UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Department of Information Engineering,

Computer Science and Mathematics

DEEP NEURAL NETWORKS

DNN project

Profesors:

Profesor Giovanni Stilo

Profesor Andrea Manno

Student:

Vladyslav Bulhakov #287270

Contents

1 Tasks of the project

The project aims to compare the performances of two sets of three Convolutional Neural Networks (CNNs) by coding and in the report. The CNNs in each set share the same architecture but differ in their training/initialization schema. The performance comparisons are based on the MNIST 2-D dataset, and the CNNs are built and trained using the PyTorch library.

2 Dataset

All experiments on the CNNs was performed using the MNIST 2-D dataset.

The MNIST 2-D dataset is a collection of grayscale images of handwritten digits from 0 to 9. Each image has a resolution of 28x28 pixels. The dataset is commonly used as a benchmark in computer vision and machine learning tasks. It is divided into a training set and a test set. The training set contains many labeled images used for model training, while the test set consists of a smaller number of labeled images used for evaluation. The goal of the MNIST dataset is typically to train models that can accurately classify handwritten digits.

To download our dataset, I will be use built-in method in PyTorch library

```
    train_dataset = datasets.MNIST(
    root="./data",
    train=True,
    transform=Compose([ToTensor()]),
    download=True,
)
test_dataset = datasets.MNIST(
    root="./data",
    train=False,
    transform=Compose([ToTensor()]),
    download=True)
```

By loading the MNIST dataset using the code, I can access the training and test samples along with their corresponding labels for further processing and model training.

After downloading a dataset can create a data loader

```
train_dataloader = DataLoader(train_dataset, batch_size=64)
test_dataloader = DataLoader(test_dataset, batch_size=64)
```

The `train_dataloader` and `test_dataloader` objects can be used to iterate over the training and test datasets, respectively, in batches. Each iteration provides a batch of samples and their corresponding labels, which can be fed into your model for training or evaluation.

We can print our dataset

```
    figure = plt.figure(figsize=(10, 8))
cols, rows = 5, 5
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(train_dataset), size=(1,)).item()
    img, label = train_dataset[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(label)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

The code generates a figure showing a grid of 25 randomly selected images from the MNIST training dataset.

3 Drawing graphs

For visualization, I will be using helper function `plot_training_progress`

```
    def plot_training_progress(losses, accuracies):
    epochs = range(1, len(losses) + 1)

