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: Yaroslav Karabin
MatricolaID: #295359

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.

In the project we compared the performances of two sets of three Convolutional Neural Networks.

Technical requirements that we comply with:

- The CNNs built and trained using the Pytorch library.
- Experiments on the CNNs performed using the Fashion MNIST 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. Libraries

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
import torch.nn.functional as F
import matplotlib.pyplot as plt
from tqdm import tqdm
from torchvision.transforms import ToTensor, Compose
```

Fig. 1. Imported libraries.

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

The Fashion MNIST dataset is a popular dataset used for benchmarking machine learning algorithms, particularly for image classification tasks. It was created by Zalando Research and serves as a more challenging alternative to the classic MNIST handwritten digits dataset. Here are the key details about this dataset in table 1:

Table 1. Fashion MNIST key details

Content	70,000 grayscale images of 10 different types of clothing items
Image size	Each image is 28x28 pixels
Color Channels	Each image is in grayscale (1 color channel)
Classes	There are 10 classes, each representing a different type of clothing item
Training and test split	60,000 training images and 10,000 test images

The Fashion MNIST dataset is widely used for:

1. **Benchmarking**: Comparing the performance of different machine learning algorithms and architectures, particularly in the context of image classification.
2. **Educational Purposes**: Teaching and learning about image processing, neural networks, and deep learning techniques.
3. **Testing New Methods**: Validating new approaches in computer vision and pattern recognition before applying them to more complex datasets.

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

Fig. 2. Code for loading dataset.

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.

For demonstration, we generated a figure showing first 5 images from the Fashion MNIST training dataset.

```python
def show_images(dataset, num_images=5):
    plt.figure(figsize=(10, 2))
    for i in range(num_images):
        image, label = dataset[i]
        plt.subplot(1, num_images, i+1)
        plt.imshow(image.squeeze(), cmap="gray")
        plt.title(f"Label: {label}")
        plt.axis("off")
    plt.show()
```

Fig. 2. Code for showing images.

```python
show_images(train_dataset)
```

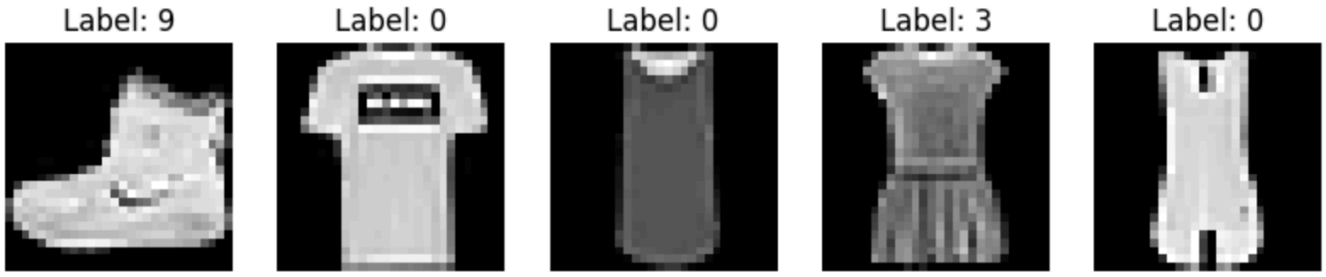


Fig. 3. First 5 images from training dataset

