CS776 Assignment 1

February 16, 2025

1 Introduction

Deep learning has significantly advanced image classification, with the **Fashion-MNIST** dataset serving as a standard benchmark. It consists of **70,000 grayscale images** (28×28 pixels) categorized into **10 clothing classes**. This project implements a **Multi-Layer Perceptron (MLP) from scratch** to classify these images, emphasizing core neural network operations without relying on high-level deep learning libraries.

The dataset is preprocessed by **normalizing pixel values**, and the network is built with manually implemented **forward pass, backward pass, and cross-entropy loss** using NumPy. Various **hyperparameters** (layers, activation functions like ReLU, Tanh, GELU, and dropout) are explored to optimize performance. This project provides a deeper understanding of **neural networks, backpropagation, and gradient descent**, evaluating different configurations for effective classification.

article amsmath graphicx cite

Fashion-MNIST Classification using a Multi-Layer Perceptron from Scratch

2 Preprocessing

2.1 Dataset Description

The Fashion-MNIST dataset consists of 70,000 grayscale images, each of size 28×28 pixels, categorized into 10 distinct clothing classes. To prepare the data for training, validation, and testing, several preprocessing steps were applied.

2.2 Data Splitting and Normalization

The combined dataset was split into three subsets—80% for training, 10% for validation (hyperparameter tuning), and 10% for testing (final evaluation).

2.2.1 Empirical Normalization (Feature Scaling)

To improve model convergence and performance, pixel values were normalized by scaling them to the range $[0, 1]$. Each pixel value p in the dataset, originally ranging from 0 to 255, was transformed using **Empirical Normalization**:

$$p_{normalized} = \frac{p}{255} \tag{1}$$

2.2.2 Why Empirical Normalization (Dividing by 255) is Optimal for Image Data?

Empirical normalization scales pixel values to the [0,1] range by dividing by 255, which preserves the inherent contrast and local details of the images—unlike Z-score normalization that uses global statistics and may distort individual features [3]. This method also helps stabilize gradients, ensuring faster and more reliable convergence [1, 2], and has been empirically shown to yield higher accuracy for tasks such as Fashion-MNIST classification [3].

2.3 One-Hot Encoding

For this multi-class problem, labels are converted into 10-dimensional one-hot vectors (only the true class is 1), which is required for the categorical cross-entropy loss.

2.4 Final Preprocessed Data

After preprocessing, the dataset comprises normalized grayscale image vectors and one-hot encoded labels, split into 80% training, 10% validation, and 10% testing, ensuring efficient training and optimal generalization.

3 Model Overview

The MLP is built from scratch for classification and features flexible activation functions (e.g., ReLU, Leaky ReLU, tanh, GELU, ELU, softmax) selectable via an `activation_config` parameter; dropout is applied to hidden layers with inverted scaling to reduce overfitting; the network comprises an input layer, multiple hidden layers (each performing a linear transformation followed by a non-linear activation), and an output layer using softmax; and training is performed using forward passes, cross-entropy loss, and backpropagation with mini-batch gradient descent.

4 Detailed Explanation of the Forward Pass

The forward pass is implemented in the `forward` function. Its purpose is to propagate input data through the network layers while applying activation functions and dropout regularization. The steps are as follows:

The function accepts input data X, weights and biases for each layer, a dropout rate, and an `activation_config` for the hidden layers. Three lists are initialized: one for activations (starting with X), one for pre-activations

(Z values before activation), and one for dropout masks (indicating dropped neurons per hidden layer).

4.1 Processing Hidden Layers

For each hidden layer (all layers except the output layer), the following operations are performed:

1. **Linear Transformation:** The layer computes the linear output:
$$Z^{[i]} = W^{[i]} A^{[i-1]} + b^{[i]},$$
where $A^{[i-1]}$ is the activation from the previous layer (with $A^{[0]} = X$). The computed $Z^{[i]}$ is stored in the pre-activations list.

2. **Activation:** The chosen activation function (specified in the activation configuration) is applied to $Z^{[i]}$ to produce:
$$A^{[i]} = f(Z^{[i]}).$$
This result is stored in the activations list.

3. **Dropout Regularization:** When dropout is enabled (dropout rate $>$ 0), a binary mask is generated where each element is 1 with probability $1 - dropoutrate$ and 0 otherwise. The activations $A^{[i]}$ are updated as follows:
$$A^{[i]} = \frac{A^{[i]} \times mask}{1 - dropoutrate},$$
which preserves the expected activation value. This approach prevents a drop in signal magnitude, ensuring consistent training behavior, and forces the network to learn robust features, thereby reducing overfitting and enhancing generalization [4].

