UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DEPARTMENT OF INFORMATION ENGINEERING

Computer Science and Mathematics

DEEP NEURAL NETWORKS DNN PROJECT

Professors: Prof. Giovanni Stilo, PhD

Teaching collaborator Andrea Manno, PhD

Student: Kateryna Karabin

MatricolaID: #287272

2023

CONTENT

Part 1. Project Overview

In the project we compared the performances of two sets of three Convolutional Neural Networks (CNNs) by coding and in the report. The CNNs in the same set share the same architecture but differ in their training/initialization schema.

Technical requirements that we comply with:

- The CNNs built and trained using the Pytorch library.
- Experiments on the CNNs performed using the MNIST 2-D dataset.
- Code submitted in a Python file or Jupyter Notebook.
- Experiments described in the report (pdf).
- Performance comparison and results are available between 2 sets of 3 CNNs.

1.1. Library

To get started, we imported the following libraries:

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
from torchvision.transforms import ToTensor, Compose
```

First, the work started on colab. research, so we did a check. In order for us to be able to train our model on a hardware accelerator, we checked whether torch.cuda or torch.backends.mps are available, otherwise we use the Central Processing Unit (CPU).

1.2. Dataset

We used the MNIST 2-D dataset with the following parameters: root, train transform, download. This dataset is a collection of grayscale images of handwritten digits from 0 to 9 and each image has a resolution of 28x28 pixels.

Such as dataset is divided into a training set and a test set. Also contains many labeled images used to train the model and consists of smaller number of labeled images used for evaluation, respectively. We used this data set to train models that can accurately classify handwritten digits.

In the PyTorch library, there are several built-in methods and utilities available for various tasks in deep learning. Classes in this library simplify the process of loading and preprocessing data for training and evaluation.

```
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)
```

For example, we created data loaders for training and testing data sets using the DataLoader class, which is responsible for iterating over a data set, providing the ability to batch access data. In our code, the packet size is set to 64, meaning that during training or testing, the data will be divided into packets, and each packet will contain 64 samples.

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

For demonstration, we generated a figure showing a grid of 25 randomly selected images from the MNIST training dataset.