```python
show_images(test_dataset)
```

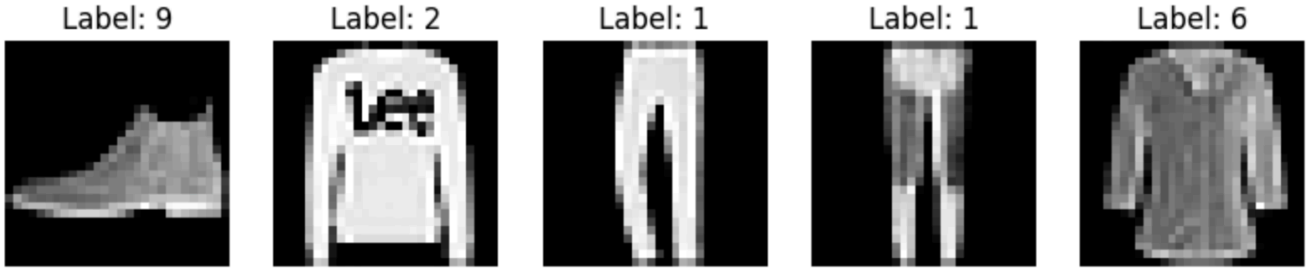


Fig. 4. First 5 images from testing dataset

Part 2. Main 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 several computing resources, including GPU and TPU accelerators.

2.1. CNNs Architectures

The project includes 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=3, padding=1)

        self.fc1 = nn.Linear(4 * 28 * 28, 128)

        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)

        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.relu(x)

        x = self.fc2(x)
        x = F.softmax(x, dim=1)

        return x
```

Fig. 5. Code for class definition of CNN (A1)

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 3*3. 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*28*28 (the output shape from the convolutional layer) and an output size of 128. Another ReLU activation function is applied, and finally,the `nn.Linear` layer is defined with input size 128, which is an output size of previous layer and the output size this layer equals 10 (10 classes). As it should be Softmax output layer, activation function here is Softmax.

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 usingthe 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 yourchoice);
3. Optionally you can decide to add other convolutionallayers;
4. A fully connected layer;
5. A final Softmax output layer;
6. Use the Cross entropy loss.

```python
class CNN_A2(nn.Module):
    def __init__(self, additional_conv_layers=0):
        super(CNN_A2, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=4, out_channels=8, kernel_size=3, padding=1)

        self.additional_convs = nn.ModuleList()
        for _ in range(additional_conv_layers):
            self.additional_convs.append(nn.Conv2d(in_channels=8, out_channels=8, kernel_size=3, padding=1))

        self.fc1 = nn.Linear(8 * 28 * 28, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)

        x = self.conv2(x)
        x = F.relu(x)

        for conv in self.additional_convs:
            x = conv(x)
            x = F.relu(x)

        x = x.view(x.size(0), -1)

        x = self.fc1(x)
        x = F.relu(x)

        x = self.fc2(x)
        x = F.softmax(x, dim=1)

        return x
```

Fig. 6. Code for class definition of CNN (A2)

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. Then, an `nn.Linear` layer is defined, which represents a fully connected layer with an input size of 8*28*28 (the output shape from the convolutional layer) and an output size of 128 . Then, an `nn.Linear` layer is defined, which represents a fully connected layer with an input size of 128 (the output shape from the previous layer) and an output size of 10 (representing 10 classes). Finally, as it should be Softmax output layer, activation function here is Softmax.

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 fully connected layer is used to flatten the tensor into a 1-dimensional shape with 128 tensors. The raw output probabilities over theclasses 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, theremaining 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, theremaining 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 initialize_weights_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)

        return model
```

Fig. 7. Code for initializing weights for HF schema.

```
def initialize_weights_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)

        return model
```

Fig. 7. Code for initializing weights for HT schema.

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

- The first code snippet (*initialize_weights_hf*) initializes only the weights andmakes them non-trainable.
- The second code snippet (*initialize_weights_ht*) initializes both the weights andbiases, making them trainable.

We also add the " *initialize_weights_default* " function to initialize the weights and offsets of theconvolution (nn.Conv2d) and linear (nn.Linear) layers in this model.

```
def initialize_weights_default(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)

    return model
```

Fig. 8. Code for initializing weights for DT schema.

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
```

Fig. 9. Code for initializing Cross Entropy Loss.

