

EE449 HW1

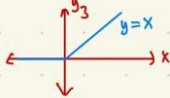
Yusuf Baran

1.

1.1

$$y_1 = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$
$$\frac{\partial y_1}{\partial x} = \frac{2e^{2x}(e^{2x} + 1) - (2 \cdot e^{2x})(e^{2x} - 1)}{(e^{2x} + 1)^2} = \frac{2 \cdot e^{4x} + 2e^{2x} - 2e^{4x} + 2e^{2x}}{(e^{2x} + 1)^2}$$
$$= \frac{2 \cdot e^{4x} + 2e^{2x} - 2e^{4x} + 2e^{2x}}{(e^{2x} + 1)^2} = \frac{4 \cdot e^{2x}}{(e^{2x} + 1)^2} = \frac{e^{4x} + 2e^{2x} + 1}{(e^{2x} + 1)^2} - \frac{e^{4x} - 2e^{2x} + 1}{(e^{2x} + 1)^2}$$
$$= 1 - [\tanh(x)]^2$$

$$y_2 = \sigma(x) = \frac{1}{1 + e^{-x}} \quad \frac{\partial y_2}{\partial x} = \frac{-e^{-x}}{(1 + e^{-x})^2} = \frac{1}{(1 + e^{-x})^2} - \frac{1 + e^{-x}}{(1 + e^{-x})^2} = [\sigma(x)]^2 - \sigma(x) = \sigma(x)(\sigma(x) - 1)$$

$$y_3 = \max(0, x)$$

$$\frac{\partial y_3}{\partial x} = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \\ \text{undefined}, & x = 0 \end{cases}$$

for plotting assume 0

Figure 1. Derivative calculations

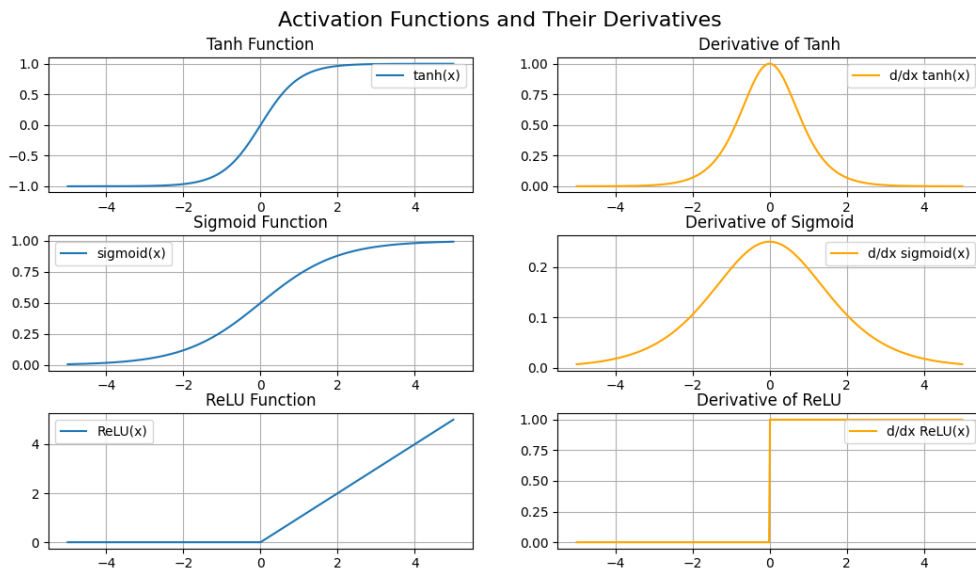


Figure 2. Functions' and derivatives' plots

1.2

i. Sigmoid:

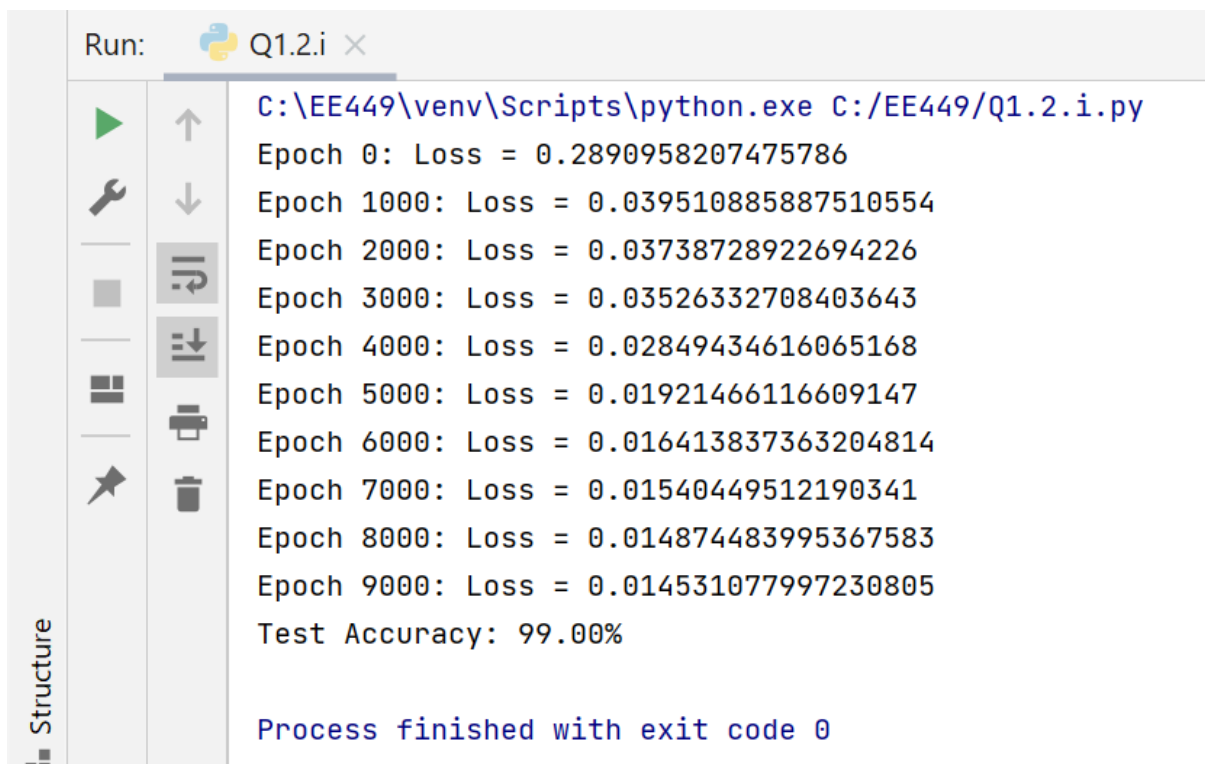


Figure 3. Loss and accuracy of sigmoid activation function

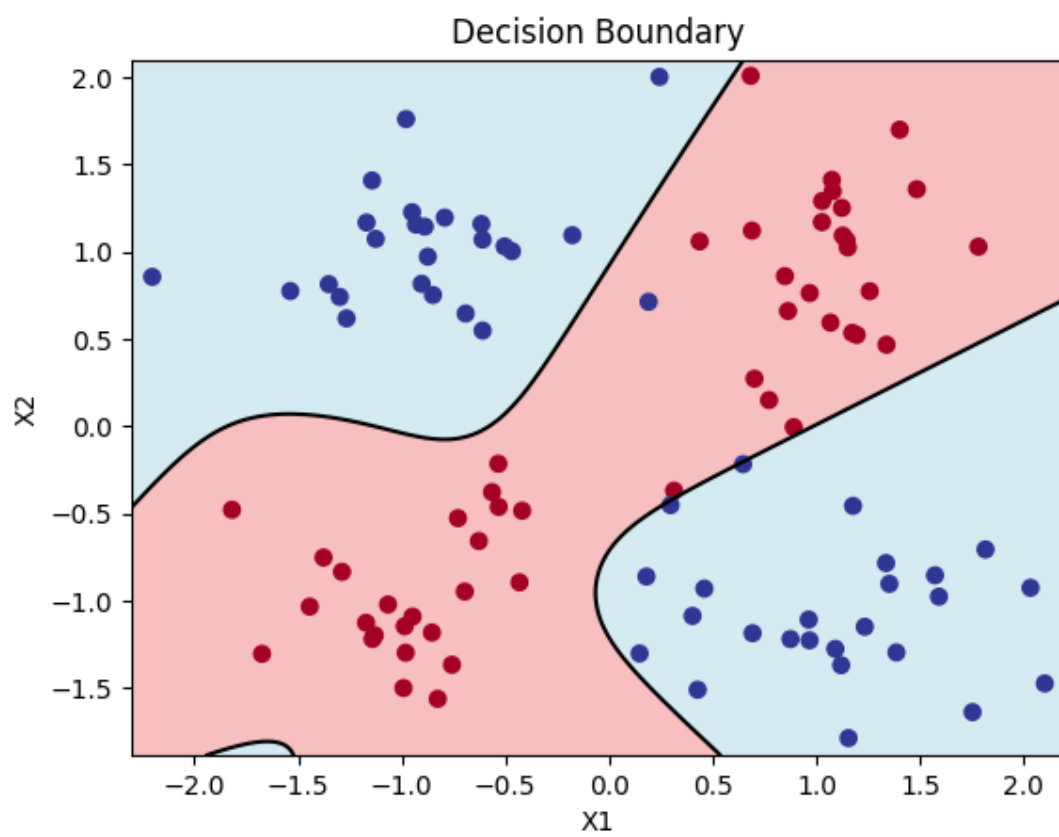


Figure 4. Decision boundary of sigmoid activation function

ii. Tanh:

```
Run: Q1.2.ii x
C:\EE449\venv\Scripts\python.exe C:/EE449/Q1.2.ii.py
Epoch 0: Loss = 0.9482960661031181
Epoch 1000: Loss = 0.03150777068112321
Epoch 2000: Loss = 0.025302497869819973
Epoch 3000: Loss = 0.029117866840111088
Epoch 4000: Loss = 0.023997349598860584
Epoch 5000: Loss = 0.023686354572419965
Epoch 6000: Loss = 0.02314861521639253
Epoch 7000: Loss = 0.02495479201681959
Epoch 8000: Loss = 0.02313572429114692
Epoch 9000: Loss = 0.02309535873492234
Test Accuracy: 97.00%

Process finished with exit code 0
```

Figure 5. Loss and accuracy of the tanh activation function

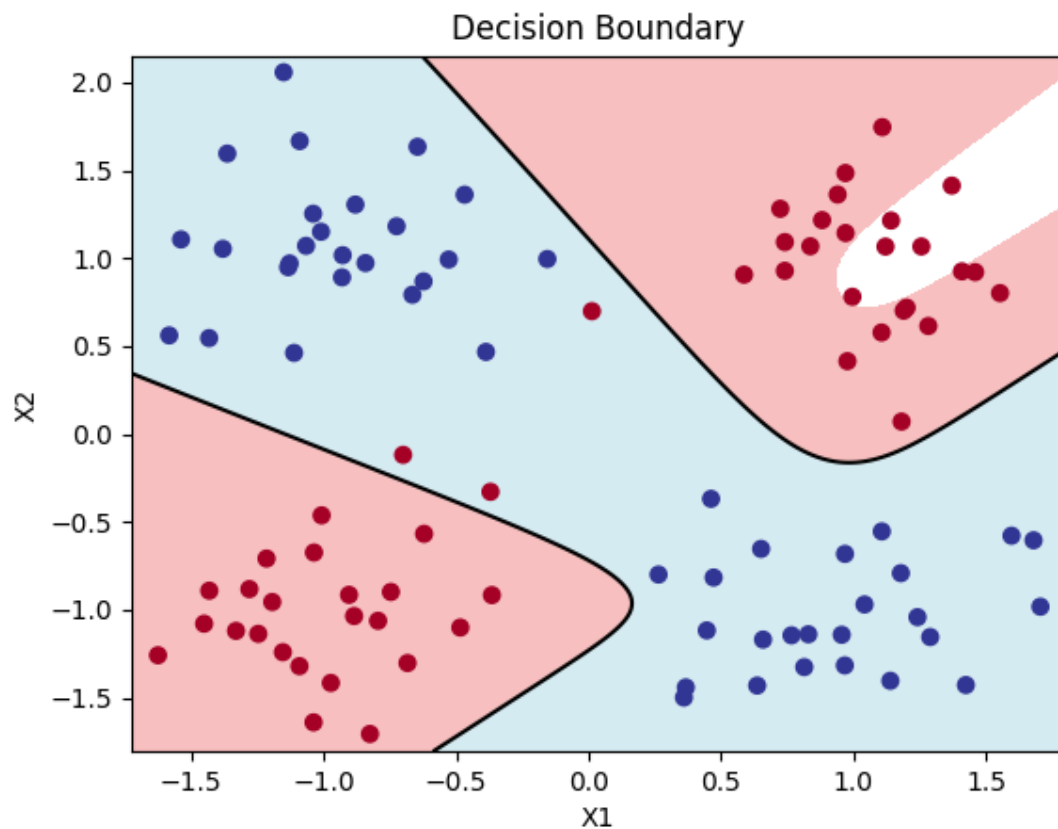
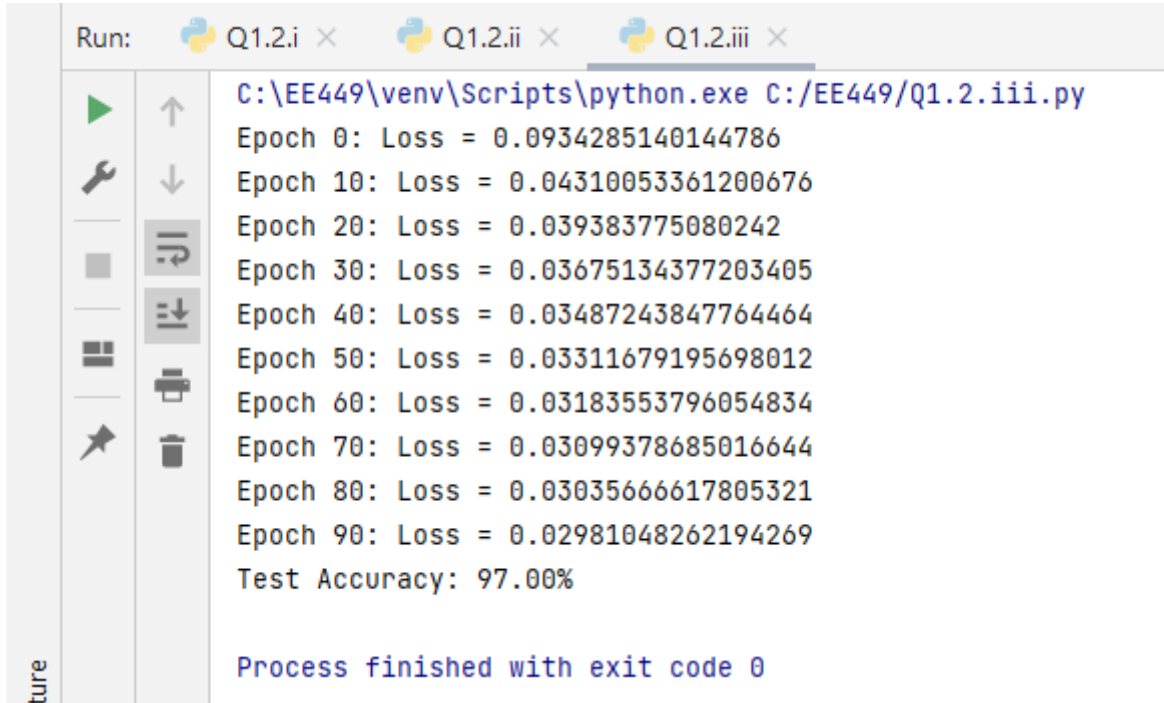


