Neural Networks

Neurons as logic gates

In this exercise we will experiment with neuron computations. lets see how to represent basic logic functions like AND, OR, and XOR using single neurons (or more complicated structures). Finally, at the lets find how to represent neural networks as a chain of matrix computations.

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt
```

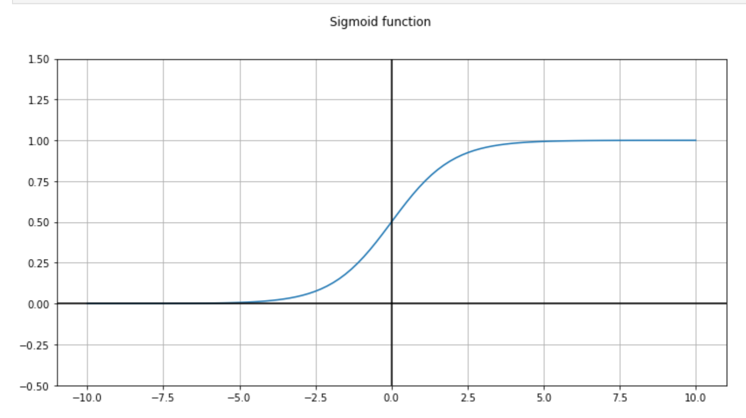
Sigmoid function:

$$\sigma = \frac{1}{1+e^{-x}}$$

σ ranges from (0, 1). When the input x is negative, σ is close to 0. When x is positive, σ is close to 1. At $x = 0$, $\sigma = 0.5$

In [6]:
```
## Quickly define the sigmoid function
def sigmoid(x):
    """Sigmoid function"""
    return 1.0 / (1.0 + np.exp(-x))
```

In [9]:
```
# Plot the sigmoid function
vals = np.linspace(-10, 10, num=100, dtype=np.float32)
activation = sigmoid(vals)
fig = plt.figure(figsize=(12,6))
fig.suptitle('Sigmoid function')
plt.plot(vals, activation)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.yticks()
plt.ylim([-0.5, 1.5]);
```



Thinking of neurons as boolean logic gates

Logic gates:A logic gate is an idealized model of computation or a physical electronic device implementing a Boolean function, a logical operation performed on one or more binary inputs that produces a single binary output. (i.e) logic gate takes in two boolean (true/false or 1/0) inputs, and returns either a 0 or 1 depending on its rule. The truth table for a logic gate shows the outputs for each combination of inputs, (0, 0), (0, 1), (1,0), and (1, 1). For example, let's look at the truth table for an "OR" gate:

OR Gate

OR gate truth table

Input		Output
0	0	0
0	1	1
1	0	1
1	1	1

A neuron that uses the sigmoid activation function outputs a value between (0, 1). This naturally leads us to think about boolean values. Imagine a neuron that takes in two inputs, x_1 and x_2, and a bias term:

By limiting the inputs of x_1 and x_2 to be in $\{0, 1\}$, we can simulate the effect of logic gates with our neuron. The goal is to find the weights (represented by ? marks above), such that it returns an output close to 0 or 1 depending on the inputs.

What numbers for the weights would we need to fill in for this gate to output OR logic? Observe from the plot above that $\sigma(z)$ is close to 0 when z is largely negative (around -10 or less), and is close to 1 when z is largely positive (around +10 or greater).

$$z = w_1x_1 + w_2x_2 + b$$

Let's think this through:

- When x_1 and x_2 are both 0, the only value affecting z is b. Because we want the result for (0, 0) to be close to zero, b should be negative (at least -10)
- If either x_1 or x_2 is 1, we want the output to be close to 1. That means the weights associated with x_1 and x_2 should be enough to offset b to the point of causing z to be at least 10.
- Let's give b a value of -10. How big do we need w_1 and w_2 to be?
  - At least +20
- So let's try out $w_1 = 20$, $w_2 = 20$, and $b = -10$!

In [12]:
```
def logic_gate(w1, w2, b):
    # Helper to create logic gate functions
    # Plug in values for weight_a, weight_b, and bias
    return lambda x1, x2: sigmoid(w1 * x1 + w2 * x2 + b)

def test(gate):
    # Helper function to test out our weight functions.
    for a, b in (0, 0), (0, 1), (1, 0), (1, 1):
        print("{}, {}: {}".format(a, b, np.round(gate(a, b))))
```

In [13]:
```
or_gate = logic_gate(20, 20, -10)
test(or_gate)
```

