Deep Learning

CS-898BD

Yassine Ibork

Y723u998

Assignment 2

Supervised by: Dr. Lokesh Das

I. Introduction

This assignment provides practical experience with Convolutional Neural Networks (CNNs). It enhances our understanding of how CNNs operate and enables us to implement batch normalization effectively. We explored various optimization techniques, including Stochastic Gradient Descent (SGD), Adam, and RMSprop. Additionally, the assignment guided us in implementing our own custom deep learning architectures.

The code related to this assignment will be found in this github repository: https://github.com/yibork/WSU-DL-CNNs

II. Methodology

This assignment contains two main parts the first part is about training a deep neural network on the cifar-10 architecture and the second part is about training a deeplearning model in 15 scene dataset by using three different optimizers. In this section I will talk about the methodology that I have adopted in each part of the assignment.

2.1. Part 1

In the first part of this assignment, I utilized PyTorch to load and manage my dataset effectively.

2.1.1. Dataset

The dataset comprises the CIFAR-10 collection, which is structured into two primary directories: one for training and another for testing. To enhance the model evaluation process, I have created a validation set derived from the initial training data. Specifically, I extracted 5% of the training data to form this validation set. This approach allows for a more accurate assessment of the model's performance during training without compromising the integrity of the test set.

- **Validation Set Size**: 1,000 images
- **Test Set Size**: 9,000 images
- **Training Set Size**: 50,000 images

I have also applied several transformations to my dataset to ensure that it is processed correctly by my CNN and to mitigate the issue of gradient explosion.

- **Random Horizontal Flip**: This augments the dataset by flipping images horizontally, helping the model learn orientation invariance.
- **Random Crop**: It randomly crops images with padding, allowing the model to learn from different parts of each image.
- **ToTensor**: Converts images to PyTorch tensors, scaling pixel values from [0, 255] to [0, 1].

- **Normalize**: Standardizes pixel values to have a mean and standard deviation of 0.5, aiding in stable training and faster convergence.

```
transform = transforms.Compose(
    [transforms.RandomHorizontalFlip(),
     transforms.RandomCrop(32, padding=4),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
)
```

Figure 1: Transformation Pipeline

2.1.2. Model Implementation and Optimization Techniques

I implemented the architecture specified by our professor using PyTorch, which will be elaborated upon in the "Deep Learning Architecture" section. One of the key components integrated into the model is batch normalization, which aids in stabilizing the learning process and accelerating convergence.

To optimize the training efficiency, I employed mini-batches of size 32. This strategy leverages vectorization while allowing for frequent model updates without the need to process the entire dataset at once. For the optimization process, I utilized Stochastic Gradient Descent (SGD) as my optimizer, coupled with a cosine annealing learning rate scheduler to dynamically adjust the learning rate throughout training.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNNModel().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.6)
scheduler = CosineAnnealingLR(optimizer, T_max=100, eta_min=0.001)
```

Figure 2: Model loading

2.2. Part 2

The second part mainly focused on implementing our own architecture and validating it against the 15-scene classification dataset using three different optimizers. To achieve this, I had to go over the following parts:

2.2.1. Data Preprocessing

The dataset is organized into a directory structure containing subdirectories for each class of images. To create balanced training, validation, and test sets, I implemented a function that splits the dataset based on specified ratios (80% training, 10% validation, 10% testing). This approach allows for a more accurate assessment of the model's performance by validating unseen data. Notably, I extracted images uniformly across classes, ensuring that the smallest class's size determined the number of samples from each class.

To improve the model's robustness and generalization capabilities, I applied various transformations to the images during training. This included:

- **Random Resized Crop**: Resizes and crops images to a fixed size, enhancing the model's ability to learn from different scales and aspects of the images.
- **Random Horizontal Flip**: Randomly flips images horizontally, helping to simulate variations in object orientations.
- **Normalization**: Each image is normalized using specific mean and standard deviation values tailored for the dataset, ensuring that the pixel values fall within a standard range for better convergence during training.

```
transform = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

train_dataset = datasets.ImageFolder('data/preprocessed-data/train', transform=transform)
val_dataset = datasets.ImageFolder('data/preprocessed-data/val', transform=transform)
test_dataset = datasets.ImageFolder('data/preprocessed-data/test', transform=transform)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Figure 3: Data Preprocessing

2.2.2. Model Implementation and Optimization Techniques

This methodology outlines the process for training the Scene Classification CNN model, incorporating various optimization algorithms and initialization strategies.

- **Optimizer Selection**: The get_optimizer function enables the selection of different optimization algorithms, such as SGD, Adam, and RMSProp, based on user input. This flexibility allows for thorough testing to determine the most effective optimizer for the scene classification task.
- **Loss Function**: The **CrossEntropyLoss** function is employed, as it is well-suited for multi-class classification problems.
- **Training Steps**:
    - **Model Initialization**: The model's state is reset to ensure consistent starting parameters for each training run.