Figure 6. Decision boundary of tanh activation function

iii. ReLU:



```
Run: Q1.2.i x Q1.2.ii x Q1.2.iii x
C:\EE449\venv\Scripts\python.exe C:/EE449/Q1.2.iii.py
Epoch 0: Loss = 0.0934285140144786
Epoch 10: Loss = 0.04310053361200676
Epoch 20: Loss = 0.039383775080242
Epoch 30: Loss = 0.03675134377203405
Epoch 40: Loss = 0.03487243847764464
Epoch 50: Loss = 0.03311679195698012
Epoch 60: Loss = 0.03183553796054834
Epoch 70: Loss = 0.03099378685016644
Epoch 80: Loss = 0.03035666617805321
Epoch 90: Loss = 0.02981048262194269
Test Accuracy: 97.00%

Process finished with exit code 0
```

Figure 7. Loss and accuracy of the ReLU activation function

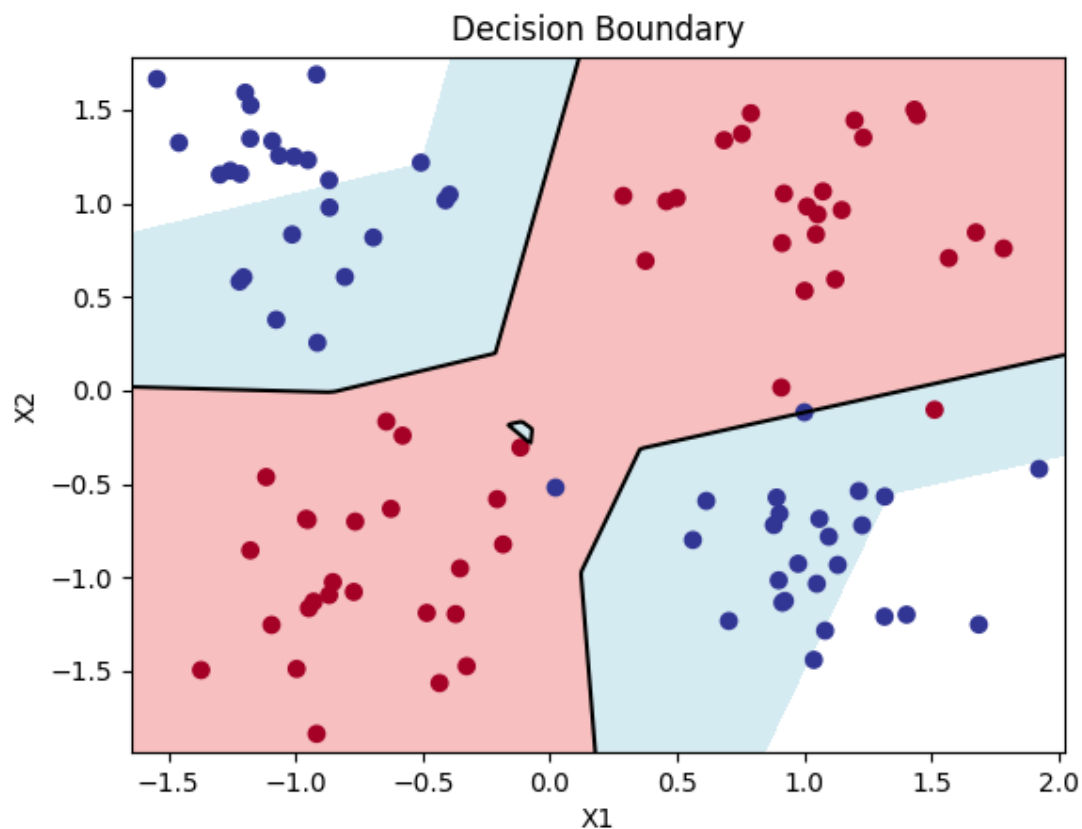


Figure 8. Decision boundary of ReLU activation function

1.3

1.

- **Sigmoid:**

The sigmoid function outputs values between 0 and 1, which is especially useful for representing probabilities. Because it is smooth and differentiable, it can be very effective for binary classification tasks. In our experiments, using sigmoid resulted in about 99% test accuracy. However, a significant drawback is the vanishing gradient problem: when the input values are too high or too low, the derivative approaches zero, which can severely slow down learning. However, it has a smooth curve so that it performs very well for fine tuning. To conclude, since it is prone to represent binary values, it performed very well for the XOR problem. We can use the sigmoid function at the last layer of the model for representing probabilities. By this way we can escape from the vanishing gradient problem which causes significant problems in hidden layers with large gradients.

- **Tanh:**

The tanh function is similar in shape to the sigmoid function but outputs values in

the range between -1 and 1. Because it's centered around zero, updates to the weights can be more balanced, which often leads to faster learning than with sigmoid. This is different than sigmoid which is strictly positive. In our trials, tanh reached around 97% accuracy—close to the performance of sigmoid. Since tanh is more steep than sigmoid, gradients are greater than sigmoid. Which makes the learning faster while decreasing the ability to fine tuning. So that, we have reached to 97% accuracy which is quite acceptable. Also, as we can see from the Figure 3 and the Figure 5 it is faster to train the model with the tanh function. Meanwhile tanh can still encounter vanishing gradients. With its zero-centered output and easily computable gradient tanh is frequently used in recurrent neural networks.

- **ReLU:**

ReLU (Rectified Linear Unit) is computationally simpler since it outputs the input directly for positive values (with a constant gradient of 1) and zero for negative values. It goes to infinity for positive x values. This characteristic largely prevents the vanishing gradient problem, allowing for faster training in deep networks. However, ReLU's output is unbounded, and its large gradients can be problematic when the task requires outputs to remain within a limited range such as probabilities in binary classification. In our XOR experiments, ReLU led to 97% test accuracy. However, it is not stable because the vanishing gradient problem is significantly affecting the ReLU since it gives directly 0 for negative x values. This phenomenon vanishes the learning and for some random starting points and training data the model can stuck and stop learning which significantly decreases the accuracy. This suggests that while ReLU is excellent for deep architectures like CNNs, it may not be ideal for problems like XOR without careful tuning such as adjusting the learning rate.

2.

The XOR problem is a classic example of a binary classification task where there are two inputs and one output. In Boolean logic, the XOR (exclusive OR) function outputs 0 when both inputs are the same and 1 when they are different. In our experiments, even though the inputs are continuous, the network is still expected to produce an output of 1 when the inputs have opposite signs (one positive and one negative) and 0 when they share the same sign.

The challenge with the XOR problem is that its four regions (corresponding to the combinations of positive and negative inputs) are not separable by a single straight line. In other words, the decision boundary needed to correctly classify XOR cannot be linear. A single-layer perceptron is limited to learning linear boundaries, which makes it incapable of solving XOR since it only creates two distinct regions.

To overcome this, a multi-layer perceptron (MLP) is required. An MLP introduces one or more hidden layers with non-linear activation functions. These hidden layers transform the input space into a new representation where the previously non-linearly separable regions can become linearly separable. In that transformed space, the network can then correctly classify the different regions, effectively solving the XOR problem.

3.

Of course, for each iteration I obtained a new decision boundary and consequently new accuracy values. This fact stems from the randomness of the training data. This training set drives the model to different final points for each iteration. Also, our first weight initializations are random so that we start to training from different points which also affects the randomness of the overall process. This is quite observable at ReLU since it might get into the negative x regions which kills the gradient and the learning. The accuracy variations between iterations can exceed %45 for ReLU. To conclude, with random initial weights and training data sets we might obtain different local minima for the error minimization problem.

2.

2.1



Figure 9.CNN output digit “7”

2.2

1.

Convolutional Neural Networks (CNNs) are fundamentally important in image processing due to their ability to efficiently manage high-dimensional data while capturing spatial specifications. Traditional fully connected networks would require an overwhelming number of parameters when applied to images, as every pixel and its color channels would need individual attention. In contrast, CNNs use small convolutional filters that slide across the image and share weights, greatly reducing the number of parameters and computational load. This weight-sharing mechanism enables the network to effectively learn local features such as edges, textures, and shapes, which are then combined in deeper layers to form more abstract representations. Additionally, pooling layers further reduce dimensionality and provide invariance to minor shifts and distortions. As a result, CNNs are highly scalable and have become the standard choice for complex image-related tasks like classification, detection, and segmentation, where capturing intricate patterns and spatial relationships is essential. To conclude, we can

extract very good observations with less computationally intensive models especially for image processing.

2.

A kernel in a convolutional layer is a small matrix of weights that is used to extract local features from an input image through the convolution operation. It slides over the image, computing element-wise multiplications between the kernel and overlapping regions of the image and then sums the results to produce a single value in the output feature map. The size of the kernel—typically given by its height and width—determines the area of the input that is examined at one time, effectively setting the receptive field of the filter. Additionally, the depth of the kernel matches the number of input channels (such as three channels for an RGB image), ensuring that all aspects of the input are considered. So it might be 3 dimensional. Multiple kernels in a convolutional layer allow the network to detect various features, with each kernel generating its own feature map that highlights specific patterns or structures in the input like colour shape and so on. To conclude, they can be considered as shared weights.

3.

In this part, a 2D convolution operation was implemented using NumPy, and applied to a set of grayscale images representing the digit "7". The output image shows the result of passing these images through a convolutional layer with a predefined kernel. As we learned in lectures convolutional layers are effective at extracting important features such as edges, textures, or patterns from images by sliding a kernel across the input and computing weighted sums. In the output visualization, each row corresponds to a different input image (batch size is 5), and each column represents the response of that image to a specific filter. We have 8 different kernels totally. Although the input digits are visually similar, the filters highlight different features of them, which causes the outputs in the same row to look different. On the other hand, images in the same column appear visually similar because the same filter is applied across different inputs. This clearly demonstrates how convolutional layers reduce the dimensionality and complexity of the image while preserving essential features. Overall, the output confirms that the convolutional layer successfully extracted meaningful representations from the input digits.

4.

The numbers in the same column look similar to each other because they are the result of applying the same filter (kernel) to different input images. As explained in Lecture 4 and seen during the convolution experiments, each filter is designed to detect a specific feature such as an edge, a corner, or a certain texture. Since the same filter is used across all input images in that column, it consistently extracts the same type of feature from each image. Even though the input images might have slight variations, the filter focuses

only on the feature it is designed to capture. Therefore, the outputs appear visually similar because the extracted features are similar, despite the inputs being different. This demonstrates one of the core ideas of convolutional neural networks (CNNs): weight sharing across different parts and samples, which allows the model to detect the same pattern regardless of its location or specific input instance.

5.

The numbers in the same row appear different from each other because they are produced by applying different filters to the same input image. Each filter is specialized to extract a specific type of feature, such as horizontal edges, vertical edges, corners, or texture patterns. So even though the input image is the same across the row, each filter reacts to different aspects of that image, resulting in visually distinct outputs. One filter might emphasize the top bar of the digit "7", while another might highlight the diagonal stroke. This behaviour is a key property of convolutional neural networks (CNNs), as discussed in Lecture 4, where multiple filters learn and extract various features from the input data. This diversity in feature extraction allows CNNs to represent the same input in multiple ways and helps improve their ability to learn complex patterns.

6.

From the previous questions, we can deduce that convolutional layers are highly effective at extracting diverse and meaningful features from input images. As seen in Question 4, applying the same filter to different images results in visually similar outputs, which shows that each filter is specialized in detecting a particular type of feature regardless of the input image. On the other hand, as discussed in Question 5, using different filters on the same image produces different outputs, highlighting various parts or patterns within the same digit. This proves that convolutional layers enable a neural network to analyze the same data from multiple perspectives. It also shows the advantage of weight sharing, which reduces the number of parameters while still allowing the network to generalize well across different inputs. Overall, convolutional layers allow neural networks to learn rich, multi-level representations of input data, which is why they are widely used in image processing tasks.

3.

