CS776: DEEP LEARNING FOR COMPUTER VISION

ASSIGNMENT 1

Assignment Due Date: 16 Feb 2025

Group Name: Multi-Head Mayhem

Ansh Makwe (241110010)
Divyansh Chaurasia (241110022)
Kumari Ritika (241110039)
Prakhar Mandloi (241110051)
Swaraj Sonavane (241110075)
Hritik Chouhan (241110030)

February 16, 2025

1 Introduction

The objective of this assignment is to implement Multilayer Perceptron and Convolutional Neural Network for image classification. The code loads, preprocesses, trains, and evaluates on the Fashion MNIST dataset. The dataset is a collection of 70,000 grayscale images (28×28 pixels) of fashion items, designed as a drop-in replacement for the original MNIST digit dataset. It consists of 60,000 training images and 10,000 test images, each labeled into 10 classes representing different clothing categories.

Figure 1: Sample images from the Fashion MNIST dataset.

1.1 Problem Statement:

Image classification is a fundamental task in computer vision, which aims to categorize images into predefined classes based on their visual characteristics. Deep learning models, particularly Multi-Layer Perceptrons (MLP) and Convolutional Neural Networks (CNN), have emerged as powerful tools for image classification.

1. **Typical Multi-Layer Perceptrons (MLP) Approach :**

   - Flattens the 2D image into a 784-dimensional vector (since 28×28=784).
   - Applies fully connected (dense) layers, but lacks spatial awareness, meaning it cannot leverage local patterns in images.
   - MLPs require a large number of parameters as each neuron in a layer is fully connected to the next layer. This leads to higher memory usage and overfitting risk.
   - MLP Forward Propagation (Dense Layers) Each layer in an MLP computes:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)} \tag{1}$$

$$a^{(l)} = f(z^{(l)}) \tag{2}$$

     where:
     - $W^{(l)}$ and $b^{(l)}$ are the weights and biases for layer l.
     - $a^{(l)}$ is the activation output using a non-linear function f, typically ReLU:

$$\text{ReLU}(x) = \max(0, x) \tag{3}$$

2. **Typical Convolutional Neural Networks (CNNs) Approach :**

   - Uses convolutional layers to extract local patterns, such as edges, textures, and shapes.
   - Employs spatial hierarchy learning, making it robust to variations in position and scale.
   - CNNs employ parameter sharing through convolutional filters, significantly reducing the number of parameters and improving generalization. However, CNNs require more floating point operations per second (FLOPS), making them computationally expensive.
   - CNN Convolutional Layer Computation The convolution operation is defined as:

$$f(i,j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m,n) \tag{4}$$

     where:
     - $I(i,j)$ represents the input image pixels at location (i,j).
     - $K(m,n)$ represents the kernel (filter) weights applied over the image.

     The result is a feature map, which is later downsampled using max pooling:

$$p(i,j) = \max_{m \in M, n \in N} f(i+m, j+n) \tag{5}$$

   - This hierarchical representation allows CNNs to detect edges in the first layer, textures in deeper layers, and full objects in later layers.

2 Dataset Preprocessing

Preprocessing is a crucial step in deep learning as it ensures that the input data is clean, normalized, and in a format suitable for the model. The Fashion MNIST dataset undergoes several preprocessing steps before training, including data normalization, reshaping, augmentation, and splitting into training and validation sets.

2.1 Loading the Dataset:

The Fashion MNIST dataset consists of 60,000 training images and 10,000 test images, each of size 28×28 pixels in grayscale. It is structured as follows:

- **Training Set:** Used for training the model (80% of the data).
- **Validation Set:** A subset of the training set used to tune hyperparameters (20% of the training data).
- **Test Set:** Used to evaluate the final model performance.

2.2 Preprocessing Used:

MLP Preprocessing Steps: The dataset is stored in the IDX file format and is loaded using a custom function load_idx_file().

- **Normalization:** Converts pixel values from [0, 255] to [0, 1].
- **Flattening:** Converts each 28×28 image into a 1D vector of size 784.
- **Creating Batches for Training:** For making train, valid and test batches for training and testing.

CNN Transformation Pipeline:

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

- **transforms.ToTensor():** Converts the PIL Image (or NumPy array) into a PyTorch tensor. It changes the pixel values from (0 to 255) to (0 to 1) by dividing by 255.
- **Normalization :**transforms.Normalize((0.5,), (0.5,)) sets the mean = 0.5 and the standard deviation = 0.5, which rescales pixel values from [0,1] to [1,1].

