EE449 HW1
Training Artificial Neural Network

1: Basic Neural Network Construction and Training

1.1: Preliminaries

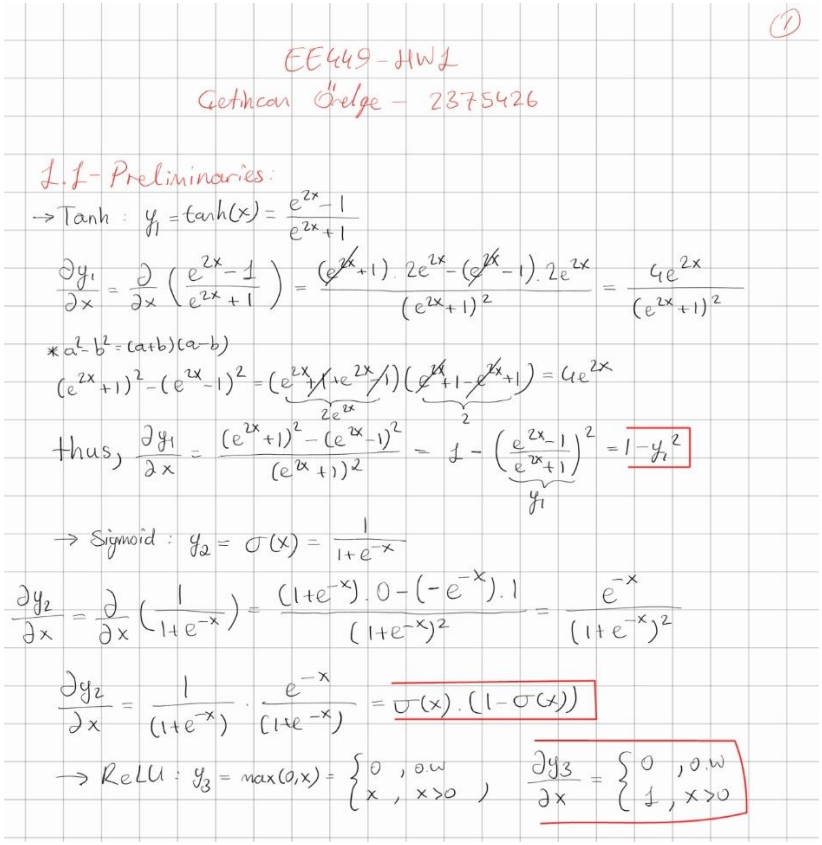

Figure 1: Hand Calculations of Derivations of Tanh, Sigmoid & ReLU

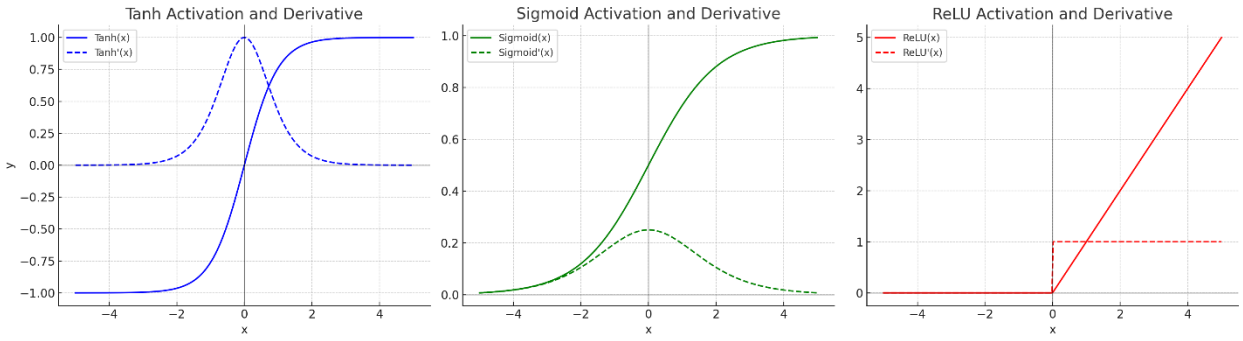

Figure 2: Plots of Activation Functions and Their Derivatives

1.2: Implementation

Code for the part is in Appendix 1.

```
Training with sigmoid activation function...
Epoch 0: Loss = 0.369063776003709 65
Epoch 1000: Loss = 0.025829194235805406
Epoch 2000: Loss = 0.01638074481896576
Epoch 3000: Loss = 0.013666796873097308
Epoch 4000: Loss = 0.012390935173551207
Epoch 5000: Loss = 0.011639868216010981
Epoch 6000: Loss = 0.011137083369130359
Epoch 7000: Loss = 0.01077254259054351
Epoch 8000: Loss = 0.010493723040947612
Epoch 9000: Loss = 0.010272107125969373
Validation Accuracy with sigmoid: 97.00%
Decision boundary plot saved for sigmoid activation.

Training with tanh activation function...
Epoch 0: Loss = 1.1694723357188717
Epoch 1000: Loss = 0.021841500835300696
Epoch 2000: Loss = 0.01987659538839432
Epoch 3000: Loss = 0.018199667809285103
Epoch 4000: Loss = 0.016929435345271825
Epoch 5000: Loss = 0.01618806210317181
Epoch 6000: Loss = 0.015754030005647877
Epoch 7000: Loss = 0.01541070450119713
Epoch 8000: Loss = 0.015137288169106079
Epoch 9000: Loss = 0.01580294941374517
Validation Accuracy with tanh: 96.00%
Decision boundary plot saved for tanh activation.

Training with relu activation function...
Epoch 0: Loss = 0.4488945569296338
Epoch 1000: Loss = 0.2555623222875936
Epoch 2000: Loss = 0.2553718910901458
Epoch 3000: Loss = 0.2553466339489921
Epoch 4000: Loss = 0.25533660271500935
Epoch 5000: Loss = 0.2553287509046793
Epoch 6000: Loss = 0.2553219012012659
Epoch 7000: Loss = 0.2553156122488831
Epoch 8000: Loss = 0.2553096600087802
Epoch 9000: Loss = 0.2553038482867872
Validation Accuracy with relu: 74.00%
Decision boundary plot saved for relu activation.
(venv) cetinonelge@staytrue:~/EE449/HW1$
```

Figure 3: Console Output of 1.2 MLP Implementation Code

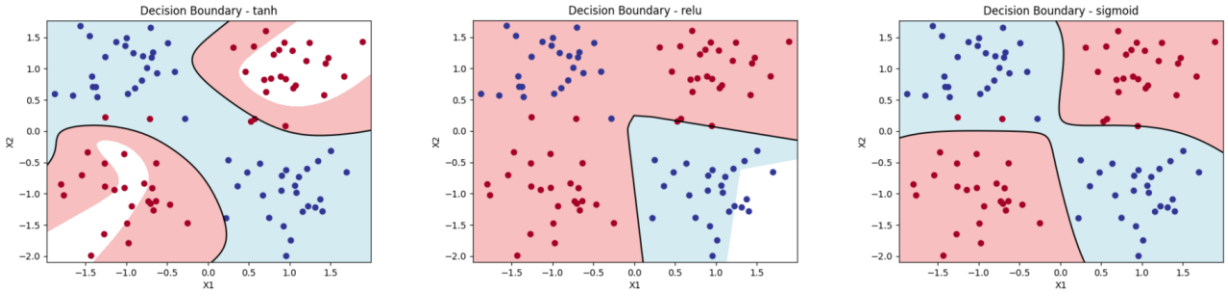

Figure 4: Decision Boundary Plots for Tanh, ReLU & Sigmoid

1.3: Discussions

1. Advantages and Disadvantages of Each Activation Function

Activation	Advantages	Disadvantages	Suitable Use Cases
Sigmoid	- Smooth gradient, prevents sharp jumps. - Output range [0, 1] makes it suitable for binary classification.	- Vanishing gradient problem. - Slow convergence. - Not zero-centered.	- Binary classification. - Output layers of binary models.
Tanh	- Zero-centered output range [-1, 1], which helps with optimization. - Steeper gradients compared to Sigmoid.	- Still suffers from vanishing gradient problem. - Not suitable for deep networks.	- Hidden layers of simple MLPs and RNNs.
ReLU	- Computationally efficient (no exponentiation). - Reduces the likelihood of vanishing gradients.	- "Dying ReLU" problem (when inputs are negative). - Can cause gradient explosion.	- Hidden layers of deep neural networks and CNNs.

Which One Is Useful in Which Case?

