Assignment 4

Gary Geunbae Lee
CSED342 - Artificial Intelligence

Contact: TA Yejin Min (`yeajinmin@postech.ac.kr`), Sujin Woo(`happysujin@postech.ac.kr`)

Before moving onto the main part of the assignment (Part B), you will complete several simple review questions. Refer to the class slides before answering the questions. Write your answers to Word and save the file as "PartA_studentID.docx" (or pdf file).

Part A

Q1. Consider a classification problem with two binary features $x_1, x_2 \in \{0, 1\}$. Suppose the following probabilities are given:

$$P(Y = y) = \frac{1}{32}, \qquad P(x_1 = 1 \mid Y = y) = \frac{y}{46}, \qquad P(x_2 = 1 \mid Y = y) = \frac{y}{62}.$$

Assume the Naive Bayes conditional independence assumption holds between $x1$ and $x2$.
Which class will the Naive Bayes classifier predict for a test item with $x_1 = 1$, and $x_2 = 0$? [4pts]

Q2. Write True or False. If False, briefly explain the reason.
Q2-1. In reinforcement learning, the agent has the current state, and the environment performs an action on the state, and updates the state to the agent. The environment reports a reward for this new state (which may be positive or negative). [2pts]

Q2-2. In reinforcement learning, the agent knows the state and performs an action, resulting in a new state that it reports to the environment. Along with this new state comes a positive reward value. [2pts]

Q2-3. In reinforcement learning, the agent performs an action on the state, and reports it to the environment which returns a positive reward value for this state. Reinforcement learning is more powerful than supervised learning. [2pts]

Q3. Let $Q(s, a)$ indicate the estimated Q-value of state–action pair (s, a) at some point during Q-learning. Now your learner receives reward r after taking action a at state s and arrives at state s'. Before updating the Q-values based on this experience, what is the relationship between $Q(s, a)$ and $r + \gamma \max_{a'} Q(s', a')$? Please indicate the most restrictive relationship that always holds. For example, if $x < y$ always holds, use $<$ instead of $\leq$. Selecting "?" means it is not possible to determine a guaranteed relationship. [4pts]

$$Q(s, a) \ \square\ r + \gamma \max_{a'} Q(s', a')$$

(a) $=$ (b) $<$ (c) $>$ (d) $\leq$ (e) $\geq$ (f) ?

Part B

1 Introduction

Why Do We Need Convolutional Neural Networks?

Images possess strong **local spatial structure**: the meaning of one pixel depends heavily on its neighbors. Fully connected neural networks ignore this structure because they treat images as flat vectors. This leads to several issues:

- **Loss of adjacency**: permuting pixels does not affect network output.
- **Excessive parameters**: a 1-megapixel RGB image requires trillions of weights.
- **No spatial invariance**: the same feature in different locations appears unrelated.

Convolutional Neural Networks (CNNs) solve these problems through:

- **Local receptive fields**: each neuron sees only a small patch.
- **Weight sharing**: the same kernel is applied across all spatial locations.
- **Spatial invariance**: features look similar regardless of where they occur.
- **Downsampling**: max pooling or stride reduces resolution while keeping salient features.

These principles dramatically reduce the number of parameters and make CNNs highly suitable for visual data.
In this assignment, we will make a simple CNN architecture, and use it as forward.

Architecture of the Simple Classification CNN in This Assignment

You will implement the following Tiny CNN:

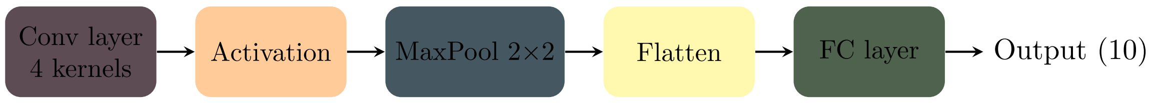

Figure 1: Tiny CNN architecture used in this assignment.

This minimal network captures the essential components of CNNs used in real-world deep learning systems.