3 Model Architecture

MLP: The MLP consists of multiple layers, each defined by:

1. **Defining the Multi-Layer Perceptron (MLP) Class :**

```
class MLP:
    def __init__(self, input_size, layers, activations, initialization="
        random", dropout=0.0, learning_rate=0.01): ...
```

2. **Initializing Weights :**
   - He initialization (np.sqrt(2 / previous_size)) is used for ReLU-based networks.
   - Glorot initialization (np.sqrt(1 / previous_size)) is suitable for Tanh/Sigmoid activations.
   - Random initialization is also provided as an alternative.
3. **Activation Functions :**
   - **ReLU (Rectified Linear Unit)**:
   $$f(x) = \max(0, x) \tag{6}$$
   Activates the input if positive, else returns zero.
   - **Tanh (Hyperbolic Tangent)**:
   $$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{7}$$
   Maps input to the range between -1 and 1.
   - **GELU (Gaussian Error Linear Unit)**:
   $$f(x) = x \cdot \Phi(x) \tag{8}$$
   where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution.
   $$f(x) = 0.5x(1 + \tanh(\sqrt{2/\pi}(x + 0.044715x^3))) \tag{9}$$
   Smooths out ReLU and sigmoid activations.
   - **Leaky ReLU**:
   $$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases} \tag{10}$$
   Allows a small, non-zero gradient for negative input values.
4. **Architecture :** The number of hidden layers and how the neurons are varied, is mentioned in the results section.

CNN: The CNN model is defined in a CNN Model class.

1. **Layer Configuration :** The CNN consists of five convolutional layers followed by batch normalization, dropout, and fully connected layers. Each convolutional layer applies ReLU activation and batch normalization. Pooling and dropout layers improve generalization.
   - Conv1 ($1 \rightarrow 64$ filters, Kernel=5x5, Stride=1, Padding=0)

- Conv2 (64 → 192 filters, Kernel=5x5, Stride=1, Padding=0)
- Conv3 (192 → 384 filters, Kernel=1x1, Stride=1, Padding=0)
- Conv4 (384 → 256 filters, Kernel=1x1, Stride=1, Padding=0)
- Conv5 (256 → 256 filters, Kernel=1x1, Stride=1, Padding=0)
- Fully Connected Layers (256 → 10 classes)

2. **Weight Initialization :**The model supports three initialization methods:
   - Xavier Initialization for better weight distribution
   - He Initialization (suitable for ReLU activations)
   - Random Uniform Initialization

4 Model Traning

MLP

1. **Forward Propagation :** Computes the output of each layer.
   - Performing a weighted sum of inputs: z = W*x + b
   - Applying activation functions ReLU , Tanh , GELU or Leaky ReLU.
2. **Backpropagation :** Computes the gradients of the loss function with respect to weights and biases.
   - The loss function is a categorical cross-entropy loss.
   - The gradients are computed using:
     (a) dA[np.arange(m), labels] -= 1 for error calculation.
     (b) Division by batch size (m) to normalize.
   - The model updates parameters using gradient descent:

```
def update_params(self, grads):
    for i in range(len(self.layers)):
        self.params[f'W{i}'] -= self.learning_rate * grads[f'dW{i}']
        self.params[f'b{i}'] -= self.learning_rate * grads[f'db{i}']...
```

3. **Epoch iteration :** The models are trained for 20 epochs using mini-batches.
   - Each epoch iterates over all training batches.
   - The total loss is calculated and printed after each epoch.
   - The total loss is calculated and printed after each epoch.

CNN

1. **Loss Function & Optimizer :**
   - CrossEntropyLoss is used for multi-class classification.
   - Adam Optimizer with L2 weight decay prevents overfitting.
2. **Training Loop :** The model is trained for 20 epochs, updating weights using backpropagation.

```
for epoch in range(1, num_epochs + 1):
    train(model, train_loader, criterion, optimizer, epoch)
    val_loss, val_accuracy = validate(model, val_loader, criterion)
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(model.state_dict(), 'best_model.pth')
```

   - Every 100 batches, it prints the training loss.
   - The best model is saved.

