Compulsory 1

KIUA 1006-1 25 Machine Learning II

Submission Format

Your report must be submitted as a single file in Portable Document Format (.pdf). All code written for this assignment must be both **(i)** included in the report (for clarity and documentation), and **(ii)** submitted as separate source files. The recommended submission is a single compressed archive (.zip) that contains your report in PDF format together with all code files. Do not include any downloaded data files in your submission.

Report Instructions

Learning to write a scientific report is an important skill that many courses, including this one, aim to strengthen. Therefore, any question you answer must be included in the written report of this compulsory assignment. Answers outside the report (for example, in code comments or inside a Jupyter Notebook) will not be considered as part of your submission. You may structure your report by creating a separate subsection for each question. **While it is not mandatory**, we recommend that you write your report in *LaTeX*. This will help you develop good scientific writing habits. You may, for example, explore Overleaf, which provides an accessible online *LaTeX* editor and many useful resources for self-learning. Remember to cite all sources you use. Be advised that your submission will be checked for plagiarism, and uncredited use of material will be treated as academic misconduct.

Plagiarism

Plagiarism is a serious academic offense and will not be tolerated. Always ensure that you provide proper attribution for any work, ideas, or concepts that are not originally yours. Should the plagiarism detection system flag your submission, your report will not be considered for assessment.
NB: Large language models may generate fabricated references, and any unclear or unverifiable citation will be considered plagiarism.

Coding Guidelines

All submitted code must be sufficiently commented so that a reader with general programming knowledge can understand how the program works. You may use standard built-in functions and packages (for example, *PyTorch*, *NumPy*, or *matplotlib* in Python) for data handling and basic calculations. However, do not use packages that trivialize the assignment. Unless explicitly stated otherwise, you are *not* permitted to rely on Python packages that provide ready-made machine learning methods such as *scikit-learn*. You may, of course, re-use code you have written for earlier parts of the course.

You are permitted to use tools such as Copilot, but the following conditions apply:

- The code must be your own work. Submit only code that you fully understand and can explain.
- Revise any machine-generated code so that it reflects your understanding and style, rather than copying it directly.
- In your report, clearly state how you used Copilot or similar tools (for example, for debugging or suggesting implementation details).
- You remain fully responsible for the correctness, clarity, and originality of the submission.

How to Cite Sources in LaTeX

You are encouraged to consult textbooks, lecture notes, research papers, or reliable online materials when solving the problems. However, you must always acknowledge and cite the sources you use to avoid unintended plagiarism. The easiest way to obtain a citation is through Google Scholar:

- Search for the paper or book in Google Scholar.
- Click on the quotation mark (“) symbol under the entry.
- Select BibTeX, and copy the entry into your .bib file (e.g., *references.bib* in your project folder). Each entry has a unique key, such as `goodfellow2016deep`.
- Use this key when citing in your report with the `~\cite{...}` command.
- For example: Neural networks are powerful`~\cite{goodfellow2016deep}`.
- For electronic sources (such as websites, online tutorials, or documentation), please refer to the following guideline: Purdue University Online Writing Lab.

This will appear in your compiled report as an in-text citation, such as [2], and the corresponding reference will automatically be added to the bibliography at the end of the document.

1 Two-layer MLP: Forward propagation by hand and in PyTorch

You will practice the forward propagation of a two-layer multilayer perceptron (MLP).

1.1 Forward propagation with identity activation function

Let the input consist of four vectors $\boldsymbol{x}_i \in \mathbb{R}^{1\times 3}$, for $i = 1, \cdots, 4$. Stack them into a matrix

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{x}_3 \\ \boldsymbol{x}_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 0 & -2 & 1 \\ 0 & 1 & 2 \end{bmatrix} \in \mathbb{R}^{4\times 3}.$$

Define the first layer with two neurons

$$\boldsymbol{W}_1 = \begin{bmatrix} 1.0 & -2.5 \\ -1.5 & -1.0 \\ 0.5 & -1.5 \end{bmatrix} \in \mathbb{R}^{3\times 2}, \qquad \boldsymbol{b}_1 = \begin{bmatrix} 0.5 & 2.0 \end{bmatrix} \in \mathbb{R}^{1\times 2}.$$

Define the second layer with one neuron

$$\boldsymbol{w}_2 = \begin{bmatrix} 0.1 \\ -2.0 \end{bmatrix} \in \mathbb{R}^{2\times 1}, \qquad b_2 = 0.5 \in \mathbb{R}.$$

