CS776: Deep Learning for Computer Vision
Group Name: TBD

DIVYANSHU 241110023

KHUSWANT KASWAN 241110035

RISHIT KUMAR 241110056

KRISHANU RAY 241110037

SENTHIL GANESH 241110089

RAJAN KUMAR 241110087

Assignment Report

Submitted To

Dr. Priyanka Bagade

1 Question 1 : MLP

This report explains how a Multi-Layer Perceptron (MLP) was built from scratch to classify images from the Fashion-MNIST dataset. It covers the implementation of the forward and backward passes, as well as a custom cross-entropy loss function. The study also explores how different hyperparameters and activation functions affect the model's performance. The main goal is to understand how design choices and optimization methods influence the model's accuracy and ability to generalize.

1.1 Data Preprocessing Module

1.1.1 Flattening Images

The Fashion-MNIST dataset contains grayscale images of size 28×28 pixels. These images were flattened into a 784-dimensional vector ($28 \times 28 = 784$) to be compatible with the MLP. This transformation allows the MLP to process the image data as a 1D input vector.

1.1.2 Preprocessing Pipeline

The preprocessing module includes:

- **Normalization**: Pixel values were rescaled to a range between 0 and 1 by dividing by 255. This step ensures that the input data is on a consistent scale, which helps in faster convergence during training.
- **Standardization or Z-score normalization** : The pixel values will be rescaled so that they'll have the properties of a standard normal distribution i.e. zero mean and unit standard deviation. We have computed mean and std from training data only and applied over both Training and test data. To avoid any case of division with zero we have used : $z = \frac{x-\mu}{\sigma+\varepsilon}$
- **Train-validation-test split**: The main dataset consist of train.csv and test.csv. Training dataset was split into 80% training, 20% validation. The validation set was used to prevent overfitting by implementing early stopping based of validation loss.

1.2 Forward Pass (Implemented from Scratch)

The forward pass computation follows:

- **Input**: Flattened image vector X of size 784×1.
- **Matrix multiplication**: For each layer, the output is computed as $Z = W.X^T + b$, where W is the weight matrix and b is the bias vector.
- **Activation function**: Applied activation function to Z. The activation functions experimented with include:
  - ReLU: $f(z) = \max(0, z)$
  - Leaky-ReLU: $f(z) = \max(0.01z, z)$
  - Tanh: $f(z) = \tanh(z)$
  - GELU: $f(z) = z \cdot \Phi(z)$, where $\Phi(z)$ is the cumulative distribution function of the standard normal distribution.
- **Dropout** : We created a dropout mask based on the dropout rate and modified the value of input X with that mask during training. We store the mask to preserves dropout pattern for backward pass.
- **Cache** : Stores pre-activation (Z) and post-activation (A) values. Retains dropout masks for backward pass.
- **Output**: The final layer uses the softmax function to produce class probabilities:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

  where C is the number of classes.

- **Mathematical Flow**:

$$Z^l = W^l A^{l-1} + b^l \tag{1}$$
$$A^l = g^l(Z^l) \tag{2}$$

Input → W1 → Z1 → ReLU → A1 → Dropout → W2 → Z2 → ReLU → A2 → W3 → Z3 → Softmax → Output

1.3 Backward Pass (Implemented from Scratch)

The backward pass updates weights by calculating gradients using the chain rule. The steps are as follows:

- Softmax Derivative (Batch Version) :

$$\frac{\partial L}{\partial Z^L} = \frac{\hat{Y} - Y}{m}$$

- Propagate gradients backward through the network using the chain rule. For each layer, compute:
  For layer l:

$$\frac{\partial \mathcal{L}}{\partial Z^{[l]}} = \underbrace{\left(W^{[l+1]}\right)^T}_{\text{Upstream gradient}} \frac{\partial \mathcal{L}}{\partial Z^{[l+1]}} \circ \underbrace{g'^{[l]}(Z^{[l]})}_{\text{Activation derivative}}$$

  Parameter Gradients

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} \frac{\partial \mathcal{L}}{\partial Z^{[l]}} (A^{[l-1]})^T$$

$$\frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$$

- Update weights and biases using gradient descent:

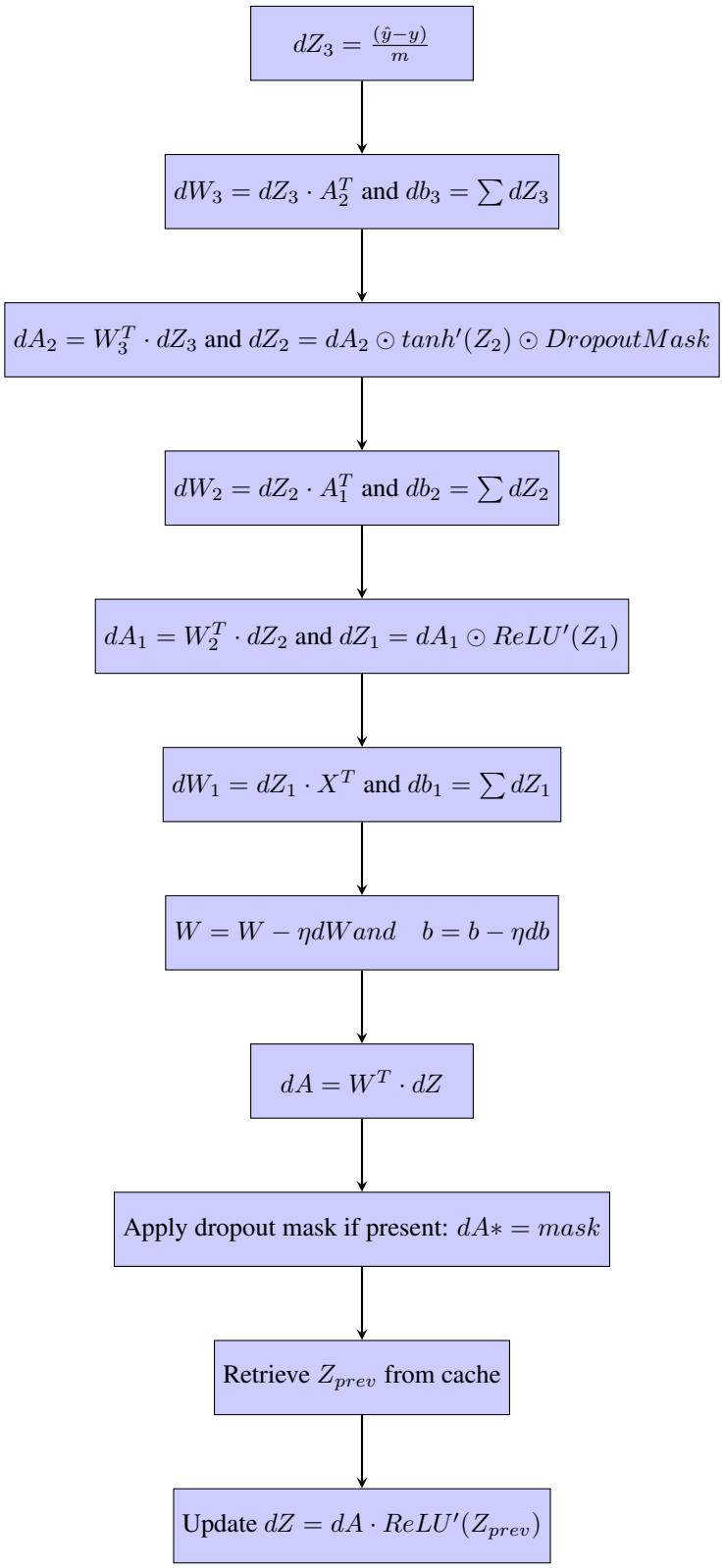
$$W_{\text{new}} = W_{\text{old}} - \alpha \frac{\partial \mathcal{L}}{\partial W}, \quad b = b - \alpha \frac{\partial \mathcal{L}}{\partial b}$$

  where α is the learning rate.

Code to Mathematics Mapping

Code Component	Mathematical Equivalent
`dZ = (y_pred - y_true).T/m`	$\frac{\partial \mathcal{L}}{\partial Z^{[l]}}$
`dW = dZ @ A_prev.T`	$\frac{\partial \mathcal{L}}{\partial W^{[l]}}$
`db = np.sum(dZ, axis=1)`	$\frac{\partial \mathcal{L}}{\partial b^{[l]}}$
`dA = W.T @ dZ`	$(W^{[l]})^T \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$
`dZ *= activation_derivative`	$\circ g'^{[l-1]}(Z^{[l-1]})$
`layer['W'] -= learning_rate * dW`	$W_{\text{new}} = W_{\text{old}} - \alpha \frac{\partial \mathcal{L}}{\partial W}$
`layer['b'] -= learning_rate * db`	$b_{\text{new}} = b_{\text{old}} - \alpha \frac{\partial \mathcal{L}}{\partial b}$

Mathematical Flow



1.4 Custom Cross-Entropy Loss Function

For softmax output layer with m sample, a custom cross-entropy loss function is implemented using the formula:

$$\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic}) \tag{3}$$

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

where y_i is the true label and $\hat{y} = softmax(Z^L)$. This loss function is used to measure the discrepancy between the predicted and true class distributions.

