Deep Learning:
Homework #01

Prof. Tongyao Pang

Deadline: November 2, 2025

Name: **Zhou Shouchen**
Student ID: 2025213446

Problem 1

1. [2pts] Consider a sequence of values $\{x_1, x_2, \ldots, x_n\}$ of some variable x, and suppose we compute an exponentially weighted moving average using the formula $\mu_n = \beta\mu_{n-1} + (1-\beta)x_n$, where $0 < \beta < 1$. By making use of the following result for the sum of a finite geometric series

$$\sum_{k=1}^{n} \beta^{k-1} = \frac{1-\beta^n}{1-\beta}$$

show that if the sequence of averages is initialized using $\mu_0 = 0$, then the estimators are biased and that the bias can be corrected using

$$\hat{\mu}_n = \frac{\mu_n}{1-\beta^n}$$

Solution

Unrolling the recurrence geometric series for μ_n, we can get that

$$\mu_n = (1-\beta)\sum_{i=1}^{n} \beta^{n-i} x_i$$

Consider the sum of the weights assigned to each x_i in this expression: each x_i is weighted by $(1-\beta)\beta^{n-i}$. Thus the total weight is

$$\sum_{i=1}^{n}(1-\beta)\beta^{n-i} = (1-\beta)\sum_{k=0}^{n-1}\beta^k = (1-\beta)\cdot\frac{1-\beta^n}{1-\beta} = 1-\beta^n$$

Since $1-\beta^n < 1$, the computed μ_n is not a properly normalized weighted average, thus results to a bias. To remove this bias, we need to normalize the weights by dividing μ_n by the total weight $1-\beta^n$. This does not change the relative weights between data points, but rescales the weightsto sum to 1. Which means we can correct the bias by letting

$$\hat{\mu}_n = \sum_{i=1}^{n}\frac{(1-\beta)\beta^{n-i}}{1-\beta^n}x_i = \frac{\mu_n}{1-\beta^n}$$

Also, we can consider this in the probabilistic way. Suppose that the x_i are drawn from a distribution with mean μ. Then, taking the expectation of μ_n, we have

$$\mathbb{E}[\mu_n] = (1-\beta)\sum_{i=1}^{n}\beta^{n-i}\mathbb{E}[x_i] = (1-\beta)\sum_{i=1}^{n}\beta^{n-i}\mu = \mu(1-\beta)\sum_{k=0}^{n-1}\beta^k = \mu(1-\beta^n) \neq \mu$$

Which shows that μ_n is a biased estimator of μ, which could also be corrected by dividing by $1-\beta^n$, which is the same result as above.

So above all, we have proved that the estimators are biased and that the bias can be corrected using

$$\hat{\mu}_n = \frac{\mu_n}{1-\beta^n}$$

Problem 2

2. [3pts] Prove the shift invariant property of convolution:

$$\mathcal{G}(f(\cdot - a))(x) = \mathcal{G}(f(\cdot))(x - a),$$

where $\mathcal{G}$ is the convolutional operator $\mathcal{G}(f(\cdot)) = \displaystyle\int_{-\infty}^{+\infty} f(\tau)h(x-\tau)\mathrm{d}\tau$

Solution

From the definition of convolution, we can get that

$$\begin{aligned}
\mathcal{G}(f(\cdot - a))(x) &= \int_{-\infty}^{+\infty} f(\tau - a)h(x-\tau)\mathrm{d}\tau \\
\text{(Let } t = \tau - a\text{)} \quad &= \int_{-\infty}^{+\infty} f(t)h(x-(t+a))\mathrm{d}t \\
&= \int_{-\infty}^{+\infty} f(t)h((x-a)-t)\mathrm{d}t \\
&= \mathcal{G}(f(\cdot))(x-a)
\end{aligned}$$

So above all, we have proved that the shift invariant property of convolution

$$\mathcal{G}(f(\cdot - a))(x) = \mathcal{G}(f(\cdot))(x - a)$$

Problem 3

3. Neural Network Architectures

(1) [3pts] Derive the number of trainable parameters in a single convolutional layer with input size $H \times W \times C_{\text{in}}$, kernel size $k \times k$, and C_{out} output channels (assume bias), and compare it with a fully connected (dense) layer of the same input and output size.