- **Weight Initialization**: The model's weights are initialized using **He initialization**, which helps maintain a balanced variance throughout the layers, promoting effective learning and faster convergence.
- **Inspired Architecture**: Overall architecture draws inspiration from the implementation of **AlexNet**, leveraging its proven effectiveness in image classification tasks.
- **Optimizer Creation**: An optimizer is instantiated through the get_optimizer function according to the selected type.
- **Learning Rate Scheduler**: A cosine annealing learning rate scheduler is implemented to dynamically adjust the learning rate during training, aiding in fine-tuning the model's performance.

This comprehensive approach allows for systematic exploration of different optimizers and initialization strategies, enhancing the model's overall efficacy.

III. Deep Learning Architecture

In this section I talk about the deep learning architecture that I have adopted in order to fulfill the requirements of the architecture.

3.1. Part 1

Starting with the first part of the assignment I have implemented a model called the CNNModel which is a Convolutional Neural Network (CNN) designed for image classification tasks, specifically targeting 10 classes. The architecture includes convolutional layers followed by pooling and fully connected layers.

3.1.1. Components of the Architecture

- **Layer 1**:
  - **Conv2d**: Takes 3 input channels (for RGB images), produces 32 output channels with a kernel size of 5 and no padding.
  - **BatchNorm**: Normalizes the 32 channels.
  - **ReLU Activation**: Applies the ReLU activation function to introduce non-linearity.

    - **MaxPool**: Reduces the size from 28x28 to 14x14 (assuming the input size is 28x28).

- **Layer 2**:
    - **Conv2d**: Takes 32 input channels and produces 64 output channels with a kernel size of 3 and no padding.
    - **BatchNorm**, **ReLU**, and **MaxPool** follow, reducing the feature map size from 14x14 to 6x6.

- **Fully Connected Layers**:
    - **self.fc1**: The first fully connected layer takes the flattened output from the previous layers it maps to 512 outputs.
    - **self.fc2**: The second fully connected layer takes the output of fc1 and produces 10 outputs corresponding to the number of classes.

