CS 4/591: Neural Network:
Implementing and Training Convolutional Neural Networks

Jyrus Cadman Sho Komiyama Robert McCourt Bethany Peña
Gabriel Urbaitis

12 December 2024

Contents

1 Introduction

1.1 Project Overview

This project presents our implementation of the LeNet-5 architecture, focusing on understanding the fundamental structure and training processes of Convolutional Neural Networks (CNNs). Built from scratch using only NumPy for core computations, our implementation encompasses the complete LeNet-5 structure with two convolutional layers (using ReLU activations), two max pooling layers, and a fully connected neural network with two hidden layers and one output layer.

Our implementation is distinguished by its detailed attention to core CNN components, particularly in the convolutional layers where we've implemented 3D convolutions with padding in the first two dimensions. The first convolutional layer employs 6 filters (each $5 \times 5 \times k$, where k represents input image color channels), while the second layer utilizes 16 filters (each $5 \times 5 \times 6$). Both max pooling layers are implemented with a 2×2 kernel size and stride of 2, maintaining the classic LeNet-5 architecture while adapting it for both MNIST and CIFAR-10 datasets.

2 CNN Architecture and Implementation

2.1 LeNet-5 Structure Overview

Our implementation follows the classic LeNet-5 architecture, modified to handle both grayscale (MNIST) and RGB (CIFAR-10) inputs through an adaptable input channel parameter. The network consists of a sequence of convolutional, pooling, and fully connected layers arranged in a feed-forward structure.

The complete network architecture can be described as:

1. Input Layer: $(batch\_size \times channels \times 32 \times 32)$
2. First Convolutional Block:
   - Conv1: 6 filters with 5×5 kernel, stride 1
   - ReLU activation
   - MaxPool1: 2×2 kernel, stride 2
3. Second Convolutional Block:
   - Conv2: 16 filters with 5×5 kernel, stride 1
   - ReLU activation
   - MaxPool2: 2×2 kernel, stride 2
4. Fully Connected Layers:
   - FC1: $400 \rightarrow 120$ with ReLU
   - FC2: $120 \rightarrow 84$ with ReLU
   - FC3: $84 \rightarrow 10$ (output layer)

2.2 Data Structures and Class Implementation

2.2.1 CNN Class Design

Our implementation utilizes a modular class structure for each layer type. The convolutional layer implementation is particularly noteworthy:

```
class Conv3d:
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        self.filters = np.random.uniform(low=-1, high=1,
            size=(out_channels, in_channels, kernel_size, kernel_size)) / (kernel_size * kernel_size)
        self.biases = np.ones(out_channels)
```

This design enables efficient handling of 3D convolutions while maintaining clear separation of concerns.

2.2.2 Layer Representations

The forward propagation through the network is implemented with careful attention to dimensionality:

1. **Convolutional Operation**: For an input volume X and filter F, the convolution operation is defined as:

$$Y_{i,j} = \sum_{c=0}^{C-1} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X_{c,i+m,j+n} \cdot F_{c,m,n} + b$$

```
    def conv3d(self, input, filters, biases, stride=1, padding=0):
        # Shape calculations
        batch_size, input_channels, input_height, input_width = self.input.shape
        num_filters, _, kernel_height, kernel_width = filters.shape
        out_height = (input_height - kernel_height) // stride + 1
        out_width = (input_width - kernel_width) // stride + 1

        output = np.zeros((batch_size, num_filters, out_height, out_width))
        # Convolution implementation
        # ...
```

There are two operations involved in the convolution, conv2d and conv3d.

Conv2d convolves a single 2D input channel with a single 2d kernel. It does this by moving the kernel across the input in steps defined by the stride. At each step the values in each region are multiplied elementwise, summed together and added to the bias for each kernel. Each result is concatenated into a feature map.

Conv3d calls conv2d for each of it's separate channels and corresponding kernel in each filter's kernel set and sums them all up together to generate a single 2D feature map for each filter. The output is 4D, the first dimension for the batch of inputs, the second for the number of filters as each one produces its own feature map, and the last two for the height and width of each feature map.

2. **Max Pooling Operation**: The max pooling operation is defined as:

$$Y_{i,j} = \max_{0 \leq m < k, 0 \leq n < k} X_{i \cdot s+m, j \cdot s+n}$$

where k is the kernel size and s is the stride.

2.3 Forward Propagation Implementation

Forward propagation is implemented as a sequence of layer-wise operations, with each layer maintaining its state for backpropagation. The complete forward pass combines convolution, activation, and pooling operations:

```
def forward(self, x):
    x = self.c1.forward(x)      # First convolution
    x = self.r1.forward(x)      # ReLU activation
    x = self.s2.forward(x)      # Max pooling

    x = self.c3.forward(x)      # Second convolution
    x = self.r3.forward(x)      # ReLU activation
    x = self.s4.forward(x)      # Max pooling

    batch_size = x.shape[0]
    x = x.reshape(batch_size, -1)  # Flatten for FC layers

    x = self.fc1.forward(x)     # Fully connected layers
    x = self.fc2.forward(x)
    x = self.fc3.forward(x)
    return x
```

This implementation ensures efficient forward propagation while maintaining all necessary information for the subsequent backpropagation phase.

3 Backpropagation Analysis

The LeNet-5 Convolutional Neural Network primarily consists of three types of layers: convolution, ReLU activation, and max-pooling. These layers progressively extract features from the input, which are then passed through a fully connected feed-forward network for classification output. Backpropagation through the ReLU activation function is straightforward, as it behaves similarly to the activation functions in traditional neural networks. The remainder of this section will focus on the backpropagation algorithms for the convolutional and max-pooling layers, which involve more complex computations.

3.0.1 Convolution Layers

During the backward pass, our goal is to calculate the gradient of the loss with respect to the filters ($\frac{\partial L}{\partial F}$). To backpropogate through the convolutional layers we can use the following equation:

$$\frac{\partial L}{\partial F} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial F}$$

In a convolution layer, the output, Y is computed by convolving the input X with the filter F. To find how changes in the filter F affect the loss, we treat the output, Y as an intermediate variable and divide the gradient computation into two parts. First, $\frac{\partial L}{\partial Y}$ measures how sensitive the loss is to changes in the ouput of the convolution. Second, $\frac{\partial Y}{\partial F}$ measures how changes in the filter values alter the layer's output. By linking these two components together via the chain rule, we can obtain the equation stated above.

The gradient $\frac{\partial L}{\partial Y}$ is the derivative of the loss L with respect to the output of the convolutional layer, and it is given to us as part of the backpropagation process. During the forward pass, the output of the convolutional layer Y is computed by convolving the input X with the filters F followed by an activation function such as ReLU

$$Y = ReLU(X * F + b)$$

where b represents the bias. This gradient is passed from the next layer in the network, and is used to update the parameters of the convolutional layer.

The term $\frac{\partial Y}{\partial F}$ represents the gradient of the output Y with respect to the filters F in the convolutional layer. When performing backpropagation, we need to compute how the output changes with respect to changes in the filter. For each position in the filter applied to the input, the output is computed by "sliding" the filter over the input, and computing the dot product at each position. This convolution operation determines the value of Y for each spatial location. We the value of an output cell, Y, by using the following equation:

$$Y_{ij} = \sum_{m=0}^{H-1} \sum_{n=0}^{W-1} X_{i+m,j+n} F_{m,n}$$

where H and W are the height and width of the filter F.

Since the output Y at a specific position is determined by the convolution operation between the input X and the filter F, the gradient of Y with respect to F at a particular position is the same as the input X at that position.

This result is then passed to the ReLU layer, and ReLU activation is applied to each cell in the matrix.