    # Plotting losses
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.plot(epochs, losses, '-o')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
```

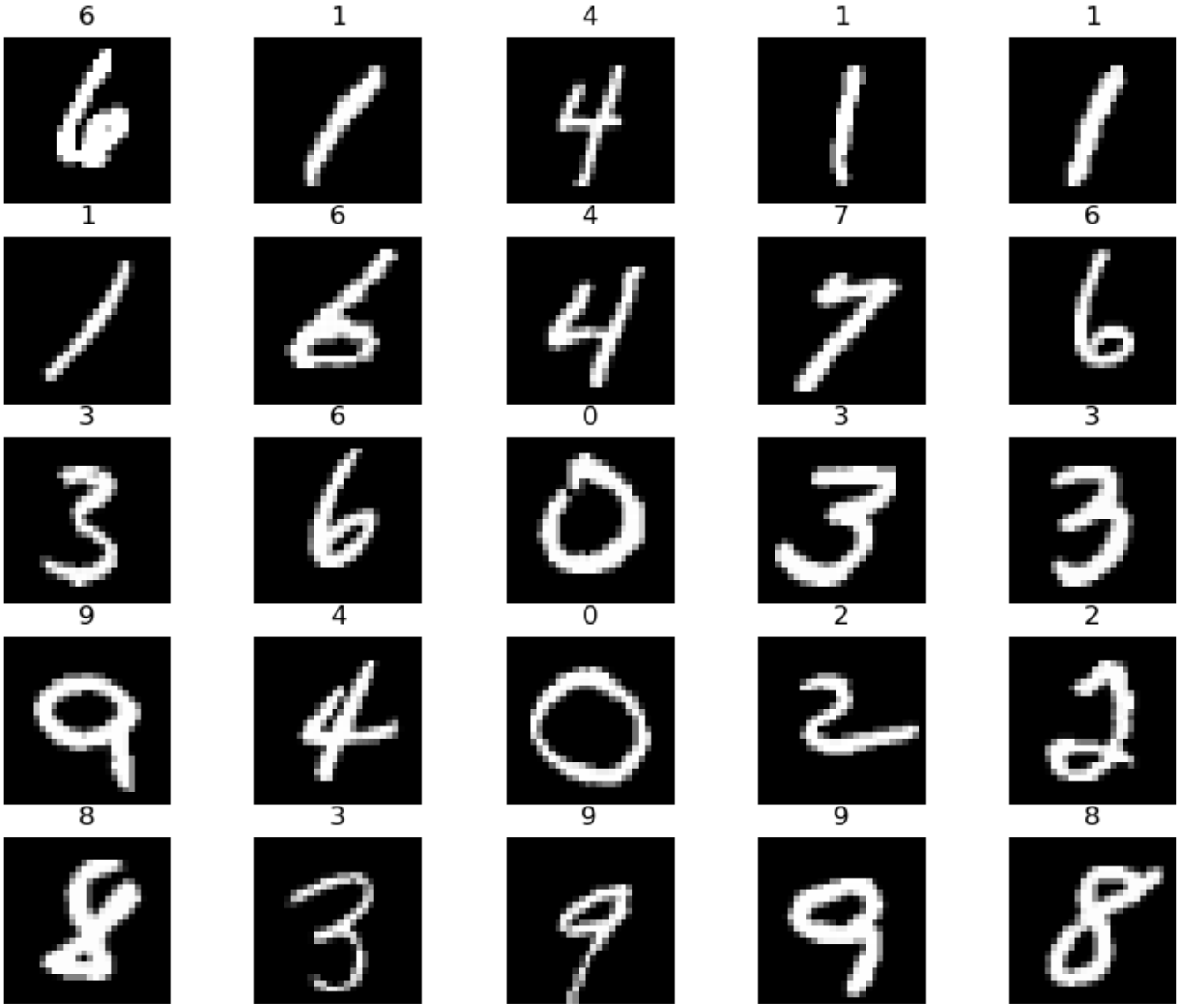

Figure 1: Example MNIST DataSet.

```
    plt.title('Training Loss')

    # Plotting accuracies
    plt.subplot(1, 2, 2)
    plt.plot(epochs, accuracies, '-o')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Training Accuracy')

    plt.tight_layout() # Adjust the padding between and around subplots.
    plt.show()
```

The function `plot_training_progress` takes two arguments:

- `losses`: A list containing the training losses for each epoch.
- `accuracies`: A list containing the training accuracies for each epoch.

The function creates a figure with two subplots: one for plotting the training losses and the other for plotting the training accuracies. It uses the `plt.plot` from MatPlotlib function to create line plots, where the x-axis represents the epochs and the y-axis represents the losses and accuracies.

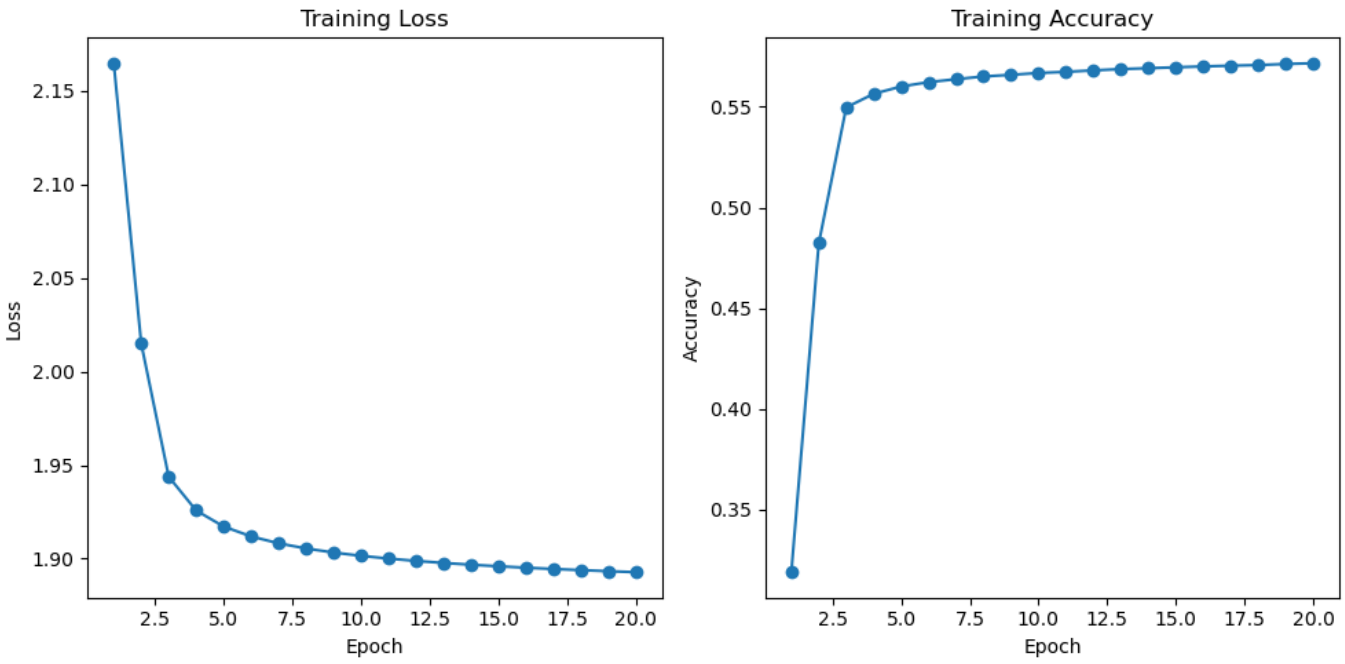

Figure 2: Example output of function.

4 CNN Architectures

Set 1: A1 CNN Architecture:

- One convolutional layer with 4 kernels
- One fully connected layer
- One Softmax output layer
- Cross entropy loss