Use the identity activation function $t = \sigma_I(t)$ for both layers. The forward propagation is then

$$\begin{aligned} &\boldsymbol{Z}_1 = \boldsymbol{X}\boldsymbol{W}_1 + \boldsymbol{b}_1, \\ &\boldsymbol{A}_1 = \sigma_I(\boldsymbol{Z}_1) = \boldsymbol{Z}_1, \quad \text{(first layer)} \end{aligned} \tag{1}$$

and,

$$\begin{aligned} &\boldsymbol{z}_2 = \boldsymbol{A}_1\boldsymbol{w}_2 + b_2, \\ &\hat{\boldsymbol{y}} = \sigma_I(\boldsymbol{z}_2) = \boldsymbol{z}_2. \quad \text{(second layer)} \end{aligned} \tag{2}$$

(Optional) $\mathbf{1}_4$ denotes the column vector of ones in $\mathbb{R}^{4\times 1}$, used to broadcast the bias terms across all samples. With this, you can rewrite Equations (1)-(2) as follows

$$\boldsymbol{Z}_1 = \boldsymbol{X}\boldsymbol{W}_1 + \mathbf{1}_4\boldsymbol{b}_1, \quad \boldsymbol{z}_2 = \boldsymbol{A}_1\boldsymbol{w}_2 + b_2\mathbf{1}_4.$$

Task.

(1) **By hand**, draw the neural networks and calculate the numerical values of $\boldsymbol{A}_1$ and the output vector $\hat{\boldsymbol{y}}$. You must show all steps, explicitly using **inner product** calculations. Clearly state the final dimensions of $\boldsymbol{A}_1$ and $\hat{\boldsymbol{y}}$. Attach a photo of your handwritten work to your report. You may use a calculator. Round numerical values to at most two decimal places.

Suppose we have one trillion input vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{1{,}000{,}000{,}000{,}000}$. Loading all these samples into your GPU memory at once may not be feasible.

(2) Propose and briefly explain a practical strategy to handle this situation so that the forward propagation can still be carried out.

1.2 PyTorch implementation and verification

Reproduce Problem 1.1 by using only PyTorch.

(3) Write a function `forward(X, activ_func1, W1, b1, activ_func2 , W2, b2)` that returns all intermediate values, such as $\boldsymbol{Z}_1, \boldsymbol{A}_1, \boldsymbol{z}_2$, and $\hat{\boldsymbol{y}}$.

$$\hat{\boldsymbol{y}} = f(\boldsymbol{X}; \sigma_1, \boldsymbol{W}_1, \boldsymbol{b}_1, \sigma_2, \boldsymbol{w}_2, b_2) = \sigma_2\Bigg(\underbrace{\underbrace{\sigma_1\big(\underbrace{\boldsymbol{X}\boldsymbol{W}_1 + \boldsymbol{b}_1}_{\boldsymbol{Z}_1} \big)}_{\boldsymbol{A}_1}\, \boldsymbol{w}_2 + b_2}_{\boldsymbol{z}_2} \Bigg). \quad (3)$$

(4) Complete Table 1 with intermediate values obtained from your PyTorch implementation, and verify that they are consistent with your hand calculations. Round numerical values to at most two decimal places.

Identity	$\boldsymbol{x}_1$		$\boldsymbol{x}_2$		$\boldsymbol{x}_3$		$\boldsymbol{x}_4$	
Var.	Hand	PyTorch	Hand	PyTorch	Hand	PyTorch	Hand	PyTorch
$\boldsymbol{a}_1$	...	...	...	...	...	...	...	...
$\hat{\boldsymbol{y}}$	...	...	...	...	...	...	...	...

Table 1: Intermediate values for four different inputs with the identity activation function.

```
% Identity activation function
def identity(x):
    return x
```

1.3 Effect of activation choice

Now keep the same weights and biases as above. Replace the activation function in both hidden layer and the output layer with one of the following

$$\sigma_{ReLU}(t) = \max\{0, t\}, \quad \sigma_{sigm}(t) = \frac{1}{1 + \exp(-t)}.$$

(5) Complete Table 2 by computing all values ($\boldsymbol{a}_1$, $\hat{\boldsymbol{y}}$) for each input vector $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_4$. Use your PyTorch implementation of the network with the four combinations of activation functions (σ_1, σ_2). Report the results as numerical values, rounded to two decimal places.