```
fig = plt. figure ( figsize =(7, 5))
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 ]
  fig.add_subplot (rows , cols , i)
  plt.title ( label )
  plt.axis ("off")
  plt.imshow (img. squeeze () , cmap ='Blues')
  plt.show ()
```

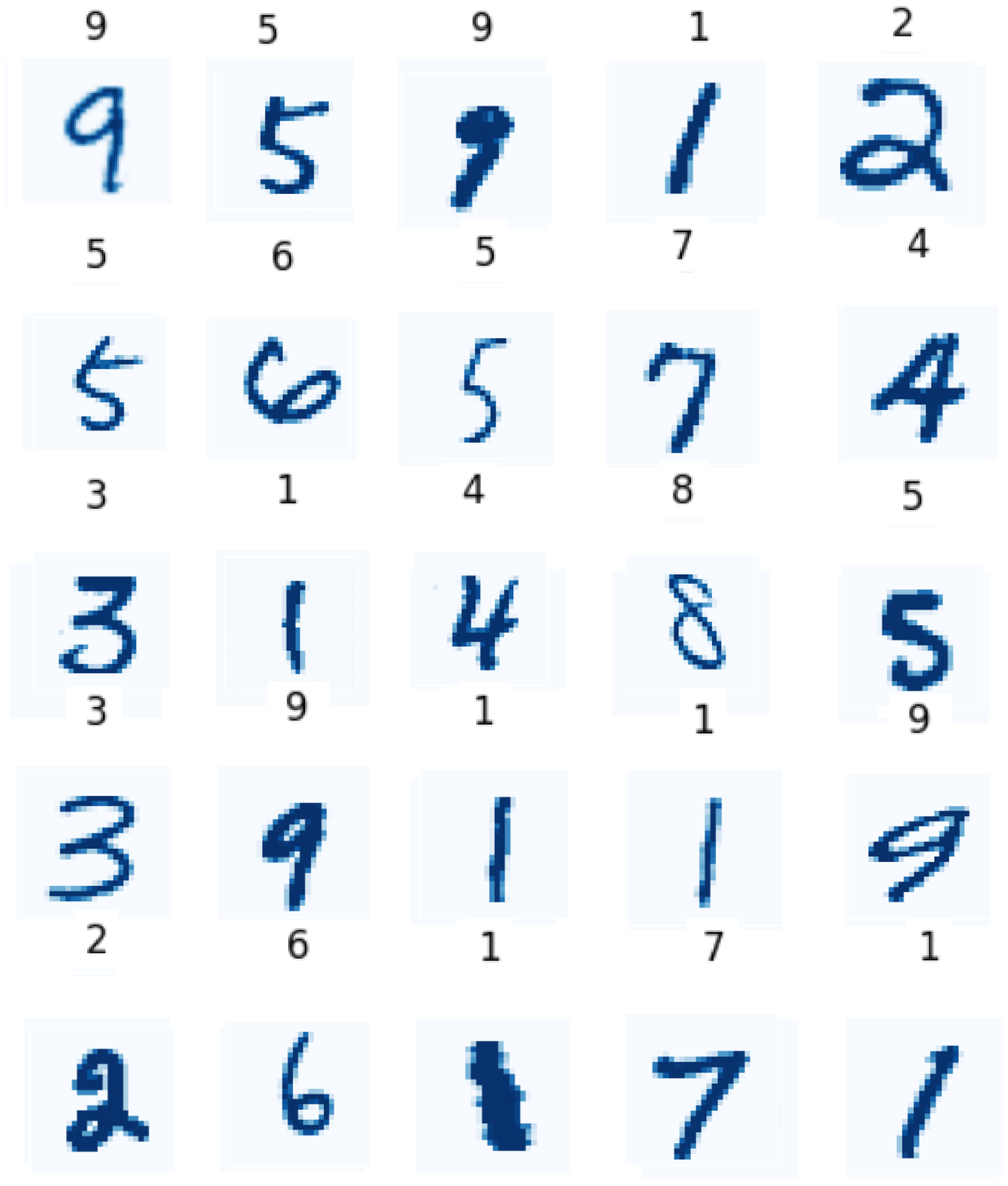


Fig. 1: example MNIST Dataset

Part 2. Code

Let's remind ourselves that Google Collaboratory is a cloud environment of Jupyter Notebook. Colab provides a hosted runtime environment that includes many popular Python libraries, including PyTorch and offers a number of computing resources, including GPU and TPU accelerators.

2.1. CNNs Architectures

The project include two sets of three CNNs. CNNs in the same set share the same architecture but have different training/initialization schema:

The CNNs in the first set share Architecture 1 (A1):

1. One convolutional layer with 4 kernels (output channels);
2. A fully connected layer;
3. A final Softmax output layer;
4. Using the Cross entropy loss.

```
class CNN_A1_(nn.Module):
  def __init__(self):
    super(CNN_A1_, self).__init__()
    self.conv1 = nn.Conv2d(in_channels=1, out_channels=4,
kernel_size=5) # One convolutional layer with 4 kernels
    self.relu1 = nn.ReLU()
    self.fc1 = nn.Linear(4 * 26 * 26, 10) # A fully connected layer
    self.relu2 = nn.ReLU()
    self.softmax = nn.Softmax(dim=1) # A final Softmax output layer


  def forward(self, x):

    x = self.conv1(x) # The convolutional layer
    x = self.relu1(x) # Activation ReLU
    x = torch.flatten(x, 1) # The torch.flatten() function reshapes a
tensor 'x'
    x = self.fc1(x) # Use fully connected layer
    x = self.relu2(x) # Activation second ReLU
    raw_output = self.softmax(x) # Softmax output

    return raw_output
```

In the `__init__` method, the `nn.Conv2d` class is used to define a convolutional layer with 1 input channel, 4 output channels, and a kernel size of 5*5. This is followed by a ReLU activation function. Then, an `nn.Linear` layer is defined, which represents a fully connected layer with an input size of 4*26*26 (the output shape from the convolutional layer) and an output size of 10 (representing 10 classes). Another ReLU activation function is applied, and finally, the `nn.Softmax` function is used for the output layer.

After that the forward method takes an input tensor x and applies a series of operations to it. It passes the input through a convolutional layer, applies ReLU activation, flattens the tensor, passes it through a fully connected layer, applies another ReLU activation, and finally computes the raw output probabilities using the softmax function. The method returns the raw output probabilities.

The CNNs in the second set share Architecture 2 (A2):

1. One convolutional layer (4 kernels);
2. A second convolutional layer (number of kernel on your choice);
3. Optionally you can decide to add other convolutional layers;
4. A fully connected layer;
5. A final Softmax output layer;
6. Use the Cross entropy loss.