We defined "*train loop*" function to make training loop possible for every batch with default size equals 64:

```
def train_loop(data, model, loss_fn, optimizer, batch_size=64):
    dataloader = DataLoader(data, batch_size=batch_size)
    size = len(dataloader.dataset)

    loss_history = []
    correct = 0

    for _, (X, y) in enumerate(dataloader):
        pred = model(X)
        loss = loss_fn(pred, y)

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

        loss_history.append(loss.item())
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    accuracy = correct / size
    return loss_history, accuracy
```

Fig. 10. Code for initializing training loop

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(data, model, loss_fn, batch_size=64):
    dataloader = DataLoader(data, batch_size)

    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    loss_history = []

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

    test_loss /= num_batches
    correct /= size

    return loss_history, correct
```

Fig. 11. Code for initializing testing loop.

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

3.3. CNNs initialization

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

Firstly, we trained the *cnn_1_hf* model using SGD optimization and the cross-entropyloss function for a specified number of epochs. "*make experiment*" function performs the training loop, evaluates the model on the test dataset after each epoch, and prints the results.

```
loss_fn = nn.CrossEntropyLoss()
cnn_1_hf = initialize_weights_hf(CNN_A1())
optimizer = torch.optim.SGD(cnn_1_hf.parameters(), lr=0.001)

make_experiment(cnn_1_hf, 'cnn_1_hf', loss_fn, optimizer)
```

Fig. 12. Code for making experiment using HF schema

In the second and third experiments, we do the same using HT and DT schemas:

```
loss_fn = nn.CrossEntropyLoss() # Initialize the loss function
cnn_1_ht = initialize_weights_ht(CNN_A1())
optimizer = torch.optim.SGD(cnn_1_ht.parameters(), lr=0.001)

make_experiment(cnn_1_ht, 'cnn_1_ht', loss_fn, optimizer)
```

Fig. 13. Code for making experiment using HT schema

```
loss_fn = nn.CrossEntropyLoss()
cnn_1_default = initialize_weights_default(CNN_A1())
optimizer = torch.optim.SGD(cnn_1_default.parameters(), lr=0.001)

make_experiment(cnn_1_default, 'cnn_1_default', loss_fn, optimizer)
```

Fig. 14. Code for making experiment using DT schema

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

```
loss_fn = nn.CrossEntropyLoss()
cnn_2_hf = initialize_weights_hf(CNN_A2())
optimizer = torch.optim.SGD(cnn_2_hf.parameters(), lr=0.001)

make_experiment(cnn_2_hf, 'cnn_2_hf', loss_fn, optimizer)
```

Fig. 15. Code for making experiment using HF schema

```
loss_fn = nn.CrossEntropyLoss()
cnn_2_ht = initialize_weights_ht(CNN_A2())
optimizer = torch.optim.SGD(cnn_2_ht.parameters(), lr=0.001)

make_experiment(cnn_2_ht, 'cnn_2_ht', loss_fn, optimizer)
```

Fig. 16. Code for making experiment using HT schema

```
loss_fn = nn.CrossEntropyLoss()
cnn_2_default = initialize_weights_default(CNN_A2())
optimizer = torch.optim.SGD(cnn_2_default.parameters(), lr=0.001)