```
class CNN_A1(nn.Module):
def __init__(self):
    super(CNN_A1, self).__init__()
    self.conv1 = nn.Conv2d(1, 4, kernel_size=3)
    # 1 input channel, 4 output channels
    self.relu = nn.ReLU()
```

```
        self.fc = nn.Linear(4 * 26 * 26, 10)
        # 4 * 26 * 26 is the output size of the convolutional layer,
        # 10 - output this labels(classes of out DataSet - numbers from 0-9)
        self.relu1 = nn.ReLU()
        self.softmax = nn.Softmax(dim=1) #SoftMax function


    def forward(self, x):
        x = self.conv1(x) # Convolutional layer
        x = self.relu1(x) # ReLU activation
        x = torch.flatten(x, 1) # Flatten the output
        x = self.fc(x) # Fully connected layer
        x = self.relu1(x) # ReLU activation
        x = self.softmax(x) # Softmax
        return x
```

Overall, the CNN_A1 class represents a CNN with one convolutional layer, one fully connected layer, and corresponding activation functions. It is designed for the MNIST dataset with 10 classes (digits 0 to 9), and the forward pass computes the output probabilities for each class.

Set 2: A2 CNN Architecture:

- One convolutional layer with 4 kernels
- One additional convolutional layer (number of kernels of your choice)
- Optionally, more convolutional layers can be added
- One fully connected layer
- One Softmax output layer
- Cross entropy loss

```
class CNN_A2(nn.Module):
def __init__(self, num_kernels):
    super(CNN_A2, self).__init__()
    self.conv1 = nn.Conv2d(1, 4, kernel_size=3)
    # 1 input channel, 4 output channels
    self.conv2 = nn.Conv2d(4, num_kernels, kernel_size=3)
    # 4 input channels, variable number of output channels
    self.fc = nn.Linear(num_kernels * 24 * 24, 10)
    # num_kernels * 24 * 24 is the output size of the last convolutional layer
    self.softmax = nn.Softmax(dim=1)
    self.relu = nn.ReLU()

def forward(self, x):
    x = self.conv1(x)
    x = self.relu(x)
    x = self.conv2(x)
    x = self.relu(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)
    x = self.softmax(x)
    return x
```

In summary, this code defines the architecture and forward pass of the CNN for Set 2 (A2). The number of output channels in the second convolutional layer is variable and determined by the num_kernels parameter. The ReLU activation function is applied after each convolutional layer, and the softmax activation is applied to the final output.

5 Training/Initialization schemas

To clarify the training/initialization schema for the CNNs in each set, there are two possible strategies: the "By-hand" strategy and the "Default" strategy.

The "By-hand" strategy refers to initializing the kernels using only 0 and 1 values. The specific pattern can be chosen, but each kernel must contain a different pattern.

The "Default" strategy refers to using the default initialization or another built-in initialization method such as Kaiming, Xavier, etc. These initialization methods are commonly used in deep learning frameworks.

Before we are going to initialization schemas, I want to represent a function for generating weights