To compute the final gradient of the loss with respect to the filters, $\frac{\partial L}{\partial F}$, we combine the two terms discussed previously. Here, $\frac{\partial L}{\partial Y}$ is passed to the current layer from the next, which tells us how much the loss changes with respect to the output. $\frac{\partial Y}{\partial F}$ is the gradient of the output with respect to the filter, which is determined by the convolution operation. The gradient $\frac{\partial Y}{\partial F}$ at a given location is simply the input X at that location. So, to calculate $\frac{\partial L}{\partial F}$, we convolve the input X with the gradient of the loss with respect to the output:

$$\frac{\partial L}{\partial F} = X * \frac{\partial L}{\partial Y}$$

In other words, we take the derivative of the loss from the next layer and convolve it with the input X, and sum over all positions where the filter was applied. This is how we are able to backpropogate the error across all convolutional layers. This results in the total gradient of the loss with respect to the filter, whcih tells us how the filter should be updated to minimize the loss.

3.0.2 Max Pooling Layers

During the backward pass, the purpose of backpropagation in a max pooling layer is to compute the gradient of the loss with respect to the input of the layer. This information allows earlier layers in the network to adjust their parameters effectively. When there is no overlap between pools, we simply need to identify which unit is the maximum value in the pool. During backpropagation, the gradient of the loss with respect to the max pooling output is propagated only to the stored indices of the maximum values. All other values in the pooling window receive a gradient of zero.

This process is repeated independently for each channel in the input feature map. The backpropagation happens separately for each channel, ensuring the gradient for each channel is handled individually while still using the same max pooling operation.

```
dL_dinput = np.zeros_like(self.input)
```

During the forward pass, the indices of the maximum values within each pooling window were stored. In the backward pass, the gradients from `dL_dout` are propagated only to the positions of these maximum values

```
max_i, max_j = self.max_indices[b, c, i, j]
dL_dinput[b, c, max_i, max_j] += dL_dout[b, c, i, j]
```

Here, `dL_dout[b,c,i,j]` represents the gradient of the loss with respect to the output of the pooling layer at a specific location. This gradient is added only to the corresponding position in `dL_dinput` that matches the stores indices (`max_i, max_j`). Because no other positions in the pooling window are updated, their gradients remain zero.

3.0.3 Integration Across All Layers

In the convolutional layers, backpropagation computes the gradient of the loss with respect to the filters, which allows the model to learn which features are most important. The error signal is passed from the next layer, and the filter gradients are computed using the chain rule. In the forward pass, the convolution produces feature maps by applying filters to the input, and in the backward pass, the gradients are used to adjust the filter weights to improve feature extraction in the subsequent passes.

ReLU layers introduce non-linearity, which is essential for learning more complex patterns. This mechanism is in place to ensure only relevant features contribute to learning, enabling the model to focus on the most important patterns in the data.

Max pooling layers serve to downsample the feature maps and retain the most important data by selecting the maximum value within each pooling window. During backpropagation, the gradients are only propagated to the positions corresponding the maximum values in each window. In the backward pass, this operation helps the model retain the most significant features.

Backpropagation across these layers allows the network to adjust its filters and weights effectively, to learn both simple and complex representations of the input data. As the error is propagated backward through the network, each layer is able to update its parameters to gradually improve the model's ability to map inputs to accurate predictions.

4 Proofs

4.1 Proof 1

Consider:

A proof to clarify that the (3D) size of $\partial L/\partial F$ computed by the method taught in our class is always same as the size of F for each filter F. We assume that the filter size is $m \times m \times k$.

Proof. We are given a convolutional filter F of size $m \times m \times k$, where:

- $m \times m$ represents the spatial dimensions (height $\times$ width)
- k represents the number of channels

We need to prove that $\partial L/\partial F$ (the gradient of loss L with respect to filter F) has these same dimensions.

First, let's consider our forward pass understanding, as implemented in `MaxPool2d`.

During the forward pass, we have:

1. The filter that slides across the input image
2. At each position, every filter element is used exactly once
3. The number of times this happens is determined by these formulas:

```
H_out = ((H_in - kernel_size) // stride) + 1
W_out = ((W_in - kernel_size) // stride) + 1
```

These formulas tell us:

- How many output positions we'll have (`H_out` $\times$ `W_out`)
- How many times each filter element will be used
- Each use of a filter element contributes to one output position Y_{ij}

Now, let's consider the gradient computation by the method taught in class.

Recall:

$$\text{For every element } F_{ij} \text{ we have that, } \frac{\partial L}{\partial F_{ij}} = \sum_{k_1}\sum_{k_2} \frac{\partial L}{\partial Y_{k_1 k_2}} \cdot \frac{\partial Y_{k_1 k_2}}{\partial F_{ij}}$$

where k_1 ranges from 1 to `H_out` and k_2 ranges from 1 to `W_out`.

Let's break this down with a concrete example shown in class.

For filter element F_{11}:

$$\frac{\partial L}{\partial F_{11}} = \frac{\partial L}{\partial Y_{11}}\frac{\partial Y_{11}}{\partial F_{11}} + \frac{\partial L}{\partial Y_{12}}\frac{\partial Y_{12}}{\partial F_{11}} + \frac{\partial L}{\partial Y_{21}}\frac{\partial Y_{21}}{\partial F_{11}} + \frac{\partial L}{\partial Y_{22}}\frac{\partial Y_{22}}{\partial F_{11}} \tag{1}$$

$$= \frac{\partial L}{\partial Y_{11}}X_{11} + \frac{\partial L}{\partial Y_{12}}X_{12} + \frac{\partial L}{\partial Y_{21}}X_{21} + \frac{\partial L}{\partial Y_{22}}X_{22} \tag{2}$$

Here, each term represents one time F_{11} was used in the forward pass, and X_{ij} represents the input value that F_{11} was multiplied with. Consequently, $\partial L/\partial Y_{ij}$ represents how much that output position contributed to the loss.

Hence, it's critical to understand that even though this sum has `H_out` $\times$ `W_out` terms, we add them all up, and get ONE final number. This single number then becomes the gradient for position $(1, 1)$ in our filter.

Finally, let's bring it all together and see how the dimensions are preserved.

For spatial dimensions ($m \times m$), each filter position (i, j) collects its own sum of gradients (Eqn. (1)). Despite summing many terms, each position gets exactly one final number. This naturally creates an $m \times m$ grid of gradients.

For the channel dimension (k), each channel in the filter operates independently. This means gradients flow back through each channel separately. This maintains K separate channels in the gradient.

To deepen our understanding, we can think of each filter element as having a "mailbox." During backpropagation, it receives "gradient mail" from every output position where it was used. It then adds up all this "mail" into one final number. This number goes into its position in the final gradient tensor.

Therefore, $\partial L/\partial F$ must have dimensions $m \times m \times k$ because:

1. Each position (i, j) in each channel gets exactly one gradient value
2. This happens for all $m \times m$ positions
3. It happens independently for all k channels
4. This naturally forms an $m \times m \times k$ tensor of gradients

This is why backpropagation through convolutions preserves the original filter dimensions, *ensuring our gradient updates can be directly applied to the filter during optimization.* □

4.2 Proof 2

A proof to clarify that the (3D) size of $\frac{\partial L}{\partial X}$ computed by the method taught in our class is always the same as the size of X for any input image X. We assume that the image size is $n \times n \times k$.

First we explain the case in 2D:

X has size $n \times n$, convolved with a filter F size $f \times f$. The output Y is size $(n - f + 1) \times (n - f + 1)$. On the backward pass, we need to make sure that each index of the filter overlaps with each index of $\frac{\partial L}{\partial Y}$. Thus we pad each side of each dimension of $\frac{\partial L}{\partial Y}$ with zeros of size $f - 1$, the front/top padding ensuring the last index of the filter overlaps with the first index of the input (and everything in between), and the back/bottom padding ensuring the last index of the input overlaps with the first index of the filter.