1.5 Hyperparameter Experiments

1.5.1 Experimental Setups

Different configurations were tested to analyze the impact of architecture, activation functions, and dropout on model performance. The configurations are summarized below:

Layers	Activation	Dropout	Learning Rate	Accuracy
784, 128, 10	ReLU, Softmax	0.2	0.001	88.25
784, 128, 10	ReLU, Softmax	0	0.001	88.15
784, 128, 10	GELU, Softmax	0.2	0.001	88.35
784, 128, 10	GELU, Softmax	0	0.001	87.87
784, 128, 10	Leaky-gelu, Softmax	0.2	0.001	87.96
784, 400, 250, 100, 10	ReLU, ReLU, ReLU, Softmax	0.2	0.001	88.55
784, 400, 250, 100, 10	Leaky-ReLU, Leaky-ReLU, Leaky-ReLU, Softmax	0.2	0.001	88.34
784, 256, 128, 10	GELU, GELU,GELU, Softmax	0.2	0.001	88.46

1.5.2 Observations

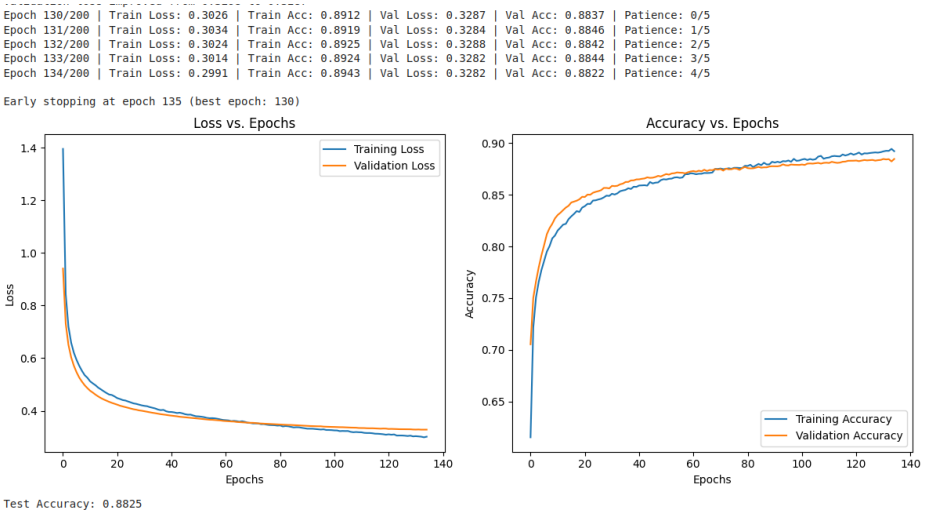
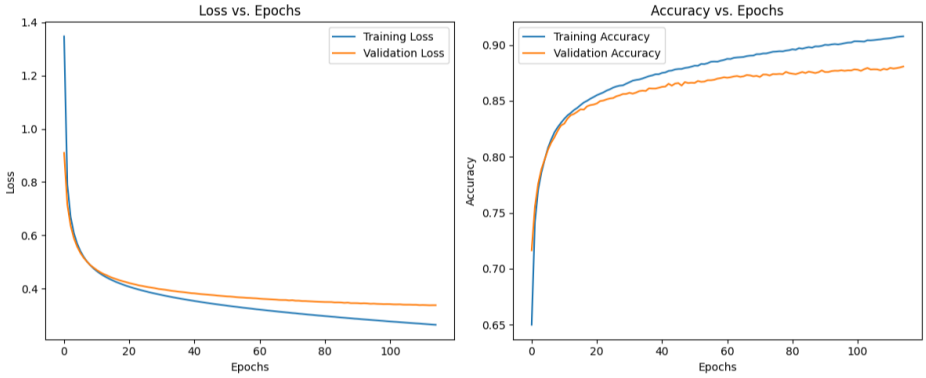


Figure 1: layer_sizes = [784, 128, 10] — activations = ['relu', 'softmax'] — dropout = 0.2

```
Epoch 110/200 | Train Loss: 0.2688 | Train Acc: 0.9057 | Val Loss: 0.3382 | Val Acc: 0.8779 | Patience: 0/5
Epoch 111/200 | Train Loss: 0.2678 | Train Acc: 0.9060 | Val Loss: 0.3386 | Val Acc: 0.8794 | Patience: 1/5
Epoch 112/200 | Train Loss: 0.2669 | Train Acc: 0.9065 | Val Loss: 0.3382 | Val Acc: 0.8788 | Patience: 2/5
Epoch 113/200 | Train Loss: 0.2660 | Train Acc: 0.9071 | Val Loss: 0.3375 | Val Acc: 0.8792 | Patience: 3/5
Epoch 114/200 | Train Loss: 0.2650 | Train Acc: 0.9074 | Val Loss: 0.3377 | Val Acc: 0.8798 | Patience: 4/5

Early stopping at epoch 115 (best epoch: 110)
```

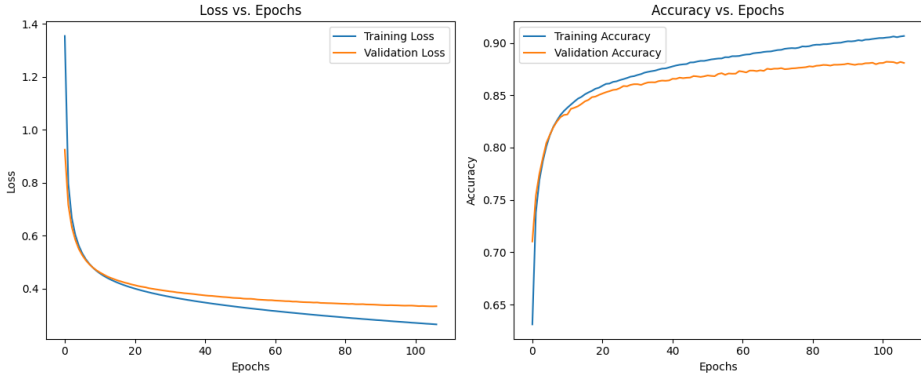



```
Test Accuracy: 0.8815
```

Figure 2: layer_sizes = [784, 128, 10] — activations = [’relu’, ’softmax’] — dropout = 0

```
Epoch 102/200 | Train Loss: 0.2691 | Train Acc: 0.9052 | Val Loss: 0.3336 | Val Acc: 0.8820 | Patience: 0/5
Epoch 103/200 | Train Loss: 0.2683 | Train Acc: 0.9055 | Val Loss: 0.3341 | Val Acc: 0.8818 | Patience: 1/5
Epoch 104/200 | Train Loss: 0.2672 | Train Acc: 0.9062 | Val Loss: 0.3334 | Val Acc: 0.8817 | Patience: 2/5
Epoch 105/200 | Train Loss: 0.2664 | Train Acc: 0.9055 | Val Loss: 0.3329 | Val Acc: 0.8806 | Patience: 3/5
Epoch 106/200 | Train Loss: 0.2655 | Train Acc: 0.9062 | Val Loss: 0.3327 | Val Acc: 0.8818 | Patience: 4/5

Early stopping at epoch 107 (best epoch: 102)
```

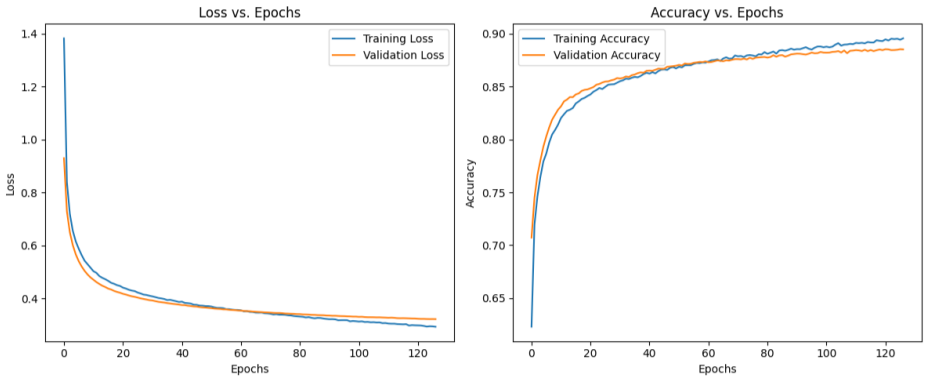



```
Test Accuracy: 0.8787
```

Figure 3: layer_sizes = [784, 128, 10] — activations = [’gelu’,’softmax’] — dropout = 0

```
Epoch 122/200 | Train Loss: 0.2984 | Train Acc: 0.8937 | Val Loss: 0.3233 | Val Acc: 0.8852 | Patience: 0/5
Epoch 123/200 | Train Loss: 0.2968 | Train Acc: 0.8954 | Val Loss: 0.3233 | Val Acc: 0.8846 | Patience: 1/5
Epoch 124/200 | Train Loss: 0.2944 | Train Acc: 0.8950 | Val Loss: 0.3228 | Val Acc: 0.8847 | Patience: 2/5
Epoch 125/200 | Train Loss: 0.2955 | Train Acc: 0.8954 | Val Loss: 0.3226 | Val Acc: 0.8849 | Patience: 3/5
Epoch 126/200 | Train Loss: 0.2950 | Train Acc: 0.8944 | Val Loss: 0.3227 | Val Acc: 0.8854 | Patience: 4/5

Early stopping at epoch 127 (best epoch: 122)
```