3.1

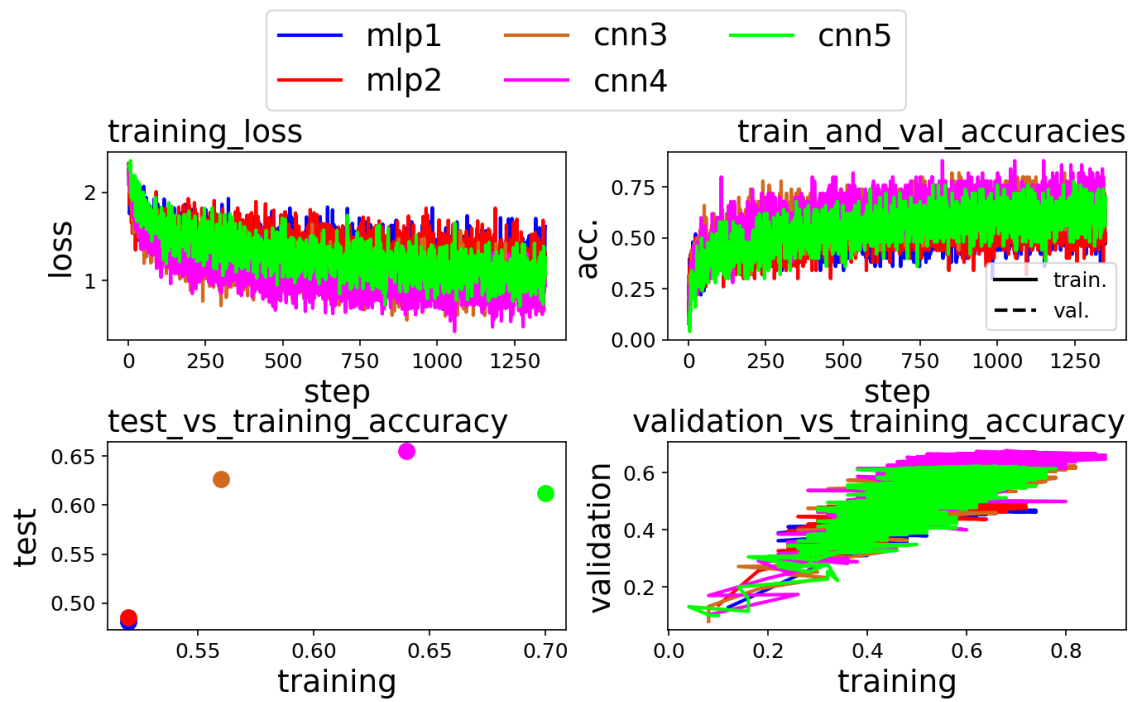


Figure 10. Performance Comparison Plots for part 3

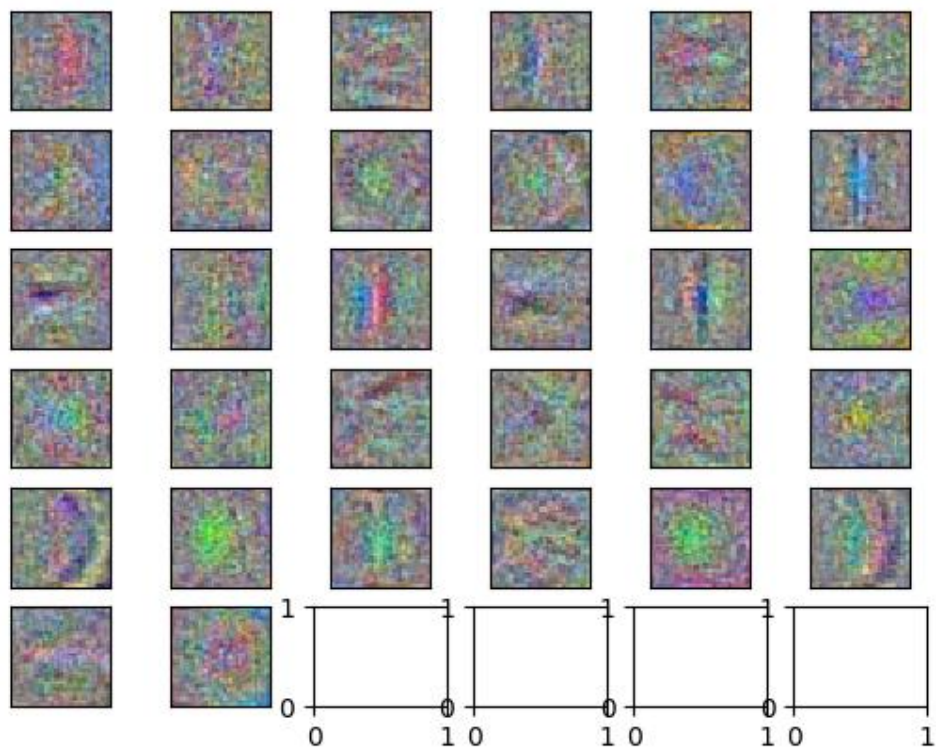


Figure 11. First layer weights for mlp1

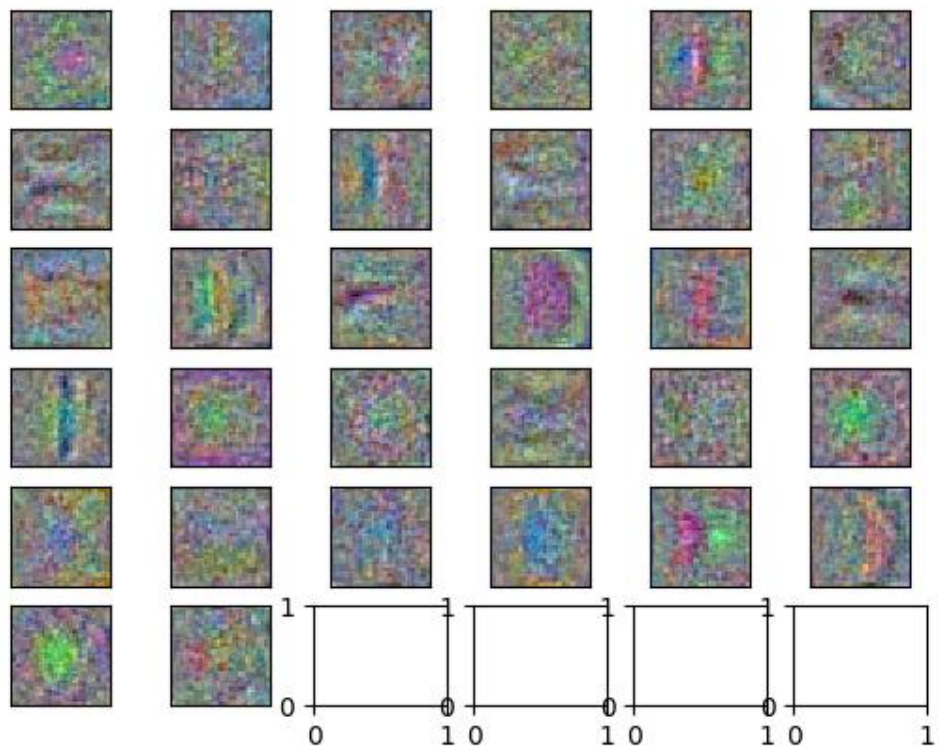


Figure 12. First layer weights for mlp2

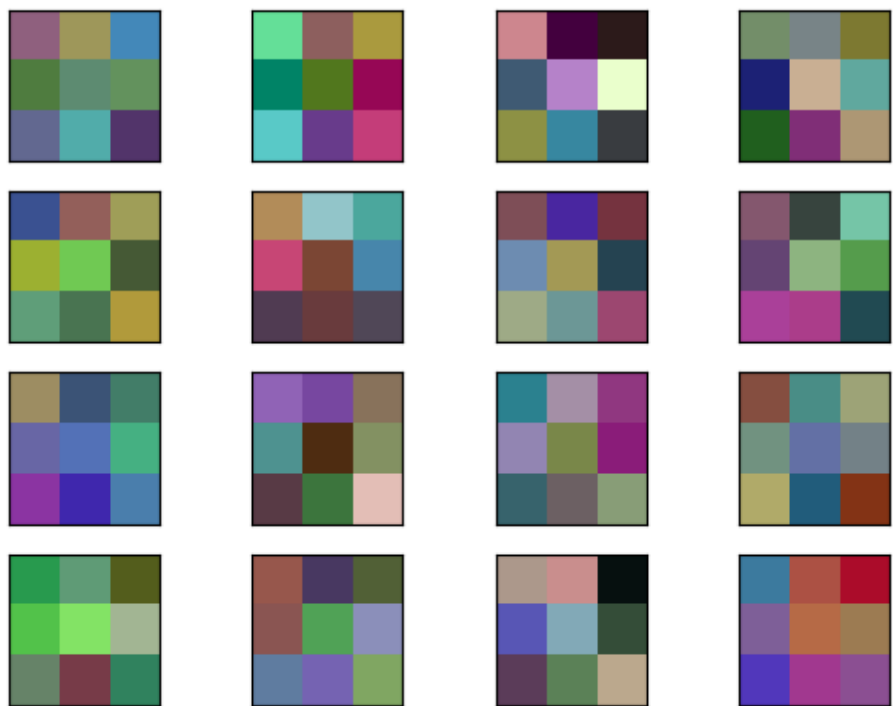


Figure 13. First layer weights for cnn3

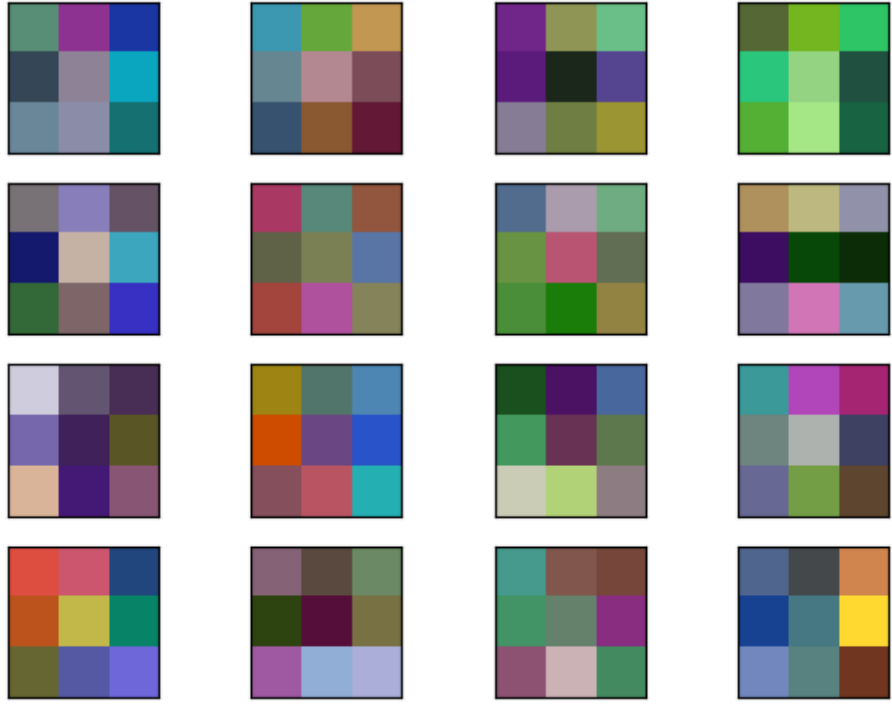


Figure 14. First layer weights for cnn4

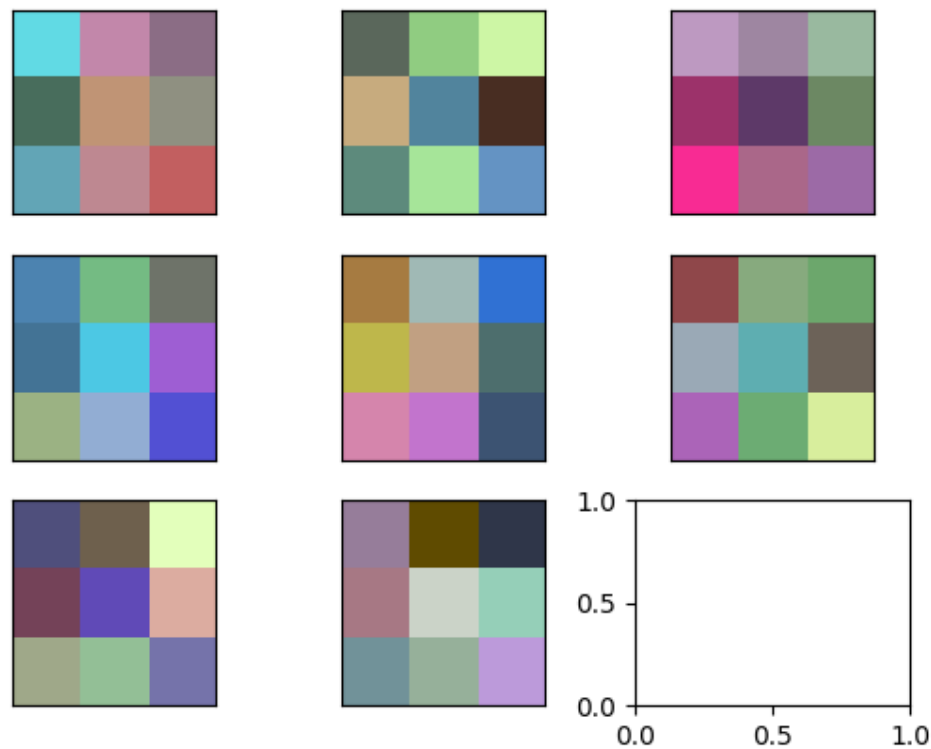


Figure 15. First layer weights for cnn5

3.2

1.

Generalization performance refers to a model's ability to perform well not only on training data but also on unseen data, such as validation or test sets. It is a crucial indicator of whether the model has learned meaningful patterns or simply memorized the training samples.

2.

To evaluate generalization performance, the most informative plots are the `train_and_val_accuracies`, `test_vs_training_accuracy`, and `validation_vs_training_accuracy` plots. The `train_and_val_accuracies` plot helps us observe how the validation accuracy tracks the training accuracy during learning. If both curves increase together and remain close, it indicates good generalization. On the other hand, if validation accuracy stagnates or drops while training accuracy keeps increasing, it suggests overfitting. The `test_vs_training_accuracy` plot directly compares the final test accuracy to the final training accuracy of each model, showing whether the model's performance on unseen data is consistent with its performance on training data. A large

gap would signal poor generalization. Lastly, the *validation_vs_training_accuracy* plot shows how validation accuracy correlates with training accuracy during the training process. A strong diagonal pattern (like in CNN models) indicates good generalization throughout training, whereas scattered points or plateaus can imply underfitting or overfitting. Together, these plots provide a clear and reliable picture of how well a model generalizes beyond the training data.