5 Adderessing the Problem Statment

MLP

1. **For experimenting with different hyperparameters like number of layers, number of neurons in a layer, dropout, activation functions like ReLU, leaky-ReLu, tanh and GeLU, we have used the following code :**

```
mlp = MLP(input_size=28*28, layers=[256, 128, 64, 10],
activations=["gelu", "gelu" , "gelu", "softmax"], initialization="random",
dropout=0.2, learning_rate=0.05)
mlp.train(train_batches, val_batches, epochs=20)
mlp.evaluate(test_batches)
```

   - Where, layers parameter expresses the number of layers used with value at each index of the layers representing the included neurons.
   - Activations parameter controls the activation function used for each layer.
   - Initialization parameter provided the reference to which type of initialization method is being used. With possible values being "random", "he", "glorot".
   - Dropout parameter contains the probability at which the neuron of a layer being active. this helps in avoiding overfitting.
   - And the last parameter is the learning rate.

CNN

1. **For experimenting with different kernel sizes, number of kernel in each layer, different pooling methods and different weight initilization technique, we are again passing these hyper parameters as an input ot the CNNModel class. The snippet of the code is provided below :**

```
kernel_sizes = [3, 3, 3, 3, 3]
strides = [2, 2, 1, 1, 1]
paddings = [2, 2, 2, 2, 0],
pooling_type = "max",
model = CNNModel(kernel_sizes, strides, paddings, pooling_type, "he")
        .to(device)
```

   - Where, the number of kernels used in each layer in coded in the CNNModel, to give further more clarity see the following line of code: nn.Conv2d(in channels = 1, out channels = 64).
   - Kernel sizes controls the size of each kernel in all 5 convolution layer.
   - Strides controls the stride of each kernel.
   - And the paddings controls the padding used in each convolution layer.
   - Pooling type tells which pooling method is being used after each conv layer, with the possibilities being "max" "average" and "gobal avg".
   - Finally, initialization parameter provided the reference to which type of initialization method is being used. With possible values being "random", "he", "xavier".

6 Model Evaluation

MLP
The test dataset is used to evaluate the trained model. The metrics are computed as follows:

$$\text{correct+} = \sum(\text{predictions} == \text{labels})$$

$$\text{total+} = \text{labels.shape}[0]$$

$$\text{accuracy} = \left(\frac{\text{correct}}{\text{total}}\right) \times 100$$

$$\text{TP} = \sum(\text{predictions} == 1)\&(\text{labels} == 1)$$

$$\text{FP} = \sum(\text{predictions} == 1)\&(\text{labels} == 0)$$

$$\text{TN} = \sum(\text{predictions} == 0)\&(\text{labels} == 0)$$

$$\text{FN} = \sum(\text{predictions} == 0)\&(\text{labels} == 1)$$

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

These are also, used in CNN.
CNN

1. **Validation :** After each epoch, the model is evaluated on the validation dataset:
   - Loss is calculated using criterion(output, target).
   - Accuracy is determined based on the highest logit score.

```
pred = output.argmax(dim=1, keepdim=True)
correct += pred.eq(target.view_as(pred)).sum().item()
```

2. **Testing :**After training, the best model is loaded and tested:

```
model.load_state_dict(torch.load('best_model.pth'))
test_loss, test_accuracy = validate(model, test_loader, criterion)
```

   - Final accuracy and loss are displayed.

7 Experiments and Results

7.1 Number of Epochs and Learning Rate:

Every model whether MLP or CNN were trained for 20 epochs. We experimented with different learning rated in MLP(the comparative analysis can be found below). For CNN we use standard ADAM optimiser with learning rate 0.001.

7.2 Best Performing MLP and CNN Models and Experimentation

7.2.1 MLP

The following hyperparameters gave best results in case of MLP:

Parameter	Value
input_size	28×28
layers	[1024,512,256, 128, 64, 10]
activations	[”gelu”,”gelu”,”gelu”, ”gelu” , ”gelu”, ”softmax”]
initialization	”random”
dropout	0.01
learning_rate	0.05

Table 1: Best PerormingMLP Model Parameters

And the performance metrics obtained are:

- **Accuracy: 87.40%**
- **Precision: 87%**
- **Recall: 87%**
- **F1 Score: 87%**

We first experimented with initialization methods, then fixed it as **random** as it performed the best. Then the following experiments were done: document

Subsequent Steps and Analysis-
Baseline Configuration:

Parameter	Value
input_size	28×28
layers	[256, 128, 64, 10]
activations	[”gelu”, ”gelu” , ”gelu”, ”softmax”]
initialization	”random”
dropout	0.01
learning_rate	0.05

Table 2: Best PerormingMLP Model Parameters

The baseline performance:

- **Accuracy: 86.92%**
- **Precision: 87%**
- **Recall: 87%**

- **F1 Score: 87%**

This serves as a baseline for comparison.