(6) Compare the outputs $\hat{\boldsymbol{y}}$ in Table 2 with those in Table 1 and discuss why the outputs differ across the three activation functions, with particular emphasis on the concepts of **nonlinear property** and **vanishing gradient**. In your answer, provide a clear discussion of the advantages and disadvantages of each activation function. Support your statements with verifiable references from established sources.

(σ_1, σ_2)				
$(\sigma_I,\ \sigma_I)$	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{x}_4$
$\boldsymbol{a}_1$	...	...	...	...
$\hat{\boldsymbol{y}}$	...	...	-4.10	...
$(\sigma_I,\ \sigma_{ReLU})$	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{x}_4$
$\boldsymbol{a}_1$	...	...	...	...
$\hat{\boldsymbol{y}}$	...	...	0.00	...
$(\sigma_I,\ \sigma_{sigm})$	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{x}_4$
$\boldsymbol{a}_1$	...	...	...	...
$\hat{\boldsymbol{y}}$	...	...	0.02	...
$(\sigma_{ReLU},\ \sigma_I)$	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{x}_4$
$\boldsymbol{a}_1$	...	...	...	...
$\hat{\boldsymbol{y}}$	...	...	-4.10	...
$(\sigma_{ReLU},\ \sigma_{ReLU})$	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{x}_4$
$\boldsymbol{a}_1$	...	...	...	...
$\hat{\boldsymbol{y}}$	...	...	0.00	...
$(\sigma_{ReLU},\ \sigma_{sigm})$	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{x}_4$
$\boldsymbol{a}_1$	...	...	...	...
$\hat{\boldsymbol{y}}$	...	...	0.02	...
$(\sigma_{sigm},\ \sigma_I)$	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{x}_4$
$\boldsymbol{a}_1$	...	...	...	...
$\hat{\boldsymbol{y}}$	...	...	-1.25	...
$(\sigma_{sigm},\ \sigma_{ReLU})$	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{x}_4$
$\boldsymbol{a}_1$	...	...	...	...
$\hat{\boldsymbol{y}}$	...	...	0.00	...
$(\sigma_{sigm},\ \sigma_{sigm})$	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{x}_4$
$\boldsymbol{a}_1$	...	...	...	...
$\hat{\boldsymbol{y}}$	...	...	0.22	...

Table 2: Intermediate values for four different inputs with different combinations of the activation functions $(\sigma_1,\ \sigma_2)$.

```
% Activation functions:
def relu(x):
    return np.maximum(0, x)
```

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

2 Two-layer MLP: Backward propagation

In this section, we use the same input vectors, weights, and biases as in the forward propagation. The setting is now supervised learning for binary classification. The label vector $\boldsymbol{y}$ corresponding to four input samples $\boldsymbol{X} = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_4]^T$ is defined as

$$\boldsymbol{y} = [1, 0, 1, 0]^T. \tag{4}$$

2.1 Binary classification with sigmoid activation function

The sigmoid function $\sigma_{\text{sigm}}(t)$ maps each real input $t \in \mathbb{R}$ to a value in the interval $(0, 1)$. Mathematically,

$$\sigma_{\text{sigm}} : \mathbb{R} \ \rightarrow \ (0, 1), \quad \hat{y} = \sigma_{\text{sigm}}(t).$$

Task.

(1) At what output value $\hat{y}$ is the decision threshold defined, and which input value t corresponds to this boundary?

(2) Interpret the label vector $\boldsymbol{y}$ in Equation (4). Which class does each input sample $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_4$ belong to in binary classification?

(3) How is the sigmoid output $\hat{\boldsymbol{y}}$ interpreted for binary classification, and how does it determine the predicted class label?

2.2 Loss functions

In binary classification, the loss function evaluates how well the predictions $\hat{y}_i$ match the true labels y_i, i.e., $\mathcal{L}(\hat{y}_i, y_i)$. Different loss choices influence optimization and learning behavior. Here we compare two standard losses, Mean Squared Error (MSE)

$$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2, \tag{5}$$

and Binary Cross Entropy (BCE)

$$\mathcal{L}_{BCE} = -\frac{1}{N} \sum_{i=1}^{N} \Big(y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i) \Big), \tag{6}$$

assuming the output layer uses the sigmoid activation function.