Thus the zero-padded-input $\frac{\partial L}{\partial Y}$ becomes size $((n-f+1)+(f-1)+(f-1))\times((n-f+1)+(f-1)+(f-1)) = (n + f - 1) \times (n + f - 1)$

Rotating F doesn't change the size, so F' is still size size $f \times f$, so plugging in the dimensions to the following equation: $\frac{\partial L}{\partial X} = F' \circledast \frac{\partial L}{\partial Y}$, the dimensions of $\frac{\partial L}{\partial X}$ become: $((n+f-1)-f+1)\times((n+f-1)-f+1)$ which simplify to $n \times n$.

To extend the logic to 3D:

When X has size $n \times n \times k$, the filter F has dimensions $f \times f \times k$. The convolution operation spans the entire depth (k) of the input, so no padding is needed along this dimension. The gradient $\frac{\partial L}{\partial Y}$ is extended to size $(n + f - 1) \times (n + f - 1) \times numfilters$. Since in the case of multiple filters, they are combined to update the weights, this also does not affect the dimensions of $\frac{\partial L}{\partial X}$.

When applying F' of size $f \times f \times k$ in the backward pass: $\frac{\partial L}{\partial X} = F' \circledast \frac{\partial L}{\partial Y}$, the third dimension of $\frac{\partial L}{\partial X}$ stays k, since the convolution fully spans the depth of the input. The final size of $\frac{\partial L}{\partial X}$ is $n \times n \times k$, which is the same as X.

5 Optimizations

In order to improve from Testing accuracies that fluctuated around 10 percent, we made many adjustments that led to the final implementation. One major change, was to use logsoftmax as the activation function for the final layer. We did this to make sure that the final output of forward would be a log probability for Negative Log-Likelihood (NLL) loss. This required changing y_pred to exp(y_pred) and then subtracting y from the result to get dL_dout in LeNet's backward, and using dl_dout without multiplying by the activation derivative in FNN's layer backward for the final layer to avoid gradient distortion.

6 Experimental Results

6.1 Training Details

In order to improve the performance of our model, we implemented a variety of strategies learned in class to avoid both exploding and vanishing gradients and to increase stability of our training.

We implemented gradient clipping in the backwards step, clipping the gradients from the fully connected layers to keep them between (-5,5), and the convolutional layers between (-1,1).

In addition to gradient clipping, we implemented batch normalization. This was implemented similar to ReLU, in that it was applied after each layer's forward call. Our BatchNormalization class re-centered and normalized the data as follows: $x = (x - mean)/(std + 1e - 8)$.

Finally, to avoid overfitting, we also implemented early stopping. We noticed that once model reached high training accuracy, it would begin to fluctuate, and would perform poorly in generalizing to the test dataset. To avoid this, we stop training once the model reaches 95% accuracy on the training dataset.

We used three runs of the model using 10 training epochs to compare the results of training with the ADAM optimizer and gradient descent. We used a learning rate of 0.01 for the ADAM optimizer; since it has the ability to decay its learning, we opted for a higher learning rate to start with. For gradient descent, we used 0.001, since a smaller learning rate may make it less likely to overstep the optimal solution.

6.2 Overview of Datasets

The project utilizes two benchmark datasets to train and test our LeNet5 model implementation: MNIST and CIFAR-10. To prepare our data, data_loader.py is implemented. The data_loader.py downloads and transforms the data using torchvision. Transformation involves: resizing all images to 32x32 pixels, which specifies height and width of images, to match the input dimension expected by LeNet5, batching data for training, which is shuffled, and testing, by specified batch size, and converting images to PyTorch tensors and normalizes the pixel values to the range [0, 1]. The produced data have shapes of (batch size, channel_depth, height, width), where channel_depth is 1 and 3 for MNIST and CIFAR-10 respectively. Here is what these datasets contain:

- MNIST: consists of 60,000 training and 10,000 test grayscale images(channel_depth = 1) of handwritten 10 digits(0-9)
- CIFAR-10: consists of 60,000 training and 10,000 test color images(channel_depth = 3) of 10 classes of objects (e.g. airplane, automobile, bird, cat, etc).

6.3 MNIST Dataset

The ADAM optimizer converged much quicker than gradient descent; it reached the stopping point for training within 4 epochs, as seen in figure 1. It also experienced much more successful generalization in its training, as demonstrated in figure 2.

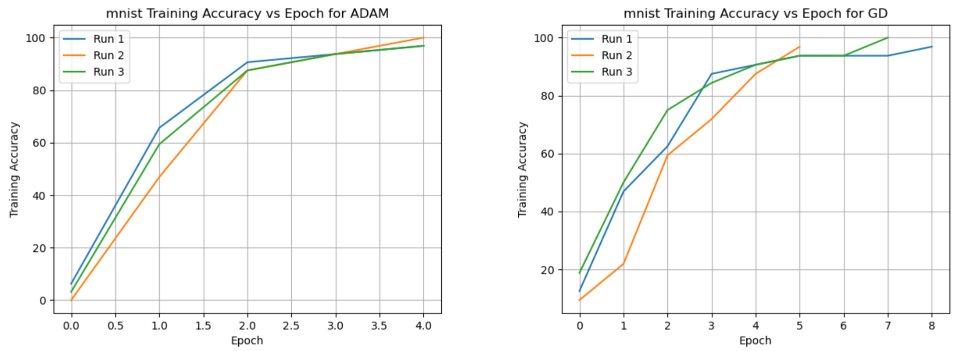

Figure 1: Comparison of Training Accuracies with MNIST for ADAM and Gradient Descent.

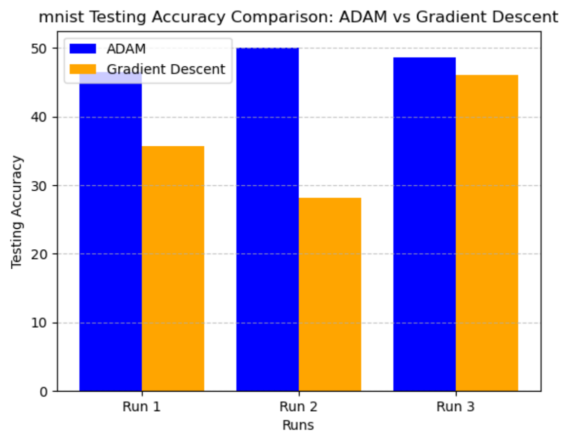

Figure 2: Comparison of Testing Accuracies with MNIST for ADAM and Gradient Descent.

6.4 CIFAR-10 Dataset

The CIFAR dataset required more training epochs for ADAM. Both ADAM and GD achieved high training accuracy, as seen in figure 3 but both methods struggled to generalize their trainings. The difference between testing accuracy achieved by ADAM and GD was less pronounced with the CIFAR dataset, but ADAM was slightly better (figure 4).

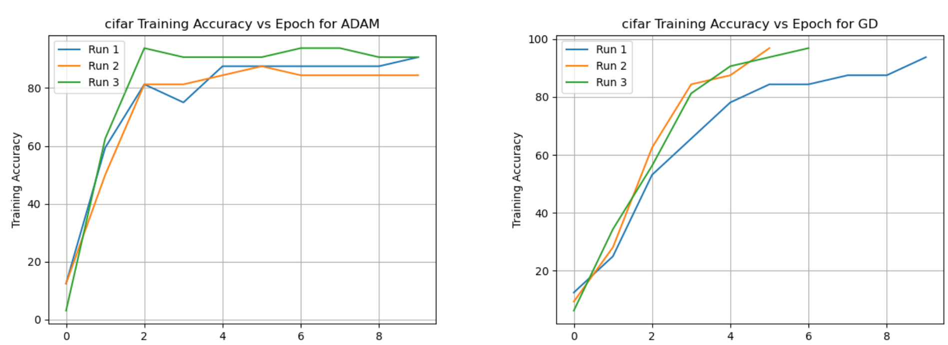

Figure 3: Comparison of Training Accuracies with MNIST for ADAM and Gradient Descent.