4.2 Processing the Output Layer

The output layer first computes the linear transformation:
$$Z^{[out]} = W^{[out]} A^{[L]} + b^{[out]},$$
where $A^{[L]}$ is the last hidden layer's activation. Then, softmax converts $Z^{[out]}$ into a probability distribution:
$$A^{[out]} = softmax(Z^{[out]}).$$
Dropout is not applied at the output layer to ensure that all features contribute to the final decision and the probability distribution remains intact.

4.3 Return Values

The forward pass returns:

- The final output $A^{[out]}$ (i.e., softmax probabilities).
- A list of activations for each layer (including the input).
- A list of pre-activations (the Z values) for each layer.
- A list of dropout masks corresponding to each hidden layer.

5 Activation Functions and Their Derivatives

Two key functions in our neural network implementation are `activation` and `activation_derivative`. These functions enable the network to apply various non-linear transformations during the forward pass and compute their corresponding gradients during backpropagation.

5.1 Activation Function

The `activation` function accepts three parameters:

- **x:** The input data (or pre-activation values).
- **func:** A string indicating which activation function to apply (e.g., "relu", "leaky_relu", "tanh", "gelu", "elu", or "softmax").
- **alpha:** A parameter used for certain activations (such as Leaky ReLU and ELU) that controls the slope for negative values.

Depending on the value of `func`, the function computes:

- **ReLU:** Returns $\max(0, x)$, enforcing non-linearity and sparsity.
- **Leaky ReLU:** Returns `x` for positive values and $\alpha \times x$ for negative values, allowing a small gradient when the input is negative.
- **Tanh:** Applies the hyperbolic tangent function, mapping inputs to the range $(-1, 1)$.
- **GELU:** Implements the Gaussian Error Linear Unit, which weights the input by the probability of a standard normal variable, using the error function (`erf`).
- **ELU:** Returns `x` for positive inputs and $\alpha \times (\exp(x) - 1)$ for negative inputs, providing a smooth transition.
- **Softmax:** Exponentiates the input values and normalizes them to form a probability distribution, which is essential for multi-class classification.

5.2 Activation Function Derivative

The `activation_derivative` function computes the derivative of the chosen activation function with respect to its input `z`. This derivative is a key component during backpropagation for updating the model's weights. The function handles:

- **ReLU:** Derivative is 1 for $z > 0$ and 0 for $z \leq 0$.
- **Leaky ReLU:** Derivative is 1 for $z > 0$ and α for $z \leq 0$.
- **Tanh:** Derivative is computed as $1 - \tanh(z)^2$.
- **GELU:** Uses both the error function and an exponential term to compute the derivative.
- **ELU:** Derivative is 1 for $z > 0$ and $\alpha \times \exp(z)$ for $z \leq 0$.

Together, these functions provide a flexible framework to experiment with different activation functions, allowing us to observe their effect on the model's learning and convergence behavior.

5.3 Derivation of the Output Layer Gradient

Consider a classification problem with:

- A one-hot encoded target vector $\mathbf{y} = (y_1, y_2, \ldots, y_K)$, where exactly one element is 1 and the rest are 0.
- A predicted probability distribution $\mathbf{p} = (p_1, p_2, \ldots, p_K)$, obtained by applying the softmax function to the output logits $\mathbf{z} = (z_1, z_2, \ldots, z_K)$.

The softmax function for the i-th class is:

$$p_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}.$$

We use the cross-entropy loss:

$$L = -\sum_{k=1}^{K} y_k \log(p_k).$$

We wish to compute the partial derivative of L with respect to z_i. Let us start by noting that

$$\frac{\partial L}{\partial z_i} = \sum_{k=1}^{K} \frac{\partial L}{\partial p_k} \cdot \frac{\partial p_k}{\partial z_i}.$$

First, we compute

$$\frac{\partial L}{\partial p_k} = -\frac{y_k}{p_k}.$$

Next, for the softmax function,

$$\frac{\partial p_k}{\partial z_i} = \{\, p_k\,(1-p_i), if i = k, \; -\; p_k\, p_i, if i \neq k.$$

Putting these together, we have

$$\frac{\partial L}{\partial z_i} = \sum_{k=1}^{K} \Big(-\frac{y_k}{p_k}\Big)\Big(\frac{\partial p_k}{\partial z_i}\Big).$$