Dropout Rate Variations: Dropout rates tested: `0.2`, `0.1`, `0.05`, `0.01`, `0.3`, `0.4`, `0.5`.

- **Observations**:
  - Lower dropout rates (0.01, 0.05) led to higher accuracy (up to 87.40%).
  - Higher dropout rates (0.3, 0.4, 0.5) led to lower accuracy, likely due to underfitting.

Learning Rate Adjustments: Learning rates tested: `0.05`, `0.01`, `0.001`, `0.2`, `0.3`, `0.4`.

- **Observations**:
  - Lower learning rates (0.01) had marginally better performance.
  - Very high learning rates (0.3, 0.4) did not improve performance significantly.

Activation Functions: Activations tested: `gelu`, `relu`, `leaky_relu`, `tanh`.

- **Observations**:
  - `GELU` and `RELU` had similar performance.
  - `Leaky ReLU` and `Tanh` showed marginally lower performance.

Layer and Neuron Configurations: Configurations tested: `[64, 10]`, `[128, 64, 10]`, `[512, 256, 128, 64, 10]`, `[1024, 512, 256, 128, 64, 10]`.

- **Observations**:
  - Increased layers and neurons improved performance, with `[1024, 512, 256, 128, 64, 10]` showing the highest accuracy (87.42%).

7.2.2 CNN

We experimented with different hyperparameters such as the number of layers, activation function being ReLU, dropout rates, and weight initialization techniques (Xavier, He, Random).

1. The best model that we have proposed turn out to be following simple and elegant AlexNet architecture.
   - **Accuracy: 92.6%**
   - **Precision: 93%**
   - **Recall: 93%**
   - **F1 Score: 93%**
2. The intuition behind using ALexNet was very simple as the question suggested to use 5 convolution layer compulsory and AlexNet tends to be the architecture of 5 convolution layer
3. To compensate with the smaller 28x28 size image of out dataset, we have changed the kernel sizes , strides and padding. The best performing hyper parameters were: kernel size = [3, 3, 3, 3, 3], strides = [1, 1, 1, 1, 1], pooling = "max" with stride 2 and initialization being "he"

4. And the architecture of our best model is same as ALexNet with number of channels in each layer being [64, 192, 384, 256, 256]
5. In the end we used the same mlp architecture with two FC layers of 256 neurons with ReLU and a output layer of 10 neurons with softmax activation function.

7.3 Tabulated data

7.3.1 MLP

Hyperparameter	Type	Values					Best
Dropout Rate	Rate	0.1	0.2	0.3	0.4	0.5	**0.1**
	Accuracy (%)	87.11	86.61	86.00	85.48	84.10	**87.11**
Learning Rate	Rate	0.05	0.1	0.2	0.3	0.4	**0.05**
	Accuracy (%)	87.11	86.12	86.88	87.06	86.73	**87.11**
Initialization	Method	He	Random	Glorot	-	-	**Random**
	Accuracy (%)	85.61	86.92	86.37	-	-	**86.92**
Activation Function	Type	Gelu	ReLU	Leaky ReLU	Tanh	-	**Gelu**
	Accuracy (%)	87.24	86.73	86.50	86.51	-	**87.24**
Hidden Layers	Number	1	2	3	4	5	**5**
	Accuracy (%)	84.87	86.35	86.83	87.24	87.42	**87.42**