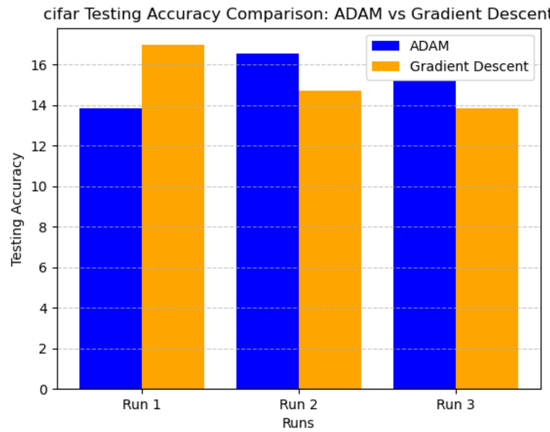

Figure 4: Comparison of Testing Accuracies with MNIST for ADAM and Gradient Descent.

7 Discussion

Overall, we were able to achieve high training performance and some generalization for the MNIST dataset, but struggled to generalize with the CIFAR set. We observed that ADAM is a more robust optimization method for our model than gradient descent.

Our model struggled to generalize with the CIFAR dataset. This dataset is a more complex dataset and classification task compared to the MNIST dataset, so this wasn't an unexpected result. In figure 3 it appears that ADAM converges around 2 epochs, so a lower early stopping threshold could possibly improve generalization accuracy.

8 Conclusion

This project successfully implemented the LeNet-5 CNN architecture from scratch using only NumPy, demonstrating both the capabilities and challenges of fundamental deep learning concepts. The implementation included complete forward and backward propagation through convolutional layers, max pooling layers, and fully connected layers, while incorporating critical optimizations such as batch normalization, gradient clipping, and early stopping.

The comparitive analysis between Adam optimizer and gradient descent revealed Adam's superior performance in both convergence speed and generalization capability, particularly with the MNIST dataset. While the model achieved strong performance on MNIST, the more complex CIFAR-10 dataset presented greater challenges, highlighting the increasing difficulty of image classification tasks with color images and more diverse object categories.

Through mathematical proofs and practical implementation, this project provided valuable insights into the core mechanisms of CNNs, establishing a solid foundation for understanding and further experimentation with deep learning architectures. The results underscore both the power of CNNs in image classification tasks and the importance of proper optimization techniques in achieving reliable model performance.

Team Contributions

Jyrus Cadman

Jyrus was responsible for establishing the foundational architecture of the CNN implementation project, including defining the initial project structure and core source code files. He implemented critical data structures and developed the `MaxPool2d` class, which handles the crucial dimensionality reduction in the LeNet-5 architecture through max pooling operations. His implementation includes efficient forward propagation logic that processes input tensors and maintains proper dimensional transformations between convolutional layers. Additionally, he worked on the mathematical proof demonstrating that the gradient of the loss with respect to a convolutional filter ($\partial L/\partial F$) maintains the same dimensions as the original filter F, providing theoretical validation for the backpropagation implementation. His work aided the subsequent implementation of backpropagation through the max pooling layers and integration with the broader CNN architecture.

Sho Komiyama

Sho was responsible for debugging, creating tests, and setting up essential components to ensure the LeNet implementation worked correctly. This include preparing `data_loader` for `MNIST` and `CIFAR-10` datasets, validating the forward pass through the network, and building a basic training sequence. The specific tasks included: developing multiple tests to verify that the convolution, ReLu, and max-pooling operations were working correctly. These tests ensures that inputs and outputs had the correct dimensions and the layers were properly connected in the forward pass. Debugging these layers was done with Gabriel's assistance. During the testing, a discrepancy was discovered between the project implementation and the example shown in Lecture 24, Slide 12. The example appeared to produce a slightly incorrect result, likely because a basis term was not applied to one element after the convolution. This confirmed that the project's implementation was accurate. Also created and tested `data_loader` to ensure that data was correctly loaded and formatted for use in training. Also designed a basic training process to validate the network functionality. The training sequence performed correctly during the forward pass but encountered an issue during backpropagation in the max-pooling layer, which revealed a disconnection between the max-pooling layer and fully connected layers, which was subsequently fixed by Bethany. Then wrote the section "Overview of Datasets."

Robert McCourt

Robert was responsible for implementing the backpropagation algorithm for the max pooling layer. Robert worked with Bethany to integrate each of their implementations of backpropagation, as well as ReLU into the project. He made modifications to the Conv3D and Conv2D functions and class to address dimensionality and shape issues, ensuring proper functionality. Robert also tested data preprocessing and conducted preliminary testing, modifying the forward function in the max pooling class to be more computationally efficient. Robert also modified the project to easily and dynamically accept the MNIST or CIFAR dataset to streamline testing, and reflect the accuracy of the model at each epoch. Robert also worked with Gabe on solutions on how to improve the accuracy of the model, such as using Xavier initialization and proper initialization of the bias, as well as contributing to the general debugging effort. Robert wrote the section on backpropagation in the report, incorporating code from both himself and Bethany and using the book as a foundational guide for his explanations.

Bethany Peña

Bethany was responsible for implementing the backpropogation algorithm for the convolutional layer, using the textbook and lecture slides as resources, which she verified with Jyrus and Gabe's proofs. She and Robert collaborated on integrating the different backpropogation pieces from the different types of layers (Convolution, ReLu, Fully Connected) into one function. Sho had pointed out an idea of how to better integrate the fully connected layers, and Bethany implemented his idea and got the initial forward and backward functions working. She also worked with Gabe to fix an error in padding the $\frac{\partial L}{\partial Y}$ matrix in the convolution backpropogation function. Additionally, Bethany worked with Gabe on testing and improving the training process. Bethany implemented approaches learned in class to improve and stabilize training performance such as the batch normalization layers, and early stopping. She also integrated Robert's ADAM optimizer code from the previous project into this project. Bethany compiled the final testing results and figures.

Gabriel Urbaitis

Gabriel debugged some of Bethany's backward method in the CNN.py file, identifying the need to not divide the f-1 padding by 2 in backpropagation, and helping her find where self.input in the Conv3d class was set incorrectly, leading to a size mismatch when the input dimensions were used for assigning dl/dx. He wrote the second proof, assisted Sho in testing 2d convolution on the example from slide 12 in lecture 24, and assisted Sho in testing and identifying bugs in Jyrus's first implementation of maxpool2d. He helped debug Sho's dataloading connection with the network initialization, including testing dimensionality for Cifar and MNIST. He added the FNN to the project, establishing the link between the CNN with the function for flattening the outputted last set of feature maps to use as input for the FNN. The FNN was later broken up by Bethany for ease of use. Gabriel wrote the conv3d, conv2d, forward, and padding functions, though the padding function was replaced by Robert, for reasons that there hadn't been time to discuss at submission time. He also wrote the Conv3d class's constructor in CNN.py. He made the changes associated with logsoftmax described in the optimizations section. He wrote the getAccuracy function as well.

