Important! Please do not remove any cells, including the test cells, even if they appear empty. They contain hidden tests, and deleting them could result in a loss of points, as the exercises are graded automatically. Only edit the cells where you are instructed to write your solution.

Exercise 1. Introduction to Gradient Calculations and PyTorch

In this exercise, the goal is to understand the basics of building a neural network and training. The exercise is divided into three stages:

Part 1. Pen and Paper: Manual Gradient Calculation (6 points): You will manually calculate the forward pass, loss computation, backward pass, and parameter updates for a simple neural network. This part will be completed in this notebook `ex1_01_pen_and_paper.ipynb` .

Part 2. NumPy Implementation for Network Training (10 points): You will implement the layers used in a simple multi-layer perceptron on NumPy. This part will be completed in the notebook `ex1_02_numpy_gradient.ipynb` .

Part 3: PyTorch for Regression (4 points): You will build a neural network in PyTorch for a toy regression problem. This will introduce you to using PyTorch for model training. This part will be completed in the notebook `ex1_03_pytorch_regression.ipynb` .

Deliverables:
Submit the completed notebooks (*ex1_01_pen_and_paper.ipynb*, *ex1_02_numpy_gradient.ipynb*, *ex1_03_pytorch_regression.ipynb*) in separate files (no zip). Do not change the name of the notebook files as it may result in 0 points for the exercise.

Quick Overview: Gradient Descent

This section will review how forward and backward passes work and how parameters are updated during training.

1. Forward pass

The forward pass involves computing the output of each layer from its inputs by traversing through each block of the neural network from the first to the last layer.

For each fully connected (linear) layer, the output is calculated using its input vector, weights, and biases as follows:

$$a^{(l)} = W^{(l)} z^{(l-1)} + b^{(l)}.$$

Here, $W^{(l)}$ and $b^{(l)}$ are the weights and the biases of the l^{th} layer, respectively, while $z^{(l-1)}$ is the output from the previous layer. For the first layer, $z^{(0)}$ represents the input to the network, commonly denoted as x. The output of the linear layer, $a^{(l)}$, is often referred to as the pre-activation. Next, we apply a nonlinearity to compute the post-activation output:

$$z^{(l)} = f(a^{(l)}),$$

where f represents the activation function (such as tanh, ReLU, etc.).

Once the output layer produces a prediction y, the loss function is computed. In this case, we are using the mean squared error (MSE) loss:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2,$$

where y_i is the predicted value, t_i is the actual target value, and N is the number of data points.

2. Backward pass

The backward pass involves computing the gradients of the loss with respect to the model parameters (weights and biases) of each layer by traversing from the last layer to the first layer. This process uses the chain rule to propagate the error gradients backward through the network and is also known as **backpropagation**. During the backward pass, we compute the gradients for both the pre-activation and the post-activation outputs.

3. Update parameters

Once the gradients are calculated, the model parameters are updated using the gradient descent algorithm. The parameters are adjusted in the direction that reduces the loss function. This update step is performed using the following rule:

$$\theta^{(\tau)} = \theta^{(\tau-1)} - \eta \cdot \nabla_\theta L^{(\tau-1)},$$

where θ represents the parameters (weights and biases), η is the learning rate, τ labels the iteration step, and $\nabla_\theta L$ is the gradient of the loss function with respect to the parameters.

In practice, training a neural network involves multiple iterations over the dataset. Each iteration is called an epoch. During each epoch, the dataset is typically divided into smaller subsets called batches. The model parameters are updated after each batch. This approach, known as mini-batch gradient descent, helps in efficient computation and faster convergence. However, for this assignment, we will perform the computation for only one iteration (one forward pass, one backward pass, and parameter update).

Part 1. Pen and Paper - Manual Gradient

Computation for a Simple MLP

You are given the initial parameters and the model architecture below. You need to manually compute the steps for one full iteration of the model training and write the final output asked in each step to the given notebook cells.

