EEE 443/543 - Project #6

Introduction

In this project the aim was to design and train a Convolutional Neural Network for geometric shape classification. The given geometry dataset contains 90.000, 200x200 images each corresponding to one of the 9 classes: Circle, Square, Octagon, Heptagon, Nonagon, Star, Hexagon, Pentagon, Triangle. Each class has 10.000 images

Methodology

First a script has been written to split the dataset into training and testing sets. Training set contains 8.000 images per class and test set contains 2000 images per class.

After that a the neural network model has been designed that takes 200x200 images as input and decides the class of the image.

In the beginning the model in the torch3.py module has been took as reference while building the architecture. I implemented the model by adjusting the sizes. I initially run the model in my pc's cpu and the algorithm run about 15 hours on AMD Ryzen 7 cpu .

First model:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(614656, 128)
        self.fc2 = nn.Linear(128, 9)

    def forward(self, x):
        x = self.conv1(x)
        x = torch.relu(x)
        x = self.conv2(x)
        x = torch.relu(x)
        x = torch.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        return x
```

Fig.1 *First Architecture*

The initial network architecture in ***Fig.1*** consists of two convolutional layers followed by dropout, flattening, and fully connected layers. The model starts by defining two convolutional layers: the first with 1 input channel and 32 output channels, and the second with 32 input channels and 64 output channels. Dropout layers are added after each convolutional layer to prevent overfitting, with a dropout rate of 25%. After the convolutional layers, the output is passed through a max-pooling layer with a pool size of 2. The flattened output is then passed through two fully connected layers, reducing the output dimension to

128, and finally to 9, which correspond to the number of classes for classification. ReLU activations are used after each convolutional and fully connected layer.

The algorithm is run with 15 number of epochs, and train bacth size of 100 and test batch size of 1000.

The loss function is Cross Entropy Loss:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

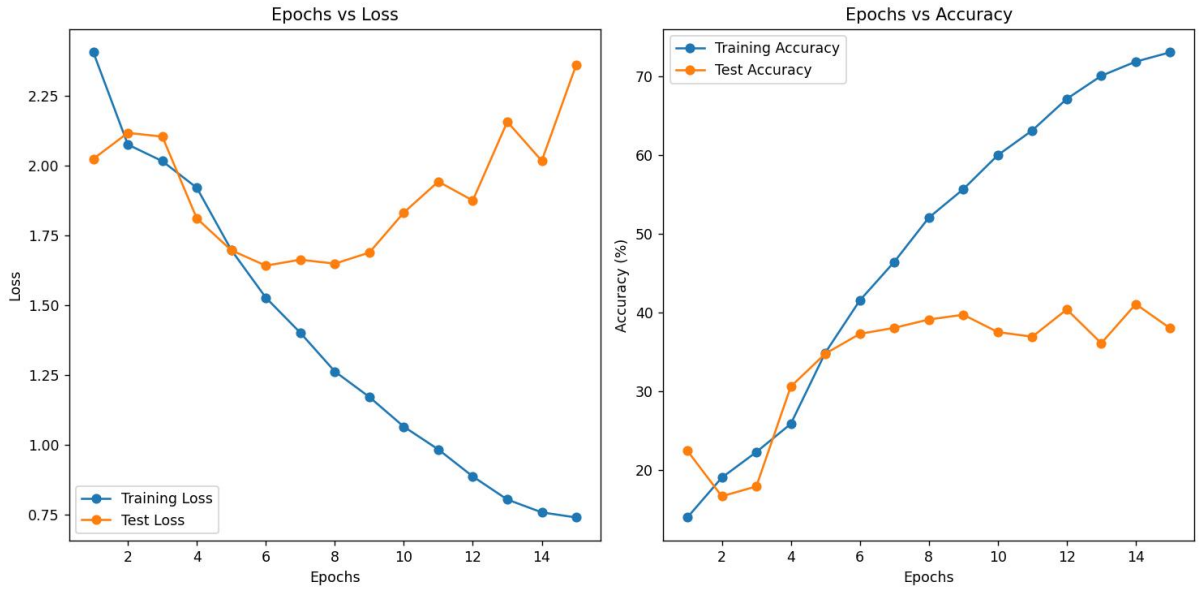
The optimizer is Adam (Adaptive Moment Estimation) optimizer learning rate lr = 0.001. Adam computes adaptive learning rates for each parameter by considering both the first moment (mean) and the second moment (variance) of the gradients.
And the model trained using backpropagation.

Before feeding the model the images are went through the data preproccessing pipeline, first the images converted to grayscale with a single output channel, then normalized to a tensor, resized to 200x200 pixels, and finally normalized. The normalization values (0.1307,) for the mean and (0.3081,) for the standard deviation in the transform pipeline come from the MNIST dataset since the model that I referenced optimized for MNIST dataset.

