Homework 4

4190.408 001, Artificial Intelligence

November 19, 2025

Submission

- In this homework we will learn and practice the basics of deep learning by building a deep learning model and training the model by ourselves.
- Assignment due: **Dec 3, 11:59 pm**
- Individual Assignment
- Submission through ETL. Please unzip the HW4.zip file provided on ETL, and inside there will be python files for each section of the homework. Please read the instructions in the PDF, and **fill in the TODOs in the corresponding files.** The files should be compressed into a zip file along with the report in pdf and named in the format "{student_ID}_{first name}_{last name}.zip", e.g., "2025-12345_Jeonghyeon_Na.zip". Please write your name in English.
- Collaborations on solving the homework is allowed. Discussions are encouraged but you should think about the problems on your own.
- If you do collaborate with someone or use a book or website, you are expected to write up your solution independently. That is, close the book and all of your notes before starting to write up your solution.
- Make sure you cite your work/collaborators at the end of the homework.
- **Using deep-learning libraries such as PyTorch, TensorFlow or Scikit-Learn is prohibited in section 1. You may and are recommended to use Numpy package.**
- **Honor Code**: This document is exclusively for Fall 2025, 4190.408 students with Professor Hanbyul Joo at Seoul National University. Any student who references this document outside of that course during that semester (including any student who retakes the course in a different semester), or who shares this document with another student who is not in that course during that semester, or who in any way makes copies of this document (digital or physical) without consent of Professor Hanbyul Joo is guilty of cheating, and therefore subject to penalty according to the Seoul National University Honor Code.

1 Multi-Layer Perceptron Implementation [40p]

In this section, your objective is to classify images of handwritten alphabets by creating a Neural Network from the ground up. You will implement the forward and backward propagation of 3-layer multi-layer perceptron, and will train the model with given dataset. You'll need to develop all the necessary components for initializing, training, assessing, and making predictions using this network.

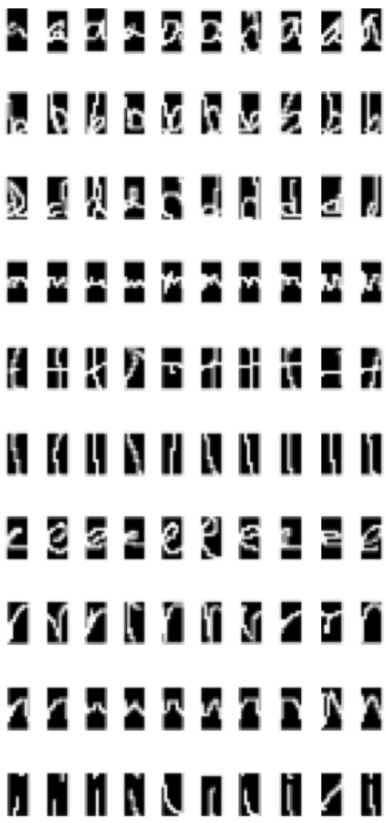

Figure 1: 10 Random Images of 10 Letters in OCR

Datasets We will employ a section of an Optical Character Recognition (OCR) dataset, which includes images of the entire alphabet written by hand. For our purpose, we'll focus on a subset comprising the letters "a," "e," "g," "i," "l," "n," "o," "r," "t," and "u." Included in your materials are three separate datasets extracted from this collection: a small set with 60 instances per character (50 for training, 10 for testing). Refer to Figure 1.1 for a selection of 10 representative images from various letters in the dataset.

Implementations

You will implement the following functions and classes in **section_1.py** file. Our model is composed of 3 linear layers, with ReLU activation layer in between each layer. Output of the final linear layer then goes into softmax operation, and cross entropy loss is used to calculate loss. You will train using gradient descent, putting all the training data into the model at once and calculating loss. **Using deep learing libraries such as PyTorch or TensorFlow is prohibited in section 1. Please use Numpy to implement all the methods in section 1.** Please make sure that all the implementation can operate in batches. You are welcome to use helper function.

```
class Linear
    ...
```

Input: Input and output dimension
Output: Single linear layer
Description: You will implement the linear layer with forward/backward propagation. When initializing, the input and output dimensions are given as input. `forward()` performs linear operation. `backward()` receives the gradient with respect to the output of the linear layer, and calculates the gradient with respect to the input of the original input of linear layer. When initializing the weights, please use He Normal Initialization, where $W \sim N(0, \sqrt{\frac{2}{n_{in}}}.\ n_{in})$ is the input feature dimension. Bias should be initialized as zero.

```
class ReLU
    ...
```