For two activation pairs $(\sigma_1, \sigma_2) \in \{(\sigma_{ReLU}, \sigma_{sigm}), (\sigma_{sigm}, \sigma_{sigm})\}$, carry out the following steps:

(4) Using corresponding predictions $\hat{\boldsymbol{y}}$ from Table 2 and the label vector in Equation (4), compute the Mean Squared Error (MSE). Show intermediate steps and report the final value rounded to two decimals.

(5) Compute the Binary Cross Entropy (BCE) for the same predictions and labels. Show how each term in the sum is calculated, and report the final value rounded to two decimals.

(6) Compare the two loss values (MSE and BCE) for each activation pair. Which loss produces larger penalties for misclassified samples? Provide a short explanation.

(7) Suppose one prediction is very close to the decision boundary (e.g., $\hat{y}_i = 0.49$ or 0.51). Which loss function is more sensitive to such small deviations, and why?

2.3 Gradients of loss functions

In this problem, the loss $\mathcal{L}$ quantifies the mismatch between predictions $\hat{y}_i$ and true labels y_i. To reduce this loss, we adjust the model parameters $\boldsymbol{\theta}$,

$$\boldsymbol{\theta} = \{\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{w}_2, b_2\}.$$

Definition of the gradient. The gradient of the loss with respect to the parameters measures how small changes in the parameters affect the value of the loss. Formally, we define

$$\nabla_{\boldsymbol{\theta}}\mathcal{L} = \{\nabla_{\boldsymbol{W}_1}\mathcal{L},\ \nabla_{\boldsymbol{b}_1}\mathcal{L},\ \nabla_{\boldsymbol{w}_2}\mathcal{L},\ \nabla_{b_2}\mathcal{L}\},$$

where each component is a partial derivative of the loss with respect to the corresponding parameter. For example,

$$\nabla_{\boldsymbol{W}_1}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_1}. \tag{7}$$

Parameters in the forward propagation. In the forward propagation, each parameter contributes to the loss through a sequence of intermediate computations. For example, both $\boldsymbol{W}_1$ and $\boldsymbol{b}_1$ determine $\boldsymbol{Z}_1$, which then propagates through the network until it influences the final loss:

$$\begin{matrix} \boldsymbol{W}_1 \searrow \\ \boldsymbol{b}_1 \nearrow \end{matrix} \quad \underbrace{\boldsymbol{Z}_1 \longrightarrow \boldsymbol{A}_1 \longrightarrow \boldsymbol{z}_2}_{embeddings} \longrightarrow \hat{\boldsymbol{y}} \longrightarrow \mathcal{L}.$$

Analogously, the parameters $\boldsymbol{w}_2$ and $\boldsymbol{b}_2$ both contribute to $\boldsymbol{z}_2$:

$$\begin{matrix} \boldsymbol{w}_2 \searrow \\ b_2 \nearrow \end{matrix} \quad \underbrace{\boldsymbol{z}_2}_{embedding} \longrightarrow \hat{\boldsymbol{y}} \longrightarrow \mathcal{L}.$$

Chain rule and backpropagation. Since the network is a composition of functions, the influence of $\boldsymbol{W}_1$ propagates forward through $\boldsymbol{Z}_1 \to \boldsymbol{A}_1 \to \boldsymbol{z}_2 \to \hat{\boldsymbol{y}}$ until it reaches the loss $\mathcal{L}$. When computing the gradient, this dependence is transmitted backward through the same chain in reverse order. By the *chain rule of differentiation*, the gradient with respect to $\boldsymbol{W}_1$ is

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_1} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} \cdot \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}_2} \cdot \frac{\partial \boldsymbol{z}_2}{\partial \boldsymbol{A}_1} \cdot \frac{\partial \boldsymbol{A}_1}{\partial \boldsymbol{Z}_1} \cdot \frac{\partial \boldsymbol{Z}_1}{\partial \boldsymbol{W}_1}. \tag{8}$$

General principle. The gradient with respect to any parameter is obtained by following the reverse path of the forward propagation and multiplying the corresponding local derivatives. The repeated application of this rule across all layers is what constitutes *backpropagation*, the standard algorithm for computing $\nabla_{\boldsymbol{\theta}} \mathcal{L}$ efficiently in neural networks.