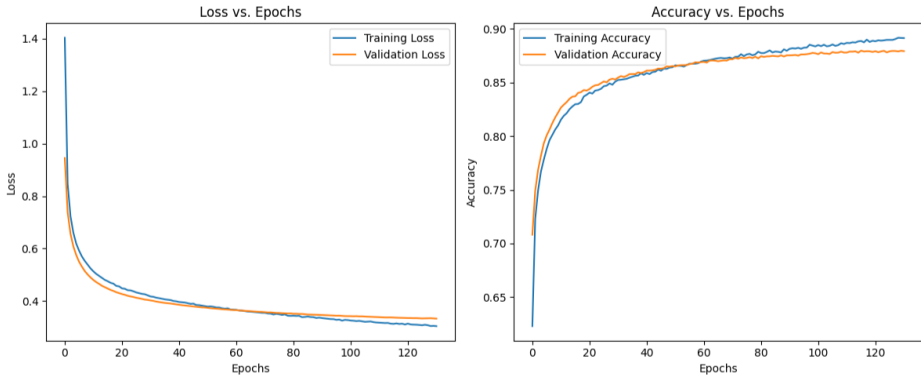



```
Test Accuracy: 0.8835
```

Figure 4: layer_sizes = [784, 128, 10] — activations = [’gelu’, ’softmax’] — dropout = 0.2

```
Epoch 126/200 | Train Loss: 0.3079 | Train Acc: 0.8896 | Val Loss: 0.3338 | Val Acc: 0.8790 | Patience: 0/5
Epoch 127/200 | Train Loss: 0.3097 | Train Acc: 0.8896 | Val Loss: 0.3341 | Val Acc: 0.8793 | Patience: 1/5
Epoch 128/200 | Train Loss: 0.3079 | Train Acc: 0.8904 | Val Loss: 0.3340 | Val Acc: 0.8792 | Patience: 2/5
Epoch 129/200 | Train Loss: 0.3048 | Train Acc: 0.8916 | Val Loss: 0.3346 | Val Acc: 0.8789 | Patience: 3/5
Epoch 130/200 | Train Loss: 0.3055 | Train Acc: 0.8914 | Val Loss: 0.3340 | Val Acc: 0.8795 | Patience: 4/5

Early stopping at epoch 131 (best epoch: 126)
```

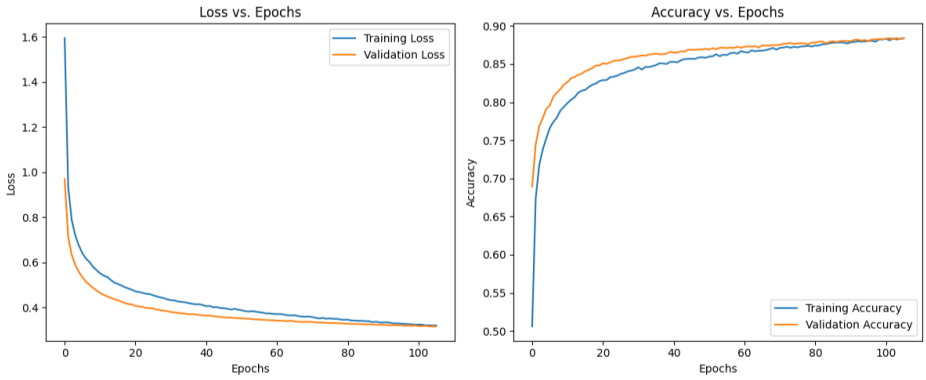



```
Test Accuracy: 0.8796
```

Figure 5: layer_sizes = [784, 128, 10] — activations = [’leaky-gelu’,’softmax’] — dropout = 0.2

```
Epoch 101/200 | Train Loss: 0.3244 | Train Acc: 0.8830 | Val Loss: 0.3174 | Val Acc: 0.8832 | Patience: 0/5
Epoch 102/200 | Train Loss: 0.3243 | Train Acc: 0.8811 | Val Loss: 0.3179 | Val Acc: 0.8839 | Patience: 1/5
Epoch 103/200 | Train Loss: 0.3211 | Train Acc: 0.8835 | Val Loss: 0.3191 | Val Acc: 0.8825 | Patience: 2/5
Epoch 104/200 | Train Loss: 0.3212 | Train Acc: 0.8822 | Val Loss: 0.3168 | Val Acc: 0.8840 | Patience: 3/5
Epoch 105/200 | Train Loss: 0.3205 | Train Acc: 0.8832 | Val Loss: 0.3166 | Val Acc: 0.8828 | Patience: 4/5

Early stopping at epoch 106 (best epoch: 101)
```

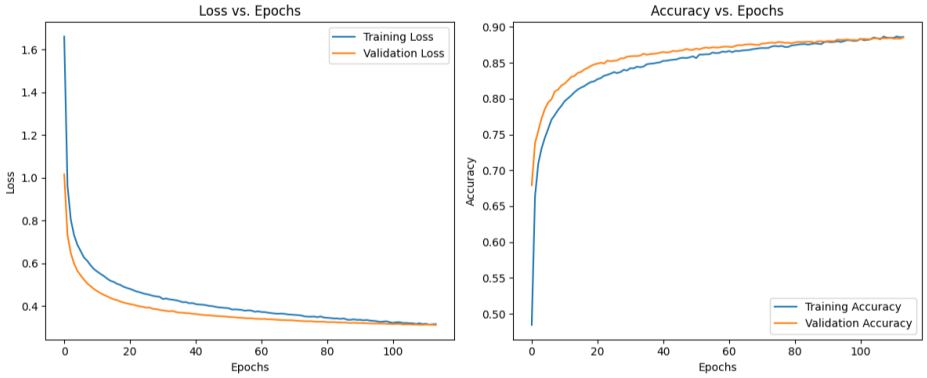



```
Test Accuracy: 0.8855
```

Figure 6: layer_sizes =[784, 400, 250, 100, 10] — activations = [’relu’, ’relu’, ’relu’, ’softmax’] — dropout = 0.2

```
Epoch 109/200 | Train Loss: 0.3199 | Train Acc: 0.8849 | Val Loss: 0.3130 | Val Acc: 0.8839 | Patience: 0/5
Epoch 110/200 | Train Loss: 0.3162 | Train Acc: 0.8846 | Val Loss: 0.3131 | Val Acc: 0.8852 | Patience: 1/5
Epoch 111/200 | Train Loss: 0.3178 | Train Acc: 0.8846 | Val Loss: 0.3142 | Val Acc: 0.8835 | Patience: 2/5
Epoch 112/200 | Train Loss: 0.3133 | Train Acc: 0.8865 | Val Loss: 0.3131 | Val Acc: 0.8838 | Patience: 3/5
Epoch 113/200 | Train Loss: 0.3153 | Train Acc: 0.8855 | Val Loss: 0.3128 | Val Acc: 0.8835 | Patience: 4/5

Early stopping at epoch 114 (best epoch: 109)
```