```
class CNN_A2_(nn.Module):
  def __init__(self):
    super(CNN_A2_, self).__init__()
    self.conv1 = nn.Conv2d(in_channels=1, out_channels=4,
kernel_size=3)#filter size 1x5x5
    self.conv2 = nn.Conv2d(in_channels=4, out_channels=8,
kernel_size=3) #filter size 4x3x3x
    self.relu = nn.ReLU()
    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(8*24*24, 10) # A fully connected layer
    self.softmax = nn.Softmax(dim=1) # A final Softmax output layer

  def forward(self, x):
    x = self.relu(self.conv1(x)) # The convolutional layer 1
    x = self.relu(self.conv2(x)) # The convolutional layer 2
    x = self.flatten(x) # The torch.flatten() function reshapes a
tensor 'x'
    raw_output = self.softmax(x) # Softmax output

    return raw_output
```

In the __init__ method, the nn.Conv2d class is used to define two convolutional layers. The first layer has 1 input channel and 4 output channels, with a kernel size of 3x3. The second layer takes the 4 output channels from the previous layer as input and has 8 output channels, also with a kernel size of 3x3. ReLU activation functions are applied after each convolutional layer. The nn.Flatten module is used to flatten the tensor before passing it through the fully connected layer. The fully connected layer (nn.Linear) has an input size of 8x24x24 (the output shape from the second convolutional layer) and an output size of 10, representing 10 classes. Finally, the nn.Softmax function is used for the output layer.

In the forward method, the input tensor x is passed through the layers sequentially. The tensor is first passed through the first convolutional layer, followed by the ReLU activation function. Then, it is passed through the second convolutional layer and another ReLU activation. The flatten layer is used to flatten the tensor into a 1-dimensional shape. The raw output probabilities over the classes are computed using the softmax function.

Part 3. Training/Initialization schemas

Each set of CNNs share the same architecture but differ by the Training/Initialization schema. Kernel initialization strategies:

- **By-hand strategy**: the kernels initialized only by 0 and 1. Used specific pattern, but each kernel must contain a different pattern;
- **Default strategy**: used the default initialization or another built-in one (e.g., Kaiming, Xavier, etc.).

HF 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 except the first one (which is Frozen) are trained.

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.

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 initialization_hf(model):
    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]]])

    weights = weights.view(4, 1, 3, 3)

    model.conv1.weight = nn.Parameter(weights, requires_grad=False)
```

```python
def initialization_ht(model):
    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]]])
        bias = torch.FloatTensor([0, 0, 0, 0])

        weights = weights.view(4, 1, 3, 3)
        bias = bias.view(4)

        model.conv1.weight = nn.Parameter(weights, requires_grad=True)
        model.conv1.bias = nn.Parameter(bias, requires_grad=True)
```

In summary, the main differences between the two pieces of code are:

- The first code snippet (initialization_hf) initializes only the weights and makes them non-trainable.
- The second code snippet (initialization_ht) initializes both the weights and biases, making them trainable.

We also add the "init_weights" function to initialize the weights and offsets of the convolution (nn.Conv2d) and linear (nn.Linear) layers in this model and we saved the default model weights in a file called "CNN_default_weights.pt".

```python
def init_weights(model):
    for m in model.modules():
        if isinstance(m, (nn.Conv2d, nn.Linear)):
            nn.init.normal_(m.weight.data)
            nn.init.normal_(m.bias.data, 0, 1)

        torch.save({"CNN_default_weights": model.state_dict()},
"CNN_default_weights.pt")
```

3.1. Training

We pass our model's output logits to "nn.CrossEntropyLoss", which will normalize the logits and compute the prediction error.

```
loss_fn = nn.CrossEntropyLoss()  # Initialize the loss function
```

We defined "train_loop" that performs the training loop.

```
def train_loop(dataloader, model, loss_fn, optimizer):
  size = len(dataloader.dataset)
  for batch, (X, y) in enumerate(dataloader):
    # Compute prediction and loss
    pred = model(X)
    loss = loss_fn(pred, y)

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if batch % 100 == 0:
      loss, current = loss.item(), (batch + 1) * len(X)
      print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

  # torch save model with torch.save()
  torch.save({'model_weights': model.state_dict()}, 'model.pt')
```

These functions provide the core logic for training and evaluating a model on a dataset. The **train_loop** performs the training loop, updates the model's parameters using backpropagation, and saves the model's weights.