3.

When comparing the generalization performance of the architectures, it is clear from the plots that CNN-based models (cnn3, cnn4, and cnn5) outperform the MLP-based models (mlp1 and mlp2). Among all, cnn4 achieves the best generalization, as it shows the highest test and validation accuracies while maintaining low training loss. In the *test_vs_training_accuracy* plot, cnn4 is positioned highest, indicating it generalized better to unseen data. It also has a smooth and closely aligned training and validation curve in the *train_and_val_accuracies* plot, showing minimal overfitting. cnn3 also performs quite well, with consistent training and validation behavior, although slightly lower than cnn4. In contrast, mlp1 and mlp2 have noticeably lower test and validation accuracies despite their training accuracies increasing. This indicates that they struggled to learn complex visual features and generalized poorly. One other note is about cnn5. Even it performed better for training set, it performed poor performance for test set because it is overfitting. So that the most optimal model is cnn4. Overall, CNN architectures, due to their ability to extract spatial features, demonstrated superior generalization compared to the fully connected MLPs in this image classification task.

4.

The number of parameters in a model plays a crucial role in both classification accuracy and generalization performance. A model with too few parameters may not have enough capacity to learn the complex patterns in the data, leading to underfitting, where both training and test accuracies remain low. This is observed in mlp1, which has a very shallow architecture and the lowest performance across all metrics. On the other hand, a model with too many parameters may memorize the training data, resulting in overfitting — high training accuracy but poor generalization to unseen data. However, in our experiments, cnn4 which has relatively more parameters and deeper structures, achieved both high training and test accuracies. To conclude, simply increasing the number of parameters does not always improve performance indicating that larger models may lead to unnecessary complexity and even worse generalization. Therefore, it is important to find an optimal number of parameters that balances learning capacity and generalization, while also reducing computational cost. Using architectures that are too large not only increases training time but also consumes more memory and energy without guaranteeing better performance.

5.

The depth of the architecture directly affects a model's ability to learn complex features from the data. In general, deeper models have more layers and non-linearities, which allow them to extract higher-level features and perform better on tasks like image classification. In our experiments, shallow architectures like `mlp1` and `mlp2` showed limited performance due to their inability to capture complex spatial patterns, leading to lower classification accuracy and weak generalization with even higher number of parameters. On the other hand, deeper models like `cnn3` and `cnn4` performed significantly better, with `cnn4` showing the best generalization and test accuracy, thanks to its deeper structure and multiple convolutional layers. However, increasing depth beyond a certain point does not always guarantee better performance. For example, `cnn5` is deeper than `cnn4` but showed worse generalization, likely due to overfitting or redundant layers. One important point is that `cnn5` is overfitted with even a smaller number of parameters. This means that this drawback stems from the large number of layers' existence in `cnn5`. This demonstrates that while depth improves learning capacity, it must be carefully balanced to avoid over-complexity and to maintain both efficient computation and strong generalization.

6.

When we look at the visualizations of the first-layer weights, we observe a clear difference in interpretability between MLP and CNN models. For MLP architectures like `mlp1` and `mlp2`, the weights appear as colorful noise. This is because each neuron in an MLP is connected to every pixel of the image, making the weights intensely located and difficult to interpret visually. On the other hand, CNN models like `cnn3`, `cnn4`, and `cnn5` have filters in their first convolutional layer that operate on small local regions of the image. In their weight visualizations, we can see more structured patterns. Overall, CNN weights are more interpretable than those of MLPs, but excessive depth or parameter size can reduce clarity in visual representations.

7.

Yes, in general, we can say that some units in the network are specialized to specific classes, especially in the output layer. Each neuron in the output layer corresponds to one of the 10 classes in the CIFAR-10 dataset. During training, these neurons are optimized to respond more strongly when the input image belongs to their associated class. For example, the neuron responsible for classifying "airplane" will become more active when the network sees an image of an airplane. In deeper CNN architectures like `cnn3`, `cnn4`, and `cnn5`, some filters in earlier layers may also become sensitive to class-specific patterns such as textures or shapes commonly found in certain classes. However, this type of specialization is more evident in the final layers, where the network combines learned features to make class-level decisions. While we cannot visually

confirm that individual filters are tied to specific classes just by looking at the weight visualizations, the structured behaviour observed in the CNNs' performance suggests that unit specialization emerges as a natural result of training on labeled data.

8.

Among all the architectures, the CNN architectures—particularly `cnn3` and `cnn4`—have the most interpretable first-layer weights. In their visualizations, we can observe meaningful patterns such as color contrasts, edge-like filters, or texture gradients. These filters operate locally over the image, making them easier to understand as they often resemble visual features found in natural images. This is consistent with what we discussed in lecture: early layers in convolutional neural networks tend to act as feature extractors, similar to biological vision systems. On the other hand, the MLP architectures (`mlp1` and `mlp2`) have fully connected layers, where each neuron is linked to every pixel in the image. This results in highly complex and dense weight patterns that appear noisy and are hard to interpret visually. Additionally, while `cnn5` is also a CNN, its first-layer weights are more complex and less interpretable compared to `cnn3` and `cnn4`, possibly due to its increased depth and number of parameters. Therefore, `cnn3` and `cnn4` provide the most interpretable weights, especially in the first layer.

9.

The five architectures used in the experiment can be categorized into two main types: MLPs (`mlp1`, `mlp2`) and CNNs (`cnn3`, `cnn4`, `cnn5`). Within each category, there are pairs of architectures that are structurally similar. For example, `mlp1` and `mlp2` both follow a fully connected design, but `mlp2` adds an extra hidden layer and increases the number of units. Similarly, `cnn3` and `cnn4` share a common convolutional structure with multiple convolutional layers followed by pooling, although `cnn4` is deeper and more layered than `cnn3`. On the other hand, `cnn5` is structurally the most complex with the highest depth and number of convolutional layers.

When we compare similarly structured models, we observe that `cnn4` outperforms `cnn3`, which shows that increasing depth and properly stacking convolutional layers improves performance—as long as the model is not excessively complex. A similar comparison between `mlp1` and `mlp2` shows a slight improvement in accuracy, but both perform worse than any CNN model, indicating that fully connected layers alone are not sufficient for image data. When comparing architectures with different structures, the difference is even more apparent: CNNs perform significantly better than MLPs, thanks to their ability to capture spatial features through local receptive fields and weight sharing. However, `cnn5`, despite its structural complexity, underperforms compared to `cnn4`, suggesting that overcomplicating the architecture can hurt generalization. In conclusion, `cnn4` is a well-balanced and efficient design, and MLPs are generally inadequate for image classification tasks like CIFAR-10.

10.

For this classification task on CIFAR-10, I would pick cnn4 as the best architecture. Among all models tested, cnn4 achieved the highest test and validation accuracy, while also maintaining low training loss and stable learning throughout training. It offers a good balance between depth and performance. Unlike cnn5, which is deeper but doesn't perform as well, cnn4 avoids unnecessary complexity. Its filters in the first layer are also interpretable, showing that it learns meaningful visual features. On the other hand, the MLP models like mlp1 and mlp2 perform much worse because they can't capture spatial structure in image data effectively. Overall, cnn4 is the best choice because it generalizes well and performs accurately.

4.

4.1

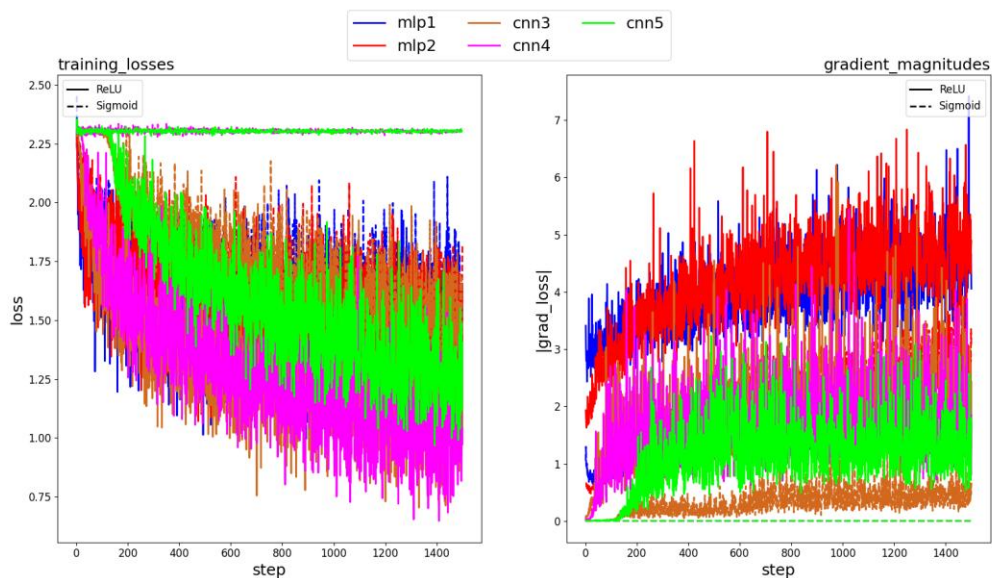


Figure 16. Performance comparison plots for part 4

4.2

1.

As shown in Figure 16, gradient behaviour varies between architectures. In MLP models, the gradient magnitudes are generally higher compared to CNNs. This is because MLPs are fully connected, meaning they contain many weights. In contrast, CNNs use weight sharing and local connections, which reduces the total number of weights. During backpropagation, gradients are accumulated over these weight connections as errors are propagated backwards. Since MLPs have many more connections, the resulting gradients are larger. Additionally, among the CNN models, we can observe that the gradient magnitudes decrease as the depth increases. The order from highest to lowest gradients is: cnn3, cnn4, and cnn5. This suggests that increasing depth in CNNs leads to smaller gradients in the earlier layers.

2.

This happens mainly because of how gradients are computed and passed backward through the layers. In deeper architectures, like cnn5, the gradients have to go through more layers during backpropagation. Each time they pass through an activation function or a weight layer, they get multiplied by values less than 1, which causes them to shrink. This is known as the vanishing gradient problem. Also, CNNs use fewer parameters due to weight sharing, so there are fewer gradient paths compared to fully connected networks like MLPs. That's why gradients are smaller in CNNs, and why deeper CNNs like cnn5 show lower gradient magnitudes in the first layer compared to shallower ones like cnn3.

3.

From Figure 16, we can see that gradients tend to vanish in deeper CNN architectures like cnn4 and cnn5 when the Sigmoid activation function is used instead of ReLU. A similar trend is also visible in shallower models, where gradients are generally smaller with Sigmoid compared to ReLU. This behavior is consistent with what we observed in the XOR problem from Part 1.2. In both cases, ReLU produces larger gradient values, which helps improve learning, while Sigmoid leads to smaller gradients and slower updates. This difference can be explained by the derivatives of the activation functions. The derivative of ReLU is either 0 or 1, and is unbounded in the positive direction, whereas the derivative of Sigmoid is always less than 0.25, and the function itself is bounded between 0 and 1. Because of this, in the XOR task, ReLU can result in large gradient values and unbounded neuron outputs. However, the XOR output is also bounded between 0 and 1, so when ReLU is used, it may not easily converge to the correct output range. Overall, the gradient behavior we observe in Part 4 with ReLU and Sigmoid is similar to the effect seen in the XOR problem.

4.

If we use inputs in the range $[0, 255]$ instead of $[0.0, 1.0]$, the training process would likely become unstable or much slower. This is because large input values can lead to large outputs in the early layers, which can cause very large (ReLU) or very small gradients during backpropagation due to saturation for sigmoid or tanh. In particular, activation functions like Sigmoid would saturate quickly when the input values are too high, meaning their outputs would be close to 0 or 1, and their gradients would be close to zero. This causes the vanishing gradient problem and slows down learning. Even ReLU can produce very large outputs, which may lead to exploding gradients or unstable weight updates. That's why input normalization, such as dividing by 255 or standardizing to zero mean and unit variance, is an important step before training deep neural networks. One other important point is for ReLU gradient might become completely 0. So training might be completely dead.

5.