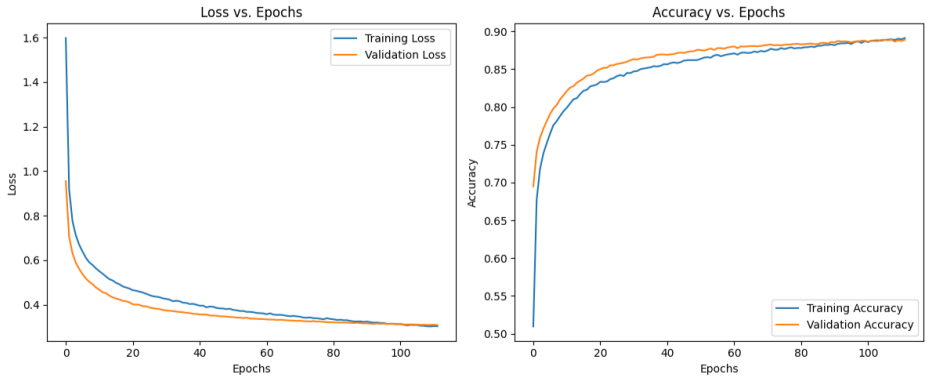



```
Test Accuracy: 0.8834
```

Figure 7: layer_sizes = [784, 400, 250, 100, 10] — activations = ['leaky relu', 'leaky relu', 'leaky relu', 'softmax'] — dropout = 0.2

```
Epoch 107/200 | Train Loss: 0.3058 | Train Acc: 0.8891 | Val Loss: 0.3095 | Val Acc: 0.8881 | Patience: 0/5
Epoch 108/200 | Train Loss: 0.3053 | Train Acc: 0.8895 | Val Loss: 0.3100 | Val Acc: 0.8890 | Patience: 1/5
Epoch 109/200 | Train Loss: 0.3030 | Train Acc: 0.8886 | Val Loss: 0.3093 | Val Acc: 0.8865 | Patience: 2/5
Epoch 110/200 | Train Loss: 0.3026 | Train Acc: 0.8903 | Val Loss: 0.3098 | Val Acc: 0.8882 | Patience: 3/5
Epoch 111/200 | Train Loss: 0.3040 | Train Acc: 0.8896 | Val Loss: 0.3095 | Val Acc: 0.8872 | Patience: 4/5

Early stopping at epoch 112 (best epoch: 107)
```




```
Test Accuracy: 0.8846
```

Figure 8: layer_sizes = [784, 400, 250, 100, 10] — activations = ['gelu', 'gelu', 'gelu', 'softmax'] — dropout = 0.2

- **Depth of the Network**: Models with multiple hidden layers (e.g., [784, 400, 250, 100, 10]) performed better than shallower architectures (e.g., [784, 128, 10]). This is because deeper networks can learn more complex features.

- **Dropout**: Dropout improved generalization by preventing overfitting. However, using too much dropout (e.g., 0.5) led to underfitting, as the model struggled to learn meaningful patterns.

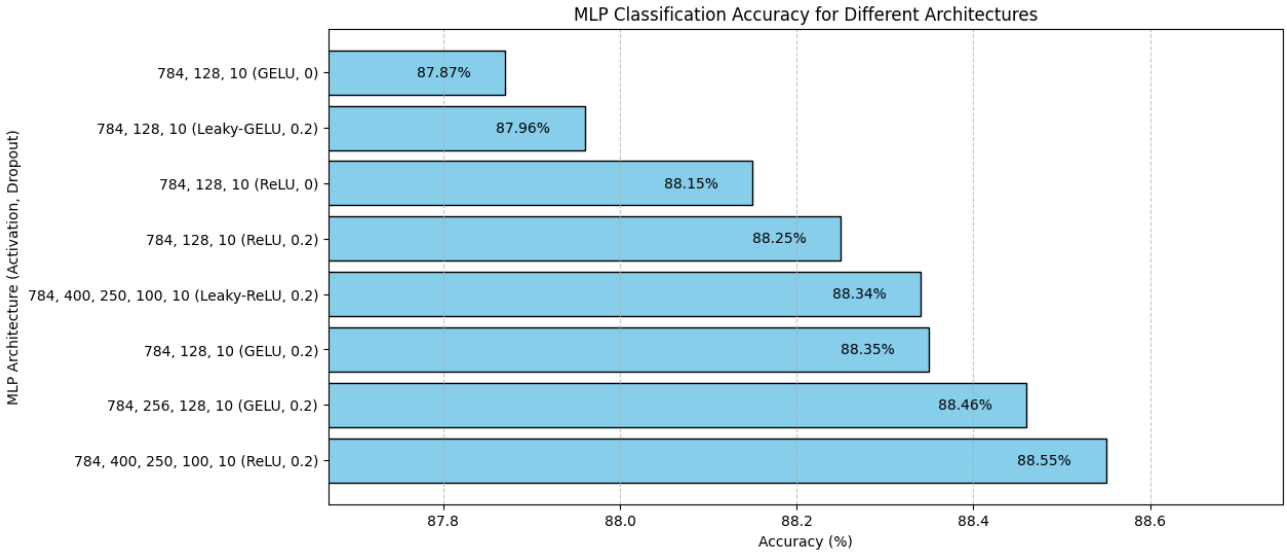
- **Activation Functions**: GELU provided slightly better performance than ReLU in deeper networks, likely due to its smooth gradient properties. Leaky-ReLU helped avoid dead neurons, performing better in some cases.

- **Learning Rate**: A learning rate of 0.001 was found to be optimal. Higher learning rates caused instability, while lower rates slowed convergence.

1.5.3 Best Performing Model

The best-performing configuration was:

- **Architecture**: [784, 400, 250, 100, 10]
- **Activation Functions**: relu, relu, relu, Softmax
- **Dropout**: 0.2
- **Learning Rate**: 0.001
- **Accuracy**: Achieved 88.55% on the test set.



2. Pytorch CNN

Introduction

This report details the process of training a 5-layered small Convolutional Neural Network (CNN) on the Fashion MNIST dataset. The experiments include variations in kernel sizes, the number of kernels, different weight initialization methods, and the use of multi-layer perceptron (MLP) from Task1 for classification.

Dataset and Preprocessing

Fashion MNIST is a dataset consisting of grayscale images of clothing items, each with a resolution of 28x28 pixels. The data set is pre-processed as follows:

- Normalization: The pixel values are normalized to the range [-1, 1] using the transformation:
- Train-Test Split: The training data set is further divided into training (80%) and validation (20%) subsets.
- Conversion to Tensors: The processed data is converted into PyTorch tensors for compatibility with the deep learning model.

5 Layered Small CNN Architecture

The implemented CNN model consists of five convolutional layers with ReLU activation functions and a specified pooling mechanism. The architecture includes:

- Layers: 5 convolutional layers with filter dimensions
- Kernel Size: Fixed at 3×3 with padding 1 to preserve spatial dimensions.
- Activation: ReLU after each convolution.
- Pooling: Max-pooling, average-pooling, or global average-pooling after each convolutional layer except the last. This reduces spatial dimensions progressively.
- The extracted features are flattened and passed through a Fully Connected Classifier for final classification.
    - First Fully Connected Layer: 256 neurons, ReLU activation
    - Second Fully Connected Layer/Output Layer: 10 neurons (for 10 Fashion MNIST classes)

Q1. A small CNN model consisting of 5 convolution layers is built using pytorch. Each convolution layer is followed by a ReLU activation. In the pooling layer, different pooling methods such as max pooling, average pooling and global average pooling is implemented. **Firstly they are implemented with the same number of filters in all the layers.** Their comparison is as follows (10 Marks)

Example run :
train_cnn_extractor(epochs=10, pool_method='max', weight_init='xavier',conv_dims=[32, 64, 128, 256,512],n6=1024)

Implementation with Max pooling,random weights, same kernels

```
(48000, 784)
Initializing Feature Extractor Training...
max random [16, 16, 16, 16, 16] MLP Hidden Layer:  32
16
Epoch 1/10 | Train Loss: 0.8691 | Train Acc: 0.6721 | Val Loss: 0.5755 | Val Acc: 0.7809
Epoch 2/10 | Train Loss: 0.5349 | Train Acc: 0.8027 | Val Loss: 0.4758 | Val Acc: 0.8214
Epoch 3/10 | Train Loss: 0.4556 | Train Acc: 0.8349 | Val Loss: 0.4382 | Val Acc: 0.8387
Epoch 4/10 | Train Loss: 0.4155 | Train Acc: 0.8495 | Val Loss: 0.4385 | Val Acc: 0.8393
Epoch 5/10 | Train Loss: 0.3893 | Train Acc: 0.8584 | Val Loss: 0.3870 | Val Acc: 0.8556
Epoch 6/10 | Train Loss: 0.3657 | Train Acc: 0.8681 | Val Loss: 0.3764 | Val Acc: 0.8625
Epoch 7/10 | Train Loss: 0.3497 | Train Acc: 0.8735 | Val Loss: 0.3535 | Val Acc: 0.8708
Epoch 8/10 | Train Loss: 0.3314 | Train Acc: 0.8815 | Val Loss: 0.3491 | Val Acc: 0.8732
Epoch 9/10 | Train Loss: 0.3189 | Train Acc: 0.8852 | Val Loss: 0.3229 | Val Acc: 0.8835
Epoch 10/10 | Train Loss: 0.3085 | Train Acc: 0.8890 | Val Loss: 0.3380 | Val Acc: 0.8736
Model Training Completed
Extracted feature shapes: (48000, 16) (12000, 16) (10000, 16)

Test Accuracy: 0.8769
```

```
Training custom MLP on CNN features...
[16, 32, 10] ['relu', 'softmax']

Early stopping at epoch 52 (best epoch: 42)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.8868
```