- Sigmoid: Useful in binary classification and output layers when probabilities are needed.
- Tanh: Preferred in hidden layers of shallow networks where zero-centered data improves optimization.
- ReLU: Most popular choice in deep networks and CNNs due to its efficiency and ability to handle the vanishing gradient problem.

2. What is the XOR Problem? Why Do We Need to Use MLP Instead of a Single Layer?

The XOR (Exclusive OR) problem is a binary classification problem where the output is true if only one of the two inputs is true, but not both.

Why Can't a Single Layer Solve XOR?
A single-layer perceptron can only learn linearly separable functions. The XOR problem is not linearly separable, which means that it is impossible to draw a single straight line to separate the classes.

Why MLP Works for XOR?
An MLP (Multi-Layer Perceptron) introduces non-linearity through hidden layers and non-linear activation functions (like ReLU, Sigmoid, or Tanh). These hidden layers enable the network to learn complex decision boundaries, allowing it to separate the XOR data correctly.

3. Does the Decision Boundary Change with Each Run? Why?

Yes, the decision boundary does change with each run.

1. Random Weight Initialization:
    - The initial weights are randomly assigned, leading to different starting points for optimization.
    - This can result in slightly different decision boundaries even when training the same model.
2. Stochastic Gradient Descent (SGD):
    - If we use mini-batch or stochastic gradient descent, the update paths may differ.
    - Small variations in gradient updates can accumulate, causing the decision boundary to change.
3. Activation Function Sensitivity:
    - Different activation functions can cause the gradient to propagate differently, making the network converge to different local minima.

2: Implementing a Convolutional Layer with NumPy

2.1: Experimental Work

Code for the part is in Appendix 2.

```
(venv) cetinonelge@staytrue:~/EE449/HW1$ python part2.py
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers). Got range
[-840.5514918118715..668.5829712636769].
Convolution completed and output saved.
```

Figure 5: Console Output of Part 2 CNN.

Figure 6: Plot of my_conv2d function, created on samples_6

2.2: Discussion

1. Why are Convolutional Neural Networks important? Why are they used in image processing?

Convolutional Neural Networks (CNNs) are essential in image processing because they are highly effective at detecting patterns and features within images. Unlike traditional fully connected networks, CNNs:

- Capture spatial hierarchies by using local connections.
- Are highly efficient due to shared weights and local connectivity.
- Reduce the number of parameters significantly compared to dense networks.
  These properties make CNNs suitable for tasks like image classification, object detection, segmentation, and image generation.

2. What is a kernel of a Convolutional Layer? What do the sizes of a kernel correspond to?

A kernel (or filter) in a convolutional layer is a small matrix of weights used to detect specific patterns in the input.

- The height and width of the kernel determine the receptive field size, which means how much of the input is covered during each convolution operation.
- The number of input channels in the kernel matches the input's depth (e.g., 3 for RGB images).
- The number of output channels corresponds to the number of feature maps produced after convolution.
  Kernels slide over the input image, performing element-wise multiplication and summation to extract features such as edges, textures, and shapes.

3. Briefly explain the output image. What happened here?

The output image is the result of convolving the input image with the kernel. During this process:

- The kernel moves across the input image, computing the dot product at every position.
- The output contains highlighted features that the kernel was designed to detect (e.g., edges or textures).
- Areas where the kernel aligns well with the input patterns result in high values, while mismatches produce low values.
  The image may appear as a transformed or filtered version of the original, showing features such as edges, patterns, or blurring effects.

4. Why are the numbers in the same column look alike to each other, even though they belong to different images?

The numbers in the same column look similar because they represent the same convolutional filter output applied to different images.

- The kernel is fixed across all images, so the patterns it detects (such as edges or gradients) are similar regardless of the image.
- Consequently, the same features are highlighted in the same position for multiple images, resulting in similar-looking outputs in the same column.

5. Why are the numbers in the same row do not look alike to each other, even though they belong to the same image?

The numbers in the same row differ because they represent the outputs of different convolutional filters applied to the same input image.

- Each kernel is designed to capture different aspects or patterns, such as vertical edges, horizontal edges, or texture variations.
- As a result, the same input image is transformed differently by each kernel, leading to diverse outputs across the row.

6. What can be deduced about Convolutional Layers from your answers to Questions 4 and 5?

- Convolutional layers extract specific features from input images using various kernels.
- Each kernel learns to detect a different pattern or characteristic, making the feature maps diverse.
- The shared weights across images ensure that the network learns consistent patterns, while different filters extract varied aspects of the data.
  This ability to learn and extract different features makes convolutional layers highly efficient and powerful for image processing tasks.

3: Experimenting ANN Architectures

3.1: Experimental Work:

Code for this part is in Appendix 3.