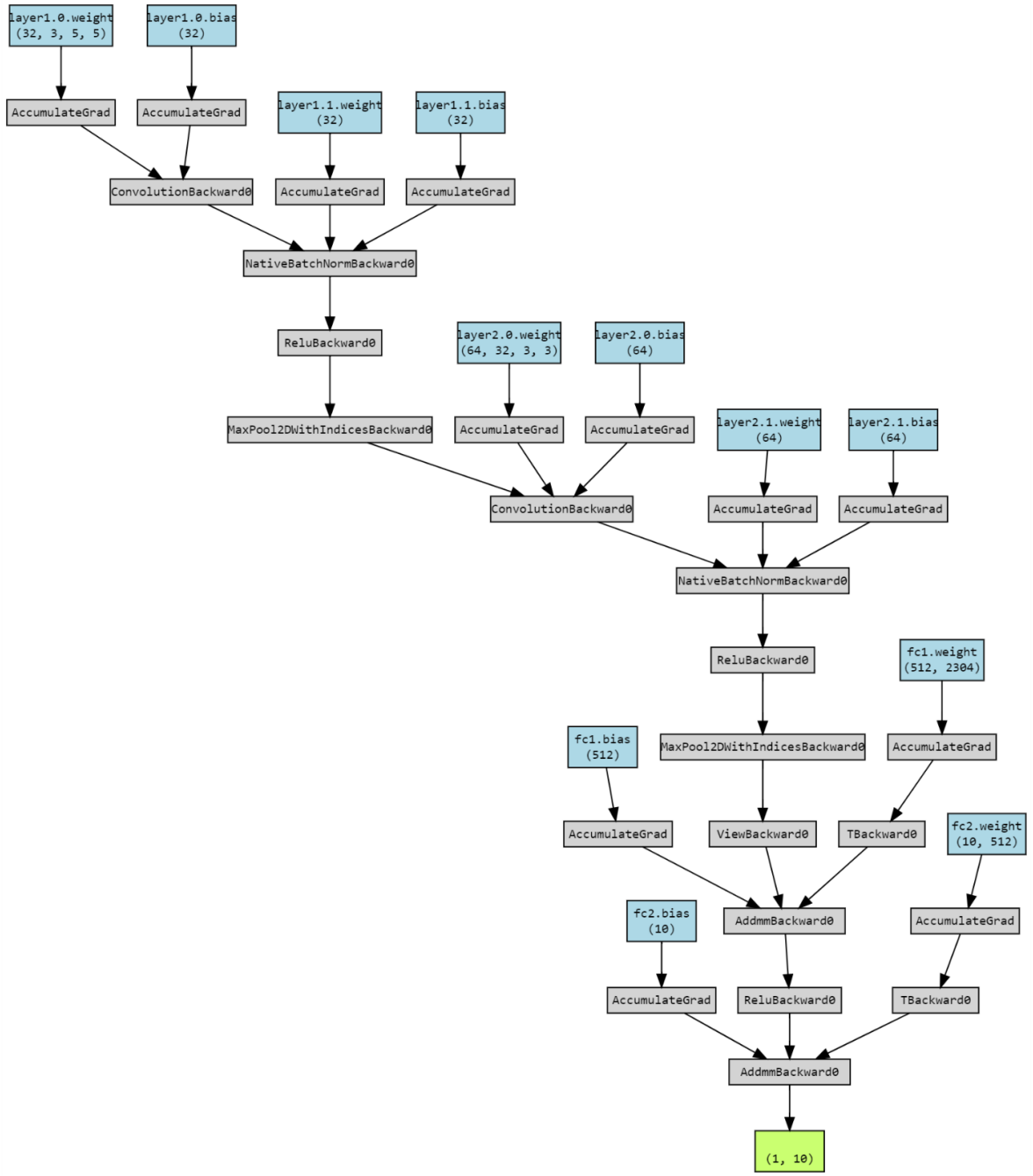

Figure 4: CIFAR-10 Model Architecture

3.2. Part 2

For the scene classification problem I proposed my architecture where I was inspired from [1]. The model that I have developed is a Convolutional Neural Network (CNN) designed for classifying images into num_classes categories (default is 15). This architecture combines

convolutional layers with batch normalization, activation functions, and pooling layers, followed by fully connected (dense) layers.

Components of Architecture

- **Convolutional Layers**
- **Activation Function**:
    - **nn.ELU()**: Exponential Linear Unit (ELU) is an activation function that helps the model learn faster and achieve better performance compared to traditional ReLU [2].
- **Batch Normalization**:
    - Normalizes the output of the previous layer by adjusting and scaling the activations, which helps stabilize the learning process and reduce the number of training epochs required.
- **Pooling Layers**:
    - Reduces the spatial dimensions (height and width) of the input, which helps reduce computation and control overfitting by summarizing the presence of features.
- **Dropout Layers**:
    - Randomly sets a fraction p of input units to 0 at each update during training time, which helps prevent overfitting.
- **Fully Connected Layers**:
    - Connects every input to every output, making it suitable for classification tasks after feature extraction through convolutional layers.
- **Weight Initialization**:
    - I have used He initializationinitializaiton

Figure 5: Scene Classification Model Architecture

IV. Experiments and Results

In this section, I will discuss the performance achieved in both parts of the assignment, along with an analysis of the corresponding losses.

4.1. Part 1:

After training the model on the CIFAR dataset, as defined in Section 3.1, over 100 epochs, the model achieved the following performance metrics:

- **Accuracy**: 82%
- **Precision**: 82%
- **Recall**: 82%
- **F1 Score**: 82%

These metrics were still improving at the end of the training process. This suggests that with further training or fine-tuning, the model could achieve even higher accuracy and overall performance. Extending the number of epochs or exploring more advanced optimization techniques may help push the model to converge at an even higher level of accuracy.

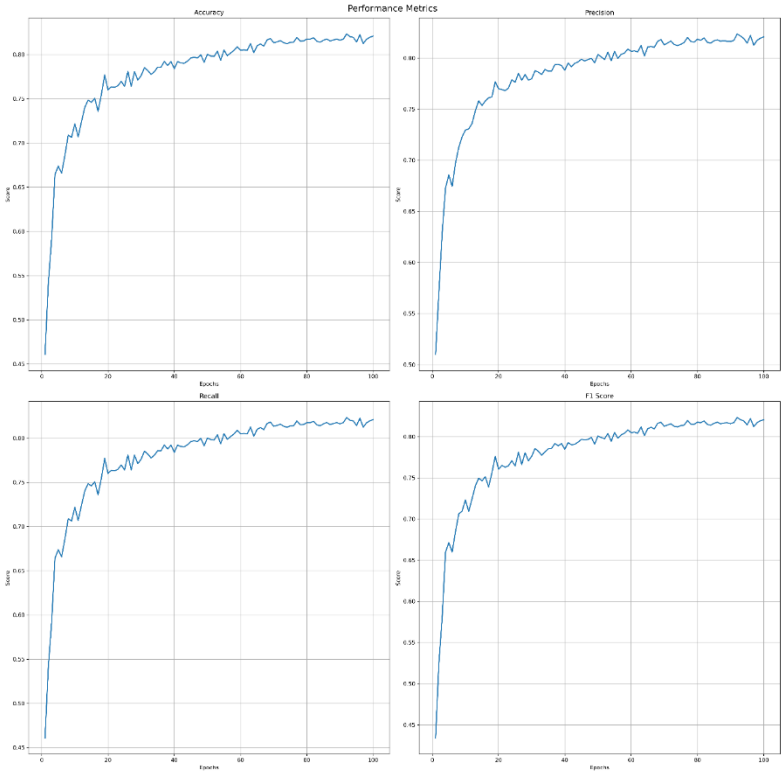

Figure 6. Performance Metrics Over Epochs

The figure illustrates the training and test loss/error over 100 epochs. As shown in the figure below, the test loss decreases during the initial epochs but then stabilizes, indicating a potential risk of overfitting. In contrast, the training loss continues to decline