3.2. Testing

And "test_loop" that used for evaluating the trained model on a test dataset.

```
def test_loop(dataloader, model, loss_fn):
  size = len(dataloader.dataset)
  num_batches = len(dataloader)
  test_loss, correct = 0, 0

  with torch.no_grad():
    for X, y in dataloader:
      pred = model(X)
      test_loss += loss_fn(pred, y).item()
      correct += (pred.argmax(1) == y).type(torch.float).sum().item()

  test_loss /= num_batches
  correct /= size
  print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
{test_loss:>8f} \n")
```

The **test_loop** evaluates the model's performance on a separate test dataset and calculates the test loss and accuracy.

To use the training function, we added some variables and then initialized our networks:

First, we trained the cnn1 model using SGD optimization and the cross-entropy loss function for a specified number of epochs. It performs the training loop, evaluates the model on the test dataset after each epoch, and prints the results.

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(cnn1.parameters(), lr=learning_rate)

epochs = 20
for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  train_loop(train_dataloader, cnn1, loss_fn, optimizer)
  test_loop(test_dataloader, cnn1, loss_fn)
print("Done!")
```

In the second and third, we do the same with cnn2

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(cnn2.parameters(), lr=learning_rate)

epochs = 20
for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  train_loop(train_dataloader, cnn2, loss_fn, optimizer)
  test_loop(test_dataloader, cnn2, loss_fn)
print("Done!")
```

and cnn3.

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(cnn3.parameters(), lr=learning_rate)

epochs = 20
for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  train_loop(train_dataloader, cnn3, loss_fn, optimizer)
  test_loop(test_dataloader, cnn3, loss_fn)
print("Done!")
```

After that, using the Architecture 2 plan, we will follow similar steps:

- We define our neural network by subclassing "nn.Module", and initialize the neural network layers in "__init__", which is run once when an instance of a Dataset object is created. Every "nn.Module" subclass implements the operations on input data in the forward method.
- We create an instance of "CNN_A2_" , and move it to the device, and print its structure.
- We use the i"nitialization_hf" function to initialize the weights of the specified convolutional neural network (CNN) model using a predefined weight tensor. The pre-defined tensor called weights has a shape of (4, 1, 3, 3), indicating 4 channels, 1 filter, and a filter size of 3x3. The weights tensor is reshaped using the view method to match the shape expected by the model's convolutional layer. The weights of the model's first convolutional layer are assigned the reshaped tensor as a "nn.Parameter" object. The requires_grad=False argument ensures that these weights are not updated during the training process and remain fixed.
- We instantiate a CNN model called cnn4 using the "CNN_A2_" class and initialize its weights using the "initialization_hf" function.

```
cnn4 = CNN_A2_()
initialization_hf(cnn4)
cnn4
```

- We instantiate a CNN model called cnn5 using the "CNN_A2_" class and initialize its weights using the "initialization_ht" function.

```
cnn5 = CNN_A2_()
initialization_ht(cnn5)
cnn5
```

- We instantiate a CNN model called cnn6 using the "CNN_A2_" class and initializes its weights and biases using a custom "init_weights" function.

```
cnn6 = CNN_A2_()
cnn6.apply(init_weights)
cnn6
```

- And finally, we use the Loss Function on all 3 CNNs.

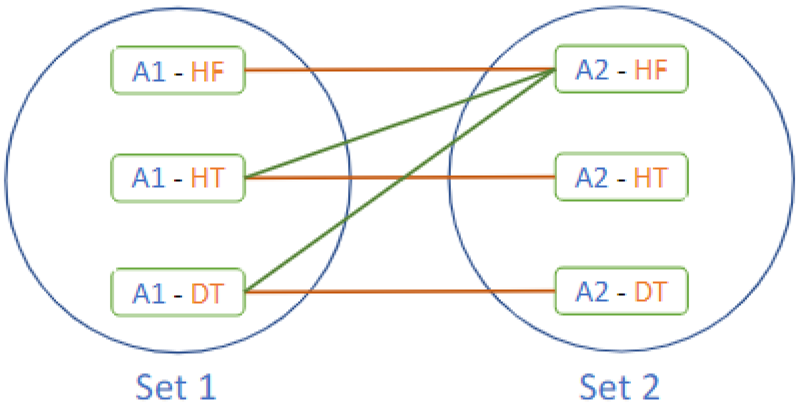
```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(cnn4.parameters(), lr=learning_rate)

epochs = 20
for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  train_loop(train_dataloader, cnn4, loss_fn, optimizer)
  test_loop(test_dataloader, cnn4, loss_fn)
print("Done!")
```

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(cnn5.parameters(), lr=learning_rate)

epochs = 20
for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  train_loop(train_dataloader, cnn5, loss_fn, optimizer)
  test_loop(test_dataloader, cnn5, loss_fn)
print("Done!")
```