```
def generate_weights_default(model):
    weights = []
    biases = []
    for module in model.modules():
        if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear):
            weight = nn.init.normal_(module.weight.data.clone())
            bias = nn.init.constant_(module.bias.data.clone(), 0)
            weights.append(weight)
            biases.append(bias)
    return weights, biases
```

The function generate_weights_default generates weights and biases for the layers of a given CNN model using the "Default" strategy. By using this function, you can generate weights and biases for the layers of a CNN model using the "Default" strategy, where the weights are initialized with random values from a normal distribution, and the biases are initialized with a constant value of 0.

HF schema: The first layer of the CNN is initialized with the by-Hand strategy, the remain-

ing layers are initialized with the default one. All the layers excep tthe first one (which is Frozen) are trained.

```
def initialize_weights_hf(model):
device = next(model.parameters()).device
with torch.no_grad():
    weights = torch.FloatTensor([[
      [1, 0, 0], [0, 1, 0], [0, 0, 1]],
      [[0, 0, 1], [0, 1, 0], [1, 0, 0]],
      [[0, 1, 0], [0, 1, 0], [0, 1, 0]],
      [[0, 0, 0], [1, 1, 1], [0, 0, 0]]]).to(device)

weights = weights.view(4, 1, 3, 3)
model.conv1.weight = nn.Parameter(weights, requires_grad=False)
```

The function initialize_weights_hf is used to initialize the weights of the convolutional layer in the CNN model according to the "By-hand" strategy. By using this function, you can initialize the weights of the convolutional layer in the CNN model according to the "By-hand" strategy with the specified pattern.

HT schema: The first layer of the CNN is initialized with the by-Hand strategy, the remaining layers are initialized with the default one. All the layers, including the first one, are trained.

```
def initialize_weights_ht(model, weights, biases):
device = next(model.parameters()).device
# Initialize first layer with by-Hand strategy
with torch.no_grad():
    hand_weights = torch.FloatTensor([
        [[1, 0, 0], [0, 1, 0], [0, 0, 1]],
        [[0, 0, 1], [0, 1, 0], [1, 0, 0]],
        [[0, 1, 0], [0, 1, 0], [0, 1, 0]],
        [[0, 0, 0], [1, 1, 1], [0, 0, 0]]
    ]).to(device)
```

```
        hand_weights = hand_weights.view(4, 1, 3, 3)
        model.conv1.weight = nn.Parameter(hand_weights, requires_grad=True)

    # Initialize remaining layers with the generated weights and biases
    index = 0
    for module in model.modules():
        if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear):
            if module is not model.conv1:
                module.weight.data.copy_(weights[index].to(device))
                module.bias.data.copy_(biases[index].to(device))
                index += 1
```

The function initialize_weights_ht is used to initialize the weights and biases of the CNN model with a combination of the "By-hand" and "Default" strategies. By using this function, you can initialize the weights and biases of the CNN model with the "By-hand" strategy for the first layer (conv1) and the generated weights and biases for the remaining layers.

DT schema: All the layers, including the first one, of the CNN are initialized with the default strategy. All the layers, including the first one, are trained.

```
def initialize_weights_dt(model, weights, biases):
index = 0
for module in model.modules():
    if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear):
        module.weight.data.copy_(weights[index])
        module.bias.data.copy_(biases[index])
        index += 1
```

The function initialize_weights_dt is used to initialize the weights and biases of the CNN model with the "Default" strategy. By using this function, you can initialize the weights and biases of the CNN model with the "Default" strategy, where the weights and biases are directly copied from the generated weights and biases.

6 Training

Firstly, we add a loss functions

```
loss_fn = nn.CrossEntropyLoss()
```

initializes an instance of the Cross Entropy Loss class from the torch.nn module. Then we add function for training our Neural Network `train_cnn`