make_experiment(cnn_2_default, 'cnn_2_default', loss_fn, optimizer)
```

Fig. 17. Code for making experiment using DT schema

Part 4. Report

This project includes 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:

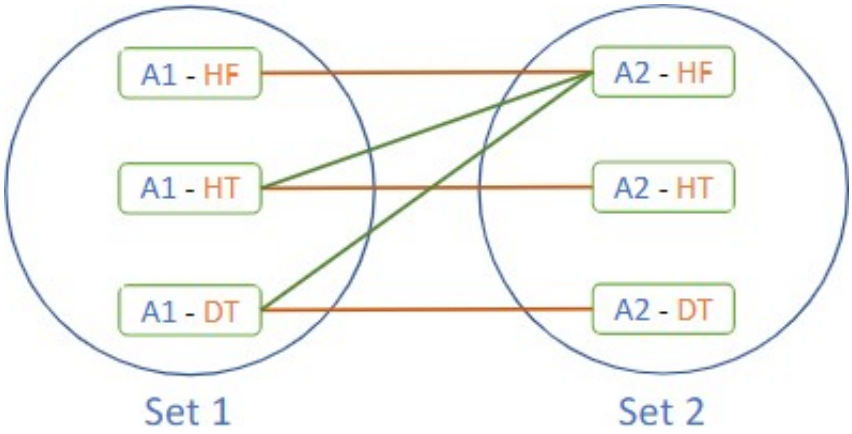


Fig. 18. Experiments and comparisons schema

4.1. Experiment 1 – Inset Comparison

Training A1 / Testing A1:

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

I. HF Schema

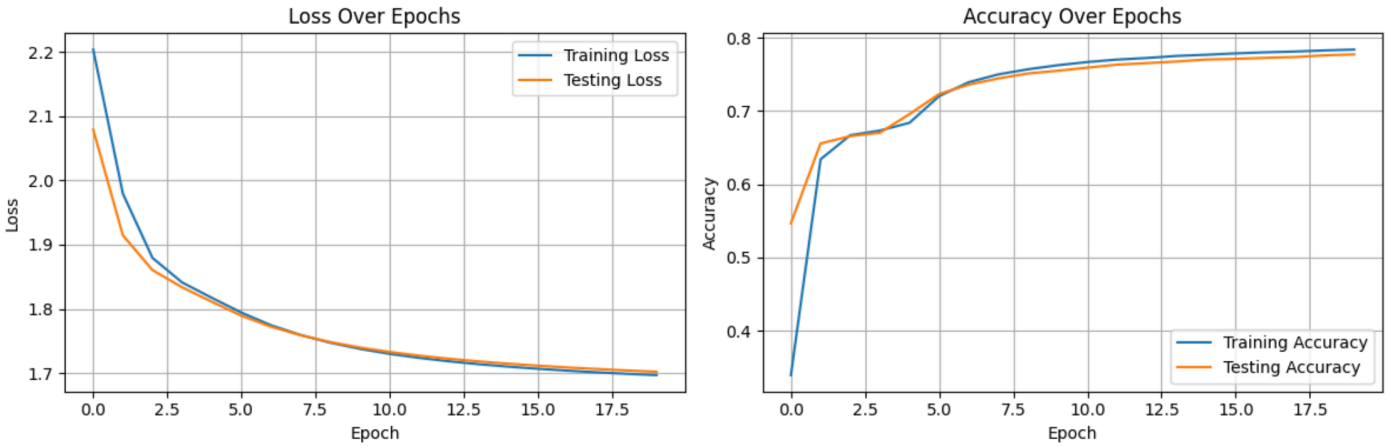


Fig. 19. Loss and accuracy over epochs diagrams

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

- Loss: The loss value gradually decreases from epoch to epoch on both training and testing datasets. This indicates that the model is learning and becoming more accurate in its predictions and there is **no overfitting**. Starting from a loss of 2.2 in the first epoch, it reduces to1.7 in the last epoch.
- Accuracy: The accuracy of the model increases as the epochs progress. It starts at 54% in the first epoch and reaches 77,7% 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

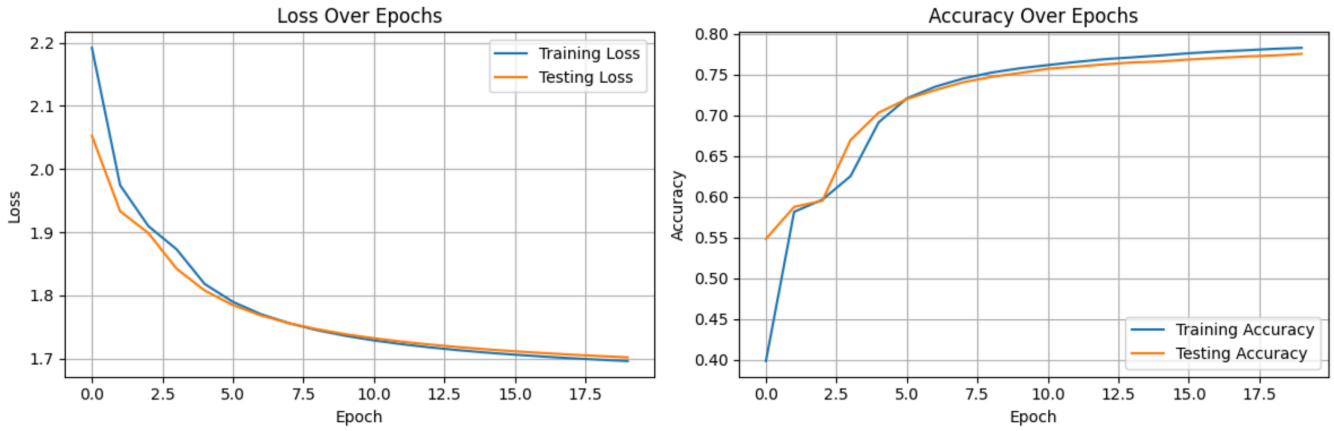


Fig. 20. Loss and accuracy over epochs diagrams

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 2.2 and an accuracy of 54.8% in the first epoch, and by the 20th epoch, the loss reduces to 1.69 and the accuracy increases to 77.5%.

III. DT Schema

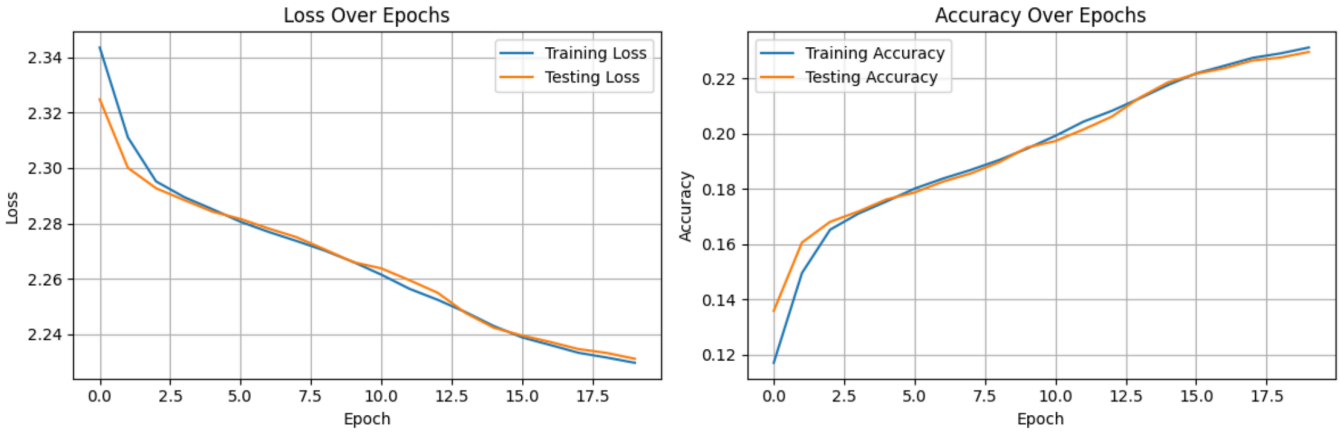


Fig. 21. Loss and accuracy over epochs diagrams

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

- Loss: The loss decreases quite slowly as the epochs progress. The initial loss of 2.34 decreases to 2.29.
- Accuracy: The accuracy of the model increases gradually with each epoch. The initial accuracy of 13.6% improves to 23% by the 20th epoch. This suggests that the model is becoming more accurate in predicting the correct labels for the given dataset but there is not enough epochs or default weights are inappropriate for our dataset.

