# Lab 2

Hüseyin Anil Gündüz

Imports

In [1]:

```python
import torch
from torch import Tensor

import matplotlib.pyplot as plt
from  matplotlib_inline.backend_inline import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')
```

Exercise 1

Consider a neural network with one hidden layer with threshold activation function (i.e., $\tau(x) = \mathbf{1}[x > 0]$) and one output neuron with no activation function. Prove that such a neural network cannot *exactly* separate the cyan ($y = 1$) and white ($y = 0$) areas in the figure below.
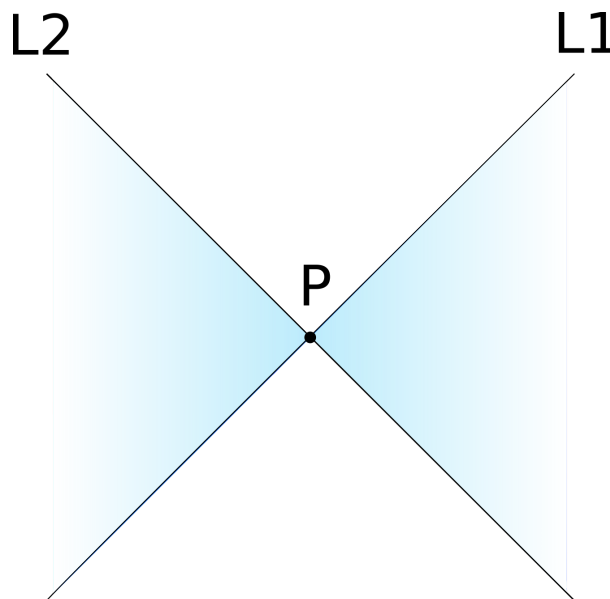
Hint: start by assuming that such a neural network exists and has two neurons in the hidden layer with a generic activation function $\tau(\cdot)$. Consider four points, one for each of the four regions, $\mathbf{x}_1, \ldots, \mathbf{x}_4$ with $y_2 = y_4 = 0$ and $y_1 = y_3 = 1$. Compute the difference between the predictions of points with different labels, $g(\mathbf{x}_1) - g(\mathbf{x}_2)$ and $g(\mathbf{x}_4) - g(\mathbf{x}_3)$. You should reach a contradiction, meaning that such a neural network does not exists.

Exercise 2

In this exercise, we are going to create a neural network with two hidden layers and threshold activation that can correctly separate the cyan region from the rest in the plot below. We will do this by composing linear classifiers just like we did in the previous lab.

```
dots = torch.tensor([[-1, 1], [-1, -1], [0, 0], [1, 1], [1, -1]])
plt.figure()
plt.scatter(dots[:, 0], dots[:, 1])
t1 = plt.Polygon([[-1, 1], [-1, -1], [0, 0]], color='#b3cde3')
plt.gca().add_patch(t1)

t2 = plt.Polygon(dots[2:5,:], color='#b3cde3')
plt.gca().add_patch(t2)
plt.show()
```
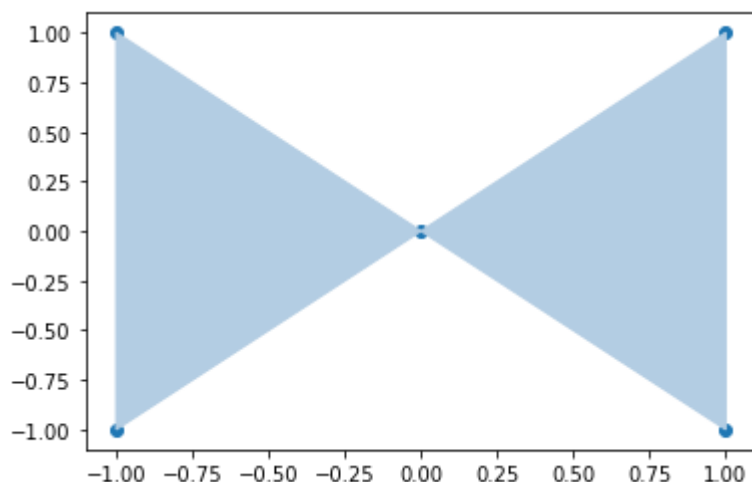