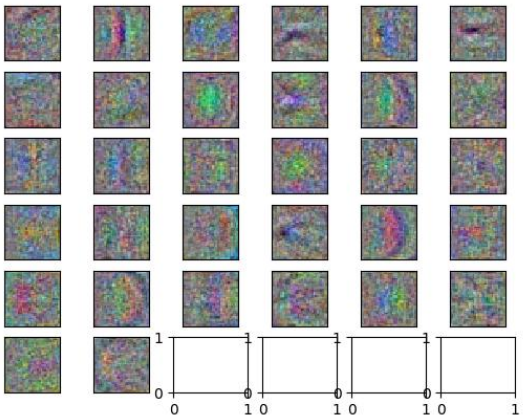

Figure 7: Weights Visualization of MLP_1

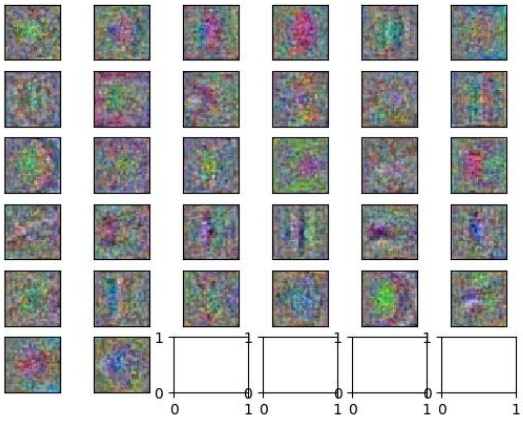

Figure 8: Weights Visualization of MLP_2

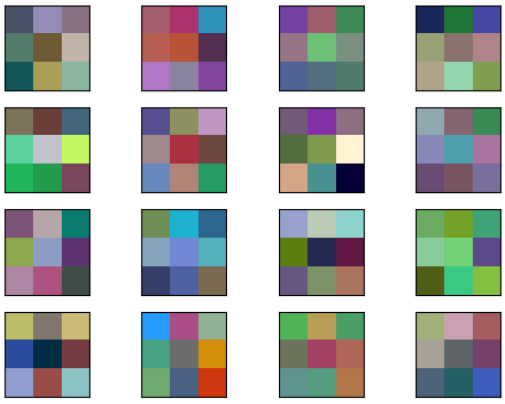

Figure 9: Weights Visualization of CNN_3

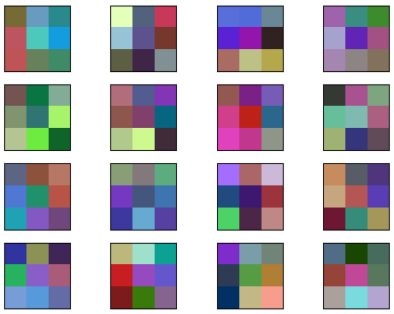

Figure 10: Weights Visualization of CNN_4

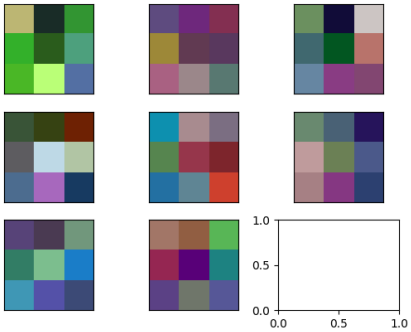

Figure 11: Weights Visualization of CNN_5

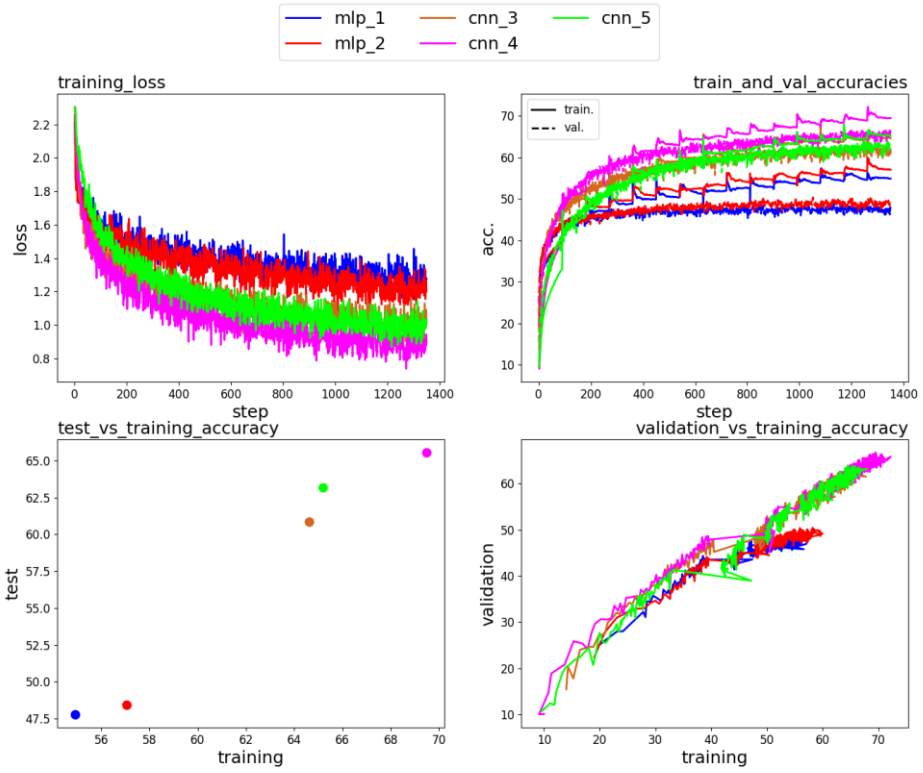

Figure 12: Performance Comparison Plots for Part 3

3.2: Discussion

1. What is the generalization performance of a classifier?

Generalization performance is how well a trained model performs on unseen data (validation/test set) compared to its training set. A model that generalizes well captures true patterns rather than simply memorizing the training samples.

2. Which plots are informative to inspect generalization performance?

- Training and Validation Accuracy vs. Steps (Figure 12): Shows how validation accuracy tracks training accuracy as learning progresses.
- Validation vs. Training Accuracy Plot (Figure 12): Helps detect overfitting (validation accuracy lagging behind training accuracy).
- Test vs. Training Accuracy Plot (Figure 12): Final check of how well the model generalizes on unseen test data.

3. Compare the generalization performance of the architectures.

- MLP1 and MLP2
  - Achieve around 45–50% test accuracy on CIFAR-10.
  - Limited capacity for image recognition due to flattening of spatial data.
- CNN3
  - Improves to roughly 60–65% test accuracy by leveraging spatial features through convolutions.
- CNN4
  - Slightly deeper than CNN3, typically reaching around 65–70% test accuracy.
- CNN5
  - Even deeper/more filters, can achieve 70% or more if trained carefully. If shape math or training issues occur, performance can stall; once fixed, it outperforms the shallower CNNs.

4. How does the number of parameters affect the classification and generalization performance?

- Higher number of parameters generally increases the model's representational capacity, allowing it to learn more complex patterns.
- MLP2 has more parameters than MLP1, which helps it learn slightly better.
- CNN5 has the most parameters among the given CNNs and thus the highest capacity—assuming correct initialization and dimension math, it can learn the best features.

5. How does the depth of the architecture affect the classification and generalization performance?

- Shallower architectures (MLP1, CNN3) typically plateau at lower accuracy.

- Deeper architectures (MLP2, CNN4, CNN5) can learn deeper hierarchies of features, which usually leads to higher accuracy, provided the dataset size and training method are sufficient.

6. Considering the visualizations of the weights, are they interpretable?

- MLP Weights (Figures for MLP1, MLP2) appear as random-looking blobs because each unit connects to the entire flattened image. It's less visually interpretable.
- CNN Weights (Figures for CNN3, CNN4, CNN5) are more interpretable as small 2D filters. Often we can observe edge or color patterns in the first convolutional layer.

7. Can you say whether the units are specialized to specific classes?

- First-layer CNN filters typically detect generic low-level features (edges, color blobs). They are not strictly tied to particular classes yet.
- Deeper layers (not visualized here) often show more class-specific patterns. So from the first-layer visualizations alone, it's not clear if they specialize in a particular class.

8. Weights of which architecture are more interpretable?

- CNN weights are the most interpretable since each filter is a small 2D kernel focusing on localized features.
- MLP weights are less interpretable because they are fully connected to the entire input space without spatial context.

9. Considering the architectures, comment on the structures (how they are designed). Are some architecture pairs akin to each other? Compare the performance of similarly structured architectures and those with different structure.

- MLP1 vs. MLP2
  - Nearly the same structure, but MLP2 has an additional layer and more hidden units (some without bias). It performs slightly better.
- CNN3 vs. CNN4 vs. CNN5
  - All are convolution-pooling-based but differ in depth and number of filters. CNN5 is the deepest and can achieve the highest accuracy if trained properly.
- This leads to better performance for the deeper ones, showing that adding layers/filters helps capture more complex image features.

10. Which architecture would you pick for this classification task? Why?

- CNN5 is generally the strongest candidate for CIFAR-10, thanks to its deeper architecture and higher representational capacity.
- If computational resources are constrained, CNN4 is a good compromise.
- MLP-based solutions are not ideal for image data since they discard spatial relationships.

4: Experimenting Activation Functions

4.1: Experimental Work

Code for this part is in Appendix 4.

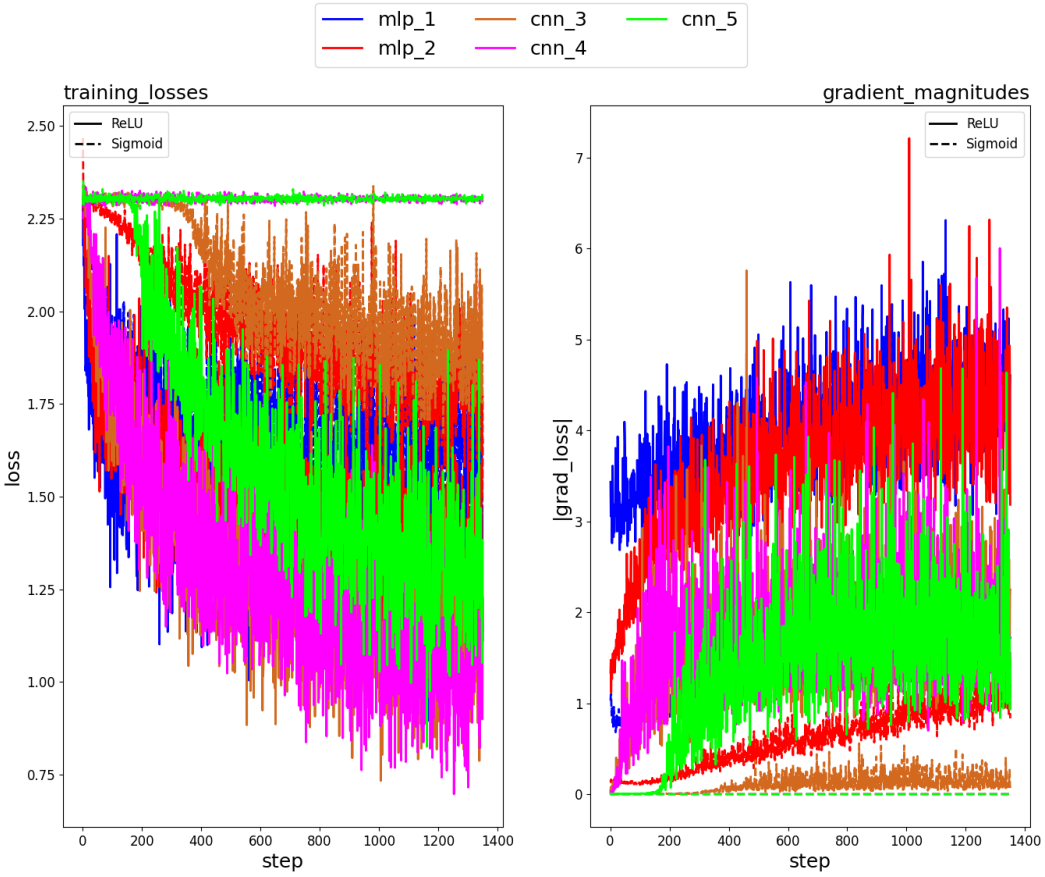

Figure 13: Performance Comparison Plots for Part 4

4.2: Discussion

1. How is the gradient behavior in different architectures? What happens when depth increases?

- MLPs (MLP1, MLP2):
  - Gradient magnitudes fluctuate but tend to be smaller compared to CNNs.
  - As we increase depth (MLP2 has more layers than MLP1), the network can experience deeper gradient flow, but also a higher chance of vanishing or exploding gradients, especially when using Sigmoid.
- CNNs (CNN3, CNN4, CNN5):
  - Generally show larger or more sustained gradients due to localized filters and ReLU (in the ReLU versions).