1. Kernels and Feature Maps

A **kernel** (or filter) is a small learnable weight matrix such as a 3×3 or 5×5 window, that slides across the input image. Each kernel acts as a local pattern detector. During training, gradient descent adjusts the kernel weights so that each filter becomes sensitive to a specific visual pattern. Some kernels learn to detect horizontal edges, others detect vertical edges, corners, or dark-to-light transitions. As a result, even a single input image produces multiple transformed outputs, each corresponding to one learned pattern.

When a kernel is convolved with the input, it produces a new 2D activation map known as a **feature map**. Each feature map highlights where the pattern represented by the kernel appears in the input image. If the network uses four kernels, the output will contain four distinct feature maps, each accounting for a different learned feature. This mechanism allows a CNN to build a rich, multi-perspective representation of the input image from the very first layer.

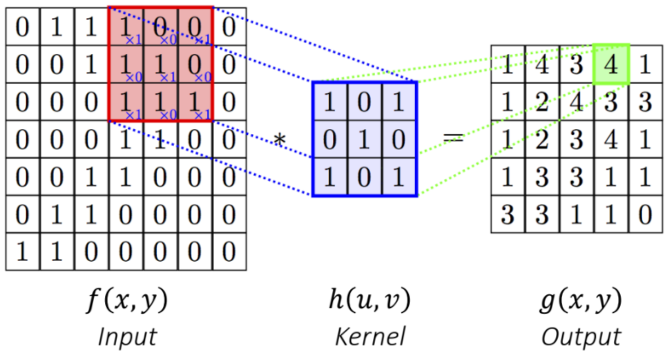

Figure 2: Convolution of input $f(x, y)$ with kernel $h(u, v)$ producing feature map $g(x, y)$.

2. Padding and Stride

Convolutional layers are defined not only by the size of the kernel, but also by two important hyperparameters: **stride** and **padding**. These parameters determine how the kernel moves across the image and how large the output feature map will be.

Stride. The **stride** parameter controls how far the kernel moves at each step. A stride of 1 means the kernel examines every overlapping $k \times k$ patch in the image, resulting in a dense, high-resolution feature map. A stride of 2 moves the kernel two pixels at a time, reducing the output resolution roughly by half in each dimension. This acts as a form of learned downsampling because the network evaluates fewer spatial positions.
Using a stride greater than 1 reduces computational cost and memory usage, but also discards fine-grained details. Thus, early convolution layers usually use stride 1, while deeper layers may use larger strides to gradually reduce spatial resolution and expand the receptive field.

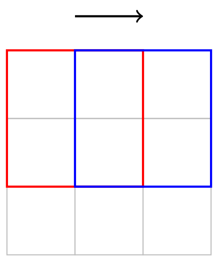

Figure 3: Sliding a 2×2 kernel across a 3×3 input with stride = 1. Padding determines whether the kernel can access the border pixels and whether the output size shrinks.

Padding. **Padding** adds artificial pixels—typically zeros—around the input image before applying the kernel. Without padding, a 3×3 kernel applied with stride 1 shrinks a 28×28 image to 26×26, because convolution only operates where the kernel completely fits inside the image. Padding serves three important purposes:

- **Preserving spatial size:** With padding $= 1$, a 3×3 kernel keeps the output size equal to the input size, e.g.,

$$28 \times 28 \xrightarrow{k=3,\, p=1,\, s=1} 28 \times 28.$$

- **Handling border information:** Padding ensures that edge pixels contribute equally to the convolution, rather than being ignored due to kernel size constraints.
- **Supporting deeper CNNs:** Without padding, repeated convolutions rapidly shrink feature maps. Padding enables arbitrarily deep stacks of convolutions without losing spatial resolution too quickly.

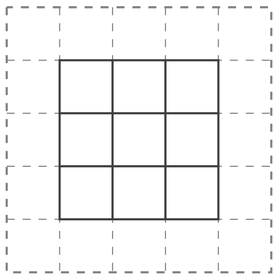

Figure 4: Zero-padding with $p = 1$: the padded region (dashed) surrounds the original input.