```
def train_cnn(model, loss_fn, optimizer, train_loader, num_epochs):
device = next(model.parameters()).device  # Get the device of the model's
# parameters

losses = []
accuracies = []

for epoch in range(num_epochs):
    running_loss = 0.0
    correct_predictions = 0
    total_predictions = 0

    for images, labels in train_loader:
        images = images.to(device)  # Move input tensor to the same device as the m
        labels = labels.to(device)  # Move label tensor to the same device as the m

        optimizer.zero_grad()

        outputs = model(images)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

```

```
            _, predicted = torch.max(outputs.data, 1)
            total_predictions += labels.size(0)
            correct_predictions += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
        epoch_accuracy = correct_predictions / total_predictions

        losses.append(epoch_loss)
        accuracies.append(epoch_accuracy)

        print(f"Epoch {epoch + 1}/{num_epochs},
        Loss: {epoch_loss:.4f}, Accuracy: {epoch_accuracy:.4f}")

    return losses, accuracies
```

The `train_cnn` function takes a CNN model, loss function, optimizer, training data loader, and the number of epochs as input. It performs the training process for the given number of epochs and returns the losses and accuracies recorded during training. The train_cnn function performs the training loop for the CNN model using the provided loss function, optimizer, and training data. It updates the model parameters based on the computed gradients, tracks the loss and accuracy for each epoch, and returns the recorded values for further analysis.

To use training function, we need to add some several variables

```
num_epochs = 20
learning_rate = 0.001
batch_size = 32
```

After this, we can initialize our Networks

```
cnn1 = CNN_A1()
cnn1 = cnn1.to(device)
initialize_weights_hf(cnn1) # Initialize with HF schema
```

add an Optimizer

```
optimizer1 = torch.optim.SGD(cnn1.parameters(), lr=learning_rate)
```

the code initializes an SGD(stochastic gradient descent) also I am using the specified learning rate (`learning_rate`). The optimizer will update the parameters of the cnn1 model during training using the SGD algorithm.

After initializing all our helper networks, I started to initialize our training loop.

```
losses1, accuracies1 = train_cnn(cnn1, loss_fn, optimizer1,
train_dataloader, num_epochs)
```

The function returns the training losses and accuracies as lists over the specified number of epochs. These lists will be used for further analysis or visualization.

I repeat the same staff to all other networks, code details is on GitHub.

7 Testing

After training our networks, we can test our Networks with specific function `test_model`