Conclusions: In summary, the HF, HT schemas all exhibit improvementsin 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 HTschema 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 but this learning is very slow.

Training A2 / Testing A2:

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

I. HF Schema

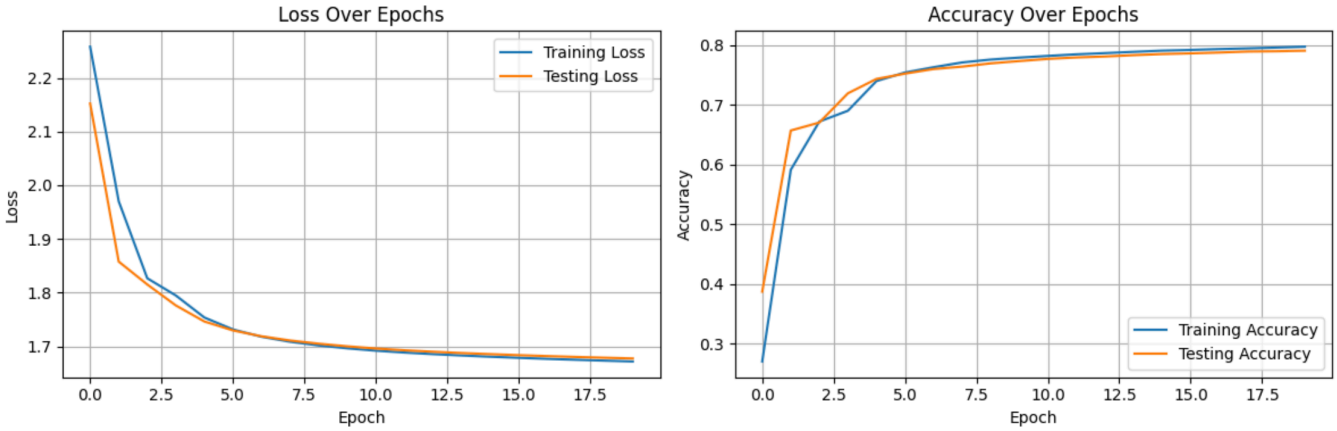


Fig. 22. Loss and accuracy over epochs diagrams

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

- The loss value gradually decreases from epoch to epoch on both training and testing datasets. This indicates that the model is learning and becoming more accurate in its predictions and there is **no overfitting**. Starting from a loss of 2.25 in the first epoch, it reduces to 1.67 in the last epoch.
- Accuracy: The accuracy of the model increases as the epochs progress. Itstarts at 38.7% in the first epoch and reaches 79% in the final epoch.

II. HT Schema

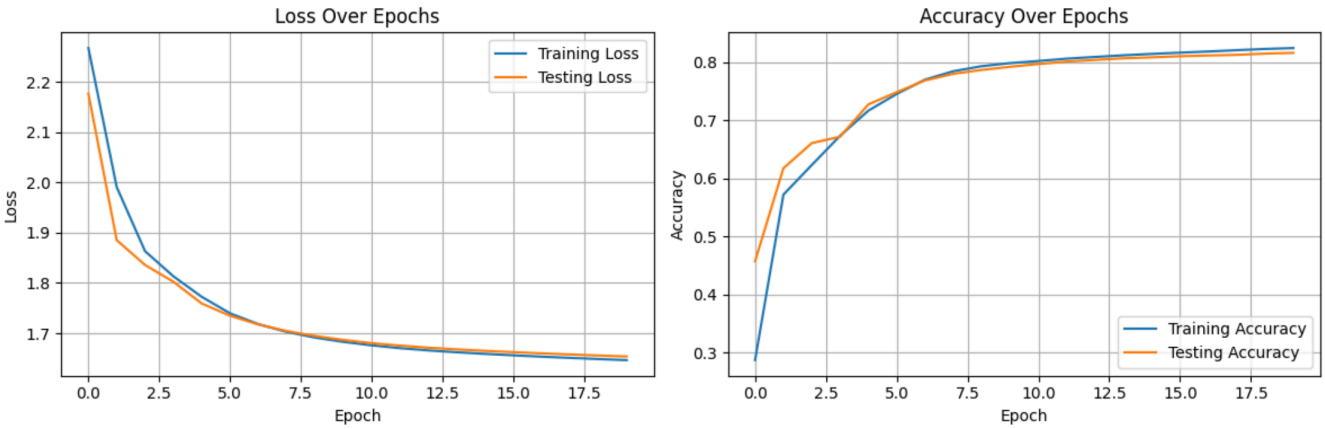


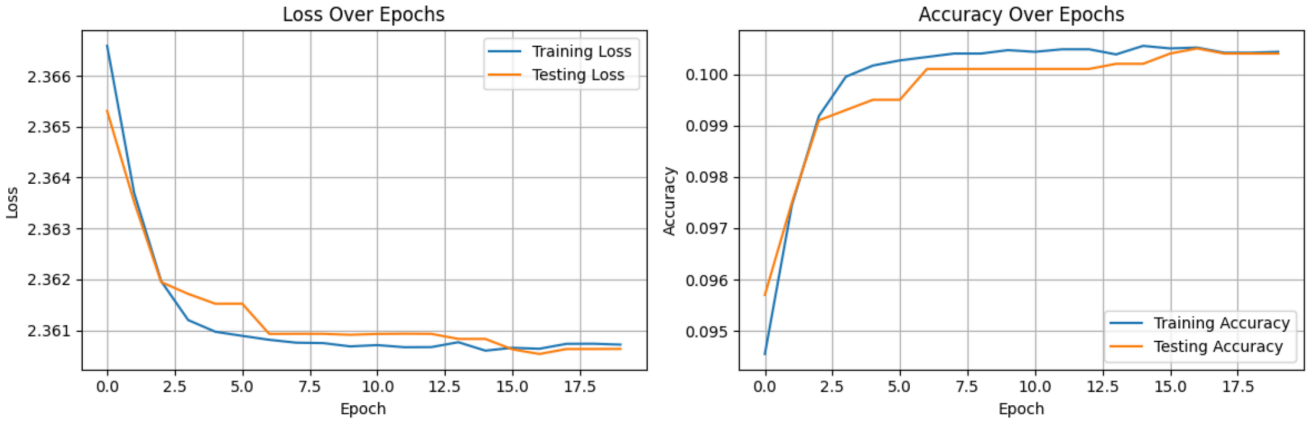
Fig. 23. Loss and accuracy over epochs diagrams

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 2.26 and an accuracy of 45.7% in the first epoch, and by the 20th epoch, the loss reduces to 1.64 and the accuracy increases to 81.5%.

III. DT Schema



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

- Loss: The loss gradually decreases till to the 10th epoch, and after that remains close to 2.36.
- The accuracy acts the same as loss. Gradually increases till to the 15th epoch, indicating that the model stops learning.

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

Conclusions: In summary, the HF, HT schemas all exhibit improvementsin 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 HTschema also demonstrates similar trends. Finally, the DT schema displays consistent decreases in loss and increases in accuracy till to the 10th and 15th epochs respectively. However, after that loss and accuracy remains the almost same which means that the model 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 2.2 in the first epoch and reduces to 1.69 in the final epoch.