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(cnn2.parameters(), lr=learning_rate)

epochs = 20
for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  train_loop(train_dataloader, cnn6, loss_fn, optimizer)
  test_loop(test_dataloader, cnn6, loss_fn)
print("Done!")
```

Part 4. Report

Project include 6 total CNNs, which vary the reference architecture (CNN A1, CNN A2) and the training initialization schema (HF, HT, DT). We conducted the following experiments and comparisons:



4.1. Experiment 1 – Inset Comparison

Training A1 / Testing A1:

- To compare the performance of the CNN 1 model based on the provided data, let's analyze the loss and accuracy values for each epoch:

I. HF Schema

Epoch 1: Loss: 2.108708 Accuracy: 37.2%	Epoch 2: Loss: 2.084791 Accuracy: 38.1%	Epoch 3: Loss: 2.073813 Accuracy: 38.5%	Epoch 4: Loss: 2.061303 Accuracy: 38.6%	Epoch 5: Loss: 1.981459 Accuracy: 53.8%
Epoch 6: Loss: 1.937937 Accuracy: 55.1%	Epoch 7: Loss: 1.927672 Accuracy: 55.3%	Epoch 8: Loss: 1.889518 Accuracy: 60.4%	Epoch 9: Loss: 1.845806 Accuracy: 63.9%	Epoch 10: Loss: 1.837838 Accuracy: 64.3%

Epoch 11: Loss: 1.832976 Accuracy: 64.5%	Epoch 12: Loss: 1.829432 Accuracy: 64.6%	Epoch 13: Loss: 1.826599 Accuracy: 64.8%	Epoch 14: Loss: 1.824184 Accuracy: 64.8%	Epoch 15: Loss: 1.821950 Accuracy: 64.9%
Epoch 16: Loss: 1.815064 Accuracy: 65.5%	Epoch 17: Loss: 1.770756 Accuracy: 71.7%	Epoch 18: Loss: 1.758998 Accuracy: 72.5%	Epoch 19: Loss: 1.753506 Accuracy: 72.8%	Epoch 20: Loss: 1.749988 Accuracy: 73.0%

By comparing the results of the epochs, we can observe the following trends:

- Loss: The loss value gradually decreases from epoch to epoch. This indicates that the model is learning and becoming more accurate in its predictions. Starting from a loss of 2.108708 in the first epoch, it reduces to 1.749988 in the last epoch.
- Accuracy: The accuracy of the model increases as the epochs progress. It starts at 37,2% in the first epoch and reaches 73,0% in the final epoch.

Based on these comparisons, we can conclude that the model's performance improves over time, with decreasing loss and increasing accuracy. Overall, the CNN shows an improvement in both loss and accuracy as the training progresses, which is a positive sign. However, it's worth noting that the performance of the model can vary depending on the specific problem and dataset.

II. HT Schema

Epoch 1: Loss: 2.308938 Accuracy: 66.7%	Epoch 2: Loss: 1.982110 Accuracy: 70.2%	Epoch 3: Loss: 1.893055 Accuracy: 71.4%	Epoch 4: Loss: 1.853914 Accuracy: 78.5%	Epoch 5: Loss: 1.780389 Accuracy: 80.4%
Epoch 6: Loss: 1.741791 Accuracy: 81.0%	Epoch 7: Loss: 1.725754 Accuracy: 81.4%	Epoch 8: Loss: 1.715123 Accuracy: 81.5%	Epoch 9: Loss: 1.707310 Accuracy: 81.7%	Epoch 10: Loss: 1.701254 Accuracy: 81.8%
Epoch 11: Loss: 1.696401 Accuracy: 82.0%	Epoch 12: Loss: 1.692418 Accuracy: 82.0%	Epoch 13: Loss: 1.689089 Accuracy: 82.1%	Epoch 14: Loss: 1.686264 Accuracy: 82.2%	Epoch 15: Loss: 1.683838 Accuracy: 82.3%
Epoch 16: Loss: 1.681732 Accuracy: 82.4%	Epoch 17: Loss: 1.679889 Accuracy: 82.4%	Epoch 18: Loss: 1.678262 Accuracy: 82.5%	Epoch 19: Loss: 1.676816 Accuracy: 82.5%	Epoch 20: Loss: 1.675521 Accuracy: 82.5%

By comparing the results of the epochs, we can observe the following trends:

- Loss: The loss gradually decreases over epochs, indicating that the model is learning and improving its ability to minimize the difference between predicted and actual values. The rate of decrease slows down as the training progresses.
- The accuracy gradually increases over epochs, indicating that the model is becoming more accurate in its predictions. The rate of increase also slows down as the training progresses.

Based on the provided data, we can see that both loss and accuracy show improvement over the 20 epochs. The model starts with a loss of 2.308938 and an accuracy of 66.7% in the first epoch, and by the 20th epoch, the loss reduces to 1.675521 and the accuracy increases to 82.5%.

III. DT Schema