```
0, 0: 0.0
0, 1: 1.0
1, 0: 1.0
1, 1: 1.0
```

OR gate truth table

Input		Output
0	0	0
0	1	1
1	0	1
1	1	1

This matches! Great! Now you try finding the appropriate weight values for each truth table. Try not to guess and check- think through it logically and try to derive values that work.

AND Gate

AND gate truth table

Input		Output
0	0	0
0	1	0
1	0	0
1	1	1

Lets Determine what values for the neurons would make this function as an AND gate.

In [15]:
```
# TO DO:lets  Fill in the w1, w2, and b parameters such that
# the truth table matches

w1 = 11
w2 = 10
b = -20
and_gate = logic_gate(w1, w2, b)
test(and_gate)
```

```
0, 0: 0.0
0, 1: 0.0
1, 0: 0.0
1, 1: 1.0
```

Lets do the same for the NOR gate and the NAND gate.

NOR (Not Or) Gate

NOR gate truth table

Input		Output
0	0	1
0	1	0
1	0	0
1	1	0

In [16]:
```
# TO DO: Fill in the w1, w2, and b parameters such that the
# truth table matches

w1 = -20
w2 = -20
b = 10
nor_gate = logic_gate(w1, w2, b)
test(nor_gate)
```

```
0, 0: 1.0
0, 1: 0.0
1, 0: 0.0
1, 1: 0.0
```

NAND (Not And) Gate

NAND gate truth table

Input		Output
0	0	1
0	1	1
1	0	1
1	1	0

In [17]:
```
# TO DO: Fill in the w1, w2, and b parameters such that the
# truth table matches

w1 = -11
w2 = -10
b = 20
nand_gate = logic_gate(w1, w2, b)
test(nand_gate)
```

```
0, 0: 1.0
0, 1: 1.0
1, 0: 1.0
1, 1: 0.0
```

The limits of single neurons

If you've taken computer science courses, you may know that the XOR gates are the basis of computation. They can be used as so-called "half-adders", the foundation of being able to add numbers together. Here's the truth table for XOR:

XOR (Exclusive Or) Gate

XOR gate truth table

Input		Output
0	0	0
0	1	1
1	0	1
1	1	0

Can we create a set of weights such that a single neuron can output this property?

It turns out we cannot, since single neurons can't correlate inputs. Can we still use neurons to somehow form an XOR gate?

What if we tried something more complex:

Here, we've got the inputs going to two separate gates: the top neuron is an OR gate, and the bottom is a NAND gate. The output of these gates then get passed to another neuron, which is an AND gate. If you work out the outputs at each combination of input values, you'll see that this is an XOR gate.

In [19]:
```
# Make sure we have or_gate, nand_gate, and and_gate working from above!
def xor_gate(a, b):
    c = or_gate(a, b)
    d = nand_gate(a, b)
    return and_gate(c, d)
test(xor_gate)
```