(Optional) Partial derivatives. As shown in Equation (8), backpropagation is built entirely from partial derivatives, where the local partial derivatives are multiplied along the path by the chain rule. At each step in the forward propagation, we compute the local derivative of the output with respect to its immediate inputs. Table 3 summarizes the forward propagation equations together with the corresponding partial derivatives required for backpropagation.

(8) Assume $\sigma_2 = \sigma_I$. Write down the expression for the partial derivative $\dfrac{\partial \mathcal{L}_{MSE}}{\partial b_2}$ using the chain rule, based on the forward path $b_2 \to \boldsymbol{z}_2 \to \hat{\boldsymbol{y}} \to \mathcal{L}$ and Table 3 (Hint: Start with one sample, then average over many).

(9) Compute $\dfrac{\partial \mathcal{L}_{MSE}}{\partial b_2}$ above analytially (Hint. The gradient with respect to a parameter always has the same dimension as the parameter itself. Start with one sample, then average over many to avoid confusing of the dimension).

(10) (Optional) Repeat (8)-(9) with $\sigma_2 = \sigma_{sigm}$.

Autograd in PyTorch. `Autograd` tracks tensor operations when `requires_grad=True`. During the forward propagation, it builds a dynamic computation graph. Calling `loss.backward()` starts backpropagation from a scalar loss. The backprop writes gradients into the `.grad` buffers of leaf tensors such as `nn.Parameter`.

Gradients accumulate across calls, so use `optimizer.zero_grad()` before the next update. After the backward propagation, the graph is freed unless you request.

Use `tensor.detach()` or the context `torch.no_grad()` during evaluation and during parameter updates to stop tracking and to save memory. The gradient of each parameter has the same shape as the parameter, which enables the update by gradient descent $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$.

Forward propagation	**Partial derivative**
$\boldsymbol{Z}_1 = \boldsymbol{X}\boldsymbol{W}_1 + \boldsymbol{b}_1$	$\dfrac{\partial \boldsymbol{Z}_1}{\partial \boldsymbol{W}_1} = \boldsymbol{X}$ $\dfrac{\partial \boldsymbol{Z}_1}{\partial \boldsymbol{b}_1} = \mathbf{1}$
$\boldsymbol{A}_1 = \sigma_I(\boldsymbol{Z}_1) = \boldsymbol{Z}_1$	$\dfrac{\partial \boldsymbol{A}_1}{\partial \boldsymbol{Z}_1} = \dfrac{\partial \boldsymbol{Z}_1}{\partial \boldsymbol{Z}_1} = \mathbf{1}$
$\boldsymbol{A}_1 = \sigma_{sigm}(\boldsymbol{Z}_1) = \dfrac{1}{1+e^{-\boldsymbol{Z}_1}}$	$\dfrac{\partial \boldsymbol{A}_1}{\partial \boldsymbol{Z}_1} = \dfrac{\partial \sigma_{sigm}(\boldsymbol{Z}_1)}{\partial \boldsymbol{Z}_1} = \boldsymbol{A}_1\big(1 - \boldsymbol{A}_1\big)$
$\boldsymbol{A}_1 = \sigma_{ReLU}(\boldsymbol{Z}_1) = \max(0, \boldsymbol{Z}_1)$	$\dfrac{\partial \boldsymbol{A}_1}{\partial \boldsymbol{Z}_1} = \dfrac{\partial \sigma_{ReLU}(\boldsymbol{Z}_1)}{\partial \boldsymbol{Z}_1} = \begin{cases} \mathbf{1} & \text{if } \boldsymbol{Z}_1 > 0 \\ \mathbf{0} & \text{if } \boldsymbol{Z}_1 \leq 0 \end{cases}$
$\boldsymbol{z}_2 = \boldsymbol{A}_1\boldsymbol{w}_2 + b_2$	$\dfrac{\partial \boldsymbol{z}_2}{\partial \boldsymbol{w}_2} = \boldsymbol{A}_1$ $\dfrac{\partial \boldsymbol{z}_2}{\partial b_2} = 1$
$\hat{\boldsymbol{y}} = \sigma_I(\boldsymbol{z}_2) = \boldsymbol{z}_2$	$\dfrac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}_2} = \dfrac{\partial \boldsymbol{z}_2}{\partial \boldsymbol{z}_2} = \mathbf{1}$
$\hat{\boldsymbol{y}} = \sigma_{sigm}(\boldsymbol{z}_2) = \dfrac{1}{1+e^{-\boldsymbol{z}_2}}$	$\dfrac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}_2} = \dfrac{\partial \sigma_{sigm}(\boldsymbol{z}_2)}{\partial \boldsymbol{z}_2} = \boldsymbol{z}_2\big(1 - \boldsymbol{z}_2\big)$
$\hat{\boldsymbol{y}} = \sigma_{ReLU}(\boldsymbol{z}_2) = \max(0, \boldsymbol{z}_2)$	$\dfrac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}_2} = \dfrac{\partial \sigma_{ReLU}(\boldsymbol{z}_2)}{\partial \boldsymbol{z}_2} = \begin{cases} \mathbf{1} & \text{if } \boldsymbol{z}_2 > 0 \\ \mathbf{0} & \text{if } \boldsymbol{z}_2 \leq 0 \end{cases}$
$\mathcal{L}_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) \leftarrow$ See Equation (5)	$\dfrac{\partial \mathcal{L}_{MSE}}{\partial \hat{\boldsymbol{y}}} = \hat{\boldsymbol{y}} - \boldsymbol{y}$
$\mathcal{L}_{BCE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) \leftarrow$ See Equation (6)	$\dfrac{\partial \mathcal{L}_{BCE}}{\partial \hat{\boldsymbol{y}}} = \dfrac{\boldsymbol{y}}{\hat{\boldsymbol{y}}} + \dfrac{1-\boldsymbol{y}}{1-\hat{\boldsymbol{y}}}$