5.1

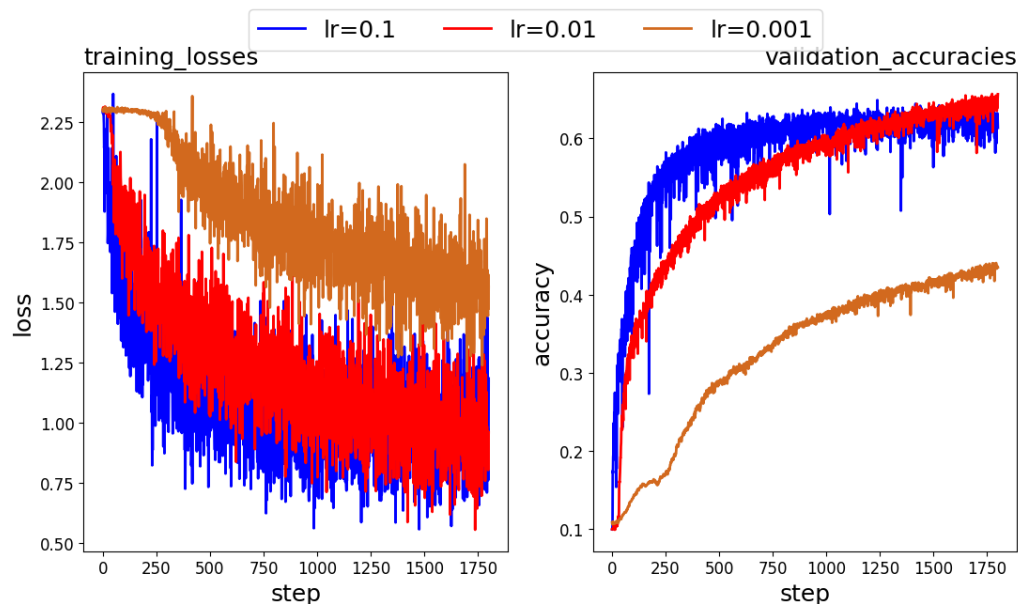


Figure 17. cnn4 with different learning rates

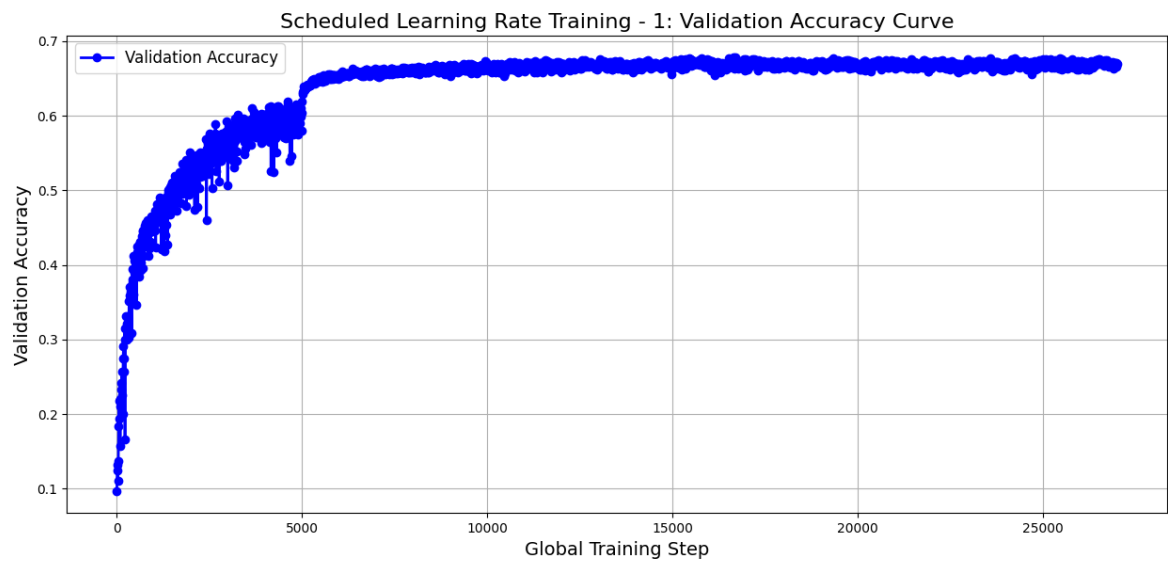


Figure 18. Scheduled learning rates with two different learning rates

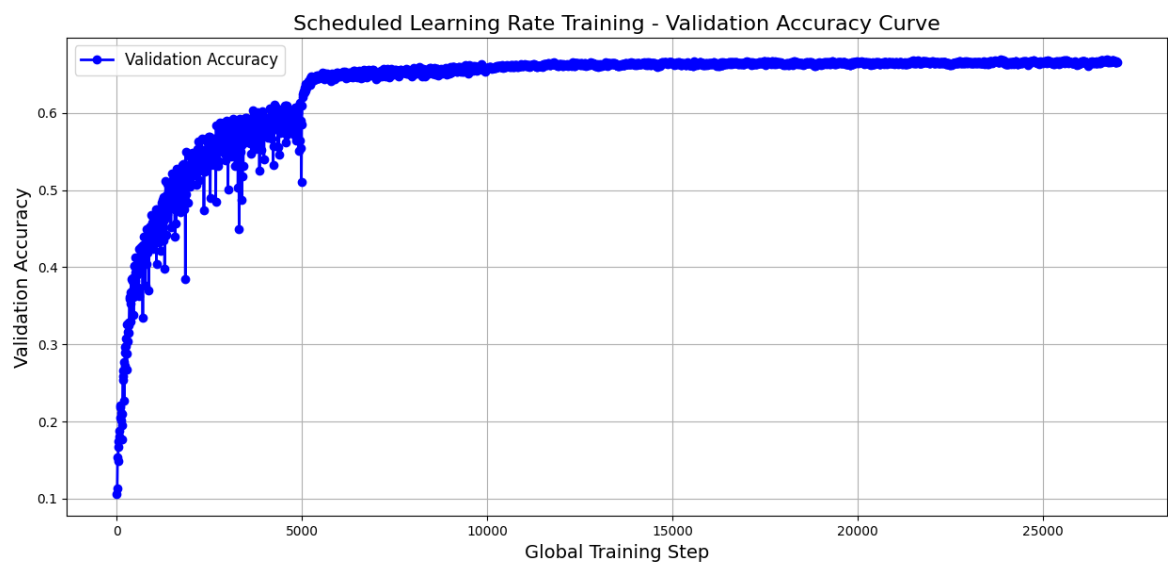


Figure 19. Scheduled learning rates with three different learning rates

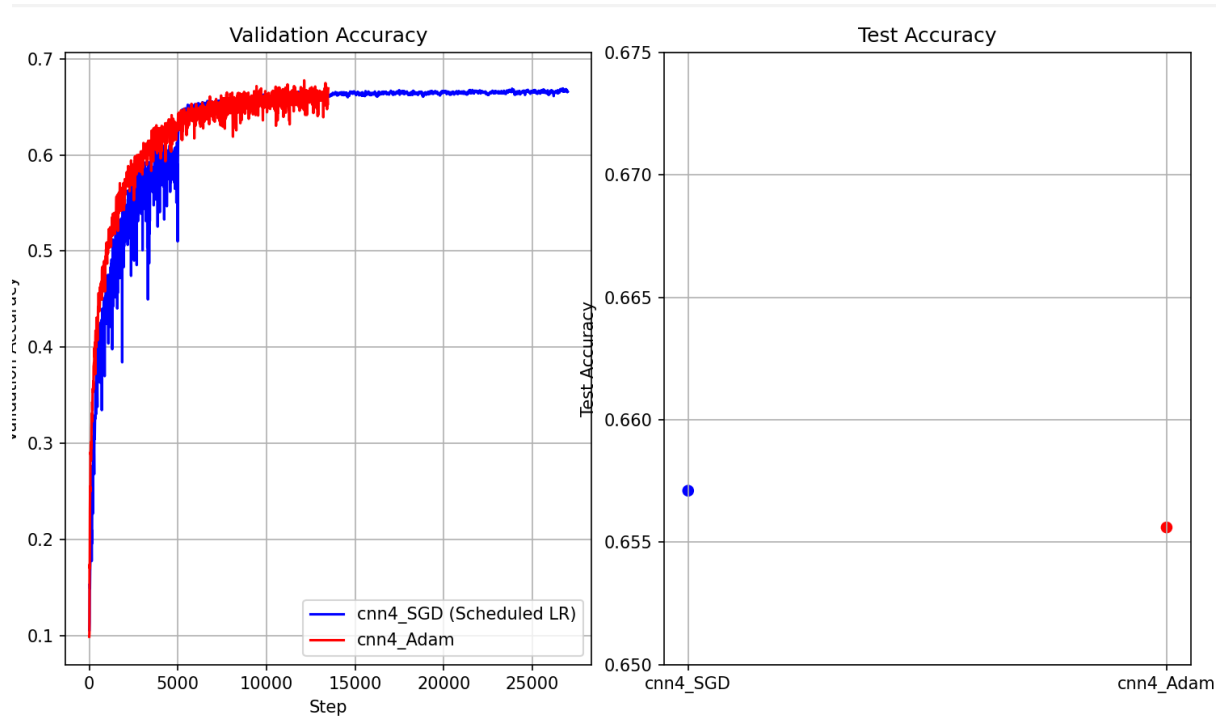


Figure 20. Comparison with scheduler and Adam

5.2

1.

The learning rate has a major effect on how fast and how well a model converges during training. A high learning rate allows the model to make larger updates, which speeds up learning in the early steps. However, if the learning rate is too high, it might overshoot the optimal point or cause unstable training. On the other hand, a low learning rate leads to smaller updates, making the learning slower but more stable. In the scheduled learning rate setup, we first use a high learning rate (0.1) to quickly reduce the loss, then gradually lower it to 0.01 and 0.001 to fine-tune the model. This helps the model converge faster and more smoothly.

As shown in Figure 21, different fixed learning rates clearly show their impact. The model trained with $lr=0.1$ converges the fastest and reaches the highest validation accuracy early. $lr=0.01$ is slower but still performs well. However, $lr=0.001$ leads to very slow convergence and much lower accuracy. This demonstrates that while smaller learning rates offer stable training, they may not reach high performance within a limited number of steps. Therefore, using a large learning rate in the beginning and reducing it over time — as done in scheduled learning rate — is an effective strategy to combine fast convergence with stability and high final accuracy.

2.

The learning rate not only affects how fast the model learns but also where it ends up after training. If the learning rate is too high, the model might skip over good solutions and fail to settle at an optimal point. This can cause unstable updates or bouncing around the loss surface, leading to poor final performance. On the other hand, if the learning rate is too low, the model may get stuck in a local minimum or a flat region and take a very long time to improve. As shown in Figure 21, the model trained with $lr=0.1$ reaches a better final accuracy than the one trained with $lr=0.001$, which converges to a worse point. A well-tuned or scheduled learning rate allows the model to first explore quickly and then gradually settle into a better solution. This helps the model not only converge faster but also reach a higher-performing state in the end.

3.

Yes, the scheduled learning rate method works well in this case. From the validation accuracy curves, we can clearly see that the model improves rapidly at the beginning when the learning rate is high. Then, after the scheduled learning rate drops (first from 0.1 to 0.01, and then to 0.001), the learning becomes more stable and the accuracy gradually improves, reaching a plateau around 67 percent. The sharp increase early on shows fast convergence, and the smooth plateau later shows that the model is fine-tuning and avoiding overshooting. This balance between speed and stability is exactly what scheduled learning rate is designed for. So, it works both in terms of fast convergence and reaching a high final validation accuracy.

4.

As can be seen from the figure, both the scheduled learning rate (SGD with step decay) and Adam optimizer show strong performance in terms of convergence. Initially, Adam converges faster in the early steps due to its adaptive learning mechanism. However, after a certain point, the validation accuracy of the scheduled learning rate surpasses that of Adam. This is because the scheduled reduction in learning rate allows the model to fine-tune the weights more effectively as training progresses, preventing overshooting and enabling better convergence to the optimum. Regarding test accuracy, the scheduled learning rate method (SGD) slightly outperforms Adam, suggesting that it generalizes better in this case. Therefore, while Adam offers faster convergence, the scheduled learning rate achieves better final accuracy and generalization.

CODE

1.

Q1.1

```

import numpy as np
import matplotlib.pyplot as plt

# Define the range of x values
x = np.linspace(-5, 5, 400)

# Tanh function and its derivative
y_tanh = np.tanh(x)
dy_tanh = 1 - y_tanh**2

# Sigmoid function and its derivative
y_sigmoid = 1 / (1 + np.exp(-x))
dy_sigmoid = y_sigmoid * (1 - y_sigmoid)

# ReLU function and its derivative
y_relu = np.maximum(0, x)
dy_relu = np.where(x > 0, 1, 0)

# Plot settings: We'll use subplots to show each function and its derivative.
fig, axs = plt.subplots(3, 2, figsize=(12, 12))
fig.suptitle('Activation Functions and Their Derivatives', fontsize=16)

# Plot for Tanh
axs[0, 0].plot(x, y_tanh, label='tanh(x)')
axs[0, 0].set_title('Tanh Function')
axs[0, 0].grid(True)
axs[0, 0].legend()

axs[0, 1].plot(x, dy_tanh, label="d/dx tanh(x)", color='orange')
axs[0, 1].set_title('Derivative of Tanh')
axs[0, 1].grid(True)
axs[0, 1].legend()

# Plot for Sigmoid
axs[1, 0].plot(x, y_sigmoid, label='sigmoid(x)')
axs[1, 0].set_title('Sigmoid Function')
axs[1, 0].grid(True)
axs[1, 0].legend()

axs[1, 1].plot(x, dy_sigmoid, label="d/dx sigmoid(x)", color='orange')
axs[1, 1].set_title('Derivative of Sigmoid')
axs[1, 1].grid(True)
axs[1, 1].legend()

# Plot for ReLU
axs[2, 0].plot(x, y_relu, label='ReLU(x)')
axs[2, 0].set_title('ReLU Function')
axs[2, 0].grid(True)
axs[2, 0].legend()

```

```

axs[2, 1].plot(x, dy_relu, label="d/dx ReLU(x)", color='orange')
axs[2, 1].set_title('Derivative of ReLU')
axs[2, 1].grid(True)
axs[2, 1].legend()

fig.subplots_adjust(hspace=0.320, top=0.90, bottom=0.048, left=0.125, right=0.9)
plt.show()

```

Q1.2.i

```

import numpy as np
from matplotlib import pyplot as plt
from utils import part1CreateDataset, part1PlotBoundary

# Define the MLP class with one hidden layer
class MLP:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights and biases with random values (Gaussian initialization)
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
        self.bias_output = np.zeros((1, self.output_size))

        # Sigmoid activation function
        def sigmoid(self, x):
            return 1 / (1 + np.exp(-x))

        # Derivative of sigmoid (assuming x is the activated output)
        def sigmoid_derivative(self, x):
            return x * (1 - x)

        # Forward pass through the network
        def forward(self, inputs):
            # Hidden layer: z = X * W_input_hidden + bias_hidden, then apply sigmoid activation
            z_hidden = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
            self.hidden_output = self.sigmoid(z_hidden)

            # Output layer: z = hidden_output * W_hidden_output + bias_output, then apply sigmoid
            z_output = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
            self.output = self.sigmoid(z_output)

            return self.output

        def backward(self, inputs, targets, learning_rate):