```
0, 0: 0.0
0, 1: 1.0
1, 0: 1.0
1, 1: 0.0
```

Feedforward Networks as Matrix Computations

Feed-forward computation of a neural network can be thought of as matrix calculations and activation functions. lets do some actual computations with matrices to see this in action.

Provided below are the following:

- Three weight matrices `W_1`, `W_2` and `W_3` representing the weights in each layer. The convention for these matrices is that each $W_{i,j}$ gives the weight from neuron i in the previous (left) layer to neuron j in the next (right) layer.
- A vector `x_in` representing a single input and a matrix `x_mat_in` representing 7 different inputs.
- Two functions: `soft_max_vec` and `soft_max_mat` which apply the soft_max function to a single vector, and row-wise to a matrix.

The goals for this exercise are:

1. For input `x_in` calculate the inputs and outputs to each layer (assuming sigmoid activations for the middle two layers and soft_max output for the final layer.
2. Write a function that does the entire neural network calculation for a single input
3. Write a function that does the entire neural network calculation for a matrix of inputs, where each row is a single input.
4. Test your functions on `x_in` and `x_mat_in`.

This illustrates what happens in a NN during one single forward pass. Roughly speaking, after this forward pass, it remains to compare the output of the network to the known truth values, compute the gradient of the loss function and adjust the weight matrices `W_1`, `W_2` and `W_3` accordingly, and iterate. Hopefully this process will result in better weight matrices and our loss will be smaller afterwards.

In [22]:
```
W_1 = np.array([[2,-1,1,4],[-1,2,-3,1],[3,-2,-1,5]])
W_2 = np.array([[3,1,-2,1],[-2,4,1,-4],[-1,-3,2,-5],[3,1,1,1]])
W_3 = np.array([[-1,3,-2],[1,-1,-3],[3,-2,2],[1,2,1]])
x_in = np.array([.5,.8,.2])
x_mat_in = np.array([[.5,.8,.2],[.1,.9,.6],[.2,.2,.3],
                     [.6,.1,.9],[.5,.5,.4],[.9,.1,.9],[.1,.8,.7]])

def soft_max_vec(vec):
    return np.exp(vec)/(np.sum(np.exp(vec)))

def soft_max_mat(mat):
    return np.exp(mat)/(np.sum(np.exp(mat),axis=1).reshape(-1,1))

print('the matrix W_1\n')
print(W_1)
print('-'*30)
print('vector input x_in\n')
print(x_in)
print ('-'*30)
print('matrix input x_mat_in -- starts with the vector `x_in`\n')
print(x_mat_in)
```

```
the matrix W_1

[[ 2 -1  1  4]
 [-1  2 -3  1]
 [ 3 -2 -1  5]]
------------------------------
vector input x_in

[0.5 0.8 0.2]
------------------------------
matrix input x_mat_in -- starts with the vector `x_in`