Appendix

Code Listings

```
import numpy as np

from activations import ReLU, BatchNormalize
from CNN import Conv3d, MaxPool2d
from FNN.fnn import FNN
from FNN.layer import Layer as FFLayer

class CNN:
    """
    Convolutional Neural Network.
    """

    def __init__(self, input_shape, num_classes):
        """
        Initialize the CNN.
        """
        channels, height, width = input_shape

        # Use dynamic in_channels for CIFAR (3 channels) or MNIST (1 channel)
        self.c1 = Conv3d(in_channels=channels, out_channels=6, kernel_size=5, stride=1, padding=0)
        self.r1 = ReLU()
        self.b1 = BatchNormalize()
        self.s2 = MaxPool2d(kernel_size=2, stride=2)


        self.c3 = Conv3d(in_channels=6, out_channels=16, kernel_size=5, stride=1, padding=0)
        self.r3 = ReLU()
        self.b3 = BatchNormalize()
        self.s4 = MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = FFLayer(n_input=400, n_output=120, activation='relu')
        self.b4 = BatchNormalize()
        self.fc2 = FFLayer(n_input=120, n_output=84, activation='relu')
        self.b5 = BatchNormalize()
        self.fc3 = FFLayer(n_input=84, n_output=num_classes, activation='logsoftmax')

        self.layers = [self.c1, self.r1, self.s2, self.c3, self.r3, self.s4, self.fc1, self.fc2, self
            .fc3]
        self.output = None

    def forward(self, x):
        """
        Forward pass for LeNet.
        """
        # Transpose input to NCHW format (batch_size, channels, height, width)
        #x = np.transpose(x, (0, 3, 1, 2))

        # Layer 1: Convolution -> ReLU -> Max Pooling
        x_conv1 = self.c1.forward(x)  # Convolution
        x = self.r1.forward(x_conv1)  # ReLU activation
        x = self.b1.forward(x)
        x = self.s2.forward(x)        # Max Pooling

        # Save input for backpropagation (only input to the convolution is needed)
        #self.c1.input = x_conv1

        # Layer 2: Convolution -> ReLU -> Max Pooling
        x_conv2 = self.c3.forward(x)  # Convolution
        x = self.r3.forward(x_conv2)  # ReLU activation
        x = self.b3.forward(x)
```

```
        x = self.s4.forward(x)        # Max Pooling

        # Save input for backpropagation
        # self.c3.input = x_conv2

        # Flatten for Fully Connected Layers
        batch_size = x.shape[0]
        x = x.reshape(batch_size, -1)

        # Fully Connected Layers
        x = self.fc1.forward(x)
        x = self.b4.forward(x)
        x = self.fc2.forward(x)
        x = self.b5.forward(x)
        x = self.fc3.forward(x)


        return x



    def backward(self, y, y_pred, learning_rate, loss_func ="nll"):
        """
        Perform backpropagation through all layers of the CNN
        params:
            dL_dout: Gradient of the loss with respect to the output of the CNN (shape: batch_size x
                num_classes).
            learning_rate: Learning rate.
        """
        if loss_func == 'mse':
            #Regular
            dL_dout = 2 * (y_pred - y) / y.shape[0]
        elif loss_func == 'nll':
            p = np.exp(y_pred)
            dL_dout = p - y

        # Feed Forward Layers
        grad_W, dL_dout = self.fc3.backward(dL_dout)
        print("Mean abs grad FC3 weights:", np.mean(np.abs(grad_W)))
        grad_W = np.clip(grad_W, -5, 5)
        self.fc3.weights -= learning_rate * grad_W

        grad_W, dL_dout = self.fc2.backward(dL_dout)
        print("Mean abs grad FC2 weights:", np.mean(np.abs(grad_W)))
        grad_W = np.clip(grad_W, -5, 5)
        self.fc2.weights -= learning_rate * grad_W

        grad_W, dL_dout = self.fc1.backward(dL_dout)
        print("Mean abs grad FC1 weights:", np.mean(np.abs(grad_W)))
        grad_W = np.clip(grad_W, -5, 5)
        self.fc1.weights -= learning_rate * grad_W

        # Begin CNN layers

        # reshape for max pool?
        batch_size, original_channels, height, width = self.s4.output.shape
        dL_dout = dL_dout.reshape(batch_size, original_channels, height, width)
        dL_dout = self.s4.backward(dL_dout)

        # conv
        dL_dout, grad_filters, grad_biases = self.c3.backward(dL_dout)
        grad_filters = np.clip(grad_filters, -1, 1)
        grad_biases = np.clip(grad_biases, -1, 1)
        self.c3.filters -= learning_rate * grad_filters
        self.c3.biases -= learning_rate * grad_biases
```

```
        print(f"shape of grad filters and biases: {grad_filters.shape}, {grad_biases.shape}")

        # maxpool
        dL_dout = self.s2.backward(dL_dout)

        # conv
        dL_dout, grad_filters, grad_biases = self.c1.backward(dL_dout)
        grad_filters = np.clip(grad_filters, -1, 1)
        grad_biases = np.clip(grad_biases, -1, 1)
        self.c1.filters -= learning_rate * grad_filters
        self.c1.biases -= learning_rate * grad_biases


    def backward_adam(self, y, y_pred, t, learning_rate, rho=0.999, rho_f=0.9, epsilon=1e-8,
         loss_func ="nll"):
        """
        Perform backpropagation through all layers of the CNN
        params:
            dL_dout: Gradient of the loss with respect to the output of the CNN (shape: batch_size x
                 num_classes).
            learning_rate: Learning rate.
        """

        alpha_t = learning_rate * ((np.sqrt(1 - (rho ** t))) / (1 - (rho_f ** t) + 1e-8))
        print(f"alpha t {alpha_t}")

        if loss_func == 'mse':
            #Regular
            dL_dout = 2 * (y_pred - y) / y.shape[0]
        elif loss_func == 'nll':
            p = np.exp(y_pred)
            dL_dout = p - y
        elif loss_func == "cross_entropy":
            dL_dout = -np.sum(y * y_pred ) / y.shape[0]
        # print(dL_dout)

        # Feed Forward Layers
        grad_W, dL_dout = self.fc3.backward(dL_dout)
        grad_W = np.clip(grad_W, -5, 5)
        self.fc3.update_A(grad_W, rho)
        self.fc3.update_F(grad_W, rho_f)
        adaptive_step = self.getAdaptiveStep(self.fc3.A,
                                             self.fc3.F,
                                             alpha_t, rho,
                                             rho_f,
                                             t, epsilon)
        self.fc3.weights -= adaptive_step

        grad_W, dL_dout = self.fc2.backward(dL_dout)
        grad_W = np.clip(grad_W, -5, 5)
        self.fc2.update_A(grad_W, rho)
        self.fc2.update_F(grad_W, rho_f)
        adaptive_step = self.getAdaptiveStep(self.fc2.A,
                                             self.fc2.F,
                                             alpha_t, rho,
                                             rho_f,
                                             t, epsilon)
        self.fc2.weights -= adaptive_step

        grad_W, dL_dout = self.fc1.backward(dL_dout)
        grad_W = np.clip(grad_W, -5, 5)
        self.fc1.update_A(grad_W, rho)
        self.fc1.update_F(grad_W, rho_f)
        adaptive_step = self.getAdaptiveStep(self.fc1.A,
                                             self.fc1.F,
```

```
                                             alpha_t, rho,
                                             rho_f,
                                             t, epsilon)
        self.fc1.weights -= adaptive_step

        # Begin CNN layers

        # reshape for max pool?
        batch_size, original_channels, height, width = self.s4.output.shape
        dL_dout = dL_dout.reshape(batch_size, original_channels, height, width)
        dL_dout = self.s4.backward(dL_dout)

        # conv
        dL_dout, grad_filters, grad_biases = self.c3.backward(dL_dout)
        grad_filters = np.clip(grad_filters, -1, 1)
        grad_biases = np.clip(grad_biases, -1, 1)
        self.c3.update_A(grad_filters, rho)
        self.c3.update_F(grad_filters, rho_f)
        adaptive_step = self.getAdaptiveStep(self.c3.A,
                                             self.c3.F,
                                             alpha_t, rho,
                                             rho_f,
                                             t, epsilon)
        self.c3.filters -= adaptive_step
        self.c3.biases -= learning_rate * grad_biases
        print(f"shape of grad filters and biases: {grad_filters.shape}, {grad_biases.shape}")

        # maxpool
        dL_dout = self.s2.backward(dL_dout)

        # conv
        dL_dout, grad_filters, grad_biases = self.c1.backward(dL_dout)
        grad_filters = np.clip(grad_filters, -1, 1)
        grad_biases = np.clip(grad_biases, -1, 1)
        self.c1.update_A(grad_filters, rho)
        self.c1.update_F(grad_filters, rho_f)
        adaptive_step = self.getAdaptiveStep(self.c1.A,
                                             self.c1.F,
                                             alpha_t, rho,
                                             rho_f,
                                             t, epsilon)
        self.c1.filters -= adaptive_step
        self.c1.biases -= learning_rate * grad_biases


    def train(self, input, labels, epochs: int, learning_rate: int = 0.01, optimizer = "gd"):
        input = (input - 0.5) / 0.5
        training_accuracy = []
        for epoch in range(epochs):
            loss_sum = 0  # accumulated loss within epoch
            correct = 0   # correctly classified samples
            total = 0     # total samples

            # Forward pass
            out = self.forward(input)

            # Convert raw class indices to one-hot encoding
            if labels.ndim == 1:  # If labels are raw class indices
                num_classes = out.shape[1]
                labels = np.eye(num_classes)[labels]  # Convert to one-hot

            # Backward pass
            if optimizer == "gd":
                self.backward(labels, out, learning_rate, loss_func="nll")
            elif optimizer == "adam":
```

```
                    self.backward_adam(labels, out, epoch+1, learning_rate, loss_func="nll")
                else:
                    raise ValueError(f"{optimizer} not an available optimizer")


                # Calculate predictions and accuracy
                predictions = np.argmax(out, axis=1)
                true_labels = np.argmax(labels, axis=1)
                correct += np.sum(predictions == true_labels)
                total += labels.shape[0]
                accuracy = (correct / total) * 100

                loss = -np.mean(np.sum(labels * out, axis=1))

                # Print accuracy after each epoch
                print(f"Epoch {epoch + 1}/{epochs}- Loss: {loss:.4f}, Accuracy: {accuracy:.2f}%")

                training_accuracy.append(accuracy)

                if accuracy > 95:
                    print("Reached high enough accuracy")
                    break

        return training_accuracy


    def getAccuracy(self, test_loader):
        correct = 0
        total = 0

        for idx, test_batch in enumerate(test_loader):
            test_input, test_labels = test_batch
            test_input = test_input.numpy()
            test_labels = test_labels.numpy()

            # Normalize input
            test_input = (test_input - 0.5) / 0.5  # Match normalization from training

            # Forward pass
            out = self.forward(test_input)

            # Predictions
            predicted_classes = np.argmax(out, axis=1)  # Predicted class indices
            true_classes = test_labels  # Directly use 1D array of true class indices

            # Accuracy calculation
            correct += np.sum(predicted_classes == true_classes)
            total += test_labels.shape[0]

            # Break early for debugging
            if idx > 5:  # Limit to 5 batches
                break

        accuracy = correct / total * 100
        print(f"Test Accuracy: {accuracy:.2f}%")
        return accuracy

    def clip_gradient(self, gradient, threshold=1.0):
        norm = np.linalg.norm(gradient)
        if norm > threshold:
            scaling_factor = np.clip(threshold / norm, a_min=0, a_max=1.0)

            # Scale the gradient using the factor
            gradient = gradient * scaling_factor