  - Accuracy: The accuracy starts at 54.6% in the first epoch and reaches 77.7% in the final epoch.

  CNN A2(HF):

  - Loss: The loss starts at 2.25 in the first epoch and reduces to 1.67 in the final epoch.
  - Accuracy: The accuracy starts at 38.7% in the first epoch and reaches 79% in the final epoch.

In this experiment we compared the performances (loss and accuracy at each epoch) of the CNNs. So, based on this comparison, we can see that CNN architectures A2(HF) outperforms A1(HF) by 1.3% in accuracy which is not significant improvement but it's improvement.

- CNN A1(HT) – CNN A2(HT)

  CNN A1(HT):

  - Loss: The loss starts at 2.19 in the first epoch and reduces to 1.69 in the final epoch.
  - Accuracy: The accuracy starts at 54.8% in the first epoch and reaches 77.5% in the final epoch.

  CNN A2(HT):

  - Loss: The loss starts at 2.27 in the first epoch and reduces to 1.64 in the final epoch.

- Accuracy: The accuracy starts at 45.7% in the first epoch and reaches 81.5% in the final epoch.

In this experiment we compared the performances (loss and accuracy at each epoch) of the CNNs. So, based on this comparison, we can see that CNN architectures A2(HF) outperforms A1(HF) by 4% in accuracy which is quite significant improvement.

- CNN A1(DT) – CNN A2(DT)

  CNN A1(DT):

  - Loss: The loss decreases quite slowly as the epochs progress. The initial loss of 2.34 decreases to 2.29.
  - Accuracy: The accuracy of the model increases gradually with each epoch. The initial accuracy of 13.6% improves to 23% by the 20th epoch.

  CNN A2(DT):

  - Loss: The loss gradually decreases till to the 10th epoch, and after that remains close to 2.36.
  - Accuracy: Gradually increases till to the 15th epoch, indicating that the model stops learning.

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

1) **CNN A1(DT)**, despite a slow decrease in loss, continues to improve both in terms of loss and accuracy throughout the 20 epochs. This indicates that the model is still learning and gradually getting better.