```
transform = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
    transforms.Resize((200, 200)),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

Fig.2 *First version of the data preprocessing pipeline*

After each train Epoch vs Loss and Epoch vs Accuracy graphs are plotted in order to visualize the results.




```
Epoch 15 Training Loss: 0.740920, Training Accuracy: 73.06%
Test Loss: 2.361462, Test Accuracy: 37.97%
```

Fig.3 *First Architecture Results*

From ***Fig.3*** it is visible that 73.06% training accuracy and 37.97% test accuracy were achieved. Even though train accuracy is sufficiently high the test accuracy was not sufficient and did not seem to increase. This indicated that the model was overfitting.

For the second run the biggest issue was the time it takes to run the model. In order to solve this issue I decided to use my pcs gpu (NVIDIA GeForce RTX 3050) instead of the cpu. I installed the NVIDIA CUDA on my computer for this purpose. After that the model training process sufficiently became faster, it now takes almost 1 hour to complete 15 epochs.

After that, the accuracy issue was assessed by changing the model architecture, I added another convolution layer along with new Batch Normalization and Pooling layers to the model to increase the accuracy.

Batch Normalization normalizes the inputs to a layer within a mini-batch, which helps stabilize and accelerate training.

Pooling is a technique used in CNNs to reduce the dimensions of the input feature maps, while retaining important information. It helps decrease computational load, memory usage, and reduces the risk of overfitting. The type of pooling that I used is max pooling (selecting the maximum value) from a specified window of the input.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.conv3 = nn.Conv2d(64, 128, 3, 1)

        self.batch_norm1 = nn.BatchNorm2d(32)
        self.batch_norm2 = nn.BatchNorm2d(64)
        self.batch_norm3 = nn.BatchNorm2d(128)

        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)

        # Pooling layers
        self.pool = nn.MaxPool2d(2, 2)  # Pooling

        # Adjust the size based on image dimensi
        self.fc1 = nn.Linear(67712, 256)  # Adju
        self.fc2 = nn.Linear(256, 9)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = torch.relu(x)
        x = self.batch_norm1(x)
        x = self.pool(x)  # Apply max

        x = self.conv2(x)
        x = torch.relu(x)
        x = self.batch_norm2(x)
        x = self.pool(x)  # Apply max

        x = self.conv3(x)
        x = torch.relu(x)
        x = self.batch_norm3(x)
        x = self.pool(x)  # Apply max