```

```

# Derivative of  $(y - t)^2$  is  $2 * (t - y)$ 
output_error = 2 * (targets - self.output)
output_delta = output_error * self.sigmoid_derivative(self.output)

hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)

self.weights_hidden_output += learning_rate * np.dot(self.hidden_output.T, output_delta)
self.bias_output += learning_rate * np.sum(output_delta, axis=0, keepdims=True)

self.weights_input_hidden += learning_rate * np.dot(inputs.T, hidden_delta)
self.bias_hidden += learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)

# -----
# Create the XOR dataset using the utility function from utils.py
x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)

# Define neural network parameters
input_size = 2 # XOR has 2 input features
hidden_size = 4 # Hidden layer size
output_size = 1 # Single output neuron for binary classification
learning_rate = 0.001

# Create the neural network instance
nn = MLP(input_size, hidden_size, output_size)

# Train the neural network
epochs = 10000
for epoch in range(epochs):
    # Forward propagation
    output = nn.forward(x_train)

    # Backpropagation and update weights
    nn.backward(x_train, y_train, learning_rate)

    # Print the loss (Mean Squared Error) every 1000 epochs
    if epoch % 1000 == 0:
        loss = np.mean((output - y_train) ** 2)
        print(f"Epoch {epoch}: Loss = {loss}")

# Test the trained neural network on validation data
y_predict = nn.forward(x_val)
# Convert continuous outputs to binary predictions using a threshold of 0.5
y_predict_binary = (y_predict >= 0.5).astype(int)
accuracy = np.mean(y_predict_binary == y_val) * 100
print(f"Test Accuracy: {accuracy:.2f}%")

# Plot the final decision boundary using the utility function from utils.py

```

```
part1PlotBoundary(x_val, y_val, nn)
plt.show()
```

Q1.2.ii

```
import numpy as np
from matplotlib import pyplot as plt
from utils import part1CreateDataset, part1PlotBoundary
```

```
# Define the MLP class with one hidden layer
```

```
class MLP:
```

```
    def __init__(self, input_size, hidden_size, output_size):
```

```
        self.input_size = input_size
```

```
        self.hidden_size = hidden_size
```

```
        self.output_size = output_size
```

```
        # Initialize weights and biases with random values (Gaussian initialization)
```

```
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
```

```
        self.bias_hidden = np.zeros((1, self.hidden_size))
```

```
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
```

```
        self.bias_output = np.zeros((1, self.output_size))
```

```
    # Tanh activation function
```

```
    def tanh(self, x):
```

```
        return np.tanh(x)
```

```
    # Derivative of tanh (assuming x is the activated output)
```

```
    def tanh_derivative(self, x):
```

```
        return 1 - x ** 2
```

```
    # Forward pass through the network
```

```
    def forward(self, inputs):
```

```
        # Hidden layer: z = X * W_input_hidden + bias_hidden, then apply sigmoid activation
```

```
        z_hidden = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
```

```
        self.hidden_output = self.tanh(z_hidden)
```

```
        # Output layer: z = hidden_output * W_hidden_output + bias_output, then apply sigmoid
```

```
        z_output = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
```

```
        self.output = self.tanh(z_output)
```

```
        return self.output
```

```
    def backward(self, inputs, targets, learning_rate):
```

```
        # Derivative of (y - t)^2 is 2 * (t - y)
```

```
        output_error = 2 * (targets - self.output)
```

```
        output_delta = output_error * self.tanh_derivative(self.output)
```

```
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
```

```
        hidden_delta = hidden_error * self.tanh_derivative(self.hidden_output)
```

```

self.weights_hidden_output += learning_rate * np.dot(self.hidden_output.T, output_delta)
self.bias_output += learning_rate * np.sum(output_delta, axis=0, keepdims=True)

self.weights_input_hidden += learning_rate * np.dot(inputs.T, hidden_delta)
self.bias_hidden += learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)

# -----
# Create the XOR dataset using the utility function from utils.py
x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)

# Define neural network parameters
input_size = 2 # XOR has 2 input features
hidden_size = 4 # Hidden layer size
output_size = 1 # Single output neuron for binary classification
learning_rate = 0.001

# Create the neural network instance
nn = MLP(input_size, hidden_size, output_size)

# Train the neural network
epochs = 10000
for epoch in range(epochs):
    # Forward propagation
    output = nn.forward(x_train)

    # Backpropagation and update weights
    nn.backward(x_train, y_train, learning_rate)

    # Print the loss (Mean Squared Error) every 1000 epochs
    if epoch % 1000 == 0:
        loss = np.mean((output - y_train) ** 2)
        print(f"Epoch {epoch}: Loss = {loss}")

# Test the trained neural network on validation data
y_predict = nn.forward(x_val)
# Convert continuous outputs to binary predictions using a threshold of 0.5
y_predict_binary = (y_predict >= 0.5).astype(int)
accuracy = np.mean(y_predict_binary == y_val) * 100
print(f"Test Accuracy: {accuracy:.2f}%")

# Plot the final decision boundary using the utility function from utils.py
part1PlotBoundary(x_val, y_val, nn)
plt.show()

```

Q.1.2.iii

```

import numpy as np
from matplotlib import pyplot as plt

```

```
from utils import part1CreateDataset, part1PlotBoundary
```

```
# Define the MLP class with one hidden layer
```

```
class MLP:
```

```
    def __init__(self, input_size, hidden_size, output_size):
```

```
        self.input_size = input_size
```

```
        self.hidden_size = hidden_size
```

```
        self.output_size = output_size
```

```
        # Initialize weights and biases with random values (Gaussian initialization)
```

```
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
```

```
        self.bias_hidden = np.zeros((1, self.hidden_size))
```

```
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
```

```
        self.bias_output = np.zeros((1, self.output_size))
```

```
    # ReLU activation function
```

```
    def relu(self, x):
```

```
        return np.maximum(0, x)
```

```
    # Derivative of ReLU (assuming x is the pre-activation input)
```

```
    def relu_derivative(self, x):
```

```
        return (x > 0).astype(float)
```

```
    # Forward pass through the network
```

```
    def forward(self, inputs):
```

```
        # Hidden layer: z = X * W_input_hidden + bias_hidden, then apply sigmoid activation
```

```
        z_hidden = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
```

```
        self.hidden_output = self.relu(z_hidden)
```

```
        # Output layer: z = hidden_output * W_hidden_output + bias_output, then apply sigmoid
```

```
        z_output = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
```

```
        self.output = self.relu(z_output)
```

```
        return self.output
```

```
    def backward(self, inputs, targets, learning_rate):
```

```
        # Derivative of (y - t)^2 is 2 * (t - y)
```

```
        output_error = 2 * (targets - self.output)
```

```
        output_delta = output_error * self.relu_derivative(self.output)
```

```
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
```

```
        hidden_delta = hidden_error * self.relu_derivative(self.hidden_output)
```

```
        self.weights_hidden_output += learning_rate * np.dot(self.hidden_output.T, output_delta)
```

```
        self.bias_output += learning_rate * np.sum(output_delta, axis=0, keepdims=True)
```

```
        self.weights_input_hidden += learning_rate * np.dot(inputs.T, hidden_delta)
```

```
        self.bias_hidden += learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)
```

```

# -----
# Create the XOR dataset using the utility function from utils.py
x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)

# Define neural network parameters
input_size = 2 # XOR has 2 input features
hidden_size = 4 # Hidden layer size
output_size = 1 # Single output neuron for binary classification
learning_rate = 0.001

# Create the neural network instance
nn = MLP(input_size, hidden_size, output_size)

# Train the neural network
epochs = 100
for epoch in range(epochs):
    for i in range(x_train.shape[0]):
        # Get a single training sample (and keep dimensions as 2D arrays)
        xi = x_train[i:i + 1]
        yi = y_train[i:i + 1]

        # Forward pass for the single sample
        output = nn.forward(xi)

        # Backward pass: update weights based on the single sample's error
        nn.backward(xi, yi, learning_rate)

    # Optionally, calculate and print loss on the full training set every 1000 epochs
    if epoch % 10 == 0:
        full_output = nn.forward(x_train)
        loss = np.mean((full_output - y_train) ** 2)
        print(f"Epoch {epoch}: Loss = {loss}")

# Test the trained neural network on validation data
y_predict = nn.forward(x_val)
# Convert continuous outputs to binary predictions using a threshold of 0.5
y_predict_binary = (y_predict >= 0.5).astype(int)
accuracy = np.mean(y_predict_binary == y_val) * 100
print(f"Test Accuracy: {accuracy:.2f}%")

# Plot the final decision boundary using the utility function from utils.py
part1PlotBoundary(x_val, y_val, nn)
plt.show()

```

Q.2.1


```

import numpy as np
import matplotlib.pyplot as plt
from utils import part2Plots

def my_conv2d(input_data, kernel):

    # Unpack shapes
    batch_size, in_channels, in_height, in_width = input_data.shape
    out_channels, kernel_in_channels, filter_height, filter_width = kernel.shape

    # Sanity check
    assert in_channels == kernel_in_channels, (
        f"Input has {in_channels} channels, but kernel expects {kernel_in_channels}"
    )

    # Compute output spatial dimensions (no padding, stride=1)
    out_height = in_height - filter_height + 1
    out_width = in_width - filter_width + 1

    # Initialize the output array
    out = np.zeros((batch_size, out_channels, out_height, out_width), dtype=np.float32)

    # Perform the convolution
    for b in range(batch_size):
        for oc in range(out_channels):
            for i in range(out_height):
                for j in range(out_width):
                    # "Slice" the input region
                    region = input_data[b, :, i:i + filter_height, j:j + filter_width]
                    # Element-wise multiply with kernel and sum
                    out[b, oc, i, j] = np.sum(region * kernel[oc, :, :, :])

    return out

# 1. Load your data
input_data = np.load(r"C:\EE449\data\samples_7.npy")
kernel = np.load(r"C:\EE449\data\kernel.npy")

# 2. Run convolution
out = my_conv2d(input_data, kernel)

# 3. Save output if needed
np.save("out.npy", out)

# 4. Visualize output
part2Plots(out)
plt.show()

```

Q.3.1

```

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Subset
import numpy as np
import pickle
import os
from utils import part3Plots, visualizeWeights # Import the provided utility functions

# Set device (use GPU if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# -----
# 1. Data Preparation
# -----
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
])

# Load CIFAR-10 training and test sets
train_data = torchvision.datasets.CIFAR10('./data', train=True, download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10('./data', train=False, download=True, transform=transform)

# Split 10% of training data as validation set (equal samples per class)
targets = np.array(train_data.targets)
val_indices = []
train_indices = []
for class_label in range(10):
    class_indices = np.where(targets == class_label)[0]
    np.random.shuffle(class_indices)
    n_val = len(class_indices) // 10 # 10% for each class
    val_indices.extend(class_indices[:n_val])
    train_indices.extend(class_indices[n_val:])

train_subset = Subset(train_data, train_indices)
val_subset = Subset(train_data, val_indices)

# Define DataLoaders
val_loader = DataLoader(val_subset, batch_size=50, shuffle=False)
test_loader = DataLoader(test_data, batch_size=50, shuffle=False)

# -----
# 2. Define Model Architectures
# -----

```

mlp1: [FC-32, ReLU] + PredictionLayer (FC10)

class MLP1(nn.Module):

```
def __init__(self, input_size=3 * 32 * 32, hidden_size=32, num_classes=10):
    super(MLP1, self).__init__()
    self.input_size = input_size
    self.fc1 = nn.Linear(input_size, hidden_size)
    self.relu = nn.ReLU()
    self.fc2 = nn.Linear(hidden_size, num_classes)
```

def forward(self, x):

```
    x = x.view(-1, self.input_size)
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    return x
```

mlp2: [FC-32, ReLU, FC-64 (no bias)] + PredictionLayer (FC10)

class MLP2(nn.Module):

```
def __init__(self, input_size=3 * 32 * 32, hidden1=32, hidden2=64, num_classes=10):
    super(MLP2, self).__init__()
    self.input_size = input_size
    self.fc1 = nn.Linear(input_size, hidden1)
    self.relu = nn.ReLU()
    self.fc2 = nn.Linear(hidden1, hidden2, bias=False)
    self.fc3 = nn.Linear(hidden2, num_classes)
```

def forward(self, x):

```
    x = x.view(-1, self.input_size)
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    x = self.fc3(x)
    return x
```

cnn3: [Conv-3x3x16, ReLU, Conv-5x5x8, ReLU, MaxPool-2x2, Conv-7x7x16, MaxPool-2x2] + PredictionLayer (FC10)

class CNN3(nn.Module):

```
def __init__(self, num_classes=10):
    super(CNN3, self).__init__()
    # Convolution layers (valid padding means no explicit padding)
    self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1) # Output: 32-3+1 = 30 -> 30x30
    self.relu = nn.ReLU()
    self.conv2 = nn.Conv2d(16, 8, kernel_size=5, stride=1) # 30-5+1 = 26 -> 26x26
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2) # 26//2 = 13 -> 13x13
    self.conv3 = nn.Conv2d(8, 16, kernel_size=7, stride=1) # 13-7+1 = 7 -> 7x7
    self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 7//2 = 3 -> 3x3
    self.fc = nn.Linear(16 * 3 * 3, num_classes)
```

```

def forward(self, x):
    x = self.relu(self.conv1(x))
    x = self.relu(self.conv2(x))
    x = self.pool(x)
    x = self.conv3(x)
    x = self.pool2(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x

```

cnn4: [Conv-3x3x16, ReLU, Conv-3x3x8, ReLU, Conv-5x5x16, ReLU, MaxPool-2x2, Conv-5x5x16, ReLU, MaxPool-2x2] + PredictionLayer (FC10)

```

class CNN4(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN4, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1) # 32-3+1=30
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, stride=1) # 30-3+1=28
        self.conv3 = nn.Conv2d(8, 16, kernel_size=5, stride=1) # 28-5+1=24
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) # 24//2=12
        self.conv4 = nn.Conv2d(16, 16, kernel_size=5, stride=1) # 12-5+1=8
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 8//2=4
        self.fc = nn.Linear(16 * 4 * 4, num_classes)

```

```

def forward(self, x):
    x = self.relu(self.conv1(x))
    x = self.relu(self.conv2(x))
    x = self.relu(self.conv3(x))
    x = self.pool(x)
    x = self.relu(self.conv4(x))
    x = self.pool2(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x

```

cnn5: [Conv-3x3x8, ReLU, Conv-3x3x16, ReLU, Conv-3x3x8, ReLU, Conv-3x3x16, ReLU, MaxPool-2x2, Conv-3x3x16, ReLU, Conv-3x3x8, ReLU, MaxPool-2x2] + PredictionLayer (FC10)

```

class CNN5(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN5, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, kernel_size=3, stride=1) # 32-3+1=30
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, stride=1) # 30-3+1=28
        self.conv3 = nn.Conv2d(16, 8, kernel_size=3, stride=1) # 28-3+1=26
        self.conv4 = nn.Conv2d(8, 16, kernel_size=3, stride=1) # 26-3+1=24
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) # 24//2=12