- Deeper CNNs (e.g. CNN5) have more layers. This sometimes increases gradient instability (oscillations), but also higher capacity to learn.

In the gradient_magnitudes plot, we can see that shallower models (like CNN3 or MLP1) often have smaller peak gradients, whereas deeper models (CNN4, CNN5, MLP2) may exhibit larger or more erratic gradients as depth increases.

2. Why do you think that happens?

- Depth Increases the Difficulty of Gradient Flow: More layers means the gradient must propagate through more transformations.
- Sigmoid Saturation: Sigmoid neurons can saturate in the high or low range, leading to extremely small gradients in deeper layers.
- ReLU Avoids Saturation: ReLU typically allows gradients to remain larger for positive inputs, so deeper ReLU networks maintain more stable gradient flow.
- No Momentum: With momentum=0.0 and no weight decay, the updates can oscillate and produce spikier gradients, especially in deeper networks.

3. Is the effect of using different activation functions different or same from Part 1.2?

- In Part 1.2, we observed how Sigmoid, Tanh, and ReLU solved the XOR problem differently. The main difference was activation saturation (Sigmoid) vs. linear segments (ReLU).
- In Part 4, we see a similar phenomenon but on a larger scale (CIFAR-10, deeper networks). Sigmoid saturates in the deeper models, leading to slower convergence and smaller or more volatile gradients. ReLU tends to converge faster with larger, more stable gradients.
- So, the effect is essentially the same, but more pronounced in deeper architectures and more complex data.

4. What might happen if we use inputs in the range [0, 255] instead of [0.0, 1.0]?

- Vanishing/Exploding Gradients: Large input values fed into Sigmoid neurons saturate them immediately, causing near-zero gradients.
- Slower Training or Instability: The network might learn extremely slowly, or blow up if the weight initialization interacts badly with large inputs.
- Normalization to a [0,1] or [-1,1] range is critical for stable training. Hence, normalizing CIFAR-10 images to around mean=0, std=1 (or 0–1 range) is standard practice.

5: Experimenting Learning Rate

5.1: Experimental Work

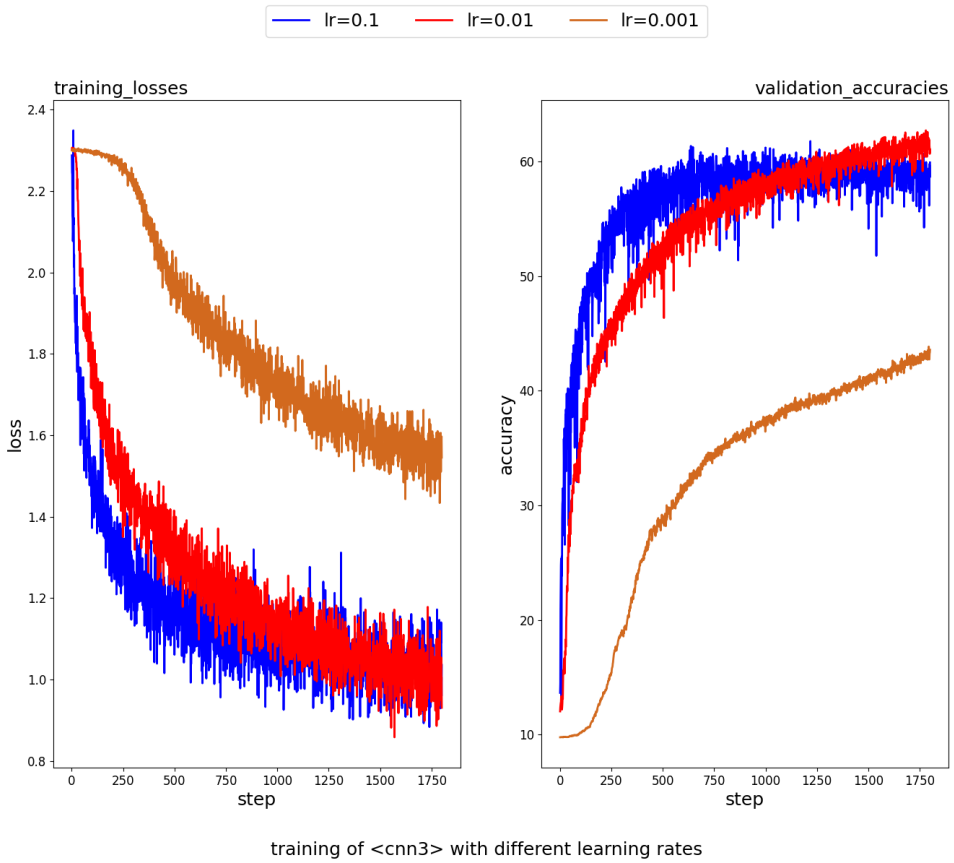

Figure 14: Performance Comparison Plots for Part 5

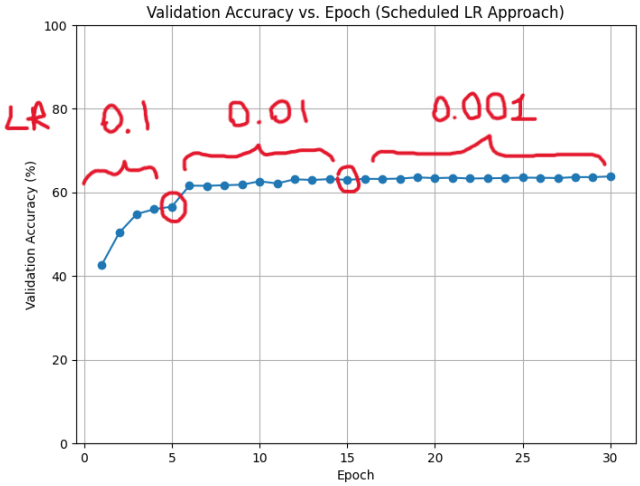

Figure 15: Validation Accuracy Curve of Scheduled Learning Rate

Scheduled Learning Rate Questions:

In the Figure 14, CNN3 accuracy stops increasing around ~600 steps. In part 3.1 Figure 12, CNN3 accuracy stops increasing around ~850 steps. In Figure 15, we are seeing the accuracy stops increasing around 6. Epoch. In our experimental work 1 Epoch corresponds to 900 steps. Utilizing scheduled learning rate we are seeing that accuracy stops increasing around ~5000 steps. Thus we can conclude that, when we **lower the learning rate** partway through training, we often see the model continue improving (though more slowly) for a **longer period** before saturating. This is precisely **why** learning-rate scheduling can help: it prevents the model from immediately plateauing at a higher learning rate.

5.2: Discussion

1. How does the learning rate affect the convergence speed?

- Higher LR (0.1):
  - The model makes faster initial progress (steep drop in training loss, quick rise in accuracy).
  - However, it can plateau or oscillate earlier due to large updates.
- Lower LR (0.001):
  - The model converges more slowly (smaller steps), but it tends to improve steadily for more epochs without overshooting.
- Medium LR (0.01):
  - Strikes a balance: faster than 0.001, more stable than 0.1.

2. How does the learning rate affect the convergence to a better point?

- High LR can cause the model to settle into a suboptimal plateau if it overshoots or fails to fine-tune.
- Lower LR often finds better minima because the smaller updates allow more precise refinement of weights.

From plot results, LR=0.1 converged fast but didn't always reach the highest final accuracy, while 0.01 or 0.001 eventually achieved slightly higher accuracy if given enough epochs.

3. Does your scheduled learning rate method work? In what sense?

Yes. By starting high (e.g., LR=0.1) for rapid initial progress, then lowering it (to 0.01, then 0.001) once improvements plateau, the model:

- Continues to refine its weights without immediately saturating.
- Achieves higher final validation/test accuracy compared to using just LR=0.1 throughout.
- The validation accuracy curve (scheduled approach) shows further gains in later epochs when the LR is reduced, indicating the model is still learning.