```
1     def test_model(model, test_loader):
2     model.eval()  # Set the model to evaluation mode
3     total_samples = len(test_loader.dataset)
4     correct_predictions = 0
5
6     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
7
8     with torch.no_grad():
9         for images, labels in test_loader:
10            images = images.to(device)  # Move input tensor to the same
11            # device as the model
12            labels = labels.to(device)  # Move labels tensor to the same
13            # device as the model
14
15            # Forward pass
16            outputs = model(images)
17
18            # Get predicted labels
19            _, predicted = torch.max(outputs.data, 1)
20
21            # Count correct predictions
22            correct_predictions += (predicted == labels).sum().item()
23
24    # Calculate accuracy
25    accuracy = (correct_predictions / total_samples) * 100
26
27    print(f"Test Accuracy: {accuracy:.2f}%")
```

this function take two arguments

- `model`: The trained model to be tested.
- `test_loader`: The data loader for the test dataset.

The function evaluates the model on the test dataset by iterating over the test loader. It sets the model to evaluation mode using `model.eval()` to disable gradient calculation and enable evaluation-specific behavior.

Inside the `with torch.no_grad()` context, the function performs the forward pass on the test images using the trained model.

The function counts the number of correct predictions by comparing the predicted labels with the true labels. Finally, it calculates the accuracy by dividing the number of correct predictions by the total number of samples in the test dataset and multiplying by 100.

8 Results

8.1 Experiment 1 - Inset Comparison

Training A1 To compare the performance of CNN_A1 with different initialization schemas, we will analyze the results obtained for each schema: HF, HT, and DT.

1. HF Schema
    - After 20 epochs, the model achieved an accuracy of 57.16%.
    - Loss decreased from 2.4076 to 1.8928 during training.
2. HT Schema
    - After 20 epochs, the model achieved an accuracy of 11.80%.
    - Loss decreased from 2.4011 to 2.3414 during training.
3. DT Schema
    - After 20 epochs, the model achieved an accuracy of 15.38%.
    - Loss decreased from 2.4011 to 2.3055 during training.

Observations:

- The HF schema, where the first layer is initialized with the "by-Hand" strategy, shows the best results. It achieves the highest accuracy of 57.16% after 20 epochs.
- The HT schema, where both the first and remaining layers are initialized with the "by-Hand" and default strategies, respectively, shows a lower accuracy of 11.80%.
- The DT schema, where all layers are initialized with the default strategy, performs better than

the HT schema but still has a lower accuracy of 15.38%.

Conclusion: The choice of initialization schema significantly impacts the performance of CNN_A1. The HF schema, with the "by-Hand" initialization for the first layer, outperforms the other schemas in terms of accuracy. It allows the model to learn and converge faster, resulting in better overall performance. Therefore, for the CNN_A1 model, it is recommended to use the HF schema for better results.

Testing A1

- Test Accuracy: 57.39% - CNN_A1 - HF Schema
- Test Accuracy: 15.55% - CNN_A1 - HT Schema
- Test Accuracy: 12.28% - CNN_A1 - DT Schema

Training A2 To compare the performance of CNN_A2 with different initialization schemas, we will analyze the results obtained for each schema: HF, HT, and DT.

1. HF Schema
    - After 20 epochs, the model achieved an accuracy of 90.01%.
    - Loss decreased from 2.0889 to 1.5726 during training.
2. HT Schema
    - After 20 epochs, the model achieved an accuracy of 11.80%.
    - Loss decreased from 2.2354 to 2.1737 during training.
3. DT Schema
    - After 20 epochs, the model achieved an accuracy of 15.38%.

- Loss stabilized at 2.3517 during training.

Observations:

- The HF schema shows significant improvement in accuracy compared to the other schemas. The accuracy starts at 45.70% and steadily increases to 90.01% by the 20th epoch. This indicates that initializing the first layer with a "by-Hand" strategy and training the subsequent layers leads to better performance.

- The HT schema demonstrates limited improvement in accuracy compared to the DT schema. The accuracy starts at 22.54% and reaches 28.71% by the 20th epoch. Although the first layer is initialized with a "by-Hand" strategy, training all layers does not significantly enhance the performance.

- The DT schema shows the lowest accuracy among the three schemas. The accuracy remains constant at 10.92% throughout all 20 epochs. Initializing all layers, including the first one, with the default strategy and training them does not lead to notable improvements.

Conclusion Based on the results, the HF schema performs the best, followed by the HT schema, while the DT schema performs the worst. Initializing the first layer with a "by-Hand" strategy and training the subsequent layers seems to contribute to improved accuracy.

Testing A1

- Test Accuracy: 91.20% - CNN_A1 - HF Schema

- Test Accuracy: 28.64% - CNN_A1 - HT Schema

- Test Accuracy: 11.30% - CNN_A1 - DT Schema

Overall, both sets A1 and A2 showed that initializing the first layer of the model with an HF schema resulted in the best performance. This highlights the importance of appropriate initialization

techniques for neural network models.

8.2 Experiment 2 – architecture comparison

1. CNN_A1(HF) → CNN_A2(HF)

CNN_A1 - HF Schema:

- Epoch 1/20, Loss: 2.1648, Accuracy: 0.3189
- Epoch 20/20, Loss: 1.8928, Accuracy: 0.5716
- Test Accuracy: 57.39

CNN_A2 - HF Schema:

- Epoch 1/20, Loss: 2.0889, Accuracy: 0.4570
- Epoch 20/20, Loss: 1.5726, Accuracy: 0.9001
- Test Accuracy: 91.20

Observations:

- Both CNN_A1 and CNN_A2 start with different initial accuracies and losses in the first epoch.
- As training progresses, CNN_A1 shows gradual improvement in accuracy and reduction in loss, although the increase in accuracy is not as significant as in CNN_A2.
- In contrast, CNN_A2 demonstrates a remarkable improvement in accuracy, starting at 45.70% and reaching an impressive accuracy of 90.01% by the 20th epoch.

Conclusion:

- The direct comparison between CNN_A1 and CNN_A2 with the HF initialization schema clearly shows that CNN_A2 outperforms CNN_A1 in terms of both accuracy and loss.

- CNN_A2 achieves a higher accuracy of 91.20% compared to 57.39% achieved by CNN_A1.

2. CNN_A1(HT) → CNN_A2(HT)

CNN_A1 - HT Schema:

- Epoch 1/20, Loss: 2.4076, Accuracy: 0.0509
- Epoch 20/20, Loss: 2.3055, Accuracy: 0.1538
- Test Accuracy: 15.55%

CNN_A2 - HT Schema:

- Epoch 1/20, Loss: 2.3517, Accuracy: 0.1092
- Epoch 20/20, Loss: 2.3517, Accuracy: 0.1092
- Test Accuracy: 28.64%

Observations:

- Both CNN_A1 and CNN_A2 with the HT initialization schema start with relatively low accuracies and high losses in the first epoch.
- As training progresses, both models show improvement in accuracy, but the increase is more significant in CNN_A2 compared to CNN_A1.
- CNN_A1 achieves a final accuracy of 15.55% after 20 epochs, while CNN_A2 achieves a higher accuracy of 28.64% within the same number of epochs.

Conclusion:

- The direct comparison between CNN_A1 and CNN_A2 with the HT initialization schema reveals

that CNN_A2 performs better in terms of both accuracy and loss.

- CNN_A2 achieves a higher accuracy of 28.64% compared to 15.55% achieved by CNN_A1.

3. CNN_A1(DT) → CNN_A2(DT)

CNN_A1 - DT Schema:

- Epoch 1/20, Loss: 2.4011, Accuracy: 0.0576
- Epoch 20/20, Loss: 2.3414, Accuracy: 0.1180
- Test Accuracy: 12.28%

CNN_A2 - HT Schema:

- Epoch 1/20, Loss: 2.2354, Accuracy: 0.2254
- Epoch 20/20, Loss: 2.1737, Accuracy: 0.2871
- Test Accuracy: 11.30

Observations:

- Both CNN_A1 and CNN_A2 with the DT initialization schema start with low accuracies and high losses in the first epoch.
- CNN_A1 shows a slight improvement in accuracy over the epochs but remains at a low accuracy of 12.28% after 20 epochs.
- CNN_A2, on the other hand, exhibits the same accuracy of 11.3% throughout all 20 epochs, indicating a lack of learning or convergence.
- Both models show minimal changes in loss values throughout training.

Conclusion:

- The comparison between CNN_A1 and CNN_A2 with the DT initialization schema reveals that neither model performs well in terms of accuracy.
- CNN_A1 achieves a slightly higher accuracy of 12.28% compared to the constant accuracy of 11.30% in CNN_A2.

8.3 Experiment 3 - Recovery Comparison

A2-HF to all the CNNs in the set A1 **CNN_A2 - HF Schema:**