Table 3: Forward propagation and their corresponding local partial derivatives.

(11) Interpret `main()` using the example code below.

```
def main():
    model = SimpleMLP(input_size=3, \
                      hidden_size=2, \
                      output_size=1, \
                      activ_func1=relu, \
                      activ_func2=sigmoid)
    model.set_weights(W1, b1, W2, b2)
    model = model.to(device)
    loss, x_grad, \
        fc1_weight_grad, fc1_bias_grad, \
            fc2_weight_grad, fc2_bias_grad \
                = model.compute_gradients(X.to(device), \
                                          binary_cross_entropy, \
                                              y_gt.to(device))
```

Example code.

```
# %%
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt

def get_device() -> torch.device:
    """Pick CUDA, MPS on Apple, or CPU in that order."""
    if torch.cuda.is_available():
        return torch.device("cuda")
    if torch.backends.mps.is_available():
        return torch.device("mps")
    return torch.device("cpu")
device = get_device()
print(f"Using device {device}")

######################################################################
# Define input matrix X (4x3)
# Define grund truth y_gt (4x1)
# Define first layer weights W1 and bias b1
# Define second layer weights W2 and bias b2
#                      w/ torch.tensor & dtype=torch.float32
# They will be plugged in to the model later.
######################################################################

# %%
######################################################################
# Define loss
def MSE(y_pred, y_true):
    return F.mse_loss(y_pred, y_true)

def binary_cross_entropy(y_pred, y_true):
    return F.binary_cross_entropy_with_logits(y_pred, y_true)

# %%
######################################################################
# Define 2-layer MLP
class SimpleMLP(nn.Module):
    def __init__(self, input_size,
                       hidden_size,
                       output_size,
                       activ_func1,
                       activ_func2):
```

```
    super(SimpleMLP, self).__init__()
    self.fc1 = nn.Linear(input_size, hidden_size)
    self.fc2 = nn.Linear(hidden_size, output_size)
    self.activ_func1 = activ_func1
    self.activ_func2 = activ_func2

def forward(self, x):
    x = self.fc1(x)
    x = self.activ_func1(x)
    x = self.fc2(x)
    x = self.activ_func2(x)
    return x

# To use our own weights and bias
def set_weights(self, W1, b1, W2, b2):
    with torch.no_grad():
        self.fc1.weight.copy_(W1.T)
        self.fc1.bias.copy_(b1)
        self.fc2.weight.copy_(W2.T)
        self.fc2.bias.copy_(b2)

# To test autograd
def compute_gradients(self, x, loss_fn, y_gt):
    x.requires_grad_(True)
    y = self.forward(x)
    loss = loss_fn(y, y_gt)
    loss.backward()
    return loss, x.grad, \
        self.fc1.weight.grad, self.fc1.bias.grad, \
            self.fc2.weight.grad, self.fc2.bias.grad
```