Epoch 1: Loss: 2.004866 Accuracy: 48.4%	Epoch 2: Loss: 1.932157 Accuracy: 55.9%	Epoch 3: Loss: 1.911804 Accuracy: 56.5%	Epoch 4: Loss: 1.903154 Accuracy: 56.9%	Epoch 5: Loss: 1.898020 Accuracy: 57.1%
Epoch 6: Loss: 1.894518 Accuracy: 57.3%	Epoch 7: Loss: 1.891934 Accuracy: 57.4%	Epoch 8: Loss: 1.889920 Accuracy: 57.4%	Epoch 9: Loss: 1.888280 Accuracy: 57.5%	Epoch 10: Loss: 1.886887 Accuracy: 57.6%
Epoch 11: Loss: 1.885639 Accuracy: 57.6%	Epoch 12: Loss: 1.884425 Accuracy: 57.6%	Epoch 13: Loss: 1.883022 Accuracy: 57.7%	Epoch 14: Loss: 1.880804 Accuracy: 57.7%	Epoch 15: Loss: 1.879302 Accuracy: 57.7%
Epoch 16: Loss: 1.822275 Accuracy: 65.6%	Epoch 17: Loss: 1.811943 Accuracy: 66.1%	Epoch 18: Loss: 1.807512 Accuracy: 66.4%	Epoch 19: Loss: 1.804672 Accuracy: 66.5%	Epoch 20: Loss: 1.802417 Accuracy: 66.5%

By comparing the results of the epochs, we can observe the following trends:

- Loss: The loss decreases consistently as the epochs progress. The initial loss of 2.299391 decreases to 1.802417.
- Accuracy: The accuracy of the model increases gradually with each epoch. The initial accuracy of 48.4% improves to 66.5% by the 20th epoch. This suggests that the model is becoming more accurate in predicting the correct labels for the given dataset.

The model demonstrates a positive trend in performance, showing improved accuracy and decreased loss throughout the training process.

Conclusions: In summary, the HF, HT, and DT schemas all exhibit improvements in both loss and accuracy metrics as the epochs progress. The models trained with the HF schema show a gradual decrease in loss and increase in accuracy. The HT schema also demonstrates similar trends, but without specific values, it is challenging to quantify the improvements accurately. Finally, the DT schema displays consistent decreases in loss and increases in accuracy, indicating steady learning and improvement of the model.

Training A2 / Testing A2:

- To compare the performance of the CNN 2 model based on the provided data, let's analyze the loss and accuracy values for each epoch:

I. HF Schema

Based on the provided data for the epochs of the HF schema, the following observations made:

- Loss: The loss remains constant throughout all epochs, with a value of approximately 8.435566. There is no noticeable improvement or change in the loss value over time.
- Accuracy: The accuracy also remains constant at 0.0% for all epochs. The model does not achieve any improvement in accuracy throughout the training process.

Based on these observations, it appears that the model is not learning or making any progress in its predictions for the HF data schema. The constant loss and accuracy values suggest that the model is not able to effectively fit the training data or generalize to the validation set.

II. HT Schema

Based on the provided data for the epochs of the HT schema, the following observations made:

- Loss: The loss remains constant throughout all epochs, with a value of approximately 8.435583. There is no noticeable improvement or change in the loss value over time.
- Accuracy: The accuracy also remains constant at 0.0% for all epochs. The model does not achieve any improvement in accuracy throughout the training process.

Based on these observations, it appears that the model is not learning or making any progress in its predictions for the HT data schema. The constant loss and accuracy values suggest that the model is not able to effectively fit the training data or generalize to the test set.

III. DT Schema

Based on the provided data for the epochs of the DT schema, the following observations made:

- Loss: The loss value remains constant throughout all epochs, with a value of 8.435800. This indicates that the model is not effectively minimizing the error or improving its predictive capability over the training iterations.
- Accuracy: The accuracy is consistently reported as 0.0% for all epochs. This means that the model is not making any correct predictions on the test data.