The test accuracy achieved in this implementation is **88.68%**

Implementation with Average pooling,random weights, same kernels

```
(48000, 784)
Initializing Feature Extractor Training...
avg random [16, 16, 16, 16, 16] MLP Hidden Layer:  32
16
Epoch 1/10 | Train Loss: 1.0396 | Train Acc: 0.6113 | Val Loss: 0.8167 | Val Acc: 0.7006
Epoch 2/10 | Train Loss: 0.7366 | Train Acc: 0.7237 | Val Loss: 0.6741 | Val Acc: 0.7399
Epoch 3/10 | Train Loss: 0.6414 | Train Acc: 0.7525 | Val Loss: 0.6021 | Val Acc: 0.7641
Epoch 4/10 | Train Loss: 0.5874 | Train Acc: 0.7744 | Val Loss: 0.5636 | Val Acc: 0.7745
Epoch 5/10 | Train Loss: 0.5542 | Train Acc: 0.7868 | Val Loss: 0.5671 | Val Acc: 0.7803
Epoch 6/10 | Train Loss: 0.5276 | Train Acc: 0.7981 | Val Loss: 0.5310 | Val Acc: 0.7933
Epoch 7/10 | Train Loss: 0.5056 | Train Acc: 0.8071 | Val Loss: 0.5024 | Val Acc: 0.8048
Epoch 8/10 | Train Loss: 0.4886 | Train Acc: 0.8163 | Val Loss: 0.4756 | Val Acc: 0.8158
Epoch 9/10 | Train Loss: 0.4730 | Train Acc: 0.8230 | Val Loss: 0.4675 | Val Acc: 0.8222
Epoch 10/10 | Train Loss: 0.4551 | Train Acc: 0.8313 | Val Loss: 0.4572 | Val Acc: 0.8232
Model Training Completed
Extracted feature shapes: (48000, 16) (12000, 16) (10000, 16)

Test Accuracy: 0.8293
```

```
Training custom MLP on CNN features...
[16, 32, 10] ['relu', 'softmax']

Early stopping at epoch 65 (best epoch: 55)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.8337
```

The best test accuracy achieved in this implementation is **83.37%**

Implementation with Global average pooling, random weights and same kernels

```
(48000, 784)
Initializing Feature Extractor Training...
global random [16, 16, 16, 16, 16] MLP Hidden Layer:  32
16
Epoch 1/10 | Train Loss: 1.9720 | Train Acc: 0.2481 | Val Loss: 1.8179 | Val Acc: 0.3097
Epoch 2/10 | Train Loss: 1.8058 | Train Acc: 0.3135 | Val Loss: 1.7819 | Val Acc: 0.3093
Epoch 3/10 | Train Loss: 1.7751 | Train Acc: 0.3215 | Val Loss: 1.7525 | Val Acc: 0.3301
Epoch 4/10 | Train Loss: 1.7426 | Train Acc: 0.3238 | Val Loss: 1.7098 | Val Acc: 0.3242
Epoch 5/10 | Train Loss: 1.6883 | Train Acc: 0.3323 | Val Loss: 1.6355 | Val Acc: 0.3527
Epoch 6/10 | Train Loss: 1.6053 | Train Acc: 0.3603 | Val Loss: 1.5533 | Val Acc: 0.3800
Epoch 7/10 | Train Loss: 1.5328 | Train Acc: 0.3815 | Val Loss: 1.6286 | Val Acc: 0.3305
Epoch 8/10 | Train Loss: 1.5108 | Train Acc: 0.3886 | Val Loss: 1.4770 | Val Acc: 0.3981
Epoch 9/10 | Train Loss: 1.4909 | Train Acc: 0.3960 | Val Loss: 1.4941 | Val Acc: 0.3938
Epoch 10/10 | Train Loss: 1.4792 | Train Acc: 0.4024 | Val Loss: 1.4537 | Val Acc: 0.4131
Model Training Completed
Extracted feature shapes: (48000, 16) (12000, 16) (10000, 16)

Test Accuracy: 0.4162

Training custom MLP on CNN features...
[16, 32, 10] ['relu', 'softmax']

Early stopping at epoch 94 (best epoch: 84)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.4129
```

The test accuracy achieved in this implementation is **41.29%** which is **the worst among all implementations.** The reason for this accuracy is that in global average pooling the feature returned after each pooling layer for all kernels is just one (the average of all).

Q2. Experiment with different kernel sizes is done. We tried (2X2) (3X3) and (5X5). **(3X3) is the best one.** Number of kernels in each layer (keep the number of filters the same in each layer, double it in each layer etc) is also experimented. **In the previous part we have already implemented the same number of filters in all the layers. In this part we have doubled the number of kernels in each layer.** Their comparison is as follows (10 Marks)

Implementation with Max pooling and random weights and double the number of kernels in each layer

```
Initializing Feature Extractor Training...
max random [16, 32, 64, 128, 256] MLP Hidden Layer:  512
256
Epoch 1/10 | Train Loss: 0.5904 | Train Acc: 0.7829 | Val Loss: 0.3855 | Val Acc: 0.8628
Epoch 2/10 | Train Loss: 0.3602 | Train Acc: 0.8684 | Val Loss: 0.3209 | Val Acc: 0.8842
Epoch 3/10 | Train Loss: 0.3025 | Train Acc: 0.8894 | Val Loss: 0.2971 | Val Acc: 0.8925
Epoch 4/10 | Train Loss: 0.2722 | Train Acc: 0.9001 | Val Loss: 0.2691 | Val Acc: 0.9024
Epoch 5/10 | Train Loss: 0.2464 | Train Acc: 0.9098 | Val Loss: 0.3301 | Val Acc: 0.8772
Epoch 6/10 | Train Loss: 0.2248 | Train Acc: 0.9171 | Val Loss: 0.2611 | Val Acc: 0.9020
Epoch 7/10 | Train Loss: 0.2117 | Train Acc: 0.9226 | Val Loss: 0.2562 | Val Acc: 0.9061
Epoch 8/10 | Train Loss: 0.1951 | Train Acc: 0.9291 | Val Loss: 0.2653 | Val Acc: 0.9081
Epoch 9/10 | Train Loss: 0.1783 | Train Acc: 0.9350 | Val Loss: 0.2468 | Val Acc: 0.9084
Epoch 10/10 | Train Loss: 0.1672 | Train Acc: 0.9378 | Val Loss: 0.2499 | Val Acc: 0.9093
Model Training Completed
Extracted feature shapes: (48000, 256) (12000, 256) (10000, 256)

Test Accuracy: 0.9155
```

```
(10000, 256) (10000, 10)
Training custom MLP on CNN features...
[256, 512, 10] ['relu', 'softmax']
Validation loss improved from inf to 0.2677

Early stopping at epoch 16 (best epoch: 11)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.9179
```

The test accuracy achieved in this implementation is **91.79%**

Implementation with Average pooling and random weights and double the number of kernels in each layer

```
(48000, 784)
Initializing Feature Extractor Training...
avg random [16, 32, 64, 128, 256] MLP Hidden Layer:  512
256
Epoch 1/10 | Train Loss: 0.7926 | Train Acc: 0.7000 | Val Loss: 0.6244 | Val Acc: 0.7562
Epoch 2/10 | Train Loss: 0.5126 | Train Acc: 0.8046 | Val Loss: 0.4689 | Val Acc: 0.8188
Epoch 3/10 | Train Loss: 0.4221 | Train Acc: 0.8415 | Val Loss: 0.3883 | Val Acc: 0.8575
Epoch 4/10 | Train Loss: 0.3634 | Train Acc: 0.8676 | Val Loss: 0.3622 | Val Acc: 0.8678
Epoch 5/10 | Train Loss: 0.3300 | Train Acc: 0.8800 | Val Loss: 0.3221 | Val Acc: 0.8828
Epoch 6/10 | Train Loss: 0.3013 | Train Acc: 0.8912 | Val Loss: 0.3040 | Val Acc: 0.8864
Epoch 7/10 | Train Loss: 0.2800 | Train Acc: 0.8982 | Val Loss: 0.2887 | Val Acc: 0.8961
Epoch 8/10 | Train Loss: 0.2680 | Train Acc: 0.9022 | Val Loss: 0.2744 | Val Acc: 0.8987
Epoch 9/10 | Train Loss: 0.2519 | Train Acc: 0.9082 | Val Loss: 0.2724 | Val Acc: 0.9005
Epoch 10/10 | Train Loss: 0.2383 | Train Acc: 0.9130 | Val Loss: 0.2718 | Val Acc: 0.8980
Model Training Completed
Extracted feature shapes: (48000, 256) (12000, 256) (10000, 256)

Test Accuracy: 0.9052
(10000, 256) (10000, 10)
Training custom MLP on CNN features...
[256, 512, 10] ['relu', 'softmax']
Validation loss improved from inf to 0.3231
Epoch 1/200 | Train Loss: 0.5643 | Train Acc: 0.8896 | Val Loss: 0.3231 | Val Acc: 0.9046 | Patience: 0/5

Early stopping at epoch 24 (best epoch: 19)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.9142
```

The test accuracy achieved in this implementation is **91.42%**.