Table 3: Summary of Hyperparameter Tuning Results

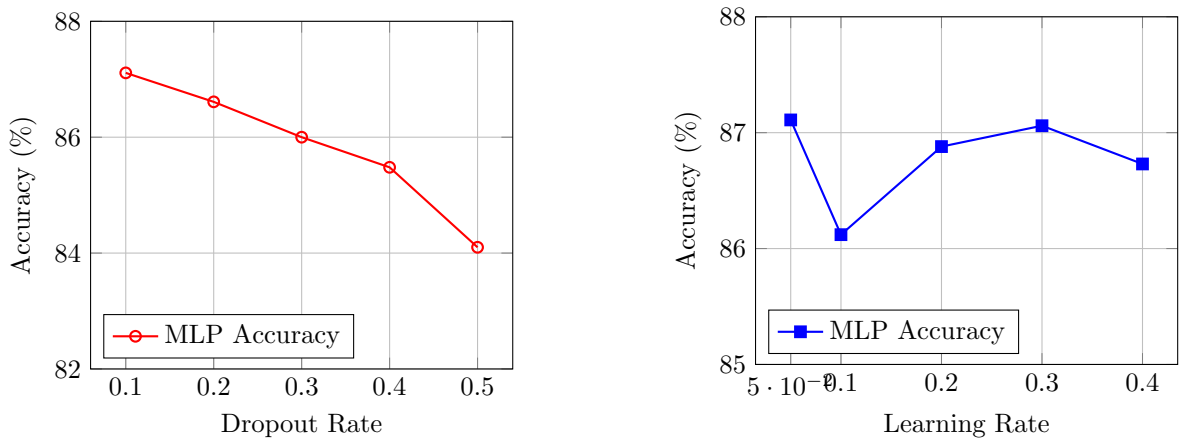
7.4 Graph Plots, Observations and Justifications

7.4.1 MLP

1. **Dropout Rate vs Accuracy:** Dropout is a regularization technique used to prevent overfitting in neural networks by randomly deactivating a fraction of neurons during training.
   - As dropout increases, fewer neurons contribute to learning, making the network more generalizable but possibly reducing accuracy.
   - The graph shows a downward trend: At 0.1 dropout, accuracy is 87.11%. At 0.5 dropout, accuracy drops to 84.10%, showing that excessive dropout reduces performance.
2. **Learning Rate vs Accuracy:** The learning rate determines how quickly the model updates weights during training.
   - A higher learning rate allows faster training but may result in instability.
   - A lower learning rate is more stable but might converge too slowly.
   - The graph shows that accuracy remains stable (around 86%) for learning rates between 0.1 and 0.3, with a slight drop for extreme values. At 0.05 LR, accuracy is 87.11%. At 0.4 LR, accuracy drops slightly to 86.73%.
3. **Initialization Methods vs Accuracy:** Compares different weight initialization methods.

   (Random) initialization gives the highest accuracy (86.92%), followed by Glarot (86.37%), and He initialization (85.61%).
4. **Activation Function vs Accuracy:** CCompares different activation functions.

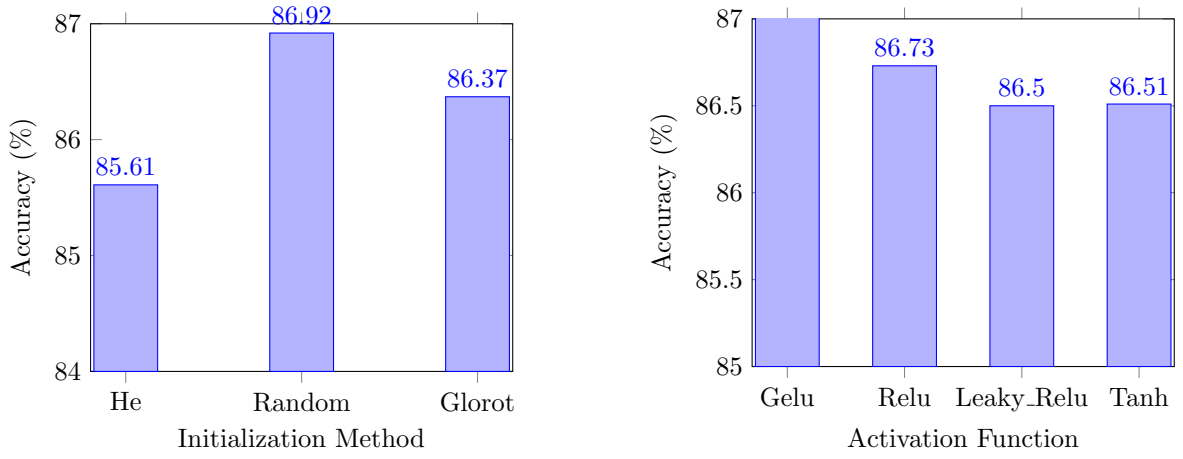
   GELU performs best (87.24%), followed by Leaky ReLU (86.50%).