```

```
        # print(gradient)
        return gradient


    def getAdaptiveStep(self, A, F, alpha_t, rho, rho_f, t, epsilon):
            A_hat = A * (1 / ((1 - (rho ** t)) + 1e-8))
            F_hat = F * (1 / ((1 - (rho_f ** t)) + 1e-8))
            adaptive_step = alpha_t * F_hat / (np.sqrt(A_hat) + epsilon)
            return adaptive_step
```

Listing 1: LeNet.py

```
import numpy as np
from FNN.layer import Layer

class Conv3d:
    """
    3D Convolutional Layer
    """

    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        self.filters = np.random.randn(out_channels, in_channels, kernel_size, kernel_size) * 0.01
        self.biases = np.zeros(out_channels)

        # for adam optimization
        self.A = np.zeros_like(self.filters)
        self.F = np.zeros_like(self.filters)

        #print(f"Initialized filters: {self.filters.shape}, biases: {self.biases.shape}")


    def pad_matrix(self, input, pad_size):
        return np.pad(input, ((0, 0), (0, 0), (pad_size, pad_size), (pad_size, pad_size)), mode='
             constant')


    def forward(self, input):
        """
        params:
            input: 4D input array of shape (batch_size, in_channels, height, width).
        return:
            conv3d output
        """
        self.input = input
        print("from forward, input shape: ", input.shape)
        return self.conv3d(input, self.filters, self.biases, self.stride, self.padding)

    def conv2d(self, input_channel, kernel, bias, stride):
        """
        params:
            input_channel: one of the 2D input channels (height x width).
            kernel: the associated 2D kernel (height x width).
            bias: bias for the kernel
            stride: stride of convolution.

        return:
            2D output matrix.
        """
```

```

        input_height, input_width = input_channel.shape
        kernel_height, kernel_width = kernel.shape

        # output dimensions
        out_height = (input_height - kernel_height) // stride + 1
        out_width = (input_width - kernel_width) // stride + 1


        output = np.zeros((out_height, out_width))

        # 2D convolution
        for i in range(out_height):
            for j in range(out_width):
                region = input_channel[
                         i * stride:i * stride + kernel_height,
                         j * stride:j * stride + kernel_width
                         ]

                region = np.clip(region, -1e3, 1e3)
                kernel = np.clip(kernel, -1e3, 1e3)

            # Perform convolution
                try:
                    output[i][j] = np.sum(region * kernel) + bias
                except RuntimeWarning as e:
                    print(f"Overflow in convolution at position ({i}, {j}):", e)
                    print("Region:", region)
                    print("Kernel:", kernel)
                    raise e

        # Debug: Print output of the convolution
        #print(f"Convolution output shape: {output.shape}")

        return output

    def conv3d(self, input, filters, biases, stride=1, padding=0):
        """
        params:
            input: 4D input array of shape (batch_size x input_channels x height x width).
            filters: filters (num_filters x input_channels x kernel_height x kernel_width).
            biases: bias for each filter
            stride: stride of convolution.
            padding: padding on height and width.

        return:
            4D output matrix (batch_size x num_filters x out_height x out_width).
        """

        # print(f"conv3d - input shape: {input.shape}")
        """
        if padding > 0:
            self.input = self.pad_matrix(input, padding)
        else:
            self.input = input
            """

        batch_size, input_channels, input_height, input_width = input.shape
        num_filters, filter_channels, kernel_height, kernel_width = filters.shape

        # print the input shape
        # print(f"Input shape: {input.shape}")
        # print("batch size: ", batch_size)
        # print("input height: ", input_height)
        # print("input channels: ", input_channels)
```

```python
        # print("input width: ", input_width)

        # print(f"Filters shape: {filters.shape}")
        # print("num filters: ", num_filters)
        # print("filter channels: ", filter_channels)
        # print("kernel height: ", kernel_height)
        # print("kernel width: ", kernel_width)

        # output dimensions
        out_height = (input_height - kernel_height + 2 * padding) // stride + 1
        out_width = (input_width - kernel_width + 2 * padding) // stride + 1

        # Debug: Print calculated output dimensions
        # print(f"Calculated output dimensions: out_height={out_height}, out_width={out_width}")

        # output initialization
        output = np.zeros((batch_size, num_filters, out_height, out_width))

        # 3D convolution
        for batch in range(batch_size):
            for filter in range(num_filters):
                global_output = np.zeros((out_height, out_width))
                for channel in range(input_channels):
                    # Sum all channels
                    global_output += self.conv2d(
                        input[batch, channel], filters[filter, channel], biases[filter], stride
                    )
                output[batch, filter] = global_output

        return output


    def backward(self, dL_dout):
        """
        params:
            dL_dout: derivative of loss w.r.t output of previous layer

        return:
        """
        batch_size, dL_dout_channels, dL_dout_height, dL_dout_width = dL_dout.shape
        num_filters, in_channels, kernel_height, kernel_width = self.filters.shape
        dL_db = np.zeros(num_filters)

        # compute dL \ dF
        dL_df = np.zeros_like(self.filters)
        for batch in range(batch_size):
            for filter in range(num_filters):
                for channel in range(in_channels):
                    dL_df[filter, channel, :,:] = self.conv2d(self.input[batch, channel,:,:],
                                                    dL_dout[batch, filter,:,:],
                                                    0, # bias is 0 because we don't need it here
                                                    stride=self.stride)

        # computer dL \ dx (this is the part that will be backpropogated)
        dL_dx = np.zeros_like(self.input)
        print("input shape ", self.input.shape)
        for batch in range(batch_size):
            for filter in range(num_filters):
                for channel in range(in_channels):
                    # rotate 90 degrees twice
                    rotated_filter = np.rot90(self.filters[filter,channel,:,:],
                                        k=2,)

                    # Add padding to dL_dout to match input size (padding = kernel size - 1)
                    # Slides say "Notice that you need to extend the matrix."
```

```
                    # print(self.filters[filter,channel,:,:])
                    pad_size = (self.filters.shape[2] - 1) #// 2
                    # print(pad_size)
                    padded_dL_dout = self.pad_matrix(dL_dout, pad_size)
                    # print(padded_dL_dout.shape)
                    dL_dx[batch,channel,:,:] = self.conv2d(padded_dL_dout[batch, filter,:,:],