Based on these comparisons, it can be concluded that the CNN model is not learning from the data or improving its performance over the epochs. The model is not able to capture the patterns and make accurate predictions on the given dataset.

Conclusions: In summary, regardless of the schema used (HF, HT, or DT), the CNN model is not learning from the data and does not show any improvement in performance. The model fails to capture the underlying patterns in the dataset and make accurate predictions.

4.2. Experiment 2 – Architecture comparison

- CNN A1(HF) – CNN A2(HF)

  CNN A1(HF):

  - Loss: The loss starts at 2.108708 in the first epoch and reduces to 1.749988 in the final epoch.
  - Accuracy: The accuracy starts at 37.2% in the first epoch and reaches 73.0% in the final epoch.

  CNN A2(HF):

  - Loss: The loss value is approximately 8.435566 and does not show any noticeable improvement or change over time.

- Accuracy: The accuracy also remains constant at 0.0% for all epochs, indicating that the model is not making any correct predictions on the test data.

In this experiment we compared the performances (loss and accuracy at each epoch) of the CNNs:

1) Based on this comparison, it is evident that CNN architecture A1(HF) outperforms A2(HF) in terms of both loss reduction and accuracy improvement.
2) A1(HF) shows a decreasing trend in loss and an increasing trend in accuracy, indicating that it is learning from the data and making better predictions.
3) On the other hand, A2(HF) shows no improvement in loss or accuracy, suggesting that it is not effectively learning or capturing the patterns in the dataset.

- CNN A1(HT) – CNN A2(HT)

CNN A1(HT):

- Loss: Loss: The loss starts at 2.308938 in the first epoch and reduces to 1.675521 in the final epoch.
- Accuracy: The accuracy starts at 66.7% in the first epoch and reaches 82.5% in the final epoch.

CNN A2(HT):

- Loss: The loss value of approximately 8.435583 does not show any noticeable improvement or change over time.
- Accuracy: The accuracy also remains constant at 0.0% for all epochs, indicating that the model is not making any correct predictions on the test data.

In this experiment we compared the performances (loss and accuracy at each epoch) of the CNNs:

1) Based on this comparison, it is evident that CNN architecture A1(HT) outperforms A2(HT) in terms of both loss reduction and accuracy improvement.

2) A1(HT) shows a decreasing trend in loss and an increasing trend in accuracy, indicating that it is learning from the data and making better predictions.
3) On the other hand, A2(HT) shows no improvement in loss or accuracy, suggesting that it is not effectively learning or capturing the patterns in the dataset.

- CNN A1(DT) – CNN A2(DT)

CNN A1(DT):

- Loss: Loss: The decrease in loss from 2.299391 to 1.802417 suggests that the model is learning and becoming more accurate in its predictions.
- Accuracy: The increase from 48.4% to 66.5% demonstrates that the model is learning and making more accurate predictions.

CNN A2(DT):

- Loss: The loss value remains constant at 8.435800 throughout all epochs, indicating that the model is not effectively minimizing the error or improving its predictive capability over the training iterations.
- The accuracy is consistently reported as 0.0% for all epochs, which means that the model is not making any correct predictions on the test data.

In this experiment we compared the performances (loss and accuracy at each epoch) of the CNNs:

1) Based on this comparison, it is evident that CNN architecture A1(DT) outperforms A2(DT) in terms of both loss reduction and accuracy improvement.
2) A1(DT) shows a consistent decrease in loss and an increase in accuracy, indicating that it is effectively learning and improving its predictions.
3) On the other hand, A2(DT) shows no improvement in loss or accuracy, suggesting that it is not effectively learning or capturing the patterns in the dataset.

4.3. Experiment 3 – Recovery Comparison