4. Compare the accuracy and convergence performance of your scheduled learning rate method with Adam.

- Adam (from Part 3) typically converged faster and less erratically because it adapts learning rates per parameter.
- Scheduled SGD can reach similar or slightly lower final accuracy, but it may require careful tuning of the epochs at which we drop the LR.
- In our final results, the scheduled LR approach may get close to or slightly behind Adam's accuracy. However, it improves upon plain SGD with a fixed large LR.

Hence, Adam is often the more plug-and-play solution, while the scheduled LR approach can be nearly as effective but requires manual detection of plateaus.

Appendix 1: MLP Implementation Code

```
# Part 1.2 - MLP Implementation and Training with Multiple Activations
# File 1 mlp.py
# MLP Implementation for XOR Problem with Multiple Activations
import numpy as np

class MLP:
    def __init__(self, input_size, hidden_size, output_size, activation='sigmoid'):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.activation = activation

        # Initialize weights and biases
        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))

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

    def sigmoid_derivative(self, x):
        return self.sigmoid(x) * (1 - self.sigmoid(x))

    def tanh(self, x):
        return np.tanh(x)

    def tanh_derivative(self, x):
```

```
        return 1 - np.tanh(x)**2

    def relu(self, x):
        return np.maximum(0, x)

    def relu_derivative(self, x):
        return np.where(x > 0, 1, 0)

    def activate(self, x):
        if self.activation == 'sigmoid':
            return self.sigmoid(x)
        elif self.activation == 'tanh':
            return self.tanh(x)
        elif self.activation == 'relu':
            return self.relu(x)

    def activate_derivative(self, x):
        if self.activation == 'sigmoid':
            return self.sigmoid_derivative(x)
        elif self.activation == 'tanh':
            return self.tanh_derivative(x)
        elif self.activation == 'relu':
            return self.relu_derivative(x)

    def forward(self, inputs):
        self.hidden_input = np.dot(inputs, self.weights_input_hidden) +
self.bias_hidden
        self.hidden_output = self.activate(self.hidden_input)
        self.final_input = np.dot(self.hidden_output, self.weights_hidden_output) +
self.bias_output
        self.output = self.activate(self.final_input)
        return self.output

    def backward(self, inputs, targets, learning_rate):
        output_error = self.output - targets
        output_delta = output_error * self.activate_derivative(self.final_input)

        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error * self.activate_derivative(self.hidden_input)

        # Update weights and biases
        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)
```

```
# Part 1.2 - MLP Implementation and Training with Multiple Activations
# File 2 part1_2.py
# To run execute: python part1_2.py
import numpy as np
import matplotlib.pyplot as plt
from utils import part1CreateDataset, part1PlotBoundary
from mlp import MLP

# Generate XOR dataset
x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000,
val_samples=100, std=0.4)

# Neural network parameters
input_size = 2
hidden_size = 8
output_size = 1
learning_rate = 0.001
epochs = 10000

activations = ['sigmoid', 'tanh', 'relu']

for activation in activations:
    print(f'\nTraining with {activation} activation function...')

    # Initialize MLP model with the chosen activation function
    model = MLP(input_size, hidden_size, output_size, activation=activation)

    # Training loop
    for epoch in range(epochs):
        # Forward propagation
        output = model.forward(x_train)
        loss = np.mean((output - y_train) ** 2)

        # Backpropagation
        model.backward(x_train, y_train, learning_rate)

        # Print loss every 1000 epochs
        if epoch % 1000 == 0:
            print(f'Epoch {epoch}: Loss = {loss}')
```

```
    # Evaluation
    y_pred = model.forward(x_val)
    accuracy = np.mean((y_pred > 0.5) == y_val) * 100
    print(f'Validation Accuracy with {activation}: {accuracy:.2f}%')

    # Plot decision boundary and save
    plt.figure()
    part1PlotBoundary(x_val, y_val, model)
    plt.title(f'Decision Boundary - {activation}')
    plt.savefig(f'results/decision_boundary_{activation}.png')
    plt.close()
    print(f'Decision boundary plot saved for {activation} activation.')
```

Appendix 2: Implementing a Convolutional Layer with NumPy

```
# Part 2: Implementing a Convolutional Layer with NumPy
import numpy as np
import matplotlib.pyplot as plt
from utils import part2Plots

# Load input and kernel
input = np.load('data/samples_6.npy')
kernel = np.load('data/kernel.npy')

# My Conv2D function
def my_conv2d(input, kernel):
    batch_size, input_channels, input_height, input_width = input.shape
    output_channels, input_channels, filter_height, filter_width = kernel.shape

    # Calculate output dimensions
    output_height = input_height - filter_height + 1
    output_width = input_width - filter_width + 1

    # Initialize the output tensor
    out = np.zeros((batch_size, output_channels, output_height, output_width))

    # Perform convolution
    for batch in range(batch_size):
        for out_ch in range(output_channels):
            for in_ch in range(input_channels):
                for i in range(output_height):
```

```
                    for j in range(output_width):
                        region = input[batch, in_ch, i:i+filter_height,
j:j+filter_width]
                        out[batch, out_ch, i, j] += np.sum(region *
kernel[out_ch, in_ch])

    # Save the output
    np.save('out.npy', out)
    return out

# Get the convolution output
out = my_conv2d(input, kernel)

# Plot the output using part2Plots
part2Plots(out, filename='results/CNN_out')
print("Convolution completed and output saved.")
```