        x = torch.flatten(x, 1)  # Fl

        x = self.fc1(x)
        x = torch.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        return x
```

Fig.4 *Second Architecture*

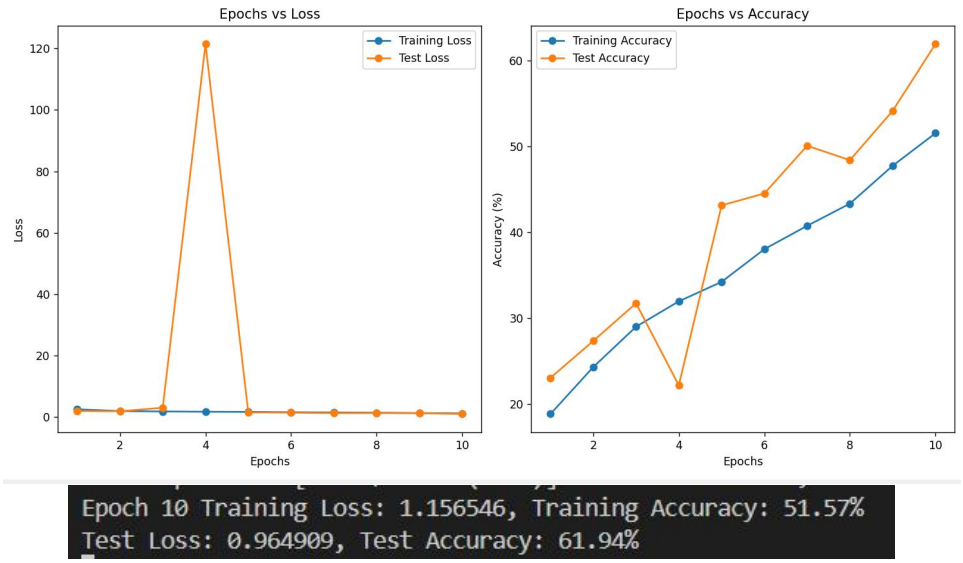


Fig.5 *Second Architecture Results*

From ***Fig.5*** it is visible that 51.57% training accuracy and 61.94% test accuracy were achieved after 10 epochs. Now the overfitting issue is resolved but accuracy was not sufficient enough.

In order to increase the accuracy the order of the ReLu and Batch Normalization steps have interchanged in the model architecture , which visible in ***Fig.6,*** to increase the accuracy.

The train and test batch sizes were reduced to 32 and 64 respectively.

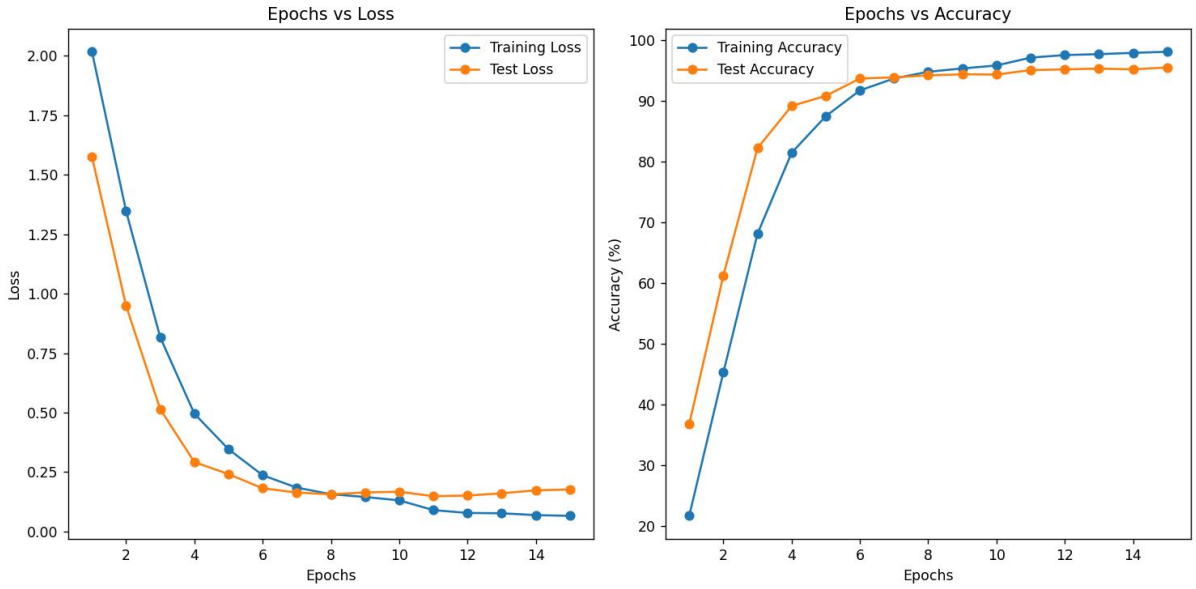
The order of the data preprocessing pipeline also rearranged from Grayscale -> ToTensor -> Resize -> Normalize to Resize -> Grayscale -> ToTensor -> Normalize as it is a more logical order, visible in ***Fig.6***.

Another addition was learning rate scheduler. Learning rate scheduler adjusts the learning rate at regular intervals (steps). It decays the learning rate by a factor of gamma = 0.7 every step_size = 5 number of epochs.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 25 * 25, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 9)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
        x = self.pool(torch.relu(self.bn3(self.conv3(x))))

        x = torch.flatten(x, 1)

        x = torch.relu(self.fc1(x))
        x = self.dropout(x)

        x = torch.relu(self.fc2(x))
        x = self.dropout(x)

        x = self.fc3(x)
        return x
```

```python
transform = transforms.Compose([
    transforms.Resize((200, 200)),
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

Fig.6 *Third Architecture and the data preprocessing pipeline*