In this experiment we compared the performances (loss and accuracy at each epoch) of A2–HF to all the CNN softheset1(A1-*). Thus the comparison must be conducted between the following couples: A2–HF/A1–HF, A2–HF/A1–HT, A2–HF/A1–DT.

- CNN A2(HF) – CNN A1(HF)

  CNN A2(HF):

  - Loss: The loss remains constant at approximately 8.435566 throughout all epochs.
  - Accuracy: The accuracy is consistently reported as 0.0% for all epochs.

  CNN A1(HF):

  - Loss: The loss gradually decreases from epoch to epoch, starting from 2.108708 in the first epoch and reducing to 1.749988 in the last epoch.
  - Accuracy: The accuracy increases from 37.2% in the first epoch to 73.0% in the final epoch.

  Conclusions:

  1) Loss: CNN A1(HF) achieves a noticeable improvement in loss, decreasing it from 2.108708 to 1.749988, while CNN A2(HF) maintains a constant loss of approximately 8.435566.
  2) Accuracy: CNN A1(HF) demonstrates a significant improvement in accuracy, increasing it from 37.2% to 73.0%, whereas CNN A2(HF) maintains a constant accuracy of 0.0%.

  These numerical comparisons indicate that CNN A1(HF) outperforms CNN A2(HF) in both loss reduction and accuracy improvement. A1(HF) shows a clear trend of learning, with decreasing loss and increasing accuracy over the epochs. On the other hand, A2(HF) does not exhibit any improvement in loss or accuracy, remaining constant throughout the training process.

- CNN A2(HF) – CNN A1(HT)

CNN A2(HF):

- Loss: The loss remains constant at approximately 8.435566 throughout all epochs.
- Accuracy: The accuracy is consistently reported as 0.0% for all epochs.

CNN A1(HT):

- Loss: The loss decreases from 2.308938 in the first epoch to 1.675521 in the 20th epoch.
- Accuracy: The accuracy starts at 66.7% in the first epoch and gradually increases to 82.5% by the 20th epoch.

Conclusions:

1) Loss: CNN A1(HT) exhibits a significant reduction in loss, with a decrease from 2.308938 to 1.675521 over the epochs. In contrast, CNN A2(HF) maintains a constant loss of approximately 8.435566.
2) Accuracy: CNN A1(HT) demonstrates notable improvement in accuracy, starting at 66.7% and gradually increasing to 82.5% over the epochs. Conversely, CNN A2(HF) maintains a constant accuracy of 0.0%.

In summary, the comparison between CNN A2(HF) and CNN A1(HT) reveals that A1(HT) outperforms A2(HF) in terms of both loss reduction and accuracy improvement. A1(HT) shows a clear trend of learning, with decreasing loss and increasing accuracy over the epochs. On the other hand, A2(HF) does not exhibit any improvement in loss or accuracy, as they remain constant throughout the training process.

- CNN A2(HF) – CNN A1(DT)

CNN A2(HF):

- Loss: The loss remains constant at approximately 8.435566 throughout all epochs.
- Accuracy: The accuracy is consistently reported as 0.0% for all epochs.

CNN A1(DT):

- Loss: There is a consistent decrease in loss from 2.299391 in the first epoch to 1.802417 in the final epoch.
- Accuracy: The accuracy increases from 48.4% in the first epoch to 66.5% in the final epoch.

Conclusions:

1) Loss: CNN A1(DT) shows a consistent decrease in loss from 2.299391 to 1.802417 over the epochs, indicating that the model is learning and improving its ability to minimize the difference between predicted and actual values. In contrast, CNN A2(HF) maintains a constant loss of approximately 8.435566, without any noticeable improvement.
2) Accuracy: CNN A1(DT) demonstrates an increase in accuracy from 48.4% to 66.5% over the epochs, suggesting that the model is becoming more accurate in its predictions. Conversely, CNN A2(HF) maintains a constant accuracy of 0.0%, indicating that it does not achieve any improvement in accuracy throughout the training process.

In summary, the comparison between CNN A2(HF) and CNN A1(DT) reveals that A1(DT) performs better in terms of both loss reduction and accuracy improvement. A1(DT) shows a clear trend of learning, with decreasing loss and increasing accuracy over the epochs. On the other hand, A2(HF) does not exhibit any improvement in loss or accuracy, as they remain constant throughout the training process.