(2) [2pts] Explain why positional information is necessary in Transformers and describe one method to inject positional information.

Solution

(1) 1. For the convolutional layer: since the input size is $H \times W$ with C_{in} channels, and the output has C_{out} channels, we could get that the size of each kernel is $k \times k \times C_{\text{in}}$, and there are C_{out} such kernels.

Also, since the channels are biased, thus for each kernel, a parameter as the bias is needed. So the total number of parameters is

$$(k^2 C_{\text{in}} + 1) \cdot C_{\text{out}} = k^2 C_{\text{in}} C_{\text{out}} + C_{\text{out}}$$

2. For a fully connected (dense) layer with the same input and output size, the number of the neurals of the input layer is

$$N_{\text{in}} = HWC_{\text{in}}$$

and the number of the neurals of the output layer is

$$N_{\text{out}} = HWC_{\text{out}}$$

And since its a fully connected layer, each neural in the output layer is connected to all the neurals in the input layer, with an additional bias term for each output neural. Thus, the total number of trainable parameters is

$$(N_{\text{in}} + 1) \cdot N_{\text{out}} = H^2 W^2 C_{\text{in}} C_{\text{out}} + HWC_{\text{out}}$$

Since the convolution layer could be regarded as shareing weights, it has much less parameters than the fully connected layer, especially when $k \ll H, W$.

(2) In the Transformer architecture, positional information is necessary because the attention mechanism is inherently permutation-equivariant. Without positional encoding, the model would get only a bag of tokens and cannot distinguish between different orderings of the same set of words. Thus it is impossible for the model to capture sequence order or relative distances, which would greatly decrease its ability to understand context and meaning in sequential data.

A common method to incorporate positional information is sinusoidal positional encoding. Given a position *pos* and dimension index i, with model dimension d, the positional encoding(PE) is defined as

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

The positional encoding is then added elementwise to the token embedding, which means that for the tokens in the position *pos*, its input to the model(the 0-th layer) is

$$x_{pos}^{(0)} = \text{Embed}(token_{pos}) + PE(pos)$$

With such positional encodings, the model can learn to attend to tokens based on their positions in the sequence without introducing additional parameters. And modern methods such as rotary positional encodings (RoPE)[1] is also widely used, which encodes relative positional information by rotating the query and key vectors in multi-head attention instead of adding positional encodings to the token embeddings.

[1] https://arxiv.org/abs/2104.09864

Problem 4

4. [15pts] Training a Convolutional Neural Network (CNN) on EuroSAT for Image Classification
In this assignment, you will train a deep learning model from scratch for EuroSAT dataset classification. EuroSAT is a dataset of 27,000 RGB satellite images (64×64 pixels) across 10 land cover classes, derived from Sentinel-2 satellite data for remote sensing classification tasks.
You are required to complete the following code by **filling in your own architecture and training function**. In the sections that specify **"To be implemented by students"**, you should replace pass with your own implementation.
After completing the implementation, answer the following questions and submit a report in Markdown/PDF format.