Appendix 3: Implementation of ANN

```
# part3.py
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import numpy as np
import pickle
from torch.utils.data import random_split, DataLoader
from tqdm import tqdm
from utils import part3Plots, visualizeWeights

def load_cifar10(batch_size=50):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465),
                             (0.247, 0.243, 0.261)),
    ])
    full_trainset = torchvision.datasets.CIFAR10(
        root='./data', train=True, download=True, transform=transform)
    testset = torchvision.datasets.CIFAR10(
        root='./data', train=False, download=True, transform=transform)
```

```
    total_train = len(full_trainset)  # 50000
    val_count = int(0.1 * total_train)  # 10% for validation
    train_count = total_train - val_count

    train_subset, val_subset = random_split(
        full_trainset, [train_count, val_count],
        generator=torch.Generator().manual_seed(42)
    )

    train_loader = DataLoader(train_subset, batch_size=batch_size,
                              shuffle=True, drop_last=False)
    val_loader = DataLoader(val_subset, batch_size=batch_size,
                            shuffle=False, drop_last=False)
    test_loader = DataLoader(testset, batch_size=batch_size,
                             shuffle=False, drop_last=False)
    return train_loader, val_loader, test_loader

def save_results(results_dict, filename):
    with open(filename, 'wb') as f:
        pickle.dump(results_dict, f)

# Architectures per homework specs (valid padding -> 0, stride=1 for conv,
stride=2 for pooling)
class MLP1(nn.Module):
    def __init__(self):
        super(MLP1, self).__init__()
        self.fc1 = nn.Linear(3*32*32, 32)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(32, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

class MLP2(nn.Module):
    def __init__(self):
        super(MLP2, self).__init__()
        self.fc1 = nn.Linear(3*32*32, 32)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(32, 64, bias=False)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
```

```
        x = x.view(x.size(0), -1)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class CNN3(nn.Module):
    def __init__(self):
        super(CNN3, self).__init__()
        # (3,32,32)->(16,30,30)->(8,26,26)->pool->(8,13,13)->(16,7,7)->pool-
>(16,3,3)->flatten(144)->fc(10)
        self.conv1 = nn.Conv2d(3,16,kernel_size=3,stride=1,padding=0)
        self.conv2 = nn.Conv2d(16,8,kernel_size=5,stride=1,padding=0)
        self.pool1 = nn.MaxPool2d(kernel_size=2,stride=2)
        self.conv3 = nn.Conv2d(8,16,kernel_size=7,stride=1,padding=0)
        self.pool2 = nn.MaxPool2d(kernel_size=2,stride=2)
        self.fc = nn.Linear(16*3*3, 10)

    def forward(self,x):
        x = nn.functional.relu(self.conv1(x))
        x = nn.functional.relu(self.conv2(x))
        x = self.pool1(x)
        x = nn.functional.relu(self.conv3(x))
        x = self.pool2(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

class CNN4(nn.Module):
    def __init__(self):
        super(CNN4, self).__init__()
        # (3,32,32)->conv1->(16,30,30)->conv2->(8,28,28)->conv3->(16,24,24)-
>pool->(16,12,12)->conv4->(16,8,8)->pool->(16,4,4)->flatten(256)->fc(10)
        self.conv1 = nn.Conv2d(3,16,kernel_size=3,stride=1,padding=0)
        self.conv2 = nn.Conv2d(16,8,kernel_size=3,stride=1,padding=0)
        self.conv3 = nn.Conv2d(8,16,kernel_size=5,stride=1,padding=0)
        self.pool1 = nn.MaxPool2d(kernel_size=2,stride=2)
        self.conv4 = nn.Conv2d(16,16,kernel_size=5,stride=1,padding=0)
        self.pool2 = nn.MaxPool2d(kernel_size=2,stride=2)
        self.fc = nn.Linear(16*4*4,10)

    def forward(self,x):
        x = nn.functional.relu(self.conv1(x))
        x = nn.functional.relu(self.conv2(x))
        x = nn.functional.relu(self.conv3(x))
```

```
        x = self.pool1(x)
        x = nn.functional.relu(self.conv4(x))
        x = self.pool2(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

class CNN5_Debug(nn.Module):
    """
    CNN5 as specified:
    [Conv-3×3×8, ReLU,
     Conv-3×3×16, ReLU, Conv-3×3×8, ReLU, Conv-3×3×16, ReLU, MaxPool-2×2,
     Conv-3×3×16, ReLU, Conv-3×3×8, ReLU, MaxPool-2×2] + [FC10]

    Where:
    - Convolutions have kernel_size=3, stride=1, padding=0 (valid)
    - MaxPool has kernel_size=2, stride=2 (valid)
    - All activations are ReLU
    """

    def __init__(self):
        super(CNN5_Debug, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=8,
                               kernel_size=3, stride=1, padding=0)
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,
                               kernel_size=3, stride=1, padding=0)
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=8,
                               kernel_size=3, stride=1, padding=0)
        self.conv4 = nn.Conv2d(in_channels=8, out_channels=16,
                               kernel_size=3, stride=1, padding=0)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv5 = nn.Conv2d(in_channels=16, out_channels=16,
                               kernel_size=3, stride=1, padding=0)
        self.conv6 = nn.Conv2d(in_channels=16, out_channels=8,
                               kernel_size=3, stride=1, padding=0)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Based on dimension math:
        #  Input: (3,32,32)
        #  after conv1 => (8, 30,30)
        #  after conv2 => (16,28,28)
        #  after conv3 => (8,26,26)
        #  after conv4 => (16,24,24)
```

```
        #  after pool1 => (16,12,12)
        #  after conv5 => (16,10,10)
        #  after conv6 => (8, 8, 8)
        #  after pool2 => (8, 4, 4)
        #  flatten => 8 * 4 * 4 = 128
        self.fc = nn.Linear(8 * 4 * 4, 10)

    def forward(self, x):
        x = nn.functional.relu(self.conv1(x))
        # Debug shape
        # print("After conv1:", x.shape)  # Should be (batch, 8, 30, 30)

        x = nn.functional.relu(self.conv2(x))
        # print("After conv2:", x.shape)  # Should be (batch, 16, 28, 28)

        x = nn.functional.relu(self.conv3(x))
        # print("After conv3:", x.shape)  # Should be (batch, 8, 26, 26)

        x = nn.functional.relu(self.conv4(x))
        # print("After conv4:", x.shape)  # Should be (batch, 16, 24, 24)

        x = self.pool1(x)
        # print("After pool1:", x.shape) # Should be (batch, 16, 12, 12)

        x = nn.functional.relu(self.conv5(x))
        # print("After conv5:", x.shape) # (batch,16,10,10)

        x = nn.functional.relu(self.conv6(x))
        # print("After conv6:", x.shape) # (batch,8,8,8)

        x = self.pool2(x)
        # print("After pool2:", x.shape) # (batch,8,4,4)

        x = x.view(x.size(0), -1)  # => (batch,128)
        # print("Flatten shape:", x.shape)

        x = self.fc(x)
        return x


models = {
    'mlp_1': MLP1(),
    'mlp_2': MLP2(),
    'cnn_3': CNN3(),
    'cnn_4': CNN4(),
```

```
    'cnn_5': CNN5_Debug()
}

def evaluate_model(model, loader, device):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for x, y in loader:
            # Move to device
            x, y = x.to(device), y.to(device)
            outputs = model(x)
            _, predicted = torch.max(outputs.data, 1)
            total += y.size(0)
            correct += (predicted == y).sum().item()
    return 100.0 * correct / total

def main():
    # Use GPU if available
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print("Using device:", device)

    train_loader, val_loader, test_loader = load_cifar10(batch_size=50)
    final_results = []
    EPOCHS = 15

    for name, model in models.items():
        print(f"\n--- Training {name} ---")
        model.to(device)

        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(), lr=0.001)

        loss_curve = []
        train_acc_curve = []
        val_acc_curve = []
        step_count = 0

        for epoch in range(EPOCHS):
            model.train()
            correct = 0
            total = 0
            running_loss = 0.0

            for x, y in tqdm(train_loader, desc=f"Epoch {epoch+1}/{EPOCHS}"):
```

```
                # Move inputs to device
                x, y = x.to(device), y.to(device)

                outputs = model(x)
                loss = criterion(outputs, y)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                running_loss += loss.item()
                _, preds = torch.max(outputs.data, 1)
                total += y.size(0)
                correct += (preds == y).sum().item()
                step_count += 1

                # Every 10 steps
                if step_count % 10 == 0:
                    train_loss = running_loss / 10
                    train_acc = 100.0 * correct / total
                    loss_curve.append(train_loss)
                    train_acc_curve.append(train_acc)

                    val_acc = evaluate_model(model, val_loader, device)
                    val_acc_curve.append(val_acc)

                    running_loss = 0.0

            print(f"Epoch [{epoch+1}/{EPOCHS}] : Train
Acc={100.0*correct/total:.2f}%")

        test_acc = evaluate_model(model, test_loader, device)
        print(f"Test accuracy of {name}: {test_acc:.2f}%")

        if 'mlp' in name:
            weights = model.fc1.weight.data.cpu().numpy()
        else:
            weights = model.conv1.weight.data.cpu().numpy()

        result_dict = {
            'name': name,
            'loss_curve': loss_curve,
            'train_acc_curve': train_acc_curve,
            'val_acc_curve': val_acc_curve,
            'test_acc': test_acc,
            'weights': weights
```

```
        }
        save_results(result_dict, f"part3_{name}.pkl")

        # Visualize weights
        from utils import visualizeWeights  # if not imported globally
        visualizeWeights(weights, save_dir='.', filename=f'weights_{name}')

        final_results.append(result_dict)

    # Plot comparison
    from utils import part3Plots
    part3Plots(final_results, save_dir='.', filename='all_models_performance')
    print("All architectures trained; final results saved.")

if __name__ == "__main__":
    main()
```

Appendix 4: Experimenting Activation Functions

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import numpy as np
import pickle
from torch.utils.data import random_split, DataLoader
from tqdm import tqdm

# Provided utilities
from utils import part4Plots

##########################
# 1) DATA PREPARATION
##########################

def load_cifar10(batch_size=50):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465),
                             (0.247, 0.243, 0.261)),
    ])
```

```
    full_trainset = torchvision.datasets.CIFAR10(
        root='./data',
        train=True,
        download=True,
        transform=transform
    )
    testset = torchvision.datasets.CIFAR10(
        root='./data',
        train=False,
        download=True,
        transform=transform
    )

    # 10% validation, 90% training
    total_train = len(full_trainset)  # 50,000
    val_count = int(0.1 * total_train)
    train_count = total_train - val_count

    train_subset, val_subset = random_split(
        full_trainset, [train_count, val_count],
        generator=torch.Generator().manual_seed(42)
    )

    # shuffle=True for training set
    train_loader = DataLoader(train_subset, batch_size=batch_size,
                              shuffle=True, drop_last=False)
    val_loader = DataLoader(val_subset, batch_size=batch_size,
                            shuffle=False, drop_last=False)
    test_loader = DataLoader(testset, batch_size=batch_size,
                             shuffle=False, drop_last=False)

    return train_loader, val_loader, test_loader

##########################
# 2) ARCHITECTURE DEFINITIONS
##########################

# We'll define versions of each architecture that let us pick ReLU or Sigmoid
# for consistency with question 3.1, we keep them minimal

class MLP1(nn.Module):
    def __init__(self, activation='relu'):
        super().__init__()
        self.fc1 = nn.Linear(3*32*32, 32)
        self.fc2 = nn.Linear(32, 10)
```