                                                    rotated_filter,
                                                    0, # bias is zero because we handle is separately
                                                    stride=self.stride)

        # compute bias gradient
        for filter in range(num_filters):
            dL_db[filter] = np.sum(dL_dout[:, filter, :, :])


        return dL_dx, dL_df, dL_db

    def update_A(self, grad_filter, rho=0.999):
        grad_filter = np.clip(grad_filter, -3, 3)
        num_filters, in_channels, kernel_height, kernel_width = grad_filter.shape
        for filter in range(num_filters):
            for channel in range(in_channels):
                self.A[filter, channel, :, :]  = (rho * self.A[filter, channel, :, :]
                                                  + (1 - rho) * (grad_filter[filter, channel, :, :]**2))


    def update_F(self, grad_filter, rho_f=0.9):
        grad_filter = np.clip(grad_filter, -3, 3)
        num_filters, in_channels, kernel_height, kernel_width = grad_filter.shape
        for filter in range(num_filters):
            for channel in range(in_channels):
                self.F[filter, channel, :, :]  = (rho_f * self.F[filter, channel, :, :]
                                                  + (1 - rho_f) * (grad_filter[filter, channel, :, :]))




class MaxPool2d:
    """
    2D Max Pooling Layer.
    """

    def __init__(self, kernel_size, stride):
        self.kernel_size = kernel_size
        self.stride = stride
        self.input = None
        self.max_indices = None  # To store indices of max values during forward pass

    def forward(self, input):
        """
        Perform max pooling and store argmax indices.
        """
        self.input = input
        batch_size, channels, H_in, W_in = input.shape

        # Calculate the output dimensions
        H_out = ((H_in - self.kernel_size) // self.stride) + 1
        W_out = ((W_in - self.kernel_size) // self.stride) + 1

        # Initialize output and indices
        output = np.zeros((batch_size, channels, H_out, W_out))
        self.max_indices = np.zeros((batch_size, channels, H_out, W_out, 2), dtype=int)

```

```
        for b in range(batch_size):
            for c in range(channels):
                for i in range(H_out):
                    for j in range(W_out):
                        h_start = i * self.stride
                        h_end = h_start + self.kernel_size
                        w_start = j * self.stride
                        w_end = w_start + self.kernel_size

                        window = input[b, c, h_start:h_end, w_start:w_end]
                        # Find max and store it
                        max_idx_flat = np.argmax(window)
                        max_idx = np.unravel_index(max_idx_flat, (self.kernel_size, self.kernel_size)
                            )

                        output[b, c, i, j] = window[max_idx]
                        # Store the exact indices in the original input
                        self.max_indices[b, c, i, j, 0] = h_start + max_idx[0]
                        self.max_indices[b, c, i, j, 1] = w_start + max_idx[1]

        self.output = output
        return output

    def backward(self, dL_dout):
        """
        Backpropagation using stored max indices.
        """
        batch_size, channels, H_in, W_in = self.input.shape
        _, _, H_out, W_out = dL_dout.shape

        # Initialize gradient w.r.t input
        dL_dinput = np.zeros_like(self.input)

        # Directly use the stored max indices
        for b in range(batch_size):
            for c in range(channels):
                for i in range(H_out):
                    for j in range(W_out):
                        max_i, max_j = self.max_indices[b, c, i, j]
                        dL_dinput[b, c, max_i, max_j] += dL_dout[b, c, i, j]

        return dL_dinput
```

Listing 2: `CNN.py`

```
import numpy as np

from FNN.layer import Layer

class FNN:
    """
    A Feed-Forward Neural Network.
    """

    # Initialize the network with a list of layers
    def __init__(self, layers):
        self.layers = layers

    # Perform forward propagation through all layers
    def forward(self, X):
        for layer in self.layers:
            X = layer.forward(X)
        return X

```

```python
        """
        Calculate gradients for all layers.
        X: Input data
        y: True labels
        y_pred: Predicted output from the forward pass
        loss_func: Loss function ('mse' or 'nll')
        """
        def backward(self, y, y_pred, loss_func='mse'):
            if loss_func == 'mse':
                #NewtonCases
                dL_dout = (y_pred - y) / y.shape[0]
                #Regular
                #dL_dout = 2 * (y_pred - y) / y.shape[0]
            elif loss_func == 'nll':
                dL_dout = y_pred - y
            gradients_W = []
            # Proceeding backward through the layers, add each new calculation to the front
            # to create the gradients array
            for layer in reversed(self.layers):
                grad_W, dL_dout = layer.backward(dL_dout)
                gradients_W.insert(0, grad_W)
            return gradients_W

        # Update weights and biases using gradient descent
        def gd(self, gradients_W, learning_rate):
            for layer, grad_W in zip(self.layers, gradients_W):
                layer.weights -= learning_rate * grad_W


        def sgd(self, X, y, batch_size, learning_rate, loss_func='mse'):
            indices = np.arange(X.shape[0])
            np.random.shuffle(indices)

            for start_idx in range(0, X.shape[0] - batch_size + 1, batch_size):
                batch_indices = indices[start_idx:start_idx + batch_size]
                X_batch = X[batch_indices]
                y_batch = y[batch_indices]

                # Forward pass
                y_pred = self.forward(X_batch)

                # Backward pass
                gradients = self.backward(y_batch, y_pred, loss_func)

                # Update weights
                for layer, gradient in zip(self.layers, gradients):
                    layer.weights -= learning_rate * gradient

        # Train the network using forward and backward propagation
        def train(self, X, y, learning_rate, epochs):
            for _ in range(epochs):
                y_pred = self.forward(X)
                gradients_W = self.backward(y,y_pred)
                self.gd(gradients_W, learning_rate)
    # Train the network using stochastic gradient descent
        def trainsgd(self, X, y, learning_rate, epochs, batch_size, loss_func='mse'):
            for epoch in range(epochs):
                self.sgd(X, y, batch_size, learning_rate, loss_func)

                # Calculate and print loss for monitoring
                y_pred = self.forward(X)
                loss = self._calculate_loss(y, y_pred, loss_func)
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss}")

        def _calculate_loss(self, y, y_pred, loss_func):
```

```
        if loss_func == 'mse':
            return np.mean((y_pred - y) ** 2)
        elif loss_func == 'nll':
            return -np.mean(y * np.log(y_pred + 1e-8))
        else:
            raise ValueError("Unsupported loss function")
```

