **Remember:** There are two regions where weights can change:
  1. For neurons in the **output layer**: define \( \delta_j^L = (a_k^L - a_k^L)\sigma'(z_k^L) \)

    *change of cost due to weight change in the output layer*
  2. For neurons in **hidden layers**: define \( \delta_j^l = \delta_{kj}^{l+1} w_{kj}^{l+1} \sigma'(z_j^l) \)

    *change of cost due to weight change in layer \( l \)*
At the output layer $\delta^L_j$ is small if:

- $\delta^L_j = (d^L_k - a^L_k)\sigma'(z^L_k)$

1. The difference between the desired final outcome and the actual final outcome (error) is small or zero (network converged)

2. The derivative of the sigmoid function $\sigma'$ is small or zero which means $z^L_k$ is close to 0 or close to 1
At the hidden layers $\delta_j^l$ is small if:

1. The change of cost due to weight change in later layers is small
2. The weights of later layers are small
3. The derivative of the sigmoid is small or close to zero which means that the net input of the neuron $z_j^l$ is close to zero or close to 1
Homework 3:

• Q2: Consider the following Neural Network:
Initial weights and biases:

\[ w^1 = 1, \quad b^1 = 1, \quad w^2 = -2, \quad b^2 = 1. \]

An input/target pair is given to be

\[ ((p = 1), (t = 1)). \]

i. Find the squared error \((e)^2\) as an explicit function of all weights and biases.

ii. Using part (i) find \(\partial(e)^2 / \partial w^1\) at the initial weights and biases.

iii. Repeat part (ii) using backpropagation and compare results.
i. Find the squared error

- First find \( e = t - a^2 \)
  - \( a^2 = f^2(n^2) \) where \( f^2 \) is a linear function. Hence, \( a^2 = n^2 \)
- \( e = t - n^2 \)
  - \( n^2 = W^2 a^1 + b^2 \)
- \( a^1 = f^1(n^1) \) where \( f^1 \) is a sigmoid function. Hence, \( a^1 = \sigma(n^1) = \frac{1}{1 + \exp(-n^1)} \)
  - \( n^1 = W^1 p + b^1 \)
- Hence, \( e = t - W^2 \frac{1}{1 + \exp(-(W^1 p + b^1))} + b^2 \)
  \[
  (e)^2 = (t - W^2 \frac{1}{1 + \exp(-(W^1 p + b^1))} + b^2)^2
  \]
ii. Find the derivative \( \frac{\partial (e)^2}{\partial w^1} \)

\[
\frac{\partial (e)^2}{\partial w^1} = 2e \frac{\partial e}{\partial w^1}
\]

\[
\frac{\partial e}{\partial w^1} = \frac{\partial}{\partial w^1} (t - W^2 \sigma(n^1) + b^2) = \frac{\partial e}{\partial n^1} \times \frac{\partial n^1}{\partial w^1} = W^2 \sigma'(n^1)(-p)
\]

\[
\text{Where } \sigma'(n^1) = \sigma(n^1) \star (1 - \sigma(n^1)) = \frac{1}{1 + \exp(-(W^1p+b^1))} \times (1 - \frac{1}{1 + \exp(-(W^1p+b^1))}) = \frac{1}{1 + \exp(-2)} \times (1 - \frac{1}{1 + \exp(-2)}) = 0.104993
\]

\[
\frac{\partial (e)^2}{\partial w^1} = 2e \times W^2 \sigma'(n^1)(-p) = 2 \times e \times (-2) \times 0.104993 \times (-1) = 0.419972 \times e
\]
To calculate $e$

- Forward step through the network
  - $n^1 = W^1 p + b^1 = 1 \times 1 + 1 = 2$
  - $a^1 = \sigma(n^1) = \frac{1}{1+\exp(-n^1)} = \frac{1}{1+\exp(-2)} = \frac{1}{1+0.135335283} = 0.880798$
  - $a^2 = n^2 = W^2 a^1 + b^2 = (-2) \times 0.880798 + 1 = -0.761598$
  - $e = t - a^2 = 1 - (-0.761598) = 1.761598$

$$\frac{\partial(e)^2}{\partial w^1} = 0.419972 \times e = 0.419972 \times 1.761598 = 0.73982$$
iii. Using Backpropagation

• Use the given Rules:

\[
\frac{\partial e}{\partial w^1} = \delta^1_j a^0_k = \delta^2_k w^2_{kj} \sigma'(n^1_j) a^0_k
\]

\[
\delta^2_k = -2(d^2_k - a^2_k)\sigma'(n^2_k)
\]
1. Find $\delta_k^2$

- $\delta_k^2 = (d_k^2 - a_k^2)\sigma'(n_k^2)$
- $d_k^2 = t = 1$

- $a^2 = n^2 = W^2 a^1 + b^2 = (-2) \times 0.880798 + 1 = -0.761598$
- $\sigma'(n_k^2)$ is $f'(n_k^2)$ where $f$ is a linear function so $f'(n_k^2) = 1$

$$\delta_k^2 = -2(d_k^2 - a_k^2)\sigma'(n_k^2) = (-2) \times (1 - (-0.761598)) \times 1 = (-2) \times 1.761598 = -3.523196$$
2. Find \( \frac{\partial e}{\partial w^1} \)

- Given Rule: \( \frac{\partial e}{\partial w^1} = \delta^2_k w^2_{kj} \sigma'(n^1_j) a^0_k \)
- \( \delta^2_k = -3.523196 \)
- \( w^2_{kj} = -2 \)
- \( a^0_k = p = 1 \)
- \( \sigma'(n^1_j) = \sigma(n^1_j) * (1 - \sigma(n^1_j)) \) (from the code given in lecture 4)
- \( \sigma'(n^1_j) = \frac{1}{1 + \exp(-n^1_j)} * \left(1 - \frac{1}{1 + \exp(-n^1_j)}\right) = \frac{1}{1 + \exp(-2)} * \left(1 - \frac{1}{1 + \exp(-2)}\right) = 0.880798 \times 0.119202 = 0.104993 \)
- \( \frac{\partial e}{\partial w^1} = (-3.523196) \times (-2) \times 0.104993 \times 1 = 0.73982 \)
We reached the same result using both methods