(a) Dropout vs Accuracy

(b) Learning Rate vs Accuracy

Figure 2: Comparison of Dropout and Learning Rate Effects on Accuracy



(a) Initialization Methods vs Accuracy

(b) Activation Function vs Accuracy

Figure 3: Comparison of Initialization Methods and Activation Functions on MLP Accuracy

5. **Effect of Number of Hidden Layers on Accuracy:** The line graph visually represents how accuracy improves initially but slightly fluctuates at 4 layers before peaking again at 5 layers (87.42%).

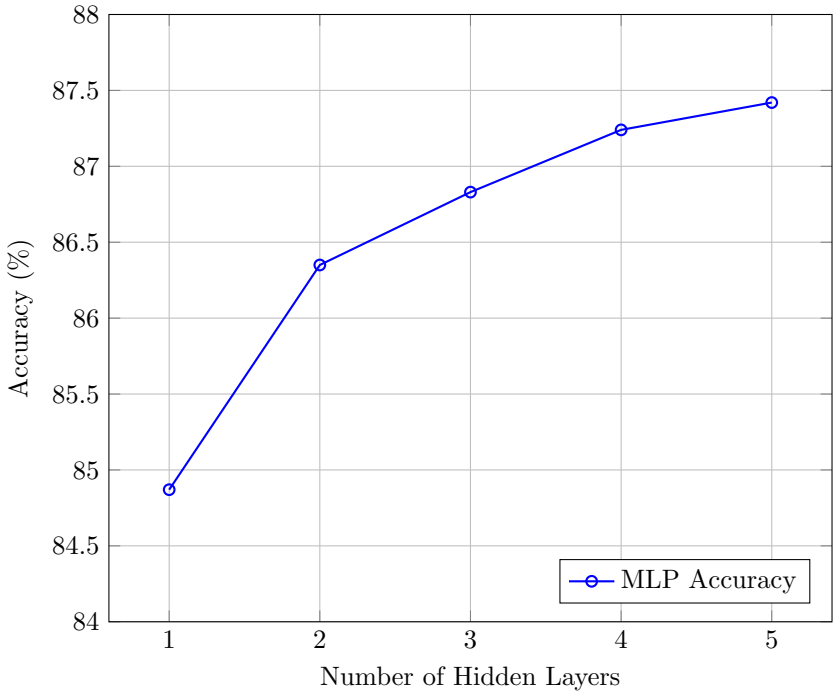


Figure 4: Impact of Hidden Layers on MLP Accuracy

7.4.2 CNN

1. **Training and Validation accuracy:** The following line graph(Figure 5) represents the training and validation accuracy of the best CNN Model that we have founded. The line graph also contains the accuracies for all three different pooling methods.
   - As the training progresses max pooling tends to perform better then other pooling methods.
   - With training accuracy being 96.7% , while the validation set accuracy being 92.6%.
2. **Experimenting with different kernel sizes:** We experimented with different kernel sizes, strides and paddings. The initialization that we used is random and number of channels in the CNN Model being 64 for all the five layers.
   - Thus for random initialization we checked for 3 cases of different kernel sizes and find out the configuration with max accuracy.
   - For the best configuration we also checked accuracy with different initialization parameters. The bar graph for the results can be seen in Figure 6.
3. **Experimenting with different initialization methods:** We experimented with initialization methods, and check the test accuracy values for each method.
   - Along with changing initialization methods we have also changed kernel sizes.
   - So finally we checked our results with two different kernel sizes whose size and accuracy value can be seen in the bar graph(Figure 7) below.