Description: You will implement the ReLU layer with forward/backward propagation. `forward()` performs ReLU operation, and `backward()` calculates the gradient.

```
def softmax
    ...
return softmax_output
```

Input: Input data
Output: Softmax output
Description: You will implement softmax operation function. It will be used in training and validation to create logits.

```
class SimpleNetwork
    ...
```

Input: Dimensions of input, hidden layer1, hidden layer2, output
Output: 3-Layer MLP model
Description: You will implement simple 3-Layer MLP with forward/backward propagation. Please use the layers implemented in class `Linear` and `ReLU`.

```
def train_with_gd
    ...
return losses
```

Input: Implemented network, input train data, ground truth label, learning rate and number of epochs
Output: Losses in each training epochs
Description: You will implement training function with gradient descent. The training data and ground truth label are given as input. You will use the implemented model, softmax function and cross entropy loss to calculate loss and then calculate gradient with backpropagation. Then, you will update the parameters of the network using the gradient and learning rate.

In the **section_1.py** file, you will find the implemented code for loading the train and test datasets. We have set the initial hyperparameters, including the hidden layers sizes, learning rate, and number of epochs. You may use the provided code to test your code and identify any bugs. You are also encouraged to try different hyperparameter settings and observe how they affect training.

2 Convolutional Neural Networks [30p]

In this section, your objective is to classify various images in CIFAR-10 dataset, using neural networks with convolution layers. You will implement two models that both use convolution, and see how the different structure of the models affects the performance of the classification task.

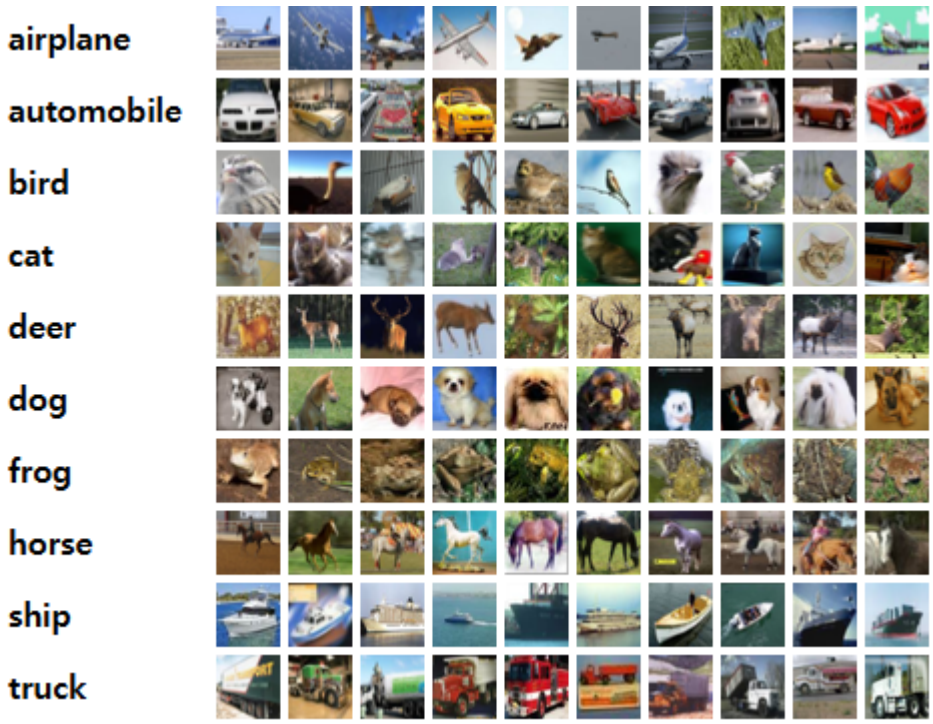

Figure 2: Example of CIFAR-10 dataset

Datasets CIFAR-10 Dataset will be used for the image classification task. CIFAR-10 is a widely used benchmark dataset for image classification in machine learning and deep learning. It contains 60,000 RGB images in total, 32x32 in resolution. It contains images of 10 classes, with 6,000 images in each class. CIFAR-10 provides substantial variability in object pose, lighting, and background while maintaining a small image resolution, making it a standard testbed for assessing the performance of convolutional neural networks and other visual recognition models.