Let us create a dataset containing points on a grid to test the network:

```
# Build a grid of equally-spaced points, plus a column for the bias
grid_range = torch.linspace(-2, 2, 50)
grid_x, grid_y = torch.meshgrid(grid_range, grid_range)
data = torch.stack([torch.ones(50**2), grid_x.flatten(), grid_y.flatten()]).T

print(data.shape)
print(data[:10])
```

```
torch.Size([2500, 3])
tensor([[ 1.0000, -2.0000, -2.0000],
        [ 1.0000, -2.0000, -1.9184],
        [ 1.0000, -2.0000, -1.8367],
        [ 1.0000, -2.0000, -1.7551],
        [ 1.0000, -2.0000, -1.6735],
        [ 1.0000, -2.0000, -1.5918],
        [ 1.0000, -2.0000, -1.5102],
        [ 1.0000, -2.0000, -1.4286],
        [ 1.0000, -2.0000, -1.3469],
        [ 1.0000, -2.0000, -1.2653]])
```
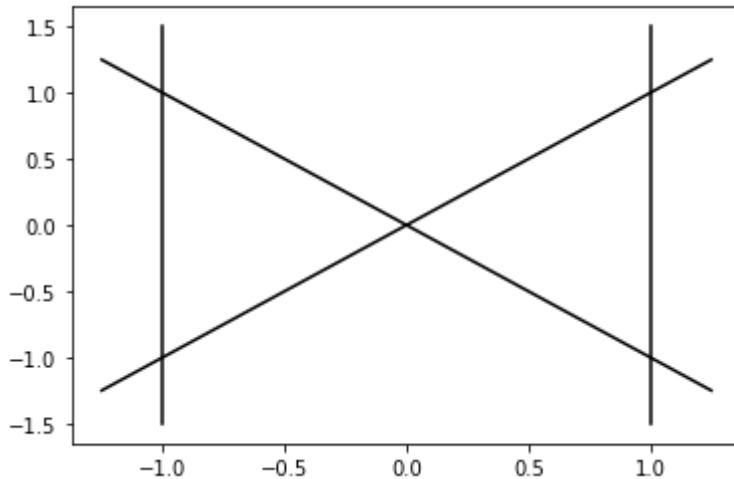
First hidden layer

The first hidden layer contains four neurons, each of which corresponds to a line in the plot below.

```python
def plot_bowtie() -> None:
    plt.plot((1.25,-1.25),(1.25,-1.25),'k')
    plt.plot((1.25,-1.25),(-1.25,1.25),'k')
    plt.plot((-1,-1),(-1.5,1.5),'k')
    plt.plot((1,1),(-1.5,1.5),'k')

plot_bowtie()
plt.show()
```


The following functions visualizes the decision boundary of a neuron with sigmoid activation, $y = \sigma(a + bx_1 + cx_2)$. You can use it to help you find the right values for the weights.

In [5]:

```python
def plot_decision_boundary(grid_x: Tensor, grid_y, pred: Tensor) -> None:
    """Plot the estimated decision boundary for a 2D grid with predictions."""
    plot_bowtie()
    plt.contourf(grid_x, grid_y, pred.view(grid_x.shape))
    plt.show()

def activation(x: Tensor) -> Tensor:
# TODO apply the threshold activation to x.

def plot_decision_boundary_first_hidden(a: int, b: int, c: int) -> None:
    """Take 3 weights for one input neuron and plot resulting decision boundary."""
    neuron_output = (
# TODO compute the output of the neuron with input weights given by a, b and c on
# each point of the grid.
    )
    plot_decision_boundary(grid_x, grid_y, neuron_output)

plot_decision_boundary_first_hidden(1, -2, 1)
```

For convenience, we group the parameters of the four neurons into a matrix with three rows and five columns (one is for the bias), so that their output can be computed in a single matrix multiplication. Each column

contains the weights of a different neuron. The first column contains a "fake" hidden neuron for the bias, whose value is always one. Note that the first row of the weight matrix is connected to the bias of the previous layer.

In [6]:

```
weights_1 = torch.tensor([
    [1, 0, 0],    # bias neuron connected to the bias of the inputs
# TODO find the coefficients for the first neuron.
# TODO find the coefficients for the second neuron.
# TODO find the coefficients for the third neuron.
# TODO find the coefficients for the fourth neuron.
], dtype=torch.float).T
# Note: The weight matrix is transposed in the end. (5 x 3 --> 3 x 5)
# This gives more readability when entering your weights manually.

print(weights_1.shape)
```

Let us plot the predictions of the four neurons:

In [7]:

```
plot_decision_boundary_first_hidden(*weights_1[:, 1])
```

In [8]:

```
plot_decision_boundary_first_hidden(*weights_1[:, 2])
```

In [9]:

```
plot_decision_boundary_first_hidden(*weights_1[:, 3])
```

In [10]:

```
plot_decision_boundary_first_hidden(*weights_1[:, 4])
```

And this is the first hidden layer of the network. Let us compute its predictions for each point of the grid:

```
hidden_1 = (
# TODO compute the output of the first hidden layer using a matrix multiplication.
)

# Make sure that the number of rows is not changed,
print(len(hidden_1) == len(data))

# that there are five columns,
print(hidden_1.shape[1] == 5)

# and that the values are between zero and one
print(float(torch.min(hidden_1)), float(torch.max(hidden_1)))
```

Second hidden layer

The second hidden layer is composed of two neurons, each activating for inputs inside one of the two triangles that make up our figure. These two neurons are connected to the four neurons of the previous layer, thus each of them has five parameters.