2) **CNN A2(DT)**, on the other hand, shows a plateau in both loss and accuracy, suggesting that the model stops improving after a certain point. This could indicate that the model has reached its capacity or is facing issues like overfitting or inappropriate learning rates.

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)

Comparison conclusions:

1) Loss: Both models show a consistent decrease in loss values, indicating continuous learning and improved predictions without signs of overfitting. CNN A2(HF) starts with a slightly higher loss (2.25) compared to CNN A1(HF) (2.2) and ends with a lower final loss (1.67 vs. 1.7).
2) Accuracy: CNN A2(HF) starts with a lower initial accuracy (38.7%) compared to CNN A1(HF) (54%). However, CNN A2(HF) reaches a higher final accuracy (79%) compared to CNN A1(HF) (77.7%).
3) **CNN A2(HF)** shows significant improvement in accuracy over the epochs, achieving the highest final accuracy among the two models. This model also demonstrates effective learning and better overall performance in terms of reducing loss and increasing accuracy.
4) **CNN A1(HF)** starts with a higher initial accuracy, indicating it might have had better initial weights or architecture. But despite a good start, its final accuracy is slightly lower than CNN A2(HF), though it still shows a good learning curve and steady improvement.

- CNN A2(HF) – CNN A1(HT)

Comparison conclusions:

1) Loss: Both models show a consistent decrease in loss values, indicating continuous learning and improved predictions. CNN A2(HF) starts with a higher loss (2.25) compared to CNN A1(HT) (2.2) but ends with a slightly lower final loss (1.67 vs. 1.69).
2) Accuracy: CNN A2(HF) starts with a lower initial accuracy (38.7%) compared to CNN A1(HT) (54.8%). In addition, CNN A2(HF) reaches a higher final accuracy (79%) compared to CNN A1(HT) (77.5%).