Implementations

You will implement two convolutional neural networks in **model.py** file, a modified version of LeNet and a simplified version of ResNet. The original LeNet was designed for recognizing the handwriting in MNIST dataset, which consists of black and white images of handwriting with 28x28 resolution. CIFAR-10 however, has 3-channel RGB images with 32x32 resolution. We will implement a modified version that reflects the changes in the channels and replaces features such as TanH activation here. For the ResNet, we will implement a simplified version of it, given CIFAR-10's relatively small resolution. You can train with your implementation and CIFAR-10 dataset by running **section_2.py**. Please use the PyTorch package to implement the desired models in this section. You are welcome to use any helper function to aid the implementation.

`class LeNet5`

`...`

Description: You will implement a modified version of LeNet. Modified LeNet consists of

- Convolution layer with input channel 3, output channel 6 and kernel size 5
- Max Pooling layer with kernel size 2 and stride 2
- Convolution layer with input channel 6, output channel 16 and kernel size 5
- Max Pooling layer with kernel size 2 and stride 2
- Flattening
- Linear layer with input dim 400 and output dim 120
- Linear layer with input dim 120 and output dim 84
- Linear layer with input dim 84 and output dim 10

After each convolution and linear layer, ReLU activation is applied, except for the final linear layer.

`class RedidualBLock`

`...`

Input: Number of input/output channels of the convolution layer.
Output: Bottleneck module
Description: ResNet is made up of building blocks called "bottlenecks". The structure of a building block in the original ResNet paper is as follows:

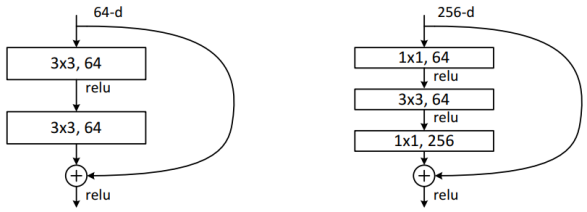

Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

You will use the structure on the left side of the figure, and change the parameters to fit our settings. After each convolution layer, batch normalization layer is followed. In each convolution, kernel size is fixed to 3 × 3, and the stride of the second convolution is fixed to 1. First convolution's stride value is determined by input. Both convolution's padding is fixed to 1. When input and output channel is different, or output height or width is different to input due to stride value, use 1 × 1 convolution and BatchNorm to match the dimension in skip connection.

`class SimplifiedResNet`

`...`

Description: You will implement a simplified version of ResNet. This version keeps a similar structure as the original ResNet, first layer consisting of a convolution layer with output channel 16, kernel size 3, and stride 1, followed by BatchNorm and ReLU activation. Then, 3 residual layer is followed. Each residual layer consists of 3 ResidualBlocks, with the later two's stride fixed to 1. The first residual block's stride for each layer is 1, 2, 2 respectively. The output channel of the residual blocks for each layer is 16, 32, 64. After the residual layers, average pooling is applied via `AdaptiveAvgPool2d` and finally linear layer.

3 Learning Rate Scheduling Design and Analysis [30p]

The learning rate is one of the most influential hyperparameters in training neural networks. A rate that is too large may cause the model to diverge, while a rate that is too small leads to slow convergence and poor minima. Instead of keeping the learning rate fixed throughout training, *Learning Rate Scheduling* adjusts it dynamically to achieve faster and more stable optimization.

A typical strategy is to start with a relatively large learning rate to explore the loss landscape, and then gradually reduce it to allow fine-grained convergence. Modern deep learning frameworks such as PyTorch provide several built-in scheduling policies, including:

- **Step Decay**: Decreases the learning rate by a constant factor every fixed number of epochs.
- **Multi-Step Decay**: Generalizes step decay by allowing multiple milestone epochs at which the learning rate drops.
- **Exponential Decay**: Applies an exponential factor to the learning rate at each epoch.
- **Cosine Annealing**: Reduces the learning rate following a cosine curve, often with periodic restarts.

These scheduling methods help control the optimization trajectory, reduce oscillations, and improve both convergence speed and final accuracy. In this section, you will design your own learning rate scheduling strategies and analyze their effects through controlled experiments.