Let us first create a new function to visualize the decision boundary of these neurons in the second hidden layer.

```
def plot_decision_boundary_second_hidden (
        a: int, b: int, c: int, d: int, e: int) -> None:
    """Take 5 weights for one hidden neuron and plot resulting decision boundary.""
    neuron_output = (
# TODO compute the output of the neuron.
# (The input is the result from the first layer)
    )
    plot_decision_boundary(grid_x, grid_y, neuron_output)

plot_decision_boundary_second_hidden(-2, 3, -1, -3, 1)
```

Now, as before, find the coefficients for the two neurons and put them into a matrix with five rows and three columns. You can use the previous function to help you find these weights.

Hint: You can think of these neurons as performing a logical AND operation on the outputs of the neurons of the previous layer. All points inside each triangle must be on the same side of three decision boundaries.

In [13]:

```
weights_2 = torch.tensor([
    [1, 0, 0, 0, 0],    # bias neuron connected to the bias of the inputs
# TODO find the coefficients for the first neuron.
# TODO find the coefficients for the second neuron.
], dtype=torch.float).T
# Note: The weight matrix is transposed in the end. (3 x 5 --> 5 x 3)
# This gives more readability when entering your weights manually.

print(weights_2.shape)
```

In [14]:

```
plot_decision_boundary_second_hidden(*weights_2[:, 1])
```

In [15]:

```
plot_decision_boundary_second_hidden(*weights_2[:, 2])
```

In [16]:

```
hidden_2 = (
# TODO compute the output of the second hidden layer.
)

# Make sure that the number of rows is not changed,
print(len(hidden_2) == len(data))

# that there are five columns,
print(hidden_2.shape[1] == 3)

# and that the values are between zero and one
print(float(torch.min(hidden_2)), float(torch.max(hidden_2)))
```

Output layer

Finally, we can seek the parameters for the output neuron. It should activate when an input is inside either one of the two triangles.

Hint: You can think of the output neuron as performing a logical OR operation on the outputs of the second hidden layer.

Let us again modify the visualization function to show the decision of the network:

```
def plot_decision_boundary_output (a: int, b: int, c: int) -> None:
    """Take 3 weights for one output neuron and plot resulting decision boundary."""
    neuron_output = (
# TODO compute the output of the output neuron.
# (The input is the result from the second layer)
    )
    plot_decision_boundary(grid_x, grid_y, neuron_output)
```

Now fill in the parameters:

```
weights_3 = torch.tensor([
# TODO find the coefficients for the output neuron.
], dtype=torch.float)

print(weights_3.shape)
```

The output of the neural network is:

```
plot_decision_boundary_output(*weights_3)
```

Let us recap how the output is computed:

```
hidden_1 = activation(data @ weights_1)
hidden_2 = activation(hidden_1 @ weights_2)
output = hidden_2 @ weights_3
```

This is, in essence, the *forward pass*.

If you did everything correctly, you should see below the bow tie image we wanted to reproduce:

```
plot_decision_boundary(grid_x, grid_y, output)
```

Exercise 3

Note: Focus on solving the second exercise first, as this exercise is of secondary importance. Essentially, it shows that the same function can be computed by many different neural networks.

Consider a neuron with incoming weights $\mathbf{w} = w_1, \ldots, w_n$ bias b, and activation $\tau(\cdot)$. This neuron is connected to the i-th neuron of the next layer with the weight v_i, and the bias of the latter neuron is c_i. We want to replace $\mathbf{w}$, v_i, b and c_i with new parameters $\mathbf{w}'$, v_i', b' and c_i' so that the output of the network is unchanged for all inputs. At least one of the new parameters must be different, but some are allowed to equal the old ones.

1. Suppose that τ is the hyperbolic tangent. Show that the network computes the same function if we let $\mathbf{w}' = -\mathbf{w}$, $v_i' = -v_i$, $b' = -b$ and $c_i' = c_i$.
2. Now suppose that τ is the logistic sigmoid function. How should you set $\mathbf{w}'$, v_i', b' and c_i'? Hint: first, find the relationship between $\sigma(x)$ and $\sigma(-x)$.
3. Can you find other ways of modifying the parameters of a neural network without altering its output? Equivalently, given a neural network computing a certain function, how can you find a different network that computes the same function?

You do not have to provide a formal answer, but you can do so if you wish.