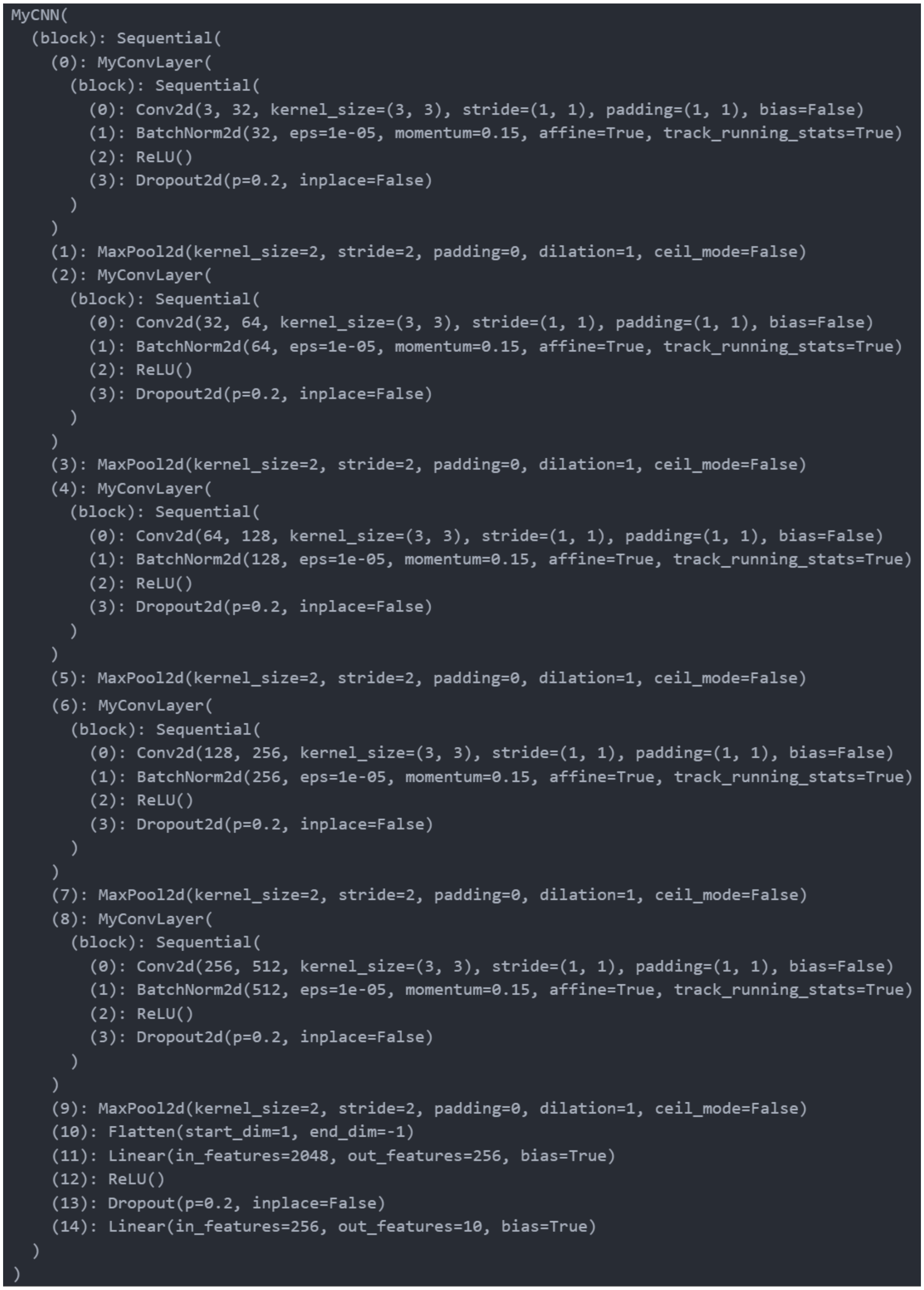
- Estimate the number of parameters and the feature map sizes at each layer.
- Report training accuracy and loss over epoches and the testing accuracy on test data.
- Compare training and test error with and without Batch Normalization and Dropout layers.
- Please also submit the Jupyter Notebook (.ipynb) with your complete, executable code.(You can just edit on this notebook file.)

Solution
The implementation code are in `hw1_coding.ipynb`.
(1) Figure 1 shows the architecture of the CNN model used for EuroSAT classification. The model's architecture is summarized as follows:

Layer	**Output Size**
Input	$3 \times 64 \times 64$
Conv(3→32)	$32 \times 64 \times 64$
MaxPool2	$32 \times 32 \times 32$
Conv(32→64)	$64 \times 32 \times 32$
MaxPool2	$64 \times 16 \times 16$
Conv(64→128)	$128 \times 16 \times 16$
MaxPool2	$128 \times 8 \times 8$
Conv(128→256)	$256 \times 8 \times 8$
MaxPool2	$256 \times 4 \times 4$
Conv(256→512)	$512 \times 4 \times 4$
MaxPool2	$512 \times 2 \times 2$
Flatten	2048
FC(2048→256)	256
FC(256→10)	10

With the methods mentioned in Problem 3, we could calculate the number of trainable parameters of the convolutional layers and the fully connected layers.
However, there have a small difference when considering applying Batch Normalization, it additionally add trainable parameters γ and β for each output channel, while the bias term in convolutional layers are removed.
So for the convolutional layers with input size (C_{in}, H, W), output size (C_{out}, H, W), kernel size $k \times k$, the number of trainable parameters is $C_{out} \times (C_{in} \times k \times k) + C_{out}$ without Batch Normalization, and $C_{out} \times (C_{in} \times k \times k) + 2 \times C_{out}$ with Batch Normalization.
For the fully connected layers, the number of trainable parameters with input size N_{in} and output size N_{out} is always $N_{out} \times N_{in} + N_{out}$.