```

```

self.conv5 = nn.Conv2d(16, 16, kernel_size=3, stride=1) # 12-3+1=10
self.conv6 = nn.Conv2d(16, 8, kernel_size=3, stride=1) # 10-3+1=8
self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 8//2=4
self.fc = nn.Linear(8 * 4 * 4, num_classes)

```

```

def forward(self, x):
    x = self.relu(self.conv1(x))
    x = self.relu(self.conv2(x))
    x = self.relu(self.conv3(x))
    x = self.relu(self.conv4(x))
    x = self.pool(x)
    x = self.relu(self.conv5(x))
    x = self.relu(self.conv6(x))
    x = self.pool2(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x

```

```

# -----

```

```

# 3. Training Function

```

```

# -----

```

```

def train_model(model, model_name, num_epochs=15):
    model = model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters()) # Adam optimizer with default parameters

```

```

train_losses = []
train_accs = []
val_accs = []
step = 0

```

```

for epoch in range(num_epochs):
    train_loader = DataLoader(train_subset, batch_size=50, shuffle=True)
    model.train() # training mode
    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if step % 10 == 0:
        train_losses.append(loss.item())
        # Calculate training accuracy for current batch

```

```

preds = torch.argmax(outputs, dim=1)
batch_acc = (preds == labels).float().mean().item()
train_accs.append(batch_acc)

# Calculate validation accuracy on full validation set
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for val_images, val_labels in val_loader:
        val_images = val_images.to(device)
        val_labels = val_labels.to(device)
        val_outputs = model(val_images)
        val_preds = torch.argmax(val_outputs, dim=1)
        correct += (val_preds == val_labels).sum().item()
        total += val_labels.size(0)
    val_acc = correct / total
    val_accs.append(val_acc)
    model.train() # switch back to training mode

step += 1

print(f"Epoch [{epoch + 1}/{num_epochs}] completed.")

# Evaluate test accuracy
model.eval()
correct_test = 0
total_test = 0
with torch.no_grad():
    for test_images, test_labels in test_loader:
        test_images = test_images.to(device)
        test_labels = test_labels.to(device)
        test_outputs = model(test_images)
        test_preds = torch.argmax(test_outputs, dim=1)
        correct_test += (test_preds == test_labels).sum().item()
        total_test += test_labels.size(0)
test_accuracy = correct_test / total_test

# Get the weights of the first layer (assume for MLPs it is fc1, for CNNs use conv1)
if model_name.lower().startswith("mlp"):
    first_layer_weights = model.fc1.weight.data.cpu().numpy()
else:
    first_layer_weights = model.conv1.weight.data.cpu().numpy()

# Form result dictionary
result_dict = {
    'name': model_name,
    'loss curve': train_losses,
    'train acc curve': train_accs,

```

```

        'val acc curve': val_accs,
        'test acc': test_accuracy,
        'weights': first_layer_weights
    }

    # Save the dictionary using pickle
    filename = f"part3_{model_name}.pkl"
    with open(filename, "wb") as f:
        pickle.dump(result_dict, f)

    print(f"{model_name}: Test Accuracy = {test_accuracy * 100:.2f}%")
    return result_dict

# -----
# 4. Train Each Architecture
# -----
results = []
architectures = {
    'mlp1': MLP1(),
    'mlp2': MLP2(),
    'cnn3': CNN3(),
    'cnn4': CNN4(),
    'cnn5': CNN5()
}

for name, model in architectures.items():
    print(f"\nTraining {name}...")
    result = train_model(model, name, num_epochs=15)
    results.append(result)

# -----
# 5. Plot Performance Comparison and Visualize Weights
# -----

# Create a directory to save the plots if it doesn't exist
save_dir = './plots'
os.makedirs(save_dir, exist_ok=True)

# We have to continue with another code because of a minor mistake: naming issue of the dictionary

```

Q3.1.appendix

```

import pickle
import os
from utils import part3Plots, visualizeWeights

# Create a list of model names you have saved
model_names = ["mlp1", "mlp2", "cnn3", "cnn4", "cnn5"]

```

```

# This will store the loaded-and-fixed result dictionaries
results_fixed = []

for model_name in model_names:
    filename = f"part3_{model_name}.pkl"
    if not os.path.isfile(filename):
        print(f"File {filename} not found, skipping.")
        continue

    with open(filename, "rb") as f:
        result_dict = pickle.load(f)

    # Fix the key names so that part3Plots does not complain:
    # (Change only if they exist in the dictionary)
    if 'loss curve' in result_dict:
        result_dict['loss_curve'] = result_dict.pop('loss curve')
    if 'train acc curve' in result_dict:
        result_dict['train_acc_curve'] = result_dict.pop('train acc curve')
    if 'val acc curve' in result_dict:
        result_dict['val_acc_curve'] = result_dict.pop('val acc curve')
    if 'test acc' in result_dict:
        result_dict['test_acc'] = result_dict.pop('test acc')

    # Append to our fixed-results list
    results_fixed.append(result_dict)

# Now call part3Plots on the fixed dictionaries
save_dir = './plots'
part3Plots(results_fixed, save_dir=save_dir, filename='part3_performance_comparison', show_plot=True)

# Finally, visualize weights for each architecture
for result in results_fixed:
    model_name = result['name']
    weights = result['weights']
    visualizeWeights(weights, save_dir=save_dir, filename=f'weights_{model_name}')

```

Q4.1

```

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import pickle
import torchvision
import torchvision.transforms as transforms
import numpy as np
import os
from utils import part4Plots

```



```

# -----
# 2. Direct copy
# -----
# mlp1: [FC-32, ReLU] + PredictionLayer (FC10)
class MLP1(nn.Module):
    def __init__(self, input_size=3 * 32 * 32, hidden_size=32, num_classes=10):
        super(MLP1, self).__init__()
        self.input_size = input_size
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = x.view(-1, self.input_size)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# mlp2: [FC-32, ReLU, FC-64 (no bias)] + PredictionLayer (FC10)
class MLP2(nn.Module):
    def __init__(self, input_size=3 * 32 * 32, hidden1=32, hidden2=64, num_classes=10):
        super(MLP2, self).__init__()
        self.input_size = input_size
        self.fc1 = nn.Linear(input_size, hidden1)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden1, hidden2, bias=False)
        self.fc3 = nn.Linear(hidden2, num_classes)

    def forward(self, x):
        x = x.view(-1, self.input_size)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x

# cnn3: [Conv-3x3x16, ReLU, Conv-5x5x8, ReLU, MaxPool-2x2, Conv-7x7x16, MaxPool-2x2] +
# PredictionLayer (FC10)
class CNN3(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN3, self).__init__()
        # Convolution layers (valid padding means no explicit padding)
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1) # Output: 32-3+1 = 30 -> 30x30
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(16, 8, kernel_size=5, stride=1) # 30-5+1 = 26 -> 26x26

```

```

self.pool = nn.MaxPool2d(kernel_size=2, stride=2) # 26//2 = 13 -> 13x13
self.conv3 = nn.Conv2d(8, 16, kernel_size=7, stride=1) # 13-7+1 = 7 -> 7x7
self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 7//2 = 3 -> 3x3
self.fc = nn.Linear(16 * 3 * 3, num_classes)

```

```

def forward(self, x):
    x = self.relu(self.conv1(x))
    x = self.relu(self.conv2(x))
    x = self.pool(x)
    x = self.conv3(x)
    x = self.pool2(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x

```

cnn4: [Conv-3x3x16, ReLU, Conv-3x3x8, ReLU, Conv-5x5x16, ReLU, MaxPool-2x2, Conv-5x5x16, ReLU, MaxPool-2x2] + PredictionLayer (FC10)

```

class CNN4(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN4, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1) # 32-3+1=30
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, stride=1) # 30-3+1=28
        self.conv3 = nn.Conv2d(8, 16, kernel_size=5, stride=1) # 28-5+1=24
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) # 24//2=12
        self.conv4 = nn.Conv2d(16, 16, kernel_size=5, stride=1) # 12-5+1=8
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 8//2=4
        self.fc = nn.Linear(16 * 4 * 4, num_classes)

```

```

def forward(self, x):
    x = self.relu(self.conv1(x))
    x = self.relu(self.conv2(x))
    x = self.relu(self.conv3(x))
    x = self.pool(x)
    x = self.relu(self.conv4(x))
    x = self.pool2(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x

```

cnn5: [Conv-3x3x8, ReLU, Conv-3x3x16, ReLU, Conv-3x3x8, ReLU, Conv-3x3x16, ReLU, MaxPool-2x2, Conv-3x3x16, ReLU, Conv-3x3x8, ReLU, MaxPool-2x2] + PredictionLayer (FC10)

```

class CNN5(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN5, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, kernel_size=3, stride=1) # 32-3+1=30
        self.relu = nn.ReLU()

```

```

self.conv2 = nn.Conv2d(8, 16, kernel_size=3, stride=1) #  $30-3+1=28$ 
self.conv3 = nn.Conv2d(16, 8, kernel_size=3, stride=1) #  $28-3+1=26$ 
self.conv4 = nn.Conv2d(8, 16, kernel_size=3, stride=1) #  $26-3+1=24$ 
self.pool = nn.MaxPool2d(kernel_size=2, stride=2) #  $24//2=12$ 
self.conv5 = nn.Conv2d(16, 16, kernel_size=3, stride=1) #  $12-3+1=10$ 
self.conv6 = nn.Conv2d(16, 8, kernel_size=3, stride=1) #  $10-3+1=8$ 
self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) #  $8//2=4$ 
self.fc = nn.Linear(8 * 4 * 4, num_classes)

```

```

def forward(self, x):
    x = self.relu(self.conv1(x))
    x = self.relu(self.conv2(x))
    x = self.relu(self.conv3(x))
    x = self.relu(self.conv4(x))
    x = self.pool(x)
    x = self.relu(self.conv5(x))
    x = self.relu(self.conv6(x))
    x = self.pool2(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x

```

```

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

```

```

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))
])
train_data = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)

```

Function to train a given model and record loss and gradient magnitude of the first layer

```

def train_activation_model(model, train_subset, num_epochs=15):
    model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.0)

    loss_curve = []
    grad_curve = []
    step = 0

    for epoch in range(num_epochs):
        train_loader = DataLoader(train_subset, batch_size=50, shuffle=True)
        model.train()
        for images, labels in train_loader:
            images = images.to(device)
            labels = labels.to(device)

```

```

optimizer.zero_grad()
outputs = model(images)
loss = criterion(outputs, labels)
loss.backward()

# Record training loss and gradient magnitude every 10 steps
if step % 10 == 0:
    loss_curve.append(loss.item())
    if hasattr(model, 'fc1'):
        grad_norm = model.fc1.weight.grad.norm().item()
    elif hasattr(model, 'conv1'):
        grad_norm = model.conv1.weight.grad.norm().item()
    else:
        grad_norm = 0.0
    grad_curve.append(grad_norm)

optimizer.step()
step += 1
print(f"Epoch [{epoch + 1}/{num_epochs}] completed.")

return loss_curve, grad_curve

# List of architectures from Part 3.
architectures = {
    'mlp1': MLP1,
    'mlp2': MLP2,
    'cnn3': CNN3,
    'cnn4': CNN4,
    'cnn5': CNN5
}

results_part4 = []

# For each architecture, create and train both activation variants
for arch_name, arch_class in architectures.items():
    print(f"\nTraining activation variants for {arch_name}...")

    # Create model instance for ReLU variant (default architecture from Part 3)
    model_relu = arch_class()

    # Create a second instance for the sigmoid variant
    model_sigmoid = arch_class()
    # Replace the activation function with Sigmoid by setting the model's 'relu' attribute
    model_sigmoid.relu = nn.Sigmoid()

    # Train the ReLU variant
    print("Training ReLU variant...")

```

```

relu_loss_curve, relu_grad_curve = train_activation_model(model_relu, train_data, num_epochs=15)

# Train the Sigmoid variant
print("Training Sigmoid variant...")
sigmoid_loss_curve, sigmoid_grad_curve = train_activation_model(model_sigmoid, train_data,
num_epochs=15)

# Create a dictionary of results for this architecture
result_dict = {
    'name': arch_name,
    'relu_loss_curve': relu_loss_curve,
    'sigmoid_loss_curve': sigmoid_loss_curve,
    'relu_grad_curve': relu_grad_curve,
    'sigmoid_grad_curve': sigmoid_grad_curve
}

# Save the dictionary to file with filename prefixed by 'part4'
filename = f"part4_{arch_name}.pkl"
with open(filename, "wb") as f:
    pickle.dump(result_dict, f)
print(f"Saved results to {filename}")

results_part4.append(result_dict)

# Create performance comparison plots using the provided part4Plots function
save_dir = './plots_4'
os.makedirs(save_dir, exist_ok=True)
part4Plots(results_part4, save_dir=save_dir, filename='part4_activation_comparison', show_plot=True)

```

Q5.1

```

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Subset
import numpy as np
import pickle
import os
from utils import part5Plots # Provided plotting function

# Set device (use GPU if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# -----
# 1. Data Preparation
# -----
transform = transforms.Compose([

```

```

transforms.ToTensor(),
transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
])

```

Load CIFAR-10 training and test sets

```

train_data = torchvision.datasets.CIFAR10('./data', train=True, download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10('./data', train=False, download=True, transform=transform)

```

Create validation set: split 10% of training data (equal samples per class)

```

targets = np.array(train_data.targets)
val_indices = []
train_indices = []
for class_label in range(10):
    class_indices = np.where(targets == class_label)[0]
    np.random.shuffle(class_indices)
    n_val = len(class_indices) // 10 # 10% per class
    val_indices.extend(class_indices[:n_val])
    train_indices.extend(class_indices[n_val:])

```

```

train_subset = Subset(train_data, train_indices)
val_subset = Subset(train_data, val_indices)

```

Define DataLoaders

```

batch_size = 50
train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_subset, batch_size=batch_size, shuffle=False)