3. ReLU Activation

A convolution layer by itself performs only a linear transformation: each output value is just a weighted sum of nearby pixels (plus a bias). If we stack multiple linear layers without any nonlinearity, the result is still a single linear mapping. In other words, no matter how many convolution layers we stack, the network would be unable to represent functions more expressive than a single matrix multiplication.
Activation functions break this limitation. By inserting a nonlinearity after each convolution, the network can combine linear filters in flexible, nonlinear ways. This enables the model to capture complex structures in images: simple edges detected in early layers can be combined into corners, contours, strokes, and eventually high-level patterns such as digits or objects.
The most common activation used in CNNs is the **ReLU** (Rectified Linear Unit):

$$\mathrm{ReLU}(x) = \max(0, x).$$

ReLU introduces nonlinearity while keeping the computation simple. It preserves positive responses from convolution filters (signals indicating a detected pattern), while suppressing negative responses. As a result, ReLU acts as a gating mechanism: only "activated" features are passed to the next layer. This selective activation allows deeper networks to build hierarchical feature representations.

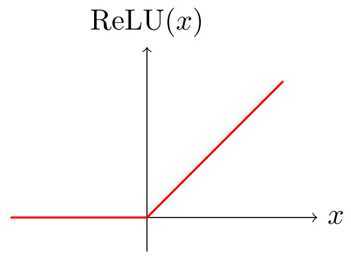

Figure 5: ReLU function

4. Pooling and Downsampling

Pooling operations reduce the spatial resolution of feature maps by summarizing local regions. The most common variant, max pooling, selects the maximum value within each window (e.g., 2×2). This captures whether a feature was present anywhere in that region, providing invariance to small translations or distortions in the input. Average pooling, on the other hand, computes the mean within each window, producing a smoother, downsampled representation.
Pooling reduces the size of feature maps, lowering computational cost and limiting the number of parameters in later layers. More importantly, it increases the receptive field of deeper layers: each pooled activation corresponds to a larger region in the original image. This progression—from local to global representation—is fundamental to how CNNs recognize increasingly abstract features.

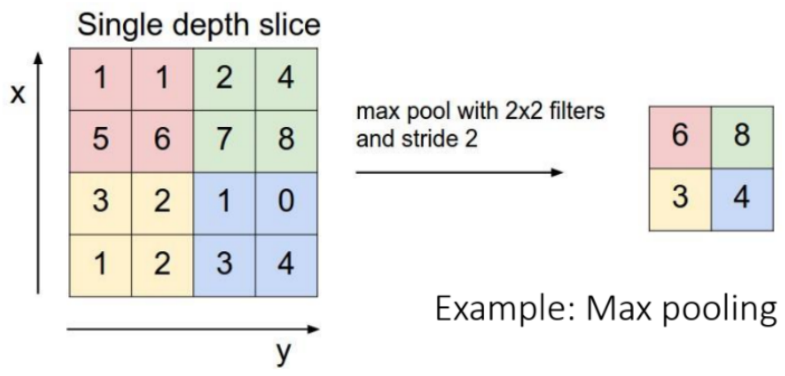

Figure 6: Example of 2×2 max pooling with stride 2. The input is divided into non-overlapping windows, and each window outputs its maximum value.

5. Flatten and Fully Connected Layers

After several convolution and pooling operations, the network produces a stack of compact feature maps summarizing the visual information extracted from the input. However, these maps are still arranged spatially in a three-dimensional tensor. A **flatten** operation reshapes this structure into a single vector so it can be processed by a fully connected (FC) layer.
The **fully connected layer** integrates information across all extracted features, allowing the network to combine local visual cues into a global interpretation. While convolutional layers are responsible for feature extraction, the FC layers often serve as the final reasoning stages that

determine which features correspond most strongly to each output class.

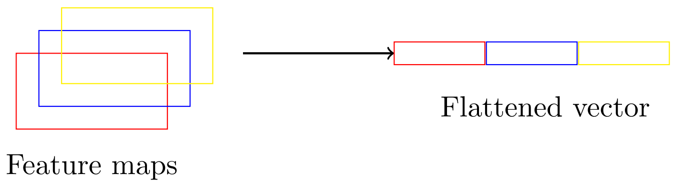

Figure 7: Flattening feature maps: the flattened vector concatenates channels in order (red → blue → yellow).