```
Epoch 15 Training Loss: 0.066129, Training Accuracy: 98.08%
Test Loss: 0.177146, Test Accuracy: 95.49%
```

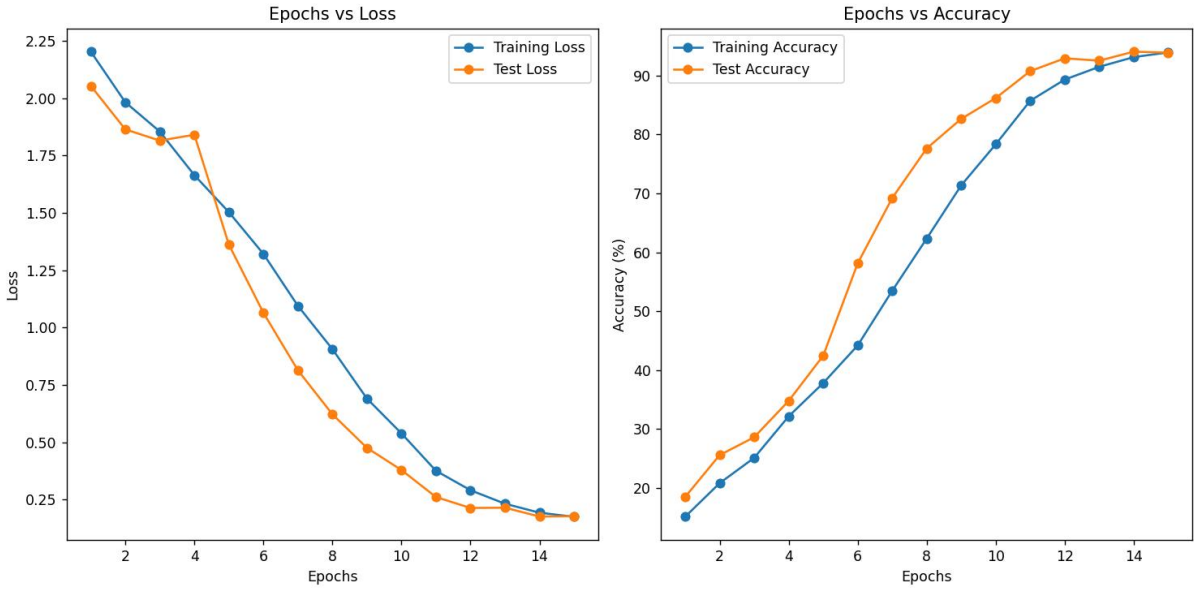
Fig.7 *Third Architecture Results*

From ***Fig.7*** it is visible that 98.08% training accuracy and 95.49% test accuracy were achieved after 15 epochs. This design achieved near perfect accuracy. However, the trained model has a size of 157MB. It is stated in the project manual that the model size should not

exceed 50MB therefore I had to reduce the size. In order to do this, one of the fully connected layers is omitted and the size of the fully connected layers are reduced.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 25 * 25, 256)
        self.fc2 = nn.Linear(256, 9)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
        x = self.pool(torch.relu(self.bn3(self.conv3(x))))
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Fig.8 *Forth Architecture*