Model Architecture

In this part of the assignment we will use a simple Multi-Layer Perceptron (MLP) with the following architecture:

- **Input dimension**: 3
- **One hidden layer** with 2 neurons and tanh activation
- **Output layer** with 1 neuron
- **Loss Function**: Mean Squared Error (MSE)

Initial Parameters

- Weights (W_1) and biases (b_1) of the hidden layer:
  $W_1 = \begin{bmatrix} 0.1 & 0.3 & -0.5 \\ -0.2 & 0.4 & 0.6 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$
- Weights (W_2) and biases (b_2) of the output layer:
  $W_2 = \begin{bmatrix} 0.7 & -0.8 \end{bmatrix}, \quad b_2 = \begin{bmatrix} 0.3 \end{bmatrix}$

Input and Target

- Input:
  $x = \begin{bmatrix} 0.3 \\ 0.4 \\ -0.1 \end{bmatrix}$
- Target output:
  $t = \begin{bmatrix} 0.5 \end{bmatrix}$

The forward and backward passes of the expected architecture are shown in Figure 1. Each gradient is computed step by step. Starting from the loss, you need to work through each layer by applying **chain rule** and propagating the errors backward through the network, such as:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial z}.$$

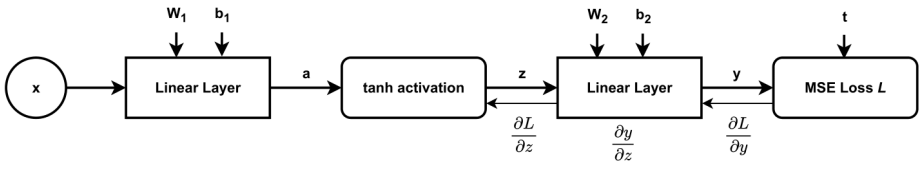

Figure 1: Forward and Backward Pass in the MLP

Steps to follow:

1. **Forward pass**: Calculate the outputs for each layer.
2. **Loss Calculation**: Calculate the MSE Loss.
3. **Backward pass**: Calculate the gradients of the loss with respect to the input of the layers and activation functions as well as the layer's parameters.
4. **Parameter update**: Update the parameters for the next iteration.

Important Notes:

1. Make sure your answers are accurate to **2 decimal places**.
2. Pay attention to the matrix shapes. The layer inputs are provided as column vectors with the shape (D_{in},), where D_{in} is the input dimension. The weights are shaped as (D_{out}, D_{in}), where D_{out} is the number of neurons in the next layer. The provided matrices are already in the correct shape with placeholders such as `a = [None, None]`. You are expected to replace the `None` value with the correct value. Do not change the given shapes.
3. We will calculate the expected values in each step based on your previous answers. If you make a mistake in an intermediate step, you will not be penalized again in the next steps if your solution based on that mistake is still consistent. However, if a necessary intermediate result is missing, the following steps that depend on it will also be affected.
4. **Do not forget** to remove `raise NotImplementedError()` sections.

Step-by-Step Manual Calculation

Step 1: Forward Pass

1. Hidden Layer Pre-Activation (a):

Compute $a = W_1 x + b_1$ by plugging in the values:

$$a = \begin{bmatrix} 0.1 & 0.3 & -0.5 \\ -0.2 & 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} 0.3 \\ 0.4 \\ -0.1 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} = \begin{bmatrix} a_{1,1} \\ a_{2,1} \end{bmatrix}$$

In [ ]:
```
# Hidden layer pre-activation (a)
a = [None, None]
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the hidden layer pre-activation. DO NOT DELETE THE CELL.
```

2. Apply tanh Activation (to a):

We apply the tanh function element-wise:

$$z = \tanh(a) = \begin{bmatrix} \tanh(a_{1,1}) \\ \tanh(a_{2,1}) \end{bmatrix} = \begin{bmatrix} z_{1,1} \\ z_{2,1} \end{bmatrix},$$

where the tanh function is defined as:

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}.$$

In [ ]:
```
# Hidden layer activation (z)
z = [None, None]
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the tanh activation. DO NOT DELETE THE CELL.
```

3. Output Layer (y):

Compute $y = W_2 z + b_2$ by plugging in the values:

$$y = \begin{bmatrix} 0.7 & -0.8 \end{bmatrix} \begin{bmatrix} z_{1,1} \\ z_{2,1} \end{bmatrix} + [0.3]$$

In [ ]:
```
# Output layer (y)
y = [None]
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the output layer. DO NOT DELETE THE CELL.
```

Step 2: Calculate the Loss

Use the Mean Squared Error (MSE) formula to compute the loss for a single data point as $L = (y_1 - t_1)^2$.