2.4 Gradient descent

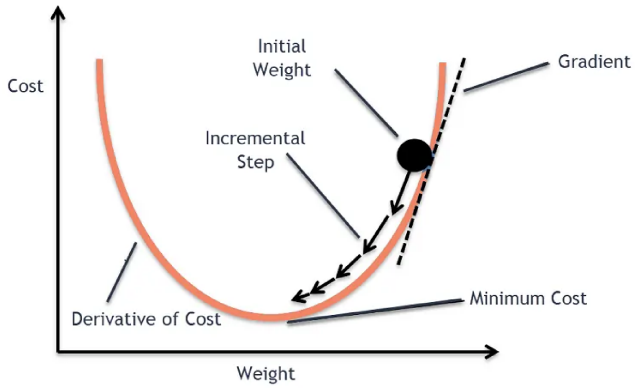

Figure 1: Motivating illustration of gradient descent [1].

Figure 1 depicts a simplified loss surface as a function of the parameter $\boldsymbol{\theta}$. The gradient at a point points toward the direction of the steepest increase in loss. A small move along this direction increases the loss. Since training aims to reduce the loss, we step in the opposite direction of the gradient, which moves us downhill.

(Optional) Mathematical derivation. By definition of the gradient, we have

$$\nabla_{\boldsymbol{\theta}} \mathcal{L} = \frac{\mathcal{L}(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) - \mathcal{L}(\boldsymbol{\theta})}{\Delta\boldsymbol{\theta}},$$

and a small step $\Delta\boldsymbol{\theta}$ changes the loss by a first order approximation

$$\mathcal{L}(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) \approx \mathcal{L}(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} \mathcal{L} \cdot \Delta\boldsymbol{\theta}.$$

Choose $\Delta\boldsymbol{\theta} = -\eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$ with step size $\eta > 0$. Then

$$\mathcal{L}\Big(\underbrace{\boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}}_{\text{gradient descent}} \Big) \approx \mathcal{L}(\boldsymbol{\theta}) - \underbrace{\eta \, \|\nabla_{\boldsymbol{\theta}} \mathcal{L}\|_2^2}_{>0} < \mathcal{L}(\boldsymbol{\theta}),$$

which is smaller than $\mathcal{L}(\boldsymbol{\theta})$ whenever the gradient is nonzero. Hence gradient descent reduces the loss for a sufficiently small step size. Gradient ascent would increase the loss.

(12) Implement a gradient descent method inside `SimpleMLP`. Use step size $\eta = 0.01$.

(13) Add a training loop in `main`. Call `compute_gradients`, then update the parameters with `gradient_descent`. Record the loss each epoch. Report the loss after 1000 epochs. Plot the loss curve. Give two to three sentences on convergence behavior. Mention whether the curve is smooth and decreasing, or noisy, or flat.

(14) Increase the step size η to 10. Overlay the new curve on the same plot. Discuss the result in two to three sentences. Comment on divergence or oscillation if present.

```
# %%
class SimpleMLP(nn.Module):
    # ...
    # To use our own weights and bias
    def set_weights(self, W1, b1, W2, b2):
        with torch.no_grad():
            self.fc1.weight.copy_(W1.T)
            self.fc1.bias.copy_(b1)
            self.fc2.weight.copy_(W2.T)
            self.fc2.bias.copy_(b2)

    # To test autograd
    def compute_gradients(self, x, loss_fn, y_gt):
        x.requires_grad_(True)
        y = self.forward(x)
        loss = loss_fn(y, y_gt)
        loss.backward()
        return loss, x.grad, \
            self.fc1.weight.grad, self.fc1.bias.grad, \
                self.fc2.weight.grad, self.fc2.bias.grad

    def gradient_descent(self, step_size,
                         fc1_weight_grad, fc1_bias_grad,
                         fc2_weight_grad, fc2_bias_grad):
        with torch.no_grad():
            self.model.fc1_weight -= step_size * fc1_weight_grad
            ...

def main():
    step_size = 1e-2
    num_epochs = 1e3
    losses = []

    for epoch in num_epochs:
        loss, x_grad, \
        gW1, gb1, gW2, gb2 = model.compute_gradients(
            X.to(device),
            binary_cross_entropy,
            y_gt.to(device),
        )
        losses.append(loss.item() if hasattr(loss, "item") else float(loss))
```

```
    model.gradient_descent(step_size, gW1, gb1, gW2, gb2)
    if epoch % 10 == 0:
        print(f"Intermediate loss at epoch {epoch}: {losses[-1]:.6f}")
    if np.isnan(losses[-1]):
        break

plt.figure()
plt.plot(losses, label="step size 1e-2")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.title("Training loss")
plt.show()
```

3 MNIST image classification with MLP

NB: This section is completely optional. Feel free to skip it if it is too much workload for you. But for people considering their career in the ML research, you may get some inspiration from it.