```
Epoch 15 Training Loss: 0.175312, Training Accuracy: 93.93%
Test Loss: 0.178132, Test Accuracy: 93.88%
```

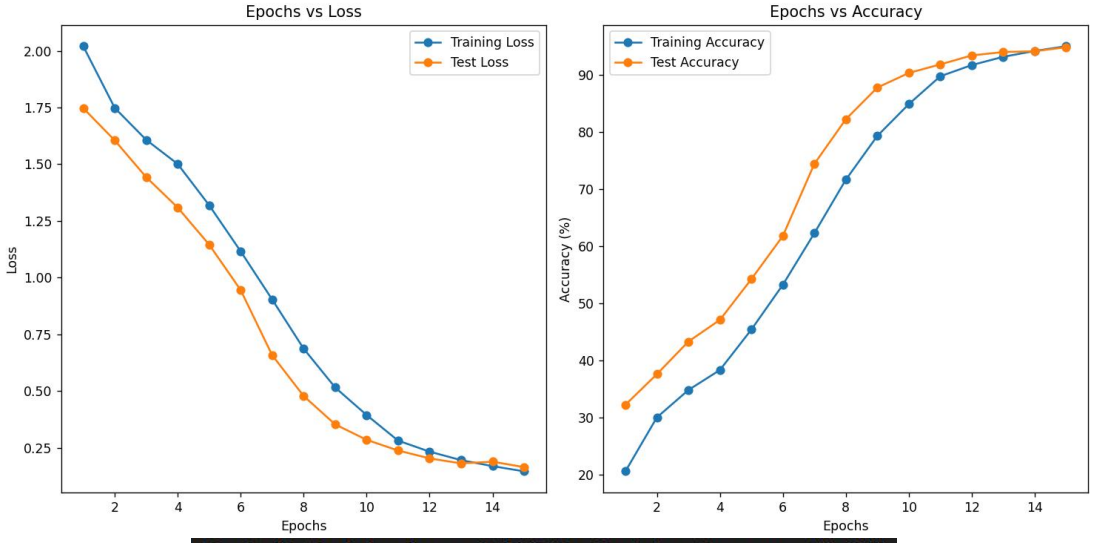
Fig.9 *Forth Architecture Results*

From ***Fig.9*** it is visible that 93.93% training accuracy and 93.88% test accuracy were achieved after 15 epochs. This design also achieved very decent results. However the size was 80 MB, still bigger than the requirement.

In the final design I reduced all the convolution layer sizes to half and rerun the model.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(64)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 25 * 25, 256)
        self.fc2 = nn.Linear(256, 9)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
        x = self.pool(torch.relu(self.bn3(self.conv3(x))))
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Fig.10 *Final Architecture*




```
Epoch 15 Training Loss: 0.146535, Training Accuracy: 95.00%
Test Loss: 0.165235, Test Accuracy: 94.82%
```

Fig.11 *Final Architecture Results*

From ***Fig.11*** it is visible that 95.00% training accuracy and 94.82% test accuracy were achieved after 15 epochs. This design also achieved very decent results. After training, the models size was 40MB which is below 50MB threshold. Therefore the algorithm was finalized.

Finally, an inference script was written as requested and it is run using a holdout set to be able to access the performance. The results were as intended. A sample output is provided below in ***Fig.12.***

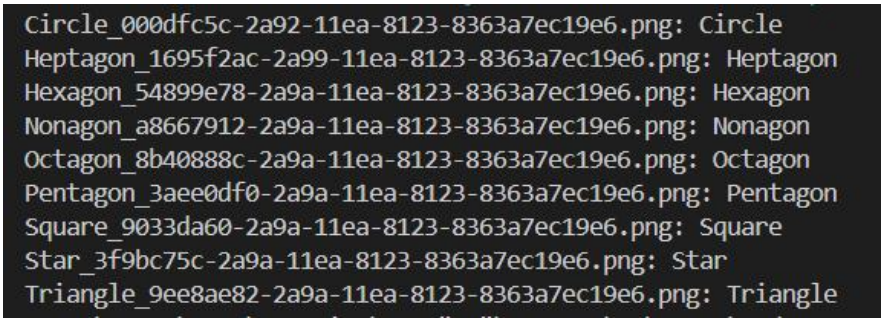


Fig.12 *Inference Script Sample Output*

It can be seen from ***Fig.12*** that the model successfully classifies the images in the inference set.

Conclusion

In conclusion, this project successfully designed, trained, and optimized a Convolutional Neural Network for geometric shape classification using a large dataset of 90,000 images. Through multiple iterations, the architecture was refined to improve both training and test accuracies while addressing issues like overfitting and model size. Initially, the model achieved satisfactory training accuracy but struggled with test accuracy, which was later improved by modifying the architecture, using batch normalization, adding pooling layers, and adjusting the data preprocessing pipeline.

By transitioning from CPU to GPU, training time was significantly reduced, allowing for faster experimentation. After optimizing the model, the final architecture achieved a remarkable 95.00% training accuracy and 94.82% test accuracy, all while ensuring the model size remained below the 50MB threshold. This design not only met the project requirements but also demonstrated the power of CNNs in geometric image classification tasks. The final inference script confirmed the model's ability to classify unseen data accurately, marking the project a success.