3) **CNN A2(HF)** Shows significant improvement in accuracy over the epochs, achieving the highest final accuracy among the two models. Demonstrates effective learning and better overall performance in terms of reducing loss and increasing accuracy.
4) **CNN A1(HT)** Starts with a higher initial accuracy, indicating it might have had better initial weights or architecture. Shows a good learning curve and steady improvement, but its final accuracy is slightly lower than CNN A2(HF).

Both models exhibit strong learning capabilities with gradual improvements in loss and accuracy. However, CNN A2(HF) ultimately achieves a slightly better performance in terms of final accuracy and loss reduction.

- CNN A2(HF) – CNN A1(DT)

Comparison conclusions:

1) Loss: CNN A2(HF) shows a significant decrease in loss from 2.25 to 1.67, indicating substantial learning and model improvement. Otherwise, CNN A1(DT) shows a very slow decrease in loss from 2.34 to 2.29, suggesting limited learning and model improvement.

2) Accuracy: CNN A2(HF) shows a strong improvement in accuracy from 38.7% to 79%, indicating effective learning and model performance. CNN A1(DT) shows a gradual increase in accuracy from 13.6% to 23%, indicating that the model is learning but at a very slow rate.
3) **CNN A2(HF)** exhibits significant improvements in both loss and accuracy, demonstrating effective learning and robust model performance over the epochs. The substantial increase in accuracy and notable decrease in loss suggest that the model is well-tuned and able to learn the dataset effectively.
4) **CNN A1(DT)** shows minimal improvements in loss and accuracy, indicating limited learning and suboptimal model performance. The slow decrease in loss and modest increase in accuracy suggest that either the model architecture is not well-suited for the dataset, the number of epochs is insufficient, or the initial weights and hyperparameters are not appropriate.

Overall, CNN A2(HF) outperforms CNN A1(DT) significantly in terms of both loss reduction and accuracy improvement, making it a much better model for the given dataset.