steadily throughout the training process. Both the training and test errors decrease after each epoch; however, the test error remains constant after epoch 40.

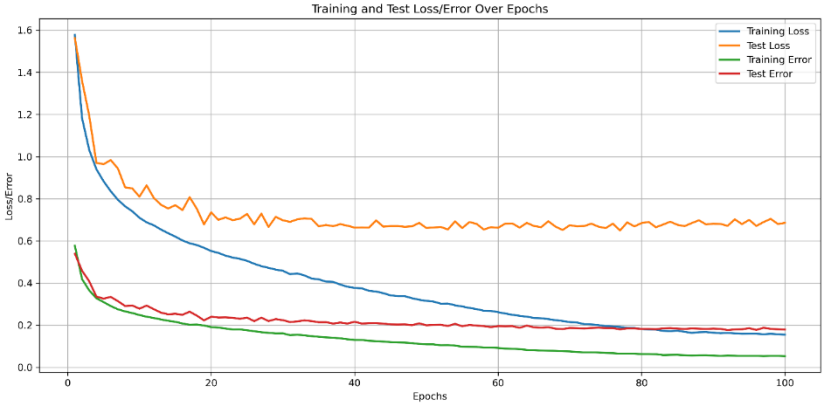

Figure 7: Training and Test Loss/Error Over Epochs

4.2. Part 2.

For the second part of the assignment, I leveraged the AlexNet architecture and refined it to meet the specific requirements. To optimize the model, I employed three different optimizers: Stochastic Gradient Descent (SGD), Adam, and RMSprop. In the following section, I will present the results obtained from each optimizer, followed by a detailed comparison of their performance to evaluate which optimizer yielded the best results

4.2.1. Stochastic gradient descent

I utilized the **Stochastic Gradient Descent (SGD)** optimizer with a batch size of 32 for training my model. The model was trained over 300 epochs, where performance metrics such as **accuracy, precision, recall,** and **F1 score** were recorded. Initially, the model experienced fluctuations across these metrics, which stabilized and improved steadily as training progressed.

By the end of the training, the model achieved the following performance metrics:

- **Accuracy**: 77%
- **Precision**: 77%
- **Recall**: 77%
- **F1 Score**: 77%

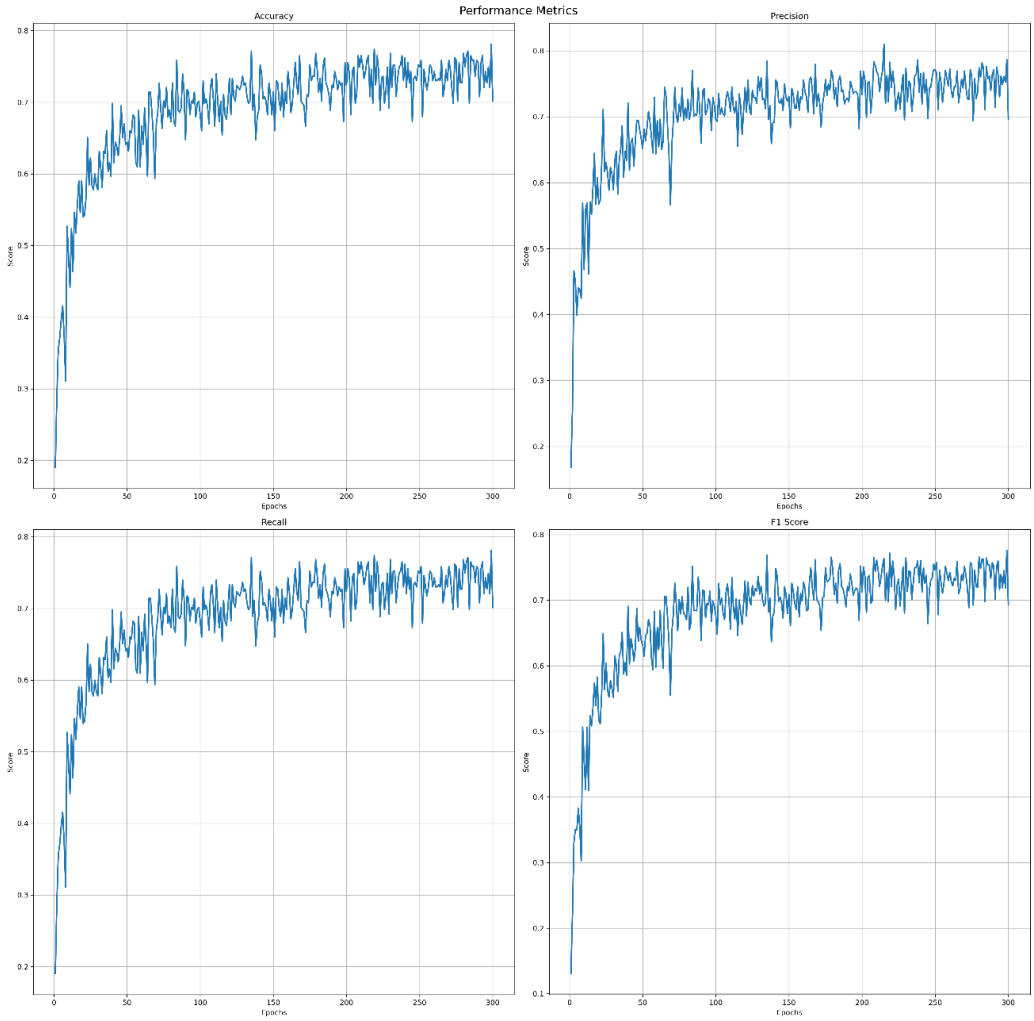

Figure 8: Performance Metrics Over Epochs

As illustrated in the figure below, both training and testing loss decrease over time. During the initial 50 epochs, the training and test losses move in tandem. However, around this point, the test loss begins to stabilize with some fluctuations, which may indicate the onset of overfitting. Meanwhile, the training error continues to decline throughout the epochs, whereas the test error

remains constant from epoch 50.

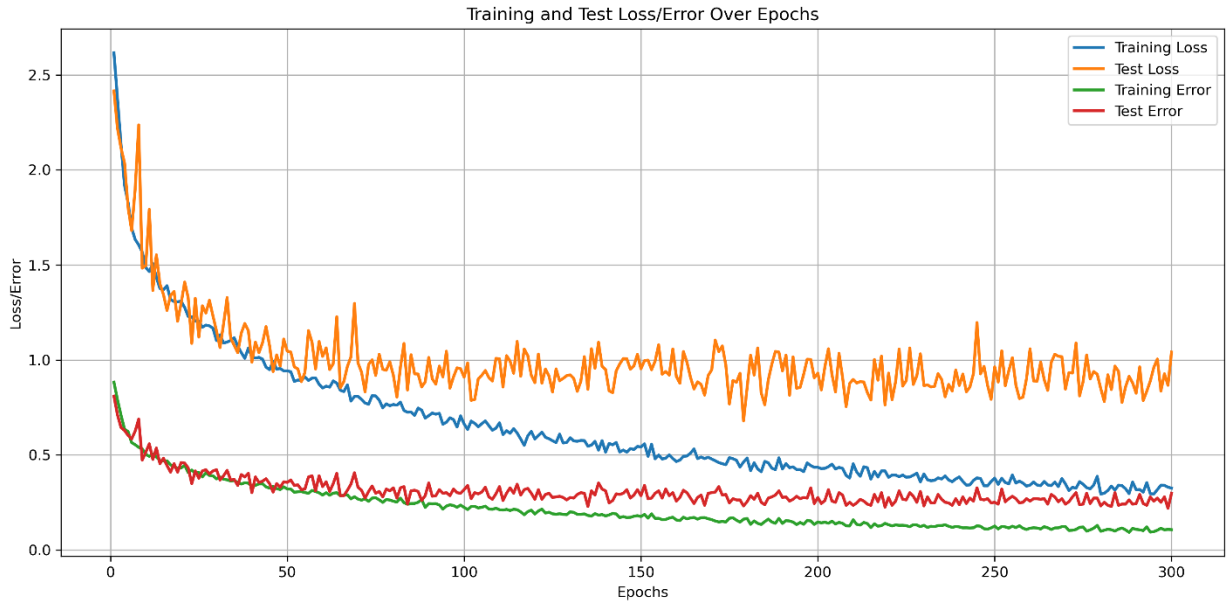

Figure 9: Training and Test Loss/Error Over Epochs

4.2.2. Adam optimizer

After training the model over 300 epochs, as shown in the performance metrics graphs, the model achieved the following approximate performance metrics:

- Accuracy: 74%
- Precision: 74%
- Recall: 74%
- F1 Score: 74%

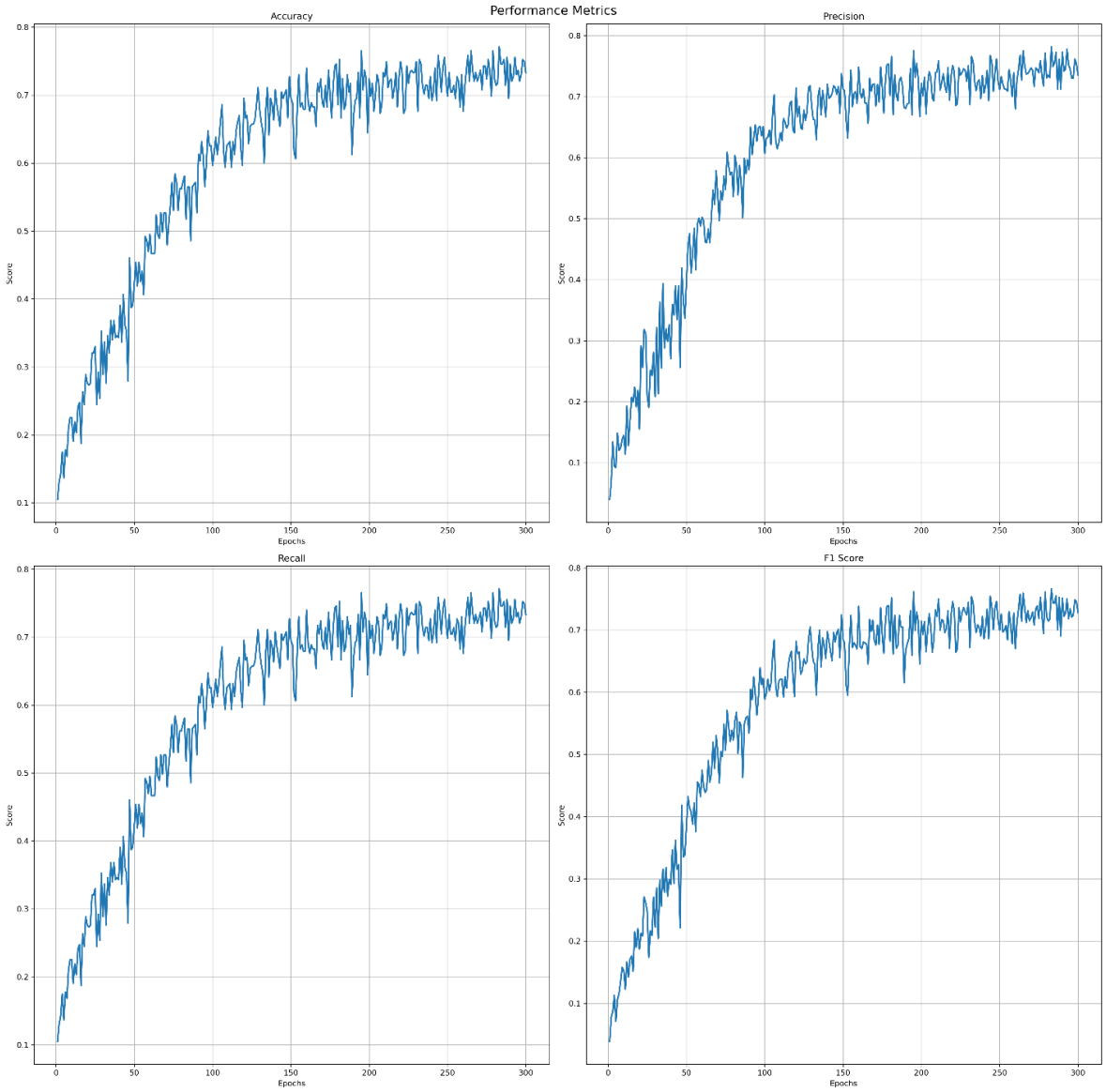

Figure 10: Performance Metrics Over Epochs

The plot in figure 11 displays the training and test loss/error over 300 epochs. Both the training loss and error decrease steadily, suggesting that the model is learning effectively. The training and test losses continue to move in tandem until epoch 175, when the test loss stabilizes, indicating a potential sign of overfitting. A similar trend is observed with the training and test errors.