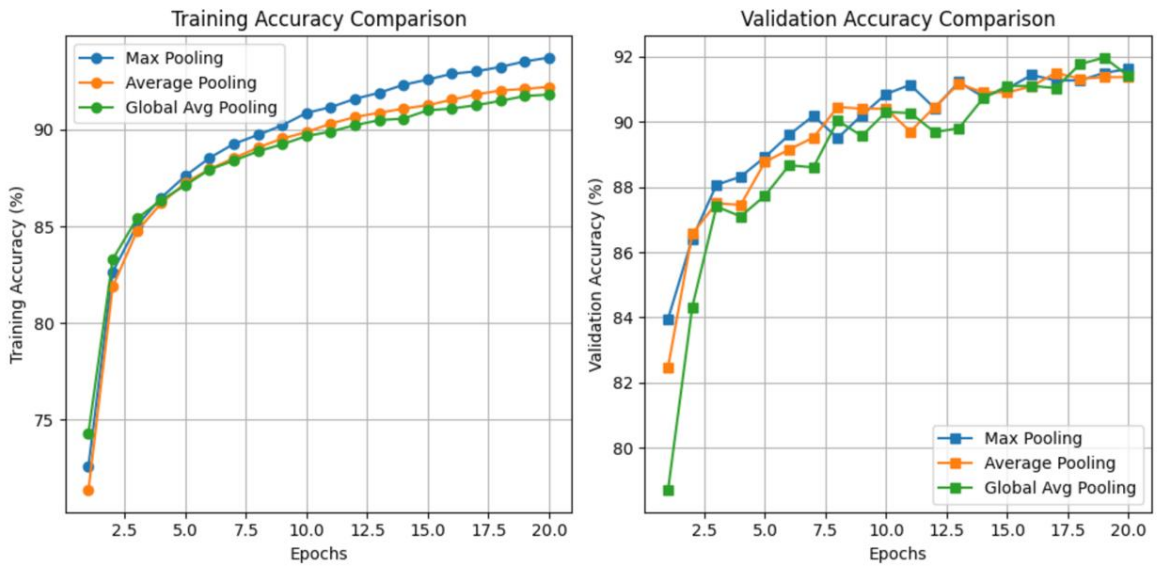
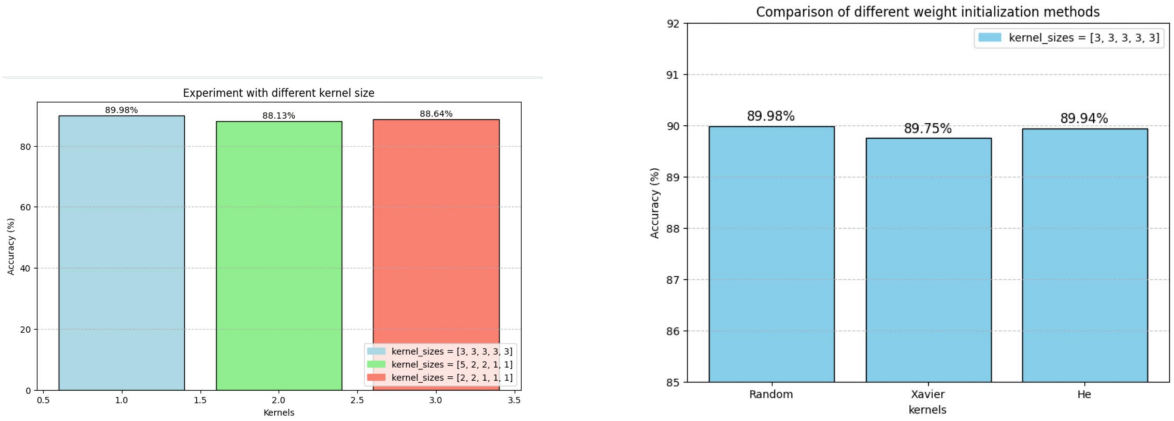


Figure 5: Train and Val Accuracy Comparison



(a) Different Kernel Sizes

(b) Different Init. on Best Configuration

Figure 6: Two images side by side

4. **Experimenting with different number of filters :** We also experimented with various number of filters, and check the test accuracy values for each architecture.
   - Among all the different architectures we have tried, we are showing the best 3 results based on test accuracy.
   - This experiment also proved our hypothesis that ALexNet architecture with sightly different kernel sizes should perform better then other architectures.
   - The bar graph(Figure 8) shows the results of our experiment.

5. **Experimenting with different pooling layers :** We also experimented with different pooling layers, and check the test accuracy values for type of pooling.
   - Among all the different pooling methods we have observed that max pooling tends to give us highest test accuracy of 92.5%.
   - This observation also makes intuitive sense as our best model is nothing but AlexNet model which uses max pooling after the convolutional layers.