[[0.5 0.8 0.2]
 [0.1 0.9 0.6]
 [0.2 0.2 0.3]
 [0.6 0.1 0.9]
 [0.5 0.5 0.4]
 [0.9 0.1 0.9]
 [0.1 0.8 0.7]]
```

In [23]:
```
z_2 = np.dot(x_in,W_1)
z_2
```

Out[23]:
```
array([ 0.8,  0.7, -2.1,  3.8])
```

In [24]:
```
a_2 = sigmoid(z_2)
a_2
```

Out[24]:
```
array([0.68997448, 0.66818777, 0.10909682, 0.97811873])
```

In [25]:
```
z_3 = np.dot(a_2,W_2)
z_3
```

Out[25]:
```
array([ 3.55880727,  4.01355384,  0.48455118, -1.55014198])
```

In [26]:
```
a_3 = sigmoid(z_3)
a_3
```

Out[26]:
```
array([0.97231549, 0.98225163, 0.61882199, 0.17506576])
```

In [27]:
```
z_4 = np.dot(a_3,W_3)
z_4
```

Out[27]:
```
array([ 2.04146788,  1.04718238, -3.47867612])
```

In [28]:
```
y_out = soft_max_vec(z_4)
y_out
```

Out[28]:
```
array([0.72780576, 0.26927918, 0.00291506])
```

In [31]:
```
# lets pass in the full matrix. So we run x_mat_in and instead of it just being one row,
z_2 = np.dot(x_mat_in,W_1)
z_2
```

Out[31]:
```
array([[ 0.8,  0.7, -2.1,  3.8],
       [ 1.1,  0.5, -3.2,  4.3],
       [ 1.1, -0.4, -0.7,  2.5],
       [ 3.8, -2.2, -0.6,  7. ],
       [ 1.7, -0.3, -1.4,  4.5],
       [ 4.4, -2.5, -0.3,  8.2],
       [ 1.5,  0.1, -3. ,  4.7]])
```

In [32]:
```
a_2 = sigmoid(z_2)
a_2
```

Out[32]:
```
array([[0.68997448, 0.66818777, 0.10909682, 0.97811873],
       [0.75026011, 0.62245933, 0.03916572, 0.98661308],
       [0.75026011, 0.40131234, 0.33181223, 0.92414182],
       [0.97811873, 0.09975049, 0.35434369, 0.99908895],
       [0.84553473, 0.42555748, 0.19781611, 0.98901306],
       [0.98787157, 0.07585818, 0.42555748, 0.99972542],
       [0.81757448, 0.52497919, 0.04742587, 0.9909867 ]])
```

In [33]:
```
z_3 = np.dot(a_2,W_2)
z_3
```

Out[33]:
```
array([[ 3.55880727,  4.01355384,  0.48455118, -1.55014198],
       [ 3.92653518,  4.10921334,  0.18688365, -0.94879275],
       [ 3.88876887,  2.2842146 ,  0.4885584 , -1.58990857],
       [ 5.37777836,  1.31317855, -0.14871063, -0.19351275],
       [ 4.4547123 ,  2.94332939,  0.11913329, -0.8567627 ],
       [ 5.38551712,  1.01435726, -0.04904456, -0.44362315],
       [ 4.32829928,  3.76620031, -0.02433132, -0.52848494]])
```

In [34]:
```
a_3 = sigmoid(z_3)
a_3
```

Out[34]:
```
array([[0.97231549, 0.98225163, 0.61882199, 0.17506576],
       [0.98066919, 0.9838446 , 0.54658541, 0.27912767],
       [0.9799401 , 0.90756123, 0.61976677, 0.16939676],
       [0.99540316, 0.78804456, 0.46289071, 0.45177222],
       [0.98850989, 0.94994727, 0.52974815, 0.29801615],
       [0.99543843, 0.73387201, 0.48774132, 0.39087798],
       [0.98698175, 0.97738352, 0.49391747, 0.37087032]])
```

In [35]:
```
z_4 = np.dot(a_3,W_3)
z_4
```

Out[35]:
```
array([[ 2.04146788,  1.04718238, -3.47867612],
       [ 1.92205929,  1.42324752, -3.54057369],
       [ 1.9563182 ,  1.13151906, -3.27363361],
       [ 1.63308573,  2.17592794, -2.97738636],
       [ 1.84869797,  1.55211843, -3.46934914],
       [ 1.59253551,  2.05871663, -2.82613226],
       [ 1.8430245 ,  1.73746743, -3.5474088 ]])
```

In [36]:
```
y_out = soft_max_vec(z_4)
y_out
```

Out[36]:
```
array([[0.09423345, 0.03486522, 0.00037743],
       [0.08362702, 0.05078266, 0.00035478],
       [0.08654163, 0.03793319, 0.00046333],
       [0.06263931, 0.10779543, 0.00062308],
       [0.07771166, 0.05776747, 0.00038097],
       [0.06015008, 0.09587296, 0.00072483],
       [0.07727201, 0.06953114, 0.00035236]])
```

In [37]:
```
## A one-line function to do the entire neural net computation

def nn_comp_vec(x):
    return soft_max_vec(sigmoid(sigmoid(np.dot(x,W_1)).dot(W_2)).dot(W_3))

def nn_comp_mat(x):
    return soft_max_mat(sigmoid(sigmoid(np.dot(x,W_1)).dot(W_2)).dot(W_3))
```

In [38]:
```
nn_comp_vec(x_in)
```

Out[38]:
```
array([0.72780576, 0.26927918, 0.00291506])
```

In [40]:
```
nn_comp_mat(x_mat_in)
```

Out[40]:
```
array([[0.72780576, 0.26927918, 0.00291506],
       [0.62054212, 0.37682531, 0.00263257],
       [0.69267581, 0.30361576, 0.00370844],
       [0.36618794, 0.63016955, 0.00364252],
       [0.57199769, 0.4251982 , 0.00280411],
       [0.38373781, 0.61163804, 0.00462415],
       [0.52510443, 0.4725011 , 0.00239447]])
```