3.1 Implementing a Custom Scheduler

PyTorch provides the `LRScheduler` base class for implementing learning rate schedules. This class handles updating the optimizer's learning rate at each epoch according to a defined schedule.

```
torch.optim.lr_scheduler.LRScheduler(optimizer, last_epoch=-1)
```

Parameters:

- `optimizer` (`Optimizer`): Wrapped optimizer whose learning rate will be scheduled.
- `last_epoch` (`int`): The index of the last epoch. Default: -1 (starts from epoch 0).

Key Methods:

- `step(epoch=None)`: Updates the learning rate. Should be called after `optimizer.step()`.
- `get_last_lr()`: Returns the last computed learning rate(s) as a list.

Algorithm 1 Typical LR Scheduler Usage Pattern

```
Initialize optimizer with model parameters
scheduler ← LRScheduler(optimizer)
for epoch = 1 to epoch_num do
    Perform training step
    scheduler.step()
end for
```

For more detailed visualizations of how the learning rate changes across epochs, see **plot_scheduler.py**.

Custom schedulers can be implemented conveniently using `torch.optim.lr_scheduler.LambdaLR`. This scheduler takes a function `lr_lambda(epoch)` that computes a multiplicative factor for the learning rate at each epoch. The learning rate is then updated as:

$$\text{lr}_{\text{epoch}} = \text{lr}_{\text{initial}} \times \texttt{lr\_lambda}(\text{epoch})$$

Implementation Template:
Your task is to implement a **scheduler factory function** in **scheduler.py** file, that creates and returns a `lr_lambda` function configured with the given parameters:

```
def your_scheduler(param):
    # Extract hyperparameters from param dictionary
    # e.g., gamma = param["gamma"]

    def lr_lambda(epoch):
        # Implement your scheduling logic here
        # epoch: current epoch number (0-indexed)
        # Returns: a multiplier for the learning rate (typically 0 < value <= 1)
        return multiplier

    return lr_lambda
```

Example: A simple step decay scheduler that halves the learning rate every 10 epochs:

```
def StepDecay(param):
    step_size = param.get("step_size", 10)
    gamma = param.get("gamma", 0.5)

    def lr_lambda(epoch):
        return gamma ** (epoch // step_size)

    return lr_lambda
```

For epoch 0-9: returns $0.5^0 = 1.0$ (lr = initial_lr $\times$ 1.0)
For epoch 10-19: returns $0.5^1 = 0.5$ (lr = initial_lr $\times$ 0.5)
For epoch 20-29: returns $0.5^2 = 0.25$ (lr = initial_lr $\times$ 0.25)

Registration:
After implementing your scheduler, add it to the scheduler dictionary:

```
scheduler_func_dict = {
    "StepDecay": StepDecay,
    "YourScheduler1": YourScheduler1,
    "YourScheduler2": YourScheduler2,
    # Add more schedulers here
}
```

Assignment Requirements:

1. Implement **three different** custom schedulers with distinct scheduling strategies
2. Each scheduler should have a descriptive name reflecting its behavior
3. For each scheduler, include in your report:
    - A plot showing how the learning rate evolves over epochs (use **plot_scheduler.py**)
    - A brief explanation (2-3 sentences) describing how the learning rate changes and the motivation or use case for this scheduling strategy

3.2 Experiments

You will use **section_3.py** to evaluate how your learning rate schedulers influence training performance.

Experiment Configuration:
Each experiment must be added to the experiment dictionary in **section_3.py**:

```
exp_dict = {
    "experiment_name": {
        "Learning_rate": 0.1,
        "Scheduler": "YourSchedulerName",
        "Scheduler_param": {
            "param1": value1,
            "param2": value2
        }
    },
    # Add more experiments here
}
```

Requirements:

1. For each of your three schedulers, conduct **at least three experiments** with different hyperparameters
2. Run **section_3.py** to generate training curves comparing all experiments

Report:
Include the following in your report:

- The generated training curves showing loss and accuracy over epochs for all experiments
- Analysis of how different scheduling methods and hyperparameters affect:
    - Convergence speed (how quickly the model reaches good performance)
    - Training stability (smoothness of the learning curves)
    - Final performance (best accuracy achieved)
- Discussion of which scheduler configurations worked best and why