Implementation with Global average pooling and random weights and double the number of kernels in each layer

```
Initializing Feature Extractor Training...
global random [16, 32, 64, 128, 256] MLP Hidden Layer:  512
256
Epoch 1/10 | Train Loss: 1.7754 | Train Acc: 0.2934 | Val Loss: 1.5312 | Val Acc: 0.3993
Epoch 2/10 | Train Loss: 1.2774 | Train Acc: 0.5025 | Val Loss: 1.0739 | Val Acc: 0.5743
Epoch 3/10 | Train Loss: 1.0460 | Train Acc: 0.6026 | Val Loss: 1.0045 | Val Acc: 0.6141
Epoch 4/10 | Train Loss: 0.9995 | Train Acc: 0.6197 | Val Loss: 0.9527 | Val Acc: 0.6377
Epoch 5/10 | Train Loss: 0.9696 | Train Acc: 0.6303 | Val Loss: 0.9373 | Val Acc: 0.6366
Epoch 6/10 | Train Loss: 0.9524 | Train Acc: 0.6345 | Val Loss: 0.9302 | Val Acc: 0.6428
Epoch 7/10 | Train Loss: 0.9432 | Train Acc: 0.6400 | Val Loss: 0.9329 | Val Acc: 0.6330
Epoch 8/10 | Train Loss: 0.9244 | Train Acc: 0.6451 | Val Loss: 0.9114 | Val Acc: 0.6462
Epoch 9/10 | Train Loss: 0.9147 | Train Acc: 0.6477 | Val Loss: 0.9164 | Val Acc: 0.6384
Epoch 10/10 | Train Loss: 0.9072 | Train Acc: 0.6504 | Val Loss: 0.9189 | Val Acc: 0.6328
Model Training Completed
Extracted feature shapes: (48000, 256) (12000, 256) (10000, 256)

Test Accuracy: 0.6428
(10000, 256) (10000, 10)
Training custom MLP on CNN features...
[256, 512, 10] ['relu', 'softmax']
```

```
Early stopping at epoch 64 (best epoch: 59)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.6742
```

The best test accuracy achieved for global average pooling implementation is **67.42%** which is **the worst among all other implementations like the previous part wherein we have the same number of filters in all layers.** The reason for this accuracy is that in global average pooling the feature returned after each pooling layer for all kernels is just one (the average of all).

Q3. Different weight initialization methods (random, Xavier, He) are tried. **In the earlier sections random weights have already been tried, of which max-pooling with doubling the number of filters in each layer has performed the best.**
Hence , we tried Xavier and He weight initialization with only double the number of filters in each layer

Implementation with Max pooling and ‘he’ weights and double the number of filters in each layer

Z

```
Initializing Feature Extractor Training...
max he [16, 32, 64, 128, 256] MLP Hidden Layer:  512
256
Epoch 1/10 | Train Loss: 0.6156 | Train Acc: 0.7725 | Val Loss: 0.3627 | Val Acc: 0.8662
Epoch 2/10 | Train Loss: 0.3561 | Train Acc: 0.8704 | Val Loss: 0.3283 | Val Acc: 0.8808
Epoch 3/10 | Train Loss: 0.3008 | Train Acc: 0.8892 | Val Loss: 0.3176 | Val Acc: 0.8868
Epoch 4/10 | Train Loss: 0.2671 | Train Acc: 0.9038 | Val Loss: 0.2724 | Val Acc: 0.9013
Epoch 5/10 | Train Loss: 0.2434 | Train Acc: 0.9113 | Val Loss: 0.2836 | Val Acc: 0.8985
Epoch 6/10 | Train Loss: 0.2195 | Train Acc: 0.9193 | Val Loss: 0.2673 | Val Acc: 0.9033
Epoch 7/10 | Train Loss: 0.2053 | Train Acc: 0.9245 | Val Loss: 0.2689 | Val Acc: 0.9040
Epoch 8/10 | Train Loss: 0.1844 | Train Acc: 0.9314 | Val Loss: 0.2624 | Val Acc: 0.9103
Epoch 9/10 | Train Loss: 0.1715 | Train Acc: 0.9371 | Val Loss: 0.2612 | Val Acc: 0.9100
Epoch 10/10 | Train Loss: 0.1569 | Train Acc: 0.9416 | Val Loss: 0.2678 | Val Acc: 0.9059
Model Training Completed
Extracted feature shapes: (48000, 256) (12000, 256) (10000, 256)

Test Accuracy: 0.9130

Training custom MLP on CNN features...
[256, 512, 10] ['relu', 'softmax']
```

```
Early stopping at epoch 11 (best epoch: 6)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.9169
```

The test accuracy achieved in this implementation is **91.69%.**

Implementation with average pooling and ‘he’ weights and double the number of filters in each layer

```
avg he [16, 32, 64, 128, 256] MLP Hidden Layer:  512
256
Epoch 1/10 | Train Loss: 0.7644 | Train Acc: 0.7069 | Val Loss: 0.5335 | Val Acc: 0.7960
Epoch 2/10 | Train Loss: 0.5096 | Train Acc: 0.8084 | Val Loss: 0.4528 | Val Acc: 0.8324
Epoch 3/10 | Train Loss: 0.4249 | Train Acc: 0.8423 | Val Loss: 0.4444 | Val Acc: 0.8321
Epoch 4/10 | Train Loss: 0.3696 | Train Acc: 0.8642 | Val Loss: 0.3596 | Val Acc: 0.8625
Epoch 5/10 | Train Loss: 0.3316 | Train Acc: 0.8791 | Val Loss: 0.3277 | Val Acc: 0.8784
Epoch 6/10 | Train Loss: 0.3063 | Train Acc: 0.8900 | Val Loss: 0.3160 | Val Acc: 0.8867
Epoch 7/10 | Train Loss: 0.2843 | Train Acc: 0.8951 | Val Loss: 0.2914 | Val Acc: 0.8932
Epoch 8/10 | Train Loss: 0.2689 | Train Acc: 0.9018 | Val Loss: 0.2837 | Val Acc: 0.8964
Epoch 9/10 | Train Loss: 0.2521 | Train Acc: 0.9073 | Val Loss: 0.2689 | Val Acc: 0.9022
Epoch 10/10 | Train Loss: 0.2378 | Train Acc: 0.9135 | Val Loss: 0.2559 | Val Acc: 0.9051
Model Training Completed
Extracted feature shapes: (48000, 256) (12000, 256) (10000, 256)

Test Accuracy: 0.9092

Training custom MLP on CNN features...
[256, 512, 10] ['relu', 'softmax']

Early stopping at epoch 23 (best epoch: 18)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.9119
```

The test accuracy achieved in this implementation is **91.19%.**

Implementation with global average pooling and ‘he’ weights and double the number of filters in each layer.

```
Initializing Feature Extractor Training...
global he [16, 32, 64, 128, 256] MLP Hidden Layer:  512
256
Epoch 1/10 | Train Loss: 1.7800 | Train Acc: 0.3023 | Val Loss: 1.4840 | Val Acc: 0.4103
Epoch 2/10 | Train Loss: 1.2209 | Train Acc: 0.5296 | Val Loss: 1.2195 | Val Acc: 0.5107
Epoch 3/10 | Train Loss: 1.0772 | Train Acc: 0.5921 | Val Loss: 1.0285 | Val Acc: 0.6080
Epoch 4/10 | Train Loss: 1.0202 | Train Acc: 0.6150 | Val Loss: 0.9937 | Val Acc: 0.6184
Epoch 5/10 | Train Loss: 0.9854 | Train Acc: 0.6276 | Val Loss: 0.9521 | Val Acc: 0.6399
Epoch 6/10 | Train Loss: 0.9607 | Train Acc: 0.6354 | Val Loss: 0.9764 | Val Acc: 0.6261
Epoch 7/10 | Train Loss: 0.9384 | Train Acc: 0.6446 | Val Loss: 0.9562 | Val Acc: 0.6348
Epoch 8/10 | Train Loss: 0.9198 | Train Acc: 0.6530 | Val Loss: 0.9018 | Val Acc: 0.6504
Epoch 9/10 | Train Loss: 0.9123 | Train Acc: 0.6531 | Val Loss: 0.8931 | Val Acc: 0.6574
Epoch 10/10 | Train Loss: 0.9058 | Train Acc: 0.6548 | Val Loss: 0.8707 | Val Acc: 0.6579
Model Training Completed
Extracted feature shapes: (48000, 256) (12000, 256) (10000, 256)

Test Accuracy: 0.6616

(10000, 256) (10000, 10)
Training custom MLP on CNN features...
[256, 512, 10] ['relu', 'softmax']

Early stopping at epoch 43 (best epoch: 38)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.6761
```

The test accuracy achieved in this implementation is **67.61%**.