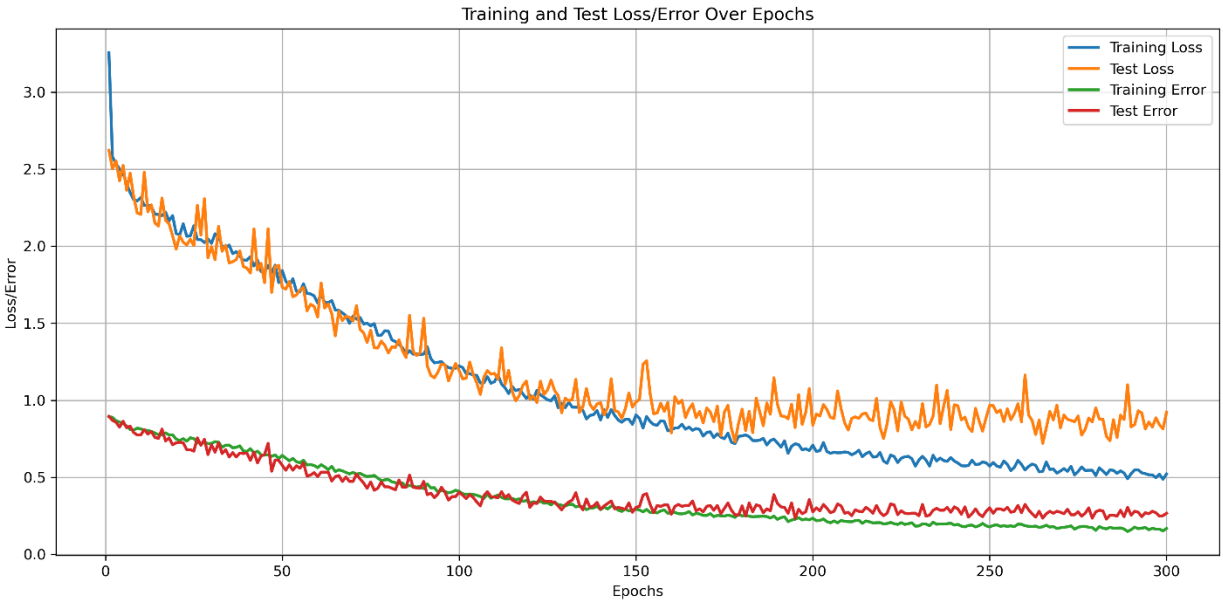

Figure 11: Training and Test Loss/Error Over Epochs

4.2.3. RMSprop optimizer

Now moving to the RMSprop optimizer, and analyzing the performance metrics from the figure 12, we can see that the model's performance has reached approximately 75% for accuracy, precision, recall, and F1 score over 300 epochs.