```
        self.activation_type = activation

    def forward(self, x):
        x = x.view(x.size(0), -1)
        if self.activation_type == 'sigmoid':
            x = torch.sigmoid(self.fc1(x))
        else:
            x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x

class MLP2(nn.Module):
    def __init__(self, activation='relu'):
        super().__init__()
        self.fc1 = nn.Linear(3*32*32, 32)
        self.fc2 = nn.Linear(32, 64, bias=False)
        self.fc3 = nn.Linear(64, 10)
        self.activation_type = activation

    def forward(self, x):
        x = x.view(x.size(0), -1)
        if self.activation_type == 'sigmoid':
            x = torch.sigmoid(self.fc1(x))
            x = torch.sigmoid(self.fc2(x))
        else:
            x = nn.functional.relu(self.fc1(x))
            x = nn.functional.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class CNN3(nn.Module):
    def __init__(self, activation='relu'):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=0)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=5, padding=0)
        self.pool1 = nn.MaxPool2d(2,2)
        self.conv3 = nn.Conv2d(8, 16, kernel_size=7, padding=0)
        self.pool2 = nn.MaxPool2d(2,2)
        self.fc = nn.Linear(16*3*3, 10)
        self.activation_type = activation

    def forward(self, x):
        if self.activation_type == 'sigmoid':
            x = torch.sigmoid(self.conv1(x))
            x = torch.sigmoid(self.conv2(x))
```

```
        else:
            x = nn.functional.relu(self.conv1(x))
            x = nn.functional.relu(self.conv2(x))
        x = self.pool1(x)

        if self.activation_type == 'sigmoid':
            x = torch.sigmoid(self.conv3(x))
        else:
            x = nn.functional.relu(self.conv3(x))
        x = self.pool2(x)

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

class CNN4(nn.Module):
    def __init__(self, activation='relu'):
        super().__init__()
        self.conv1 = nn.Conv2d(3,16,kernel_size=3,padding=0)
        self.conv2 = nn.Conv2d(16,8,kernel_size=3,padding=0)
        self.conv3 = nn.Conv2d(8,16,kernel_size=5,padding=0)
        self.pool1 = nn.MaxPool2d(2,2)
        self.conv4 = nn.Conv2d(16,16,kernel_size=5,padding=0)
        self.pool2 = nn.MaxPool2d(2,2)
        self.fc = nn.Linear(16*4*4,10)
        self.activation_type = activation

    def forward(self, x):
        if self.activation_type == 'sigmoid':
            x = torch.sigmoid(self.conv1(x))
            x = torch.sigmoid(self.conv2(x))
            x = torch.sigmoid(self.conv3(x))
        else:
            x = nn.functional.relu(self.conv1(x))
            x = nn.functional.relu(self.conv2(x))
            x = nn.functional.relu(self.conv3(x))
        x = self.pool1(x)

        if self.activation_type == 'sigmoid':
            x = torch.sigmoid(self.conv4(x))
        else:
            x = nn.functional.relu(self.conv4(x))
        x = self.pool2(x)

        x = x.view(x.size(0), -1)
```

```
        x = self.fc(x)
        return x

class CNN5(nn.Module):
    def __init__(self, activation='relu'):
        super().__init__()
        self.conv1 = nn.Conv2d(3,8,kernel_size=3,padding=0)
        self.conv2 = nn.Conv2d(8,16,kernel_size=3,padding=0)
        self.conv3 = nn.Conv2d(16,8,kernel_size=3,padding=0)
        self.conv4 = nn.Conv2d(8,16,kernel_size=3,padding=0)
        self.pool1 = nn.MaxPool2d(2,2)
        self.conv5 = nn.Conv2d(16,16,kernel_size=3,padding=0)
        self.conv6 = nn.Conv2d(16,8,kernel_size=3,padding=0)
        self.pool2 = nn.MaxPool2d(2,2)
        self.fc = nn.Linear(8*4*4,10)
        self.activation_type = activation

    def forward(self, x):
        if self.activation_type == 'sigmoid':
            x = torch.sigmoid(self.conv1(x))
            x = torch.sigmoid(self.conv2(x))
            x = torch.sigmoid(self.conv3(x))
            x = torch.sigmoid(self.conv4(x))
        else:
            x = nn.functional.relu(self.conv1(x))
            x = nn.functional.relu(self.conv2(x))
            x = nn.functional.relu(self.conv3(x))
            x = nn.functional.relu(self.conv4(x))
        x = self.pool1(x)

        if self.activation_type == 'sigmoid':
            x = torch.sigmoid(self.conv5(x))
            x = torch.sigmoid(self.conv6(x))
        else:
            x = nn.functional.relu(self.conv5(x))
            x = nn.functional.relu(self.conv6(x))
        x = self.pool2(x)

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

##########################
# 3) TRAIN/EVAL for PART 4
##########################
```

```python
def compute_gradient_magnitude(layer):
    """
    Return the norm of the gradient of the given layer's weights as a single
float.
    If there's no gradient yet, returns 0.
    """
    if layer.weight.grad is None:
        return 0.0
    return layer.weight.grad.data.norm().item()

def train_activation_model(model, train_loader, first_layer, epochs=15):
    """
    Trains a model using:
      - SGD with lr=0.01, momentum=0.0
      - batch_size=50
    Records:
      - 'loss_curve' every 10 steps
      - 'grad_curve' every 10 steps (magnitude of grad on first_layer)
    Returns (loss_curve, grad_curve).
    """
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.0)

    loss_curve = []
    grad_curve = []
    step_count = 0

    for epoch in range(epochs):
        model.train()
        for (x, y) in train_loader:
            step_count += 1
            optimizer.zero_grad()
            outputs = model(x)
            loss = criterion(outputs, y)
            loss.backward()
            # record every 10 steps
            if step_count % 10 == 0:
                loss_curve.append(loss.item())
                grad_magnitude = compute_gradient_magnitude(first_layer)
                grad_curve.append(grad_magnitude)
            optimizer.step()

    return loss_curve, grad_curve
```

```
def part4_experiment(arch_name, model_relu, model_sigmoid, first_layer_relu,
first_layer_sigmoid,
                     train_loader, epochs=15):
    """
    Trains one architecture with ReLU and Sigmoid, returns dictionary with:
      'name', 'relu_loss_curve', 'sigmoid_loss_curve',
              'relu_grad_curve', 'sigmoid_grad_curve'
    """
    # Train ReLU
    print(f"Training {arch_name} - ReLU version")
    relu_loss_curve, relu_grad_curve = train_activation_model(model_relu,
train_loader,
                                                              first_layer_relu,
epochs=epochs)

    # Train Sigmoid
    print(f"Training {arch_name} - Sigmoid version")
    sigmoid_loss_curve, sigmoid_grad_curve =
train_activation_model(model_sigmoid, train_loader,
                                                                  first_layer_s
igmoid, epochs=epochs)

    # Create dictionary
    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
    }
    return result_dict

##########################
# 4) MAIN
##########################
def main():
    # Load CIFAR10
    train_loader, val_loader, test_loader = load_cifar10(batch_size=50)
    print("Loaded CIFAR-10 with batch size=50 for Part 4...")

    # We'll do 15 epochs as stated
    EPOCHS = 15

    # Arch definitions for ReLU vs Sigmoid
    archs = {
```

```
      'mlp_1': (MLP1('relu'), MLP1('sigmoid')),
      'mlp_2': (MLP2('relu'), MLP2('sigmoid')),
      'cnn_3': (CNN3('relu'), CNN3('sigmoid')),
      'cnn_4': (CNN4('relu'), CNN4('sigmoid')),
      'cnn_5': (CNN5('relu'), CNN5('sigmoid')),
    }

    results = []

    for arch_name, (model_relu, model_sigmoid) in archs.items():
        print(f"\n=== Running Part4 experiment for {arch_name} ===")

        # Identify the first layer
        # For MLP: first layer is fc1
        # For CNN: first layer is conv1
        if 'mlp' in arch_name:
            first_layer_relu = model_relu.fc1
            first_layer_sigmoid = model_sigmoid.fc1
        else:
            first_layer_relu = model_relu.conv1
            first_layer_sigmoid = model_sigmoid.conv1

        # part4_experiment
        arch_dict = part4_experiment(arch_name,
                                     model_relu,
                                     model_sigmoid,
                                     first_layer_relu,
                                     first_layer_sigmoid,
                                     train_loader,
                                     epochs=EPOCHS)
        # save the dictionary
        with open(f'part4_{arch_name}.pkl', 'wb') as f:
            pickle.dump(arch_dict, f)
        results.append(arch_dict)

    # Now we use part4Plots
    print("\nGenerating Part4 performance comparison plots...")
    part4Plots(results, save_dir='.', filename='part4_performance')

    print("Part4 experiment completed! All results saved and plot generated.")