Implementation with global average pooling and ‘Xavier’ weights and double the number of filters in each layer.

```
Initializing Feature Extractor Training...
global xavier [16, 32, 64, 128, 256] MLP Hidden Layer:  512
256
Epoch 1/10 | Train Loss: 1.8994 | Train Acc: 0.2646 | Val Loss: 1.7300 | Val Acc: 0.3260
Epoch 2/10 | Train Loss: 1.6528 | Train Acc: 0.3396 | Val Loss: 1.5558 | Val Acc: 0.3611
Epoch 3/10 | Train Loss: 1.4922 | Train Acc: 0.4086 | Val Loss: 1.4004 | Val Acc: 0.4556
Epoch 4/10 | Train Loss: 1.3633 | Train Acc: 0.4780 | Val Loss: 1.2566 | Val Acc: 0.5218
Epoch 5/10 | Train Loss: 1.2031 | Train Acc: 0.5361 | Val Loss: 1.1402 | Val Acc: 0.5497
Epoch 6/10 | Train Loss: 1.0961 | Train Acc: 0.5726 | Val Loss: 1.0400 | Val Acc: 0.5929
Epoch 7/10 | Train Loss: 1.0356 | Train Acc: 0.5982 | Val Loss: 1.0008 | Val Acc: 0.6113
Epoch 8/10 | Train Loss: 0.9943 | Train Acc: 0.6183 | Val Loss: 0.9539 | Val Acc: 0.6354
Epoch 9/10 | Train Loss: 0.9628 | Train Acc: 0.6321 | Val Loss: 0.9665 | Val Acc: 0.6233
Epoch 10/10 | Train Loss: 0.9404 | Train Acc: 0.6385 | Val Loss: 0.9040 | Val Acc: 0.6492
Model Training Completed
Extracted feature shapes: (48000, 256) (12000, 256) (10000, 256)

Test Accuracy: 0.6483

 Early stopping at epoch 50 (best epoch: 45)
 Custom MLP training complete.
 Test Accuracy (Custom MLP on CNN features): 0.6580
```

The test accuracy achieved in this implementation is **65.80%**.

Implementation with max pooling and ‘Xavier’ weights and double the number of filters in each layer.

```
 Initializing Feature Extractor Training...
 max xavier [16, 32, 64, 128, 256] MLP Hidden Layer:  512
 256
 Epoch 1/10 | Train Loss: 0.7307 | Train Acc: 0.7210 | Val Loss: 0.4630 | Val Acc: 0.8280
 Epoch 2/10 | Train Loss: 0.4158 | Train Acc: 0.8497 | Val Loss: 0.3490 | Val Acc: 0.8728
 Epoch 3/10 | Train Loss: 0.3303 | Train Acc: 0.8814 | Val Loss: 0.2917 | Val Acc: 0.8935
 Epoch 4/10 | Train Loss: 0.2893 | Train Acc: 0.8953 | Val Loss: 0.2758 | Val Acc: 0.8988
 Epoch 5/10 | Train Loss: 0.2644 | Train Acc: 0.9049 | Val Loss: 0.2774 | Val Acc: 0.9026
 Epoch 6/10 | Train Loss: 0.2420 | Train Acc: 0.9125 | Val Loss: 0.2837 | Val Acc: 0.8968
 Epoch 7/10 | Train Loss: 0.2244 | Train Acc: 0.9181 | Val Loss: 0.2608 | Val Acc: 0.9098
 Epoch 8/10 | Train Loss: 0.2069 | Train Acc: 0.9242 | Val Loss: 0.2600 | Val Acc: 0.9058
 Epoch 9/10 | Train Loss: 0.1912 | Train Acc: 0.9301 | Val Loss: 0.2623 | Val Acc: 0.9076
 Epoch 10/10 | Train Loss: 0.1782 | Train Acc: 0.9343 | Val Loss: 0.2518 | Val Acc: 0.9132
 Model Training Completed
 Extracted feature shapes: (48000, 256) (12000, 256) (10000, 256)

 Test Accuracy: 0.9142

(10000, 256) (10000, 10)
Training custom MLP on CNN features...
[256, 512, 10] ['relu', 'softmax']

Early stopping at epoch 10 (best epoch: 5)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.9179
```

The test accuracy achieved in this implementation is 91.79%.

Implementation with average pooling and ‘Xavier’ weights and double the number of filters in each layer.

```
Initializing Feature Extractor Training...
avg xavier [16, 32, 64, 128, 256] MLP Hidden Layer:  512
256
Epoch 1/10 | Train Loss: 0.9264 | Train Acc: 0.6521 | Val Loss: 0.6593 | Val Acc: 0.7435
Epoch 2/10 | Train Loss: 0.5964 | Train Acc: 0.7728 | Val Loss: 0.5242 | Val Acc: 0.7926
Epoch 3/10 | Train Loss: 0.5078 | Train Acc: 0.8086 | Val Loss: 0.4659 | Val Acc: 0.8298
Epoch 4/10 | Train Loss: 0.4433 | Train Acc: 0.8372 | Val Loss: 0.4389 | Val Acc: 0.8319
Epoch 5/10 | Train Loss: 0.3977 | Train Acc: 0.8534 | Val Loss: 0.3668 | Val Acc: 0.8673
Epoch 6/10 | Train Loss: 0.3565 | Train Acc: 0.8693 | Val Loss: 0.3577 | Val Acc: 0.8615
Epoch 7/10 | Train Loss: 0.3276 | Train Acc: 0.8813 | Val Loss: 0.3161 | Val Acc: 0.8848
Epoch 8/10 | Train Loss: 0.3011 | Train Acc: 0.8894 | Val Loss: 0.2871 | Val Acc: 0.8987
Epoch 9/10 | Train Loss: 0.2809 | Train Acc: 0.8974 | Val Loss: 0.2941 | Val Acc: 0.8942
Epoch 10/10 | Train Loss: 0.2650 | Train Acc: 0.9030 | Val Loss: 0.2779 | Val Acc: 0.9006
Model Training Completed
Extracted feature shapes: (48000, 256) (12000, 256) (10000, 256)

Test Accuracy: 0.9034

Training custom MLP on CNN features...
[256, 512, 10] ['relu', 'softmax']

Early stopping at epoch 35 (best epoch: 30)
Custom MLP training complete.
Test Accuracy (Custom MLP on CNN features): 0.9100
```

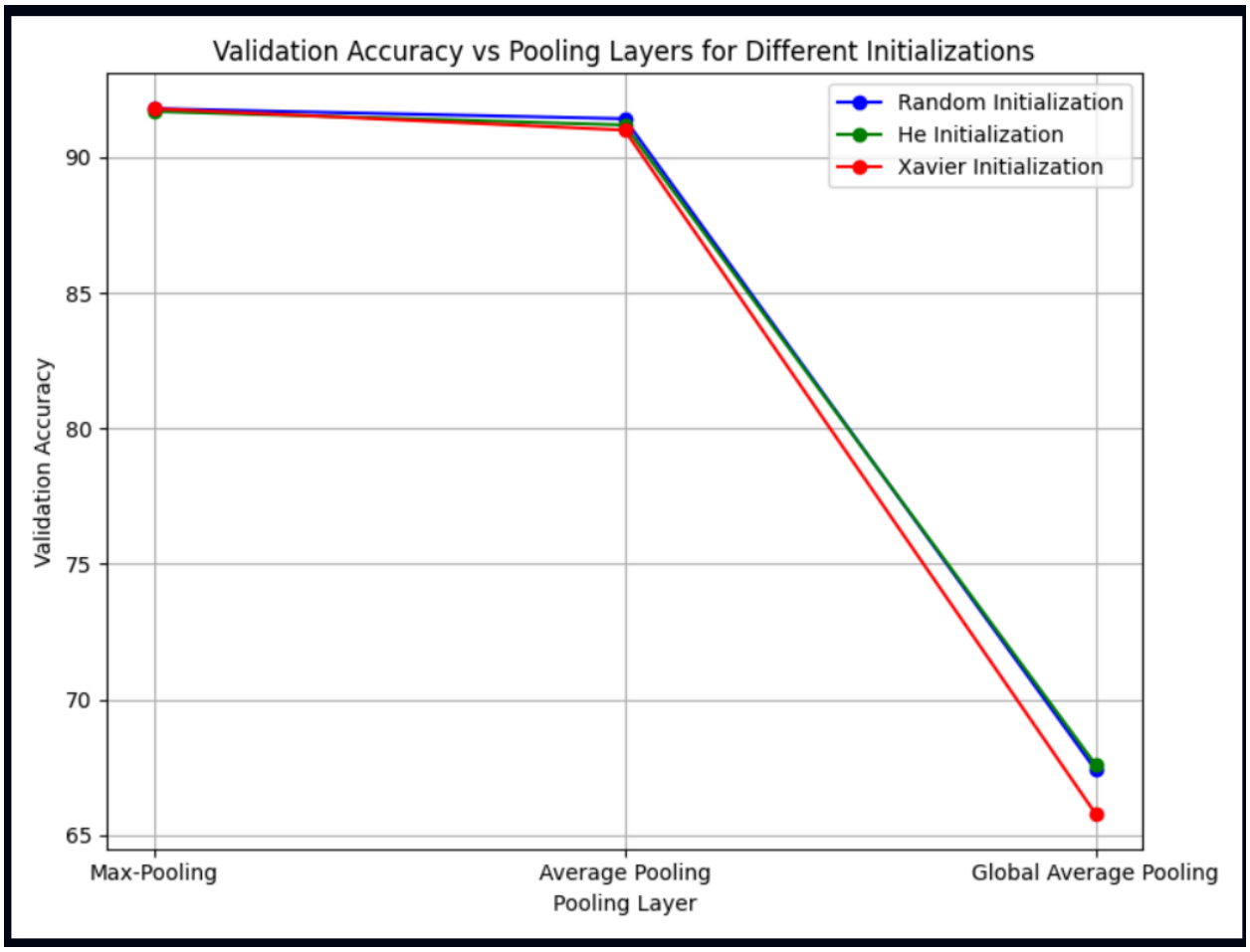
The test accuracy achieved in this implementation is **91.00%.2**