6. Softmax for Classification

In classification tasks, the final layer typically produces unnormalized scores (logits). The **softmax** function transforms these scores into a valid probability distribution:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}},$$

ensuring all probabilities are non-negative and sum to 1. This probabilistic output is essential for training with cross-entropy loss and for interpreting the model's predictions in a meaningful way. Softmax does not alter the relative ranking of logits but amplifies confidence differences, making it easier for the model to learn decisive boundaries between classes.

2 Programming Tasks (submission.py)

In this assignment, you will implement the core components of a Convolutional Neural Network using only NumPy. You must complete the functions inside `submission.py`. **Do not modify function names, arguments, or return types.** You may add helper functions if necessary.

Problem 1: Convolution Layer

In this problem, you will implement the building blocks of a single convolution layer with multiple kernels.

Problem 1a: Compute Output Spatial Dimensions [5 points]

Implement a function that computes the height and width of the convolution output given:

- Input size (H, W)
- Kernel size (k_H, k_W)
- Padding P
- Stride S

Your function should return $(\mathrm{outH}, \mathrm{outW})$.

Problem 1b: Implement `conv2d` [8 points]

Complete the function:

```
conv2d(x, w, padding, stride)
```

where:

- x: input image of shape (C_{in}, H, W)
- w: convolution kernels of shape $(C_{\text{out}}, C_{\text{in}}, k_H, k_W)$
- return an output of shape $(C_{\text{out}}, \mathrm{outH}, \mathrm{outW})$.

Your implementation must support support multi-channel input, apply zero-padding manually (using NumPy), implement stride manually.
Bias is *not* added inside `conv2d`; it will be handled in the forward pass.

Problem 2: Activation Function (ReLU) [2 points]

Implement the ReLU activation function:

$$\mathrm{ReLU}(x) = \max(0, x).$$

Your function should operate element-wise on a NumPy array of arbitrary shape.

Problem 3: Max Pooling [5 points]

Implement 2D max pooling:

$$\text{maxpool2d}(x, \text{pool\_size} = 2, \text{stride} = 2)$$

which:

- takes an input of shape (C, H, W),
- slides a pool_size $\times$ pool_size window,
- moves by `stride` each time,
- outputs the maximum in each window,
- returns a tensor of shape $(C, \text{outH}, \text{outW})$.

Problem 4: Flatten [2 points]

Implement the flatten operation:

$$(C, H, W) \;\rightarrow\; (C \cdot H \cdot W).$$

Simply reshape the tensor into a vector.

Problem 5: Fully Connected Layer [2 points]

Implement `fc2d(x, fc`$_W, fc_b$`);output` $= Wf + b$, where f is the flattened input. This layer maps the extracted convolutional features into class scores.

Problem 6: Implement the Full Forward Pass [2 points]

Complete the full CNN forward pipeline:

$$z = \text{conv2d}(x, \text{conv\_w}) + \text{conv\_b}$$

$$a = \text{ReLU}(z)$$

$$p = \text{MaxPool}(a)$$

$$f = \text{flatten}(p)$$

$$\text{logits} = Wf + b$$

Your function must return a NumPy array of shape (10).

Problem 7: Testing With Pretrained Weights [0 points] - Just take some fun!

We provide pretrained parameters for the convolution and fully connected layers (trained on a subset of MNIST). Use the script `predict_own.py` to load these weights and test your CNN forward pass.
Even if the prediction on your own handwriting is not perfect, don't worry - the learning might be imperfect, not your structure.

3 How CNN Training Works (Conceptual)

Although we provide pretrained weights, you must understand the training loop:

1. **Forward pass**: compute logits.
2. **Loss computation**: typically cross-entropy.
3. **Backward pass**: compute gradients using chain rule.
4. **Parameter update**: via SGD or Adam.
5. **Repeat for all minibatches**.

CNNs benefit from:

- local connectivity,
- weight sharing,
- pooling-based downsampling.

Thus, even deep architectures can be trained efficiently.