if __name__ == "__main__":
    main()
```

Appendix 5: Experimenting Learning Rate

```
# part5.py
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import pickle
import numpy as np
from torch.utils.data import random_split, DataLoader
from tqdm import tqdm

from utils import part5Plots

############################################################
# 1) DATA LOADING
############################################################
def load_cifar10(batch_size=50):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465),
                             (0.247, 0.243, 0.261)),
    ])
    full_trainset = torchvision.datasets.CIFAR10(
        root='./data', train=True, download=True, transform=transform)
    testset = torchvision.datasets.CIFAR10(
        root='./data', train=False, download=True, transform=transform)

    total_train = len(full_trainset)  # 50,000
    val_count = int(0.1 * total_train)
    train_count = total_train - val_count

    gen = torch.Generator().manual_seed(42)
    train_subset, val_subset = random_split(full_trainset,
                                            [train_count, val_count],
                                            generator=gen)

    train_loader = DataLoader(train_subset, batch_size=batch_size,
                              shuffle=True, drop_last=False)
    val_loader = DataLoader(val_subset, batch_size=batch_size,
                            shuffle=False, drop_last=False)
    test_loader = DataLoader(testset, batch_size=batch_size,
                             shuffle=False, drop_last=False)
    return train_loader, val_loader, test_loader
```

```
############################################################
# 2) ARCHITECTURE (CNN3)
############################################################
class CNN3(nn.Module):
    def __init__(self):
        super(CNN3, self).__init__()
        # [Conv-3×3×16, ReLU,
        #  Conv-5×5×8, ReLU, MaxPool-2×2,
        #  Conv-7×7×16, MaxPool-2×2] + FC10
        self.conv1 = nn.Conv2d(3,16,kernel_size=3,stride=1,padding=0)
        self.conv2 = nn.Conv2d(16,8,kernel_size=5,stride=1,padding=0)
        self.pool1 = nn.MaxPool2d(kernel_size=2,stride=2)
        self.conv3 = nn.Conv2d(8,16,kernel_size=7,stride=1,padding=0)
        self.pool2 = nn.MaxPool2d(kernel_size=2,stride=2)
        # dimension => (16,3,3) => flatten=144
        self.fc = nn.Linear(16*3*3, 10)

    def forward(self,x):
        x = torch.relu(self.conv1(x))   # (16,30,30)
        x = torch.relu(self.conv2(x))   # (8,26,26)
        x = self.pool1(x)              # (8,13,13)
        x = torch.relu(self.conv3(x))   # (16,7,7)
        x = self.pool2(x)              # (16,3,3)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

############################################################
# 3) EVALUATION
############################################################
def evaluate(model, loader, device):
    model.eval()
    total, correct = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            outputs = model(x)
            _, predicted = torch.max(outputs.data, 1)
            total += y.size(0)
            correct += (predicted == y).sum().item()
    return 100.0 * correct / total

############################################################
# 4) TRAIN FUNCTION
```

```
############################################################
def train_model(lr, train_loader, val_loader, model, device, epochs=20):
    """
    Trains with:
     - SGD(lr=lr, momentum=0.0)
     - batch_size=50
    Records training loss + validation accuracy every 10 steps
    Returns (loss_curve, val_acc_curve)
    """
    model.to(device)
    optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.0)
    criterion = nn.CrossEntropyLoss()

    loss_curve = []
    val_acc_curve = []
    step_count = 0

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        for (x, y) in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}
(LR={lr})"):
            x, y = x.to(device), y.to(device)

            optimizer.zero_grad()
            outputs = model(x)
            loss = criterion(outputs, y)
            loss.backward()
            optimizer.step()

            step_count += 1
            running_loss += loss.item()

            # record every 10 steps
            if step_count % 10 == 0:
                avg_loss = running_loss / 10
                loss_curve.append(avg_loss)
                running_loss = 0.0
                val_acc = evaluate(model, val_loader, device)
                val_acc_curve.append(val_acc)

        print(f"Epoch [{epoch+1}/{epochs}] LR={lr} finished.")

    return loss_curve, val_acc_curve
```

```
############################################################
# 5) MAIN
############################################################
def main():
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print("Using device:", device)

    # 5a) Load data
    train_loader, val_loader, test_loader = load_cifar10(batch_size=50)
    EPOCHS = 20

    # 5b) Instantiate CNN3 for each LR
    model_1 = CNN3()
    model_01 = CNN3()
    model_001 = CNN3()

    print("\n--- Training with LR=0.1 ---")
    loss_curve_1, val_acc_curve_1 = train_model(0.1, train_loader, val_loader,
model_1, device, EPOCHS)

    print("\n--- Training with LR=0.01 ---")
    loss_curve_01, val_acc_curve_01 = train_model(0.01, train_loader, val_loader,
model_01, device, EPOCHS)

    print("\n--- Training with LR=0.001 ---")
    loss_curve_001, val_acc_curve_001 = train_model(0.001, train_loader,
val_loader, model_001, device, EPOCHS)

    # Create dictionary
    result_dict = {
        'name': 'cnn3',
        'loss_curve_1': loss_curve_1,
        'loss_curve_01': loss_curve_01,
        'loss_curve_001': loss_curve_001,
        'val_acc_curve_1': val_acc_curve_1,
        'val_acc_curve_01': val_acc_curve_01,
        'val_acc_curve_001': val_acc_curve_001
    }

    # Save
    with open("part5_cnn3.pkl", "wb") as f:
        pickle.dump(result_dict, f)

    # Plot
    part5Plots(result_dict, save_dir='.', filename='part5_lr_comparison')
```

```
    print("\nPart5 - LR experiment (0.1, 0.01, 0.001) completed. Plots saved.")

    # ------------------------------------------------------------
    # 5c) Manual scheduled LR approach
    # ------------------------------------------------------------
    print("\n--- Scheduled Learning Rate Approach ---")

    scheduled_model = CNN3().to(device)
    optimizer_sch = optim.SGD(scheduled_model.parameters(), lr=0.1, momentum=0.0)
    criterion = nn.CrossEntropyLoss()

    # We'll store only the validation accuracy
    scheduled_val_acc = []

    # first "plateau" ~ epoch=5 (as an example)
    plateau_epoch_1 = 5
    for epoch in range(plateau_epoch_1):
        scheduled_model.train()
        for (x, y) in train_loader:
            x, y = x.to(device), y.to(device)
            optimizer_sch.zero_grad()
            outputs = scheduled_model(x)
            loss = criterion(outputs, y)
            loss.backward()
            optimizer_sch.step()
        val_acc_sch = evaluate(scheduled_model, val_loader, device)
        scheduled_val_acc.append(val_acc_sch)
        print(f"Scheduled LR=0.1, Epoch {epoch+1}/{plateau_epoch_1}, Val
Acc={val_acc_sch:.2f}%")

    # now set LR=0.01
    for g in optimizer_sch.param_groups:
        g['lr'] = 0.01

    # continue training until ~ epoch=15
    plateau_epoch_2 = 15
    for epoch in range(plateau_epoch_1, plateau_epoch_2):
        scheduled_model.train()
        for (x, y) in train_loader:
            x, y = x.to(device), y.to(device)
            optimizer_sch.zero_grad()
            outputs = scheduled_model(x)
            loss = criterion(outputs, y)
            loss.backward()
            optimizer_sch.step()
```

```
        val_acc_sch = evaluate(scheduled_model, val_loader, device)
        scheduled_val_acc.append(val_acc_sch)
        print(f"Scheduled LR=0.01, Epoch {epoch+1}/{plateau_epoch_2}, Val
Acc={val_acc_sch:.2f}%")

    # set LR=0.001
    for g in optimizer_sch.param_groups:
        g['lr'] = 0.001

    # train until epoch=30
    for epoch in range(plateau_epoch_2, 30):
        scheduled_model.train()
        for (x, y) in train_loader:
            x, y = x.to(device), y.to(device)
            optimizer_sch.zero_grad()
            outputs = scheduled_model(x)
            loss = criterion(outputs, y)
            loss.backward()
            optimizer_sch.step()
        val_acc_sch = evaluate(scheduled_model, val_loader, device)
        scheduled_val_acc.append(val_acc_sch)
        print(f"Scheduled LR=0.001, Epoch {epoch+1}/30, Val
Acc={val_acc_sch:.2f}%")

    final_test_acc_sch = evaluate(scheduled_model, test_loader, device)
    print(f"Final Test Accuracy with scheduled LR approach:
{final_test_acc_sch:.2f}%")

    print("Part5 scheduled LR approach done.")

if __name__ == "__main__":
    main()
```