Comparison Table:

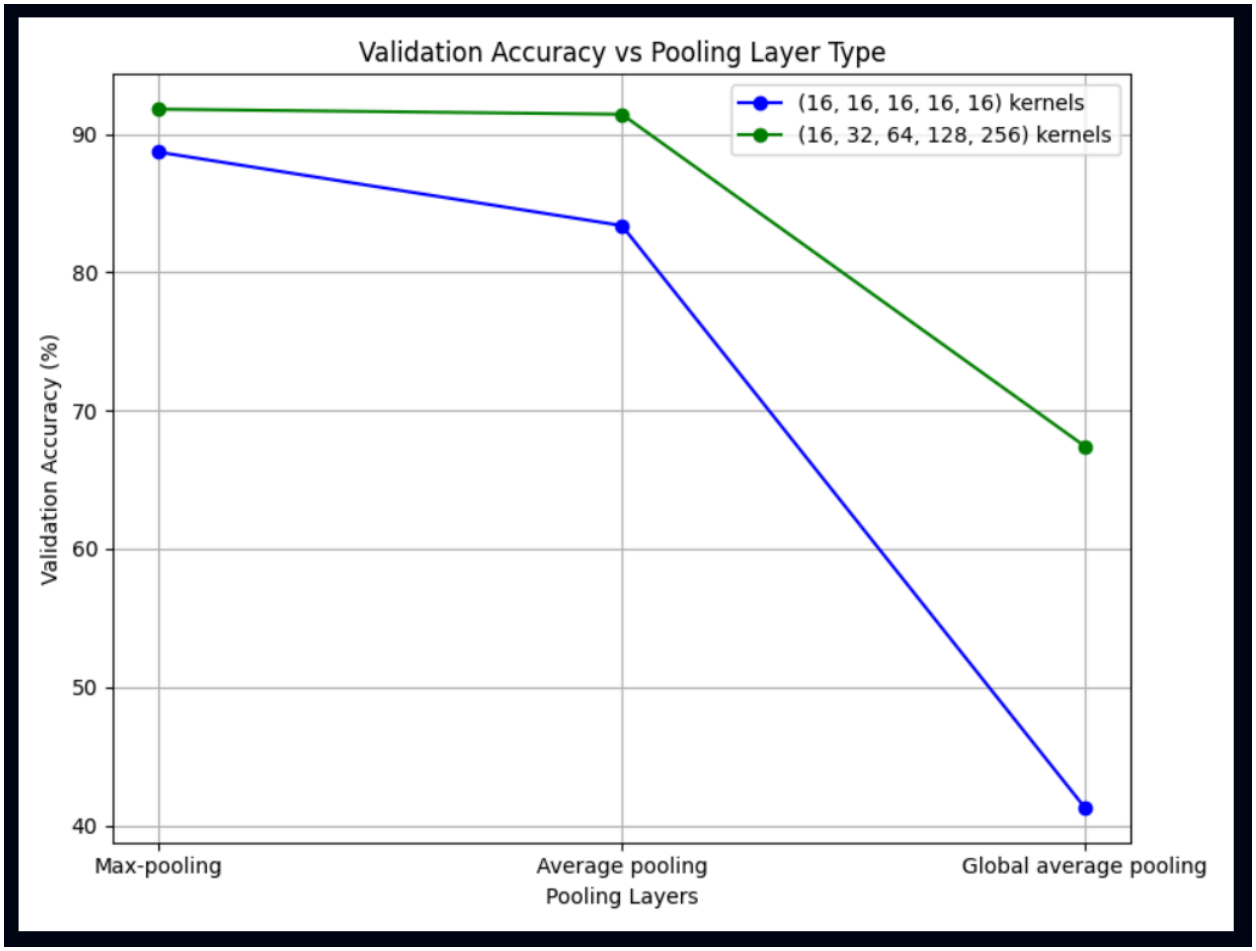
Pooling Method	Conv Layer Dimensions	Weight Initialization method	HIdden Layer size	Test Accuracy (Best)
Max-pooling	16,16,16,16,16	Random	512	88.68%
Average-pooling	16,16,16,16,16	Random	512	83.37%
Global-Average Pooling	16,16,16,16,16	Random	512	41.29%
Max-pooling	16,32,64,128,256	Random	512	91.79%
Average-pooling	16,32,64,128,256	Random	512	91.42%
Global-Average Pooling	16,32,64,128,256	Random	512	67.42%
Max-pooling	16,32,64,128,256	He	512	91.69%
Average-pooling	16,32,64,128,256	He	512	91.19%
Global-Average Pooling	16,32,64,128,256	He	512	67.61%
Max-pooling	16,32,64,128,256	Xavier	512	91.79%
Average-pooling	16,32,64,128,256	Xavier	512	91.00%

Global-Average Pooling	16,32,64,128,256	Xavier	512	65.80%
Max-pooling (3x3 Kernel)	32,64,128,256,512	Random	1024	92.37%
Max-pooling (5x5 Kernel)	32,64,128,256,512	Xavier	1024	92.35%

The validation accuracy vs different pooling layers plot have been shown below for different initialization of weights. It can be seen that **global average pooling has the worst performance and max pooling is better than both of them**.



The validation accuracy vs different pooling layers plot have been shown below for the same and double the number of filters in each subsequent layer of CNN. It can be seen that **double the number of filters in each subsequent layer of CNN performs better than the same number of filters in each layer**. Below given plot is based on random initialization of weight.



Q3. After extracting feature from CNN model , MLP built earlier is used for classification) (15 Marks)

On All the above Implementations we have tested the extracted features of training set on the custom MLP Implementation with the same Hidden Layer as used in the CNN Classifier as shown in the screenshots.

Based on the findings from the comparison table we decided to use the feature extracted from the configuration which gave us the highest accuracy on CNN model which is random initialization, max-pooling with double the number of filters in each layer. This configuration gave the **test accuracy of more than 91%**. Xavier initialization also performed almost equally.

Justification for the better accuracy achieved by above mentioned configuration :

- Max Pooling helps capture the most important features and is generally more beneficial for object recognition, making it a better choice over Average Pooling.

- Random Initialization offers more exploration, and can, in some cases, outperform He or Xavier, especially when the network isn't too deep. These methods are helpful, but not always critical.

- Doubling the number of filters allows the network to progressively learn more complex features, improving performance for many tasks, especially with large datasets and deep networks.

Percentage contribution:

S.No.	Name	Email Id	% Contribution
01	DIVYANSHU (241110023)	divyanshu24@iitk.ac.in	16.66%
02	KHUSWANT KASWAN (241110035)	khushwantk24@iitk.ac.in	16.66%
03	KRISHANU RAY (241110037)	krishanur24@iitk.ac.in	16.66%
04	MAJOR RAJAN KUMAR (241110087)	rajank24@iitk.ac.in	16.66%
05	RISHIT KUMAR (241110056)	rishitk24@iitk.ac.in	16.66%
06	SENTHIL GANESH (241110089)	senthil24@iitk.ac.in	16.66%

Resources Referred

- sentdex Youtube : Neural Networks from Scratch.
- Blog : Initializing neural networks
- Jonas Lalin Blog : Feedforward Neural Networks in Depth
- How Backpropagation Is Able To Reduce the Time Spent on Computing Gradients
- Understanding and implementing Neural Network with SoftMax
- Mohamed Elegandy. Deep learning for vision systems. In Deep Learning for Vision Systems, 1989
- Kaggle Fashion MNIST Code Section
- Youtube:Fashion MNIST - Image Classification CNN - End to End Deep Learning Project
- PyTorch Starter : MLP with CPU
- Justin Charney Blog: MLP from Scratch: Fashion-MNIST
- Medium Blog: Building A Neural Network From Scratch: Solving Fashion MNIST
- Kaggle: Fashion Data by CNN Keras&Pytorch
- Fashion MNIST with Pytorch
- StackOverflow for solving bugs