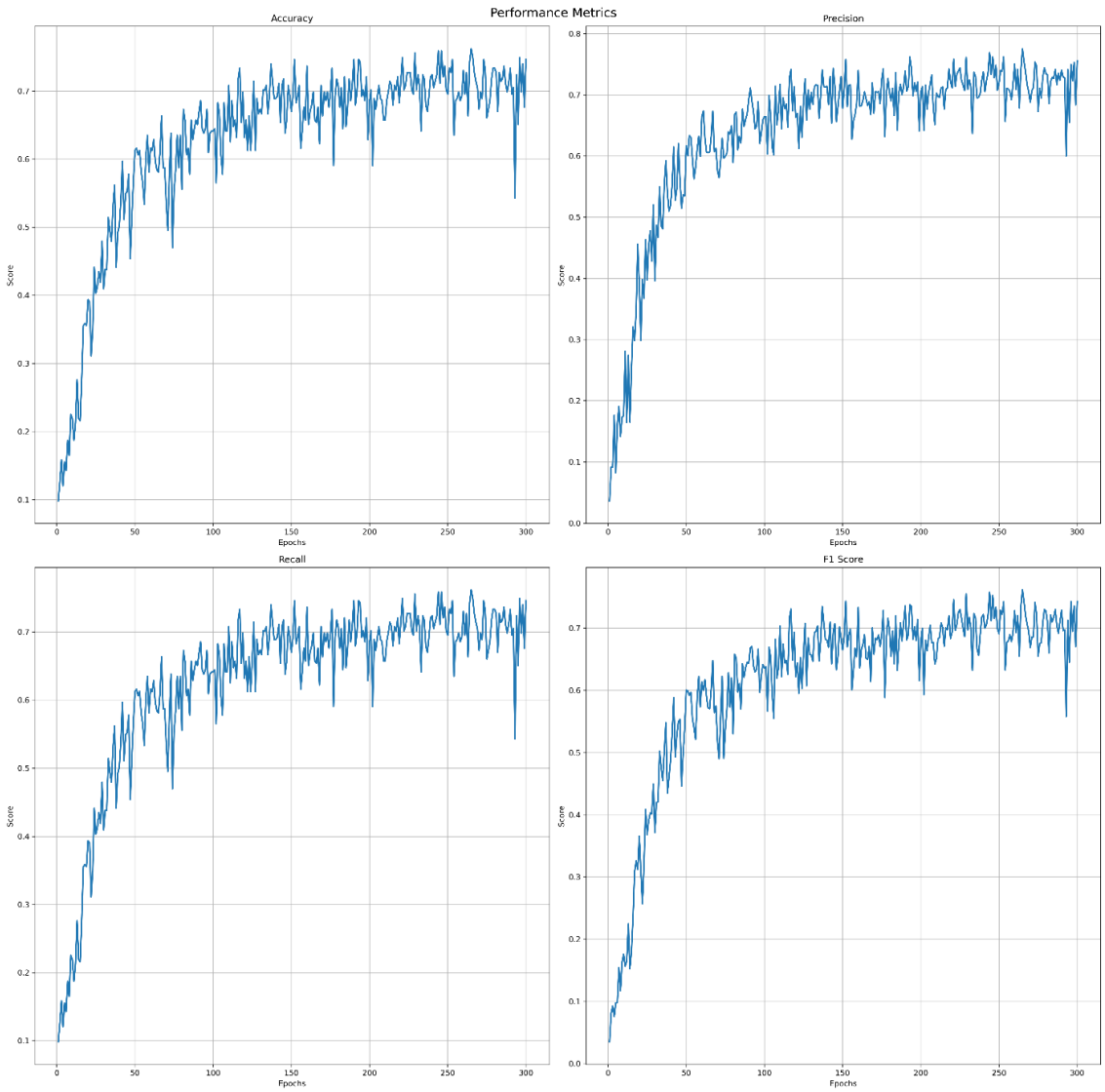

Figure 12: Performance Metrics Over Epochs

The plot in figure 13 displays training and test loss/error over 300 epochs. Both the training and test losses show a sharp initial decrease, followed by a steady decline, suggesting that the model is learning effectively. The training and test errors also decrease gradually and consistently. All metrics continue to move in tandem throughout the training process, with no clear divergence between training and test performance. This parallel trend indicates good generalization and no obvious signs of overfitting.

Figure 13: Training and Test Loss/Error Over Epochs

4.4.4. Comparison

The performance metrics for the three optimizers—Stochastic Gradient Descent (SGD), Adam, and RMSprop—revealed distinct outcomes. The SGD optimizer achieved the highest performance, with an accuracy, precision, recall, and F1 score of 77%. In contrast, the Adam optimizer yielded lower metrics of 74% across all categories, indicating effective learning but less optimal performance for this specific model. Meanwhile, the RMSprop optimizer achieved a performance level of 75%, slightly better than Adam but still below that of SGD.

In terms of training and test loss/error, the SGD model demonstrated a steady decline in both training and testing loss, with the training error continuously decreasing. However, the test loss began to stabilize after 50 epochs, suggesting signs of overfitting. The Adam optimizer exhibited a similar trend, with both training and test losses decreasing until epoch 175, after which the test loss stabilized. In contrast, the RMSprop optimizer displayed a sharp initial decrease in both training and test losses, followed by a steady decline, indicating effective learning without obvious signs of overfitting. Overall, while all optimizers effectively trained the model, SGD emerged as the best-performing optimizer, followed by RMSprop and then Adam.

Overall, while all optimizers effectively trained the model, SGD emerged as the best-performing optimizer, followed by RMSprop and then Adam. The training times for each optimizer over 300 epochs with an RTX 4070 GPU were as follows:

- **Adam**: 3873.89 seconds
- **RMSprop**: 3923.73 seconds

- **SGD**: 4075.16 seconds

V. Conclusion

In this assignment, we delved into the practical applications of Convolutional Neural Networks (CNNs), enhancing our understanding of their architectures and optimization techniques. Through two distinct parts, we trained models on the CIFAR-10 dataset and the 15-scene classification dataset, employing various strategies to improve model performance.

In Part 1, I effectively utilized PyTorch to implement a CNN model tailored for image classification, achieving an accuracy of 82% after training for 100 epochs. The implementation of batch normalization and careful data preprocessing played crucial roles in stabilizing the training process and reducing overfitting risks.

In Part 2, I explored the impact of different optimization algorithms—Stochastic Gradient Descent (SGD), Adam, and RMSprop—on a custom CNN architecture inspired by AlexNet. Each optimizer yielded unique performance metrics, with SGD resulting in the highest accuracy of 77%. The analysis of training and testing losses further emphasized the importance of monitoring model performance to mitigate overfitting and ensure generalization.

Overall, this assignment reinforced key concepts in deep learning, such as the significance of architecture design, the efficacy of various optimizers, and the need for robust data preprocessing techniques. The findings underscore the ongoing challenges and opportunities for refinement in CNN implementations, laying a solid foundation for future explorations in deep learning.

VI. Reference

[1] "Writing AlexNet from Scratch in PyTorch | DigitalOcean." Accessed: Sep. 29, 2024. [Online]. Available: https://www.digitalocean.com/community/tutorials/alexnet-pytorch

[2] "ELU — PyTorch 2.4 documentation." Accessed: Sep. 29, 2024. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.ELU.html