In [ ]:
```
# Loss calculation (MSE)
loss = None
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the loss calculation. DO NOT DELETE THE CELL.
```

Step 3: Backward Pass

1. Gradient of the Loss w.r.t the Output:

$$\frac{\partial L}{\partial y} = \frac{\partial}{\partial y}(y - t)^2 = 2 \times (y - t)$$

In [ ]:
```
# Gradient of the loss wrt the output (dL/dy)
dL_dy = [None]
```

```
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the gradient of the loss wrt output. DO NOT DELETE THE CELL.
```

2. Gradient of the Loss w.r.t the Output Layer Weights:

Apply the chain rule:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial W_2} = \frac{\partial L}{\partial y} z^T$$

In [ ]:
```
# Gradient of the loss wrt output layer weights (dL/dW2)
dL_dW2 = [[None, None]]
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the gradient of the loss wrt output layer weights. DO NOT DELETE
```

3. Gradient of the Loss w.r.t the Output Layer Bias:

Chain rule applies; compute $\frac{\partial \mathrm{L}}{\partial b_2}$.

In [ ]:
```
# Gradient of the loss wrt output layer bias (dL/db2)
dL_db2 = [None]
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the gradient of the loss wrt output layer bias. DO NOT DELETE TH
```

4. Gradient of the Loss w.r.t the Hidden Layer Weights:

To compute $\frac{\partial \mathrm{L}}{\partial W_1}$, you will need to apply the chain rule by combining the relevant partial derivatives.

Hints:

1. You need to compute four partial derivatives in this step, one of which you have already obtained.
2. Do not forget to compute the derivative of the activation function **tanh**.
3. Refer to the given computational graph to ensure that you are working through each layer to compute the full gradient.

In [ ]:
```
# Gradient of the loss wrt hidden layer weights (dL/dW1)
dL_dW1 = [[None, None, None],
          [None,  None, None]]
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the gradient of the loss wrt hidden layer weights. DO NOT DELETE
```

Step 4: Parameter Update

Using a learning rate of $\eta = 0.5$, update the parameters:

1. Update the Output Layer Weights:

$$W_2 = W_2 - \eta \times \frac{\partial \mathrm{L}}{\partial W_2}$$

In [ ]:
```
# Updated output layer weights (W2)
W2_updated = [[None, None]]
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the updated output layer weights. DO NOT DELETE THE CELL.
```

2. Update the Output Layer Bias:

$$b_2 = b_2 - \eta \times \frac{\partial \mathrm{L}}{\partial b_2}$$

In [ ]:
```
# Updated output layer bias (b2)
b2_updated = [None]
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the updated output layer bias. DO NOT DELETE THE CELL.
```

3. Update the Hidden Layer Weights:

$$W_1 = W_1 - \eta \times \frac{\partial \mathrm{L}}{\partial W_1}$$

In [ ]:
```
# Updated hidden layer weights (W1)
W1_updated = [[None, None, None],
              [None, None, None]]
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the updated hidden layer weights. DO NOT DELETE THE CELL.
```

4. Update the Hidden Layer Biases:

$$b_1 = b_1 - \eta \times \frac{\partial \mathrm{L}}{\partial b_1}$$

In [ ]:
```
# Updated hidden layer bias (b1
b1_updated = [None, None]
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell checks the updated hidden layer bias. DO NOT DELETE THE CELL.
```

In [ ]:
```
# Do not delete this cell
```