Problem 4 continued on next page...



```
MyCNN(
  (block): Sequential(
    (0): MyConvLayer(
      (block): Sequential(
        (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.15, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): Dropout2d(p=0.2, inplace=False)
      )
    )
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): MyConvLayer(
      (block): Sequential(
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.15, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): Dropout2d(p=0.2, inplace=False)
      )
    )
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): MyConvLayer(
      (block): Sequential(
        (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.15, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): Dropout2d(p=0.2, inplace=False)
      )
    )
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): MyConvLayer(
      (block): Sequential(
        (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.15, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): Dropout2d(p=0.2, inplace=False)
      )
    )
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): MyConvLayer(
      (block): Sequential(
        (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.15, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): Dropout2d(p=0.2, inplace=False)
      )
    )
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Flatten(start_dim=1, end_dim=-1)
    (11): Linear(in_features=2048, out_features=256, bias=True)
    (12): ReLU()
    (13): Dropout(p=0.2, inplace=False)
    (14): Linear(in_features=256, out_features=10, bias=True)
  )
)
```

Figure 1: The architecture of the CNN model

Problem 4 continued on next page…

So the number of trainable parameters could be calculated as follows:

Layer	**Params(without BN)**	**Params(with BN)**
Conv 3 → 32	$32 \times (3 \times 3 \times 3) + 32 = 896$	$32 \times (3 \times 3 \times 3) + 2 \times 32 = 928$
Conv 32 → 64	$64 \times (32 \times 3 \times 3) + 64 = 18,496$	$64 \times (32 \times 3 \times 3) + 2 \times 64 = 18,560$
Conv 64 → 128	$128 \times (64 \times 3 \times 3) + 128 = 73,856$	$128 \times (64 \times 3 \times 3) + 2 \times 128 = 73,984$
Conv 128 → 256	$256 \times (128 \times 3 \times 3) + 256 = 295,168$	$256 \times (128 \times 3 \times 3) + 2 \times 256 = 295,424$
Conv 256 → 512	$512 \times (256 \times 3 \times 3) + 512 = 1,180,160$	$512 \times (256 \times 3 \times 3) + 2 \times 512 = 1,180,672$
FC 2048 → 256	$2048 \times 256 + 256 = 524,544$	$2048 \times 256 + 256 = 524,544$
FC 256 → 10	$256 \times 10 + 10 = 2,570$	$256 \times 10 + 10 = 2,570$
Total Parameters	**2,095,690**	**2,096,682**

So above all, the network without Batch Normalization has $2,095,690$ trainable parameters, while the one with Batch Normalization has $2,096,682$ trainable parameters.
(2) The training accuracy and loss over epoches and the testing accuracy on test data could be found in `hw1_coding.ipynb` part 4 and part 5.
(3) The training and test error with and without Batch Normalization and Dropout layers are in the following table 1. Specifically, batch normalization are added after each convolutional layer, and the bias term in convolutional layers are removed when using batch normalization. Dropout with $p = 0.05$ is applied after the first fully connected layer and every convolutional block.

Batch Normalization	Dropout	Train Acc (%)	Train Error (%)	Test Acc (%)	Test Error (%)
without	without	**95.48**	**4.52**	90.91	9.09
with	without	94.88	5.12	92.83	7.17
without	with	93.19	6.81	91.69	8.31
with	with	92.62	7.38	**93.13**	**6.87**

Table 1: Comparison of training and test performance with and without Batch Normalization and Dropout layers.

We observe that applying Batch Normalization improves test performance by reducing generalization error. And a small of Dropout($p = 0.05$) provides additional regularization, further improving test accuracy.
The best result is achieved with applying both Batch Normalization and Dropout, although it sacrifices some training accuracy, it achieves the lowest test error.