You will study how learning setups in neural networks influence stability, speed, and generalization. Use the provided codebase. For this problem, you are asked to modify some configurations in `config.py`.

What you should log. Record the training loss and test loss at every epoch. Plot both curves together in a single figure. Write two or three sentences interpreting the trends you observe. Most of the necessary functions are already implemented in the codebase, so reuse them as you prefer.

Tip. You will repeat the same experiment many times with small hyperparameter changes. If you get familiar with Weight and bias `(wandb)` to track runs, compare metrics, and organize results, it will be helpful.

3.1 Training configuration

Baseline run.

(1) Run the code as given for 100 epochs. Use the default batch size and the default learning rate in the codebase. Evaluate on the test set.

```
TRAINING_CONFIG = {
'batch_size': 64,
'learning_rate': 0.001,
'num_epochs': 100,
'test_batch_size': 1000
}
```

(2) Attach and examine the loss curve figure you created. At which epoch does overfitting begin, and how would you decide when to stop training given the noisy test loss curve? Explain your approach to pinpoint the exact epoch and attach your code implementation. Conclude with a short discussion of the results.

Batch size sweep.

(3) Sweep the batch size in the set $B \in \{6, 6400\}$ while all other settings remain at their baseline values. Train for 100 epochs with each batch size.

(4) Compare the learning curves and report the best epoch for three batch sizes, $B \in \{6, 64, 6400\}$.

(5) Identify which batch size yields the best test accuracy and lowest test loss. Explain in two or three sentences why this batch size is most effective, considering both gradient noise and the number of parameter updates.

Learning rate sweep.

(6) Keep batch size at 64 and all other settings at baseline values. Train for 100 epochs with $\eta \in \{1, 0.00001\}$. If training diverges, define divergence as loss becomes NaN or exceeds a large threshold such as 10^3. Stop that run and record it as divergent.

(7) Compare the learning curves and report the best epoch for three learning rates, $\eta \in \{1, 0.001, 0.00001\}$.

(8) Identify which learning rate yields the best test accuracy and lowest test loss. Explain in two or three sentences why this learning rate is most effective, considering both gradient noise and the number of parameter updates.

3.2 Model configuration

While keeping the default training configuration unchanged, we now modify the model configuration.

```
# Baseline model configuration
MODEL_CONFIG = {
    'input_size': 784,
    'hidden_sizes': [512, 256, 128],
    'num_classes': 10,
    'dropout_rate': 0.2
}
```

(8) Why is the `input_size` set to 784? Can it be changed to another number?

(9) What is the total number of parameters in the default model configuration? Explain how you calculate the number.

(10) Assume your machine has limited memory, so you must reduce the model size. Set `hidden_sizes = [16]`. What is the number of parameters?

(11) Rerun the experiment with the smaller neural networks. Compare the results with the baseline configuration in terms of training loss, test loss, and test accuracy. In two or three sentences, discuss how reducing the number of hidden units affects model capacity, convergence, and generalization.

Study the code and answer the following questions. The model definition can be found in `./MNIST/models/MLP.py`.

(11) What are `nn.Dropout`? Explain its role.

(12) Remove all the dropout layers from the smaller networks of `hidden_sizes=[16]` and run the experiment. Compare the result with one with dropout and discuss.

(13) What activation function is used in the output layer? Could we add a sigmoid function? Explain why this is not appropriate.

(14) Explain the role of the softmax function in multi-class classification.

(15) Add a softmax function to the end of the network and observe how the prediction output $\hat{\boldsymbol{y}}$ changes. Compare these outputs with those from the model without softmax.

(16) Discuss the difference between raw logits and probability distributions, and explain why softmax is preferred when interpreting outputs for multi-class classification.

References

[1] Crypto1. Gradient descent algorithm how does it work in machine learning. https://www.analyticsvidhya.com/blog/2020/10/how-does-the-gradient-descent-algorithm-work-in-machine-learning/, 2025. Analytics Vidhya, last updated 04 Apr 2025.

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.