```
Epoch 1/20, Loss: 2.0889, Accuracy: 0.4570
Epoch 20/20, Loss: 1.5726, Accuracy: 0.9001
Test Accuracy: 91.20%
```

CNN_A1 - HF Schema:

```
Epoch 1/20, Loss: 2.1523, Accuracy: 0.3348
Epoch 20/20, Loss: 1.8762, Accuracy: 0.5992
Test Accuracy: 59.95%
```

CNN_A1 - HT Schema:

```
Epoch 1/20, Loss: 2.1523, Accuracy: 0.3348
Epoch 20/20, Loss: 1.8762, Accuracy: 0.5992
Test Accuracy: 59.95%
```

CNN_A1 - DT Schema:

```
Epoch 1/20, Loss: 2.4011, Accuracy: 0.0576
Epoch 20/20, Loss: 2.3414, Accuracy: 0.1180
```

```
Test Accuracy: 11.92%
```

Observations

- A2-HF achieves the highest test accuracy of 91.20% among all the models, indicating superior performance.
- CNN_A1 with HF schema achieves a test accuracy of 59.95%, while CNN_A1 with HT and DT schemas perform significantly lower with test accuracies of 15.55
- It's evident that A2-HF outperforms all the CNN_A1 models in terms of both accuracy and loss.

Conclusion

- A2-HF demonstrates superior performance compared to the CNN_A1 models in this experiment, achieving the highest test accuracy of 91.20% after 20 epochs.
- The HF initialization schema seems to contribute significantly to the improved performance of A2-HF compared to the CNN_A1 models.
- The choice of architecture and initialization schema has a substantial impact on the performance and learning capability of the CNN models. In this case, A2-HF performs the best, while CNN_A1 with HT and DT schemas show lower accuracy and slower convergence.