```

-----

2. Define CNN4 Architecture

-----

```

class CNN4(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN4, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1) # 32-3+1=30
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, stride=1) # 30-3+1=28
        self.conv3 = nn.Conv2d(8, 16, kernel_size=5, stride=1) # 28-5+1=24
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) # 24//2=12
        self.conv4 = nn.Conv2d(16, 16, kernel_size=5, stride=1) # 12-5+1=8
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 8//2=4
        self.fc = nn.Linear(16 * 4 * 4, num_classes)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.pool(x)
        x = self.relu(self.conv4(x))

```

```

        x = self.pool2(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

# -----
# 3. Set Up Experiment for Part 5
# -----
# Define the three learning rates for the experiment
learning_rates = {
    '1': 0.1,
    '01': 0.01,
    '001': 0.001,
}

num_epochs = 20

# Dictionaries to record training loss and validation accuracy curves for each learning rate
loss_curves = {'1': [], '01': [], '001': []}
val_acc_curves = {'1': [], '01': [], '001': []}

# This dictionary will store the final results of the experiment
results_dict = {'name': 'cnn4'}

# For each learning rate, create a separate model and train it
for key, lr in learning_rates.items():
    print(f"\nTraining CNN4 with learning rate {lr}")
    model = CNN4().to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.0)

    step = 0 # global training step counter
    loss_curve = []
    val_acc_curve = []

    for epoch in range(num_epochs):
        train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True)
        val_loader = DataLoader(val_subset, batch_size=batch_size, shuffle=False)
        model.train()
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        # Every 10 steps, record training loss and compute validation accuracy

```

```

if step % 10 == 0:
    loss_curve.append(loss.item())

    # Evaluate on validation set
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for val_images, val_labels in val_loader:
            val_images, val_labels = val_images.to(device), val_labels.to(device)
            val_outputs = model(val_images)
            predicted = torch.argmax(val_outputs, dim=1)
            total += val_labels.size(0)
            correct += (predicted == val_labels).sum().item()
    val_acc = correct / total
    val_acc_curve.append(val_acc)
    model.train() # Switch back to training mode
    step += 1

print(f"Epoch {epoch + 1}/{num_epochs} completed for LR {lr}.")

loss_curves[key] = loss_curve
val_acc_curves[key] = val_acc_curve

# Form the final result dictionary with the required keys
results_dict['loss curve 1'] = loss_curves['1']
results_dict['loss curve 01'] = loss_curves['01']
results_dict['loss curve 001'] = loss_curves['001']
results_dict['val acc curve 1'] = val_acc_curves['1']
results_dict['val acc curve 01'] = val_acc_curves['01']
results_dict['val acc curve 001'] = val_acc_curves['001']

# Optionally, save the dictionary to a file for later use
with open("part5_cnn4.pkl", "wb") as f:
    pickle.dump(results_dict, f)

print("Training experiments for CNN4 with different learning rates completed.")

```

Q5.1.b

```

import pickle
import os
from utils import part5Plots

# Load the recorded results from the pickle file
with open("part5_cnn4.pkl", "rb") as f:
    results_dict = pickle.load(f)

# Convert keys from "loss curve 1" to "loss_curve_1" and similarly for validation curves.
def convert_keys(results):

```



```

converted = {}
for key, value in results.items():
    if key.startswith("loss curve"):
        new_key = key.replace(" ", "_")
    elif key.startswith("val acc curve"):
        new_key = key.replace(" ", "_")
    else:
        new_key = key
    converted[new_key] = value
return converted

results_dict = convert_keys(results_dict)

# Create a directory to save plots if it doesn't exist
save_dir = './plots'
os.makedirs(save_dir, exist_ok=True)

# Generate performance comparison plots using the updated dictionary
part5Plots(results_dict, save_dir=save_dir, filename='part5_cnn4_performance', show_plot=True)

```

Q5.2

```

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import pickle
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Subset
import numpy as np
import matplotlib.pyplot as plt
import os

# -----
# 1. Data Preparation
# -----

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
])

# Load CIFAR-10 training and test sets
train_data = torchvision.datasets.CIFAR10('./data', train=True, download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10('./data', train=False, download=True, transform=transform)

# Create validation set: split 10% of training data (equal samples per class)
targets = np.array(train_data.targets)
val_indices = []

```

```

train_indices = []
for class_label in range(10):
    class_indices = np.where(targets == class_label)[0]
    np.random.shuffle(class_indices)
    n_val = len(class_indices) // 10 # 10% per class
    val_indices.extend(class_indices[:n_val])
    train_indices.extend(class_indices[n_val:])

train_subset = Subset(train_data, train_indices)
val_subset = Subset(train_data, val_indices)

# Define DataLoaders
batch_size = 50

# -----
# 2. Define CNN4 Architecture
# -----
class CNN4(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN4, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1) # 32-3+1 = 30
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, stride=1) # 30-3+1 = 28
        self.conv3 = nn.Conv2d(8, 16, kernel_size=5, stride=1) # 28-5+1 = 24
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) # 24//2 = 12
        self.conv4 = nn.Conv2d(16, 16, kernel_size=5, stride=1) # 12-5+1 = 8
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) # 8//2 = 4
        self.fc = nn.Linear(16 * 4 * 4, num_classes)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.pool(x)
        x = self.relu(self.conv4(x))
        x = self.pool2(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

# -----
# 3. Scheduled Learning Rate Training (Record every 10 steps)
# -----
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

```

```

total_epochs = 30
switch_step = 5000 # Switch learning rate after 500 training steps
global_step = 0 # Global step counter
val_acc_curve = [] # To record (global_step, validation_accuracy)

model = CNN4().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.0)

for epoch in range(total_epochs):
    train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=batch_size, shuffle=False)
    model.train()
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    global_step += 1

    # Every 10 training steps, evaluate the model on the validation set
    if global_step % 10 == 0:
        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for v_images, v_labels in val_loader:
                v_images, v_labels = v_images.to(device), v_labels.to(device)
                v_outputs = model(v_images)
                predicted = torch.argmax(v_outputs, dim=1)
                total += v_labels.size(0)
                correct += (predicted == v_labels).sum().item()
        val_acc = correct / total
        val_acc_curve.append((global_step, val_acc))
        print(f"Step {global_step} - Validation Accuracy: {val_acc:.4f}")
        model.train()

    # Switch learning rate at the designated global step
    if global_step == switch_step:
        optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.0)
        print(f"Learning rate switched to 0.01 at step {global_step}")

print("Training completed.")

# Immediately save the validation accuracy curve data
with open("scheduled_lr_val_acc.pkl", "wb") as f:

```

```

    pickle.dump(val_acc_curve, f)
print("Validation accuracy data saved.")

steps, accs = zip(*val_acc_curve)

plt.figure(figsize=(10, 6))
plt.plot(steps, accs, marker='o', linestyle='-', color='blue', linewidth=2, markersize=6, label='Validation Accuracy')
plt.xlabel('Global Training Step', fontsize=14)
plt.ylabel('Validation Accuracy', fontsize=14)
plt.title('Scheduled Learning Rate Training - 1: Validation Accuracy Curve', fontsize=16)
plt.legend(fontsize=12)
plt.grid(True)
plt.tight_layout()

# Save the plot to a designated directory.
save_dir = './plots'
os.makedirs(save_dir, exist_ok=True)
plot_path = os.path.join(save_dir, "scheduled_lr_val_acc_custom.png")
plt.savefig(plot_path)
print(f"Plot saved to {plot_path}")

plt.show()

```

Q5.3

```

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Subset
import numpy as np
import matplotlib.pyplot as plt
import os
import pickle

# -----
# 1. Data Preparation
# -----

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
])

# Load CIFAR-10 training and test sets
train_data = torchvision.datasets.CIFAR10('./data', train=True, download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10('./data', train=False, download=True, transform=transform)

# Create validation set: split 10% of training data (equal samples per class)

```

```

targets = np.array(train_data.targets)
val_indices = []
train_indices = []
batch_size = 50
for class_label in range(10):
    class_indices = np.where(targets == class_label)[0]
    np.random.shuffle(class_indices)
    n_val = len(class_indices) // 10 # 10% per class
    val_indices.extend(class_indices[:n_val])
    train_indices.extend(class_indices[n_val:])

train_subset = Subset(train_data, train_indices)
val_subset = Subset(train_data, val_indices)
test_loader = DataLoader(test_data, batch_size=50, shuffle=False)

# -----
# 2. Define CNN4 Architecture
# -----
class CNN4(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN4, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1) #  $32-3+1 = 30$ 
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, stride=1) #  $30-3+1 = 28$ 
        self.conv3 = nn.Conv2d(8, 16, kernel_size=5, stride=1) #  $28-5+1 = 24$ 
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) #  $24//2 = 12$ 
        self.conv4 = nn.Conv2d(16, 16, kernel_size=5, stride=1) #  $12-5+1 = 8$ 
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2) #  $8//2 = 4$ 
        self.fc = nn.Linear(16 * 4 * 4, num_classes)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.pool(x)
        x = self.relu(self.conv4(x))
        x = self.pool2(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

# -----
# 3. Scheduled Learning Rate Training
# -----
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

total_epochs = 30
# With 5000 training images and batch_size = 50, we have 100 steps per epoch.

```

```

# Therefore, 500 steps correspond to 5 epochs and 2000 steps correspond to 20 epochs.
switch_step_1 = 5000 # Switch from lr=0.1 to lr=0.01 at 500 steps (~5 epochs)
switch_step_2 = 10000 # Switch from lr=0.01 to lr=0.001 at 2000 steps (~20 epochs)
global_step = 0 # Global step counter
val_acc_curve = [] # Record tuples: (global_step, validation_accuracy)
loss_curve = [] # Optional: record training loss every 10 steps

model = CNN4().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.0)

for epoch in range(total_epochs):
    train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=batch_size, shuffle=False)
    model.train()
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        global_step += 1

    # Record loss and evaluate every 10 training steps
    if global_step % 10 == 0:
        loss_curve.append(loss.item())
        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for v_images, v_labels in val_loader:
                v_images, v_labels = v_images.to(device), v_labels.to(device)
                v_outputs = model(v_images)
                predicted = torch.argmax(v_outputs, dim=1)
                total += v_labels.size(0)
                correct += (predicted == v_labels).sum().item()
        val_acc = correct / total
        val_acc_curve.append((global_step, val_acc))
        print(f"Step {global_step} - Validation Accuracy: {val_acc:.4f}")
        model.train()

    # Check for learning rate switches
    if global_step == switch_step_1:
        optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.0)
        print(f"Learning rate switched to 0.01 at global step {global_step}")
    if global_step == switch_step_2:
        optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.0)
        print(f"Learning rate switched to 0.001 at global step {global_step}")

```

```

print("Training completed.")

# Evaluate on test set
model.eval()
correct_test = 0
total_test = 0
with torch.no_grad():
    for t_images, t_labels in test_loader:
        t_images, t_labels = t_images.to(device), t_labels.to(device)
        t_outputs = model(t_images)
        predicted = torch.argmax(t_outputs, dim=1)
        total_test += t_labels.size(0)
        correct_test += (predicted == t_labels).sum().item()
test_accuracy = correct_test / total_test
print(f"Test Accuracy with scheduled SGD: {test_accuracy:.4f}")

# Save the validation accuracy curve data immediately
with open("scheduled_lr_val_acc.pkl", "wb") as f:
    pickle.dump(val_acc_curve, f)
print("Validation accuracy data saved.")

# Plot the validation accuracy curve
steps, accs = zip(*val_acc_curve)
plt.figure(figsize=(10, 6))
plt.plot(steps, accs, marker='o', linestyle='-', color='blue', linewidth=2, markersize=6, label='Validation Accuracy')
plt.xlabel('Global Training Step', fontsize=14)
plt.ylabel('Validation Accuracy', fontsize=14)
plt.title('Scheduled Learning Rate Training - Validation Accuracy Curve', fontsize=16)
plt.legend(fontsize=12)
plt.grid(True)
plt.tight_layout()

save_dir = './plots'
os.makedirs(save_dir, exist_ok=True)
plot_path = os.path.join(save_dir, "scheduled_lr_val_acc_custom.png")
plt.savefig(plot_path)
print(f"Plot saved to {plot_path}")
plt.show()

```

Q.5.4

```

import pickle
import os
import matplotlib.pyplot as plt

# Load scheduled learning rate validation accuracy curve
with open("scheduled_lr_val_acc.pkl", "rb") as f:
    sched_val_curve = pickle.load(f)

```

```

sched_steps, sched_accs = zip(*sched_val_curve)

# Load Adam run result from part3_cnn4.pkl
with open("part3_cnn4.pkl", "rb") as f:
    adam_dict = pickle.load(f)

# Fix keys if needed
if 'val acc curve' in adam_dict:
    adam_dict['val_acc_curve'] = adam_dict.pop('val acc curve')
if 'test acc' in adam_dict:
    adam_dict['test_acc'] = adam_dict.pop('test acc')

# Extract Adam results
adam_accs = adam_dict['val_acc_curve']
adam_steps = list(range(0, len(adam_accs)*10, 10)) # assuming every 10 steps

# Test accuracies
test_acc_sgd = 0.6571 # from printed result or replace with actual
test_acc_adam = adam_dict['test_acc']

# Plotting
plt.figure(figsize=(14, 6))

# Left: Validation accuracy curves
plt.subplot(1, 2, 1)
plt.plot(sched_steps, sched_accs, label="cnn4_SGD (Scheduled LR)", color='blue')
plt.plot(adam_steps, adam_accs, label="cnn4_Adam", color='red')
plt.xlabel("Step")
plt.ylabel("Validation Accuracy")
plt.title("Validation Accuracy")
plt.legend()
plt.grid(True)

# Right: Test accuracy comparison
plt.subplot(1, 2, 2)
plt.scatter(["cnn4_SGD", "cnn4_Adam"], [test_acc_sgd, test_acc_adam], color=['blue', 'red'])
plt.ylim(0.65, 0.675)
plt.ylabel("Test Accuracy")
plt.title("Test Accuracy")
plt.grid(True)

# Show and save
plt.tight_layout()
plt.savefig("optimizer_comparison.png")
plt.show()

```