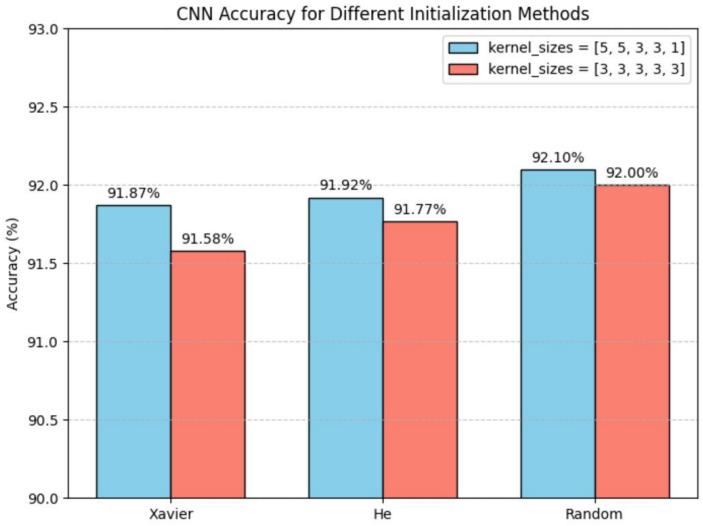


Figure 7: Test Accuracies for Two Kernel Configuration and Diff Init Methods

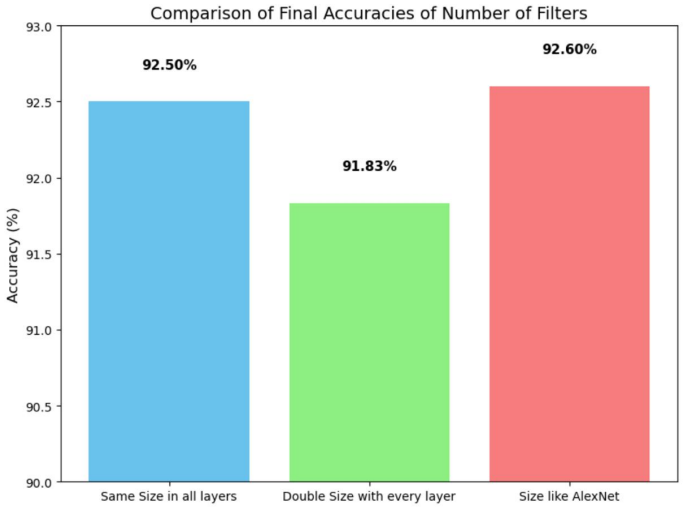


Figure 8: Test Accuracies for diff Architectures we Implemented

- The bar graph(Figure 9) shows the results of our experimentation.

8 Conclusion

The CNN model achieved better accuracy compared to the MLP model. The use of convolutional layers allowed the CNN to extract spatial features, making it more suitable for image classification tasks. Hyperparameter tuning, activation functions, and weight initialization played a crucial role in improving model performance.

9 Code for Custom MLP + CNN

- We provide a notebook that contains code for custom MLP with CNN, in this code we just change the MLP class of task1 that used numpy arrays with pytorch tenosrs.
- Also, the cvtask2final notebook has CNN models that contains nn.Linear layers.

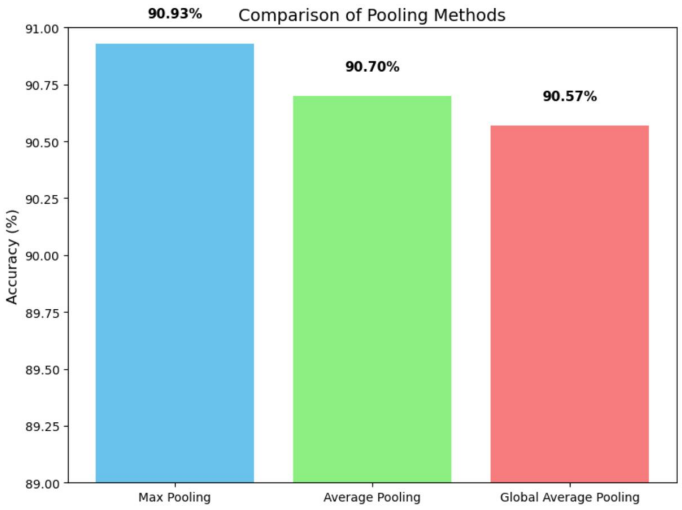


Figure 9: Test Accuracies for diff Pooling Methods

10 Instructions for Running the Code

- All the submitted Notebooks were run on Kaggle.
- So, to run and reproduce results, one can just upload the notebooks on kaggle and click on run all.

11 References

1. ImageNet Classification with Deep Convolutional Neural Networks
2. Pytorch Tutorial