Listing 3: fnn.py

```
import random

import numpy as np

class Layer:
    """
    A layer in the Feedforward Neural Network (FNN).
    """

    # Randomly initialize weights and biases
    def __init__(self, n_input, n_output, activation='relu'):
        random.seed(2400)
        self.weights = np.random.randn(n_input+1, n_output) * 0.01
        self.activation_function = activation
        self.n_input = n_input

        # for adam
        self.A = np.zeros_like(self.weights)
        self.F = np.zeros_like(self.weights)

    def forward(self, X):
        X = np.hstack([X, np.ones((X.shape[0], 1))])

        self.z = np.dot(X, self.weights)
        self.a = self.activate(self.z)
        self.input_data = X

        return self.a

    # Activation functions
    def activate(self, z):
        activations = {
            'relu': lambda z: np.maximum(0, z),
            'sigmoid': lambda z: 1 / (1 + np.exp(-z)),
            'id': lambda z: z,
            'sign': lambda z: np.sign(z),
            'tanh': lambda z: np.tanh(z),
            'hard tanh': lambda z: np.clip(z, -1, 1),
            'logsoftmax': lambda z: z - np.log(np.sum(np.exp(z - np.max(z, axis=1, keepdims=True)),
                axis=1, keepdims=True) + 1e-8)
        }

        return activations[self.activation_function](z)

    # Derivatives of activation functions
    """
    If an error arises using the 'sign' activation function, it is because the derivative is
        undefined at z = 0. (Will return NaN)
    """
    def activation_deriv(self, z):
        derivs = {
            'relu': lambda z: np.where(z > 0, 1, 0),
            'sigmoid': lambda z: (sig := 1 / (1 + np.exp(-z))) * (1 - sig),
            'id': lambda _: np.ones_like(z),
            'sign': lambda z: np.zeros_like(z),  # Derivative undefined at z = 0
```

```
            'tanh': lambda z: 1 - np.tanh(z) ** 2,
            'hard tanh': lambda z: np.where(np.abs(z) <= 1, 1, 0),
            # logsoftmax derivative here
            'logsoftmax': lambda z: np.exp(z - np.max(z, axis=1, keepdims=True)) / (
                        np.sum(np.exp(z - np.max(z, axis=1, keepdims=True)), axis=1, keepdims=True) +
                            1e-8)
        }

        return derivs[self.activation_function](z)

    def backward(self, dL_dout):
        dL_dout = np.nan_to_num(dL_dout)
        if self.activation_function != 'logsoftmax':
            activation_deriv = self.activation_deriv(self.z)
            dL_dout *= activation_deriv
        # partial derivative of the loss w.r.t. the weights
        grad_W = np.dot(self.input_data.T, dL_dout)
        # accumulation of partial derivative of the loss for each layer
        dL_din = np.dot(dL_dout, self.weights.T)

        # Remove the bias
        dL_din = dL_din[:, :-1]

        grad_W = np.clip(grad_W, -3, 3)

        return grad_W, dL_din


    def update_A(self, gradients_W, rho=0.999):
        gradients_W = np.clip(gradients_W, -3, 3)
        self.A = rho*self.A + (1 - rho) * (gradients_W ** 2)

    def update_F(self, gradients_W, rho_f=0.9):
        gradients_W = np.clip(gradients_W, -3, 3)
        self.F = rho_f * self.F + (1-rho_f) * gradients_W
```

Listing 4: `layer.py`

```
import numpy as np


class ReLU:
    """
    ReLU Activation Function.
    """

    def __init__(self):
        self.x = None

    def forward(self, x):
        self.x = x
        # return x if x > 0 else 0
        return np.maximum(0, x)

class BatchNormalize:
    """
    ReLU Activation Function.
    """

    def __init__(self):
        self.x = None

    def forward(self, x):
        mean = np.mean(x, axis=0)
```

```
        std = np.std(x, axis=0)
        x = (x - mean) / (std + 1e-8)
        return x
```

Listing 5: `activations.py`

```python
import numpy as np
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

import pickle

from LeNet import CNN
# from LeNet5 import LeNet5

def get_data_loaders(batch_size: int = 64, dataset: str = "mnist"):
    """
    Load MNIST or CIFAR-10 datasets and return DataLoaders with resized 32x32 inputs.
    :param batch_size: Batch size for DataLoader.
    :param dataset: Dataset to load ('mnist' or 'cifar').
    :return: Train and test DataLoaders for the selected dataset.
    """
    if dataset == "mnist":
        transform = transforms.Compose([
            transforms.Resize((32, 32)),
            transforms.ToTensor(),
            transforms.Lambda(lambda x: x.repeat(3, 1, 1)),  # Repeat grayscale 3 times
        ])

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

    elif dataset == "cifar":
        transform = transforms.Compose([
            transforms.Resize((32, 32)),
            transforms.ToTensor(),
        ])

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

    else:
        raise ValueError("Invalid dataset. Choose 'mnist' or 'cifar'.")

    train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False)

    return train_loader, test_loader

if __name__ == "__main__":
    batch_size = 32
    epochs = 10

    # dataset = "mnist"
    dataset = "cifar"

    train_loader, test_loader = get_data_loaders(batch_size=batch_size, dataset=dataset)

    batch = next(iter(train_loader))
    input_data, label = batch
    input_data = input_data.numpy()
    label = label.numpy()

    model = CNN(input_shape=(3, 32, 32), num_classes=10)
```

```

    # adam
    adam_runs = {}
    for i in range(3):
        model = CNN(input_shape=(3, 32, 32), num_classes=10)
        training_accuracies = model.train(input_data,
                                          label,
                                          epochs,
                                          learning_rate=0.01,
                                          optimizer="adam")

        test_accuracy = model.getAccuracy(test_loader)


        adam_runs[i] = (training_accuracies, test_accuracy)

    with open(f"{dataset}_adam_runs.pkl", "wb") as f:
        pickle.dump(adam_runs, f)

    # GD
    gd_runs = {}
    for i in range(3):
        model = CNN(input_shape=(3, 32, 32), num_classes=10)
        training_accuracies = model.train(input_data,
                                          label,
                                          epochs,
                                          learning_rate=0.001,
                                          optimizer="gd")

        test_accuracy = model.getAccuracy(test_loader)


        gd_runs[i] = (training_accuracies, test_accuracy)

    with open(f"{dataset}_gd_runs.pkl", "wb") as f:
        pickle.dump(gd_runs, f)
```

Listing 6: `data_loader.py`