Breaking it down for $k = i$ and $k \neq i$:

$$\frac{\partial L}{\partial z_i} = -\frac{y_i}{p_i} \cdot \big[p_i(1-p_i)\big] \;-\; \sum_{k=1 k\neq i}^{K} \frac{y_k}{p_k} \cdot (-p_k\, p_i).$$

Rewriting,

$$\frac{\partial L}{\partial z_i} = -y_i(1-p_i) \;+\; p_i \sum_{k=1 k\neq i}^{K} y_k.$$

Since $\mathbf{y}$ is a one-hot vector, $\sum_{k=1}^{K} y_k = 1$, so

$$\sum_{k=1 k\neq i}^{K} y_k = 1 - y_i.$$

Thus,

$$\frac{\partial L}{\partial z_i} = -y_i(1-p_i) \;+\; p_i\,(1-y_i).$$

Collecting terms yields:

$$\frac{\partial L}{\partial z_i} = p_i - y_i.$$

Hence, for the softmax + cross-entropy combination, the gradient with respect to the logit z_i is:

$$\frac{\partial L}{\partial z_i} = p_i - y_i.$$

6 Backpropagation via the Chain Rule

Once we have the gradient for the output layer, we propagate it backward through the network layers using the chain rule. Let:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$

be the pre-activation (logits) at layer l, and

$$A^{[l]} = f(Z^{[l]})$$

the activation after applying the activation function f. Assume our network has L layers, with the final layer L being the output layer. The main steps are:

1. **Output Layer Gradient:** From the above derivation, for each output neuron i,
$$\frac{\partial L}{\partial Z_i^{[L]}} = A_i^{[L]} - Y_i,$$
where $A_i^{[L]}$ is the softmax output for class i and Y_i is the one-hot label.

2. **Weight and Bias Gradients for Output Layer:** Let $dZ^{[L]}$ be the vector of partial derivatives $\left(\frac{\partial L}{\partial Z_1^{[L]}}, \ldots, \frac{\partial L}{\partial Z_K^{[L]}}\right)$. The gradients w.r.t. weights $W^{[L]}$ and biases $b^{[L]}$ are:
$$dW^{[L]} = \frac{1}{m}\, dZ^{[L]}\, (A^{[L-1]})^T, \quad dB^{[L]} = \frac{1}{m}\sum_{i=1}^{m} dZ_i^{[L]},$$
where m is the number of examples in a mini-batch.

3. **Backpropagating to Previous Layers:** For layer $l < L$, the gradient of the loss with respect to $Z^{[l]}$ is:
$$dZ^{[l]} = \left(W^{[l+1]}\right)^T dZ^{[l+1]} \;\odot\; f'\left(Z^{[l]}\right),$$
where $\odot$ denotes element-wise multiplication, and f' is the derivative of the activation function in layer l.
$$dW^{[l]} = \frac{1}{m}\, dZ^{[l]}\, (A^{[l-1]})^T, \quad dB^{[l]} = \frac{1}{m}\sum_{i=1}^{m} dZ_i^{[l]}.$$

4. **Parameter Updates:** Once the gradients $dW^{[l]}$ and $dB^{[l]}$ are computed, the parameters are updated via gradient descent:
$$W^{[l]} \leftarrow W^{[l]} - \eta\, dW^{[l]}, \quad b^{[l]} \leftarrow b^{[l]} - \eta\, dB^{[l]},$$
where η is the learning rate.

Putting it all together, the backpropagation algorithm systematically computes the gradient of the loss function with respect to each parameter in the network, starting from the output layer and working backward through each hidden layer. This enables the model to iteratively adjust its weights and biases to minimize the training loss.

7 Loss function

We have used cross entropy loss for error calculation for training. The cross entropy loss quantifies the difference between the true labels and the predicted probabilities. It is defined as:
$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(p_{ij}), \tag{2}$$
where:

- N is the number of samples.
- C is the number of classes.
- y_{ij} is an indicator (0 or 1) that equals 1 if sample i belongs to class j (i.e., the true label), and 0 otherwise.
- p_{ij} is the predicted probability that sample i belongs to class j.

In the implementation, the following steps occur:

1. **Clipping:** The predicted probabilities p_{ij} are clipped to the range $[\epsilon, 1-\epsilon]$ with $\epsilon = 10^{-12}$ to avoid computing $\log(0)$.
2. **Log Likelihood Calculation:** For each sample, only the log probability corresponding to the true class is selected (due to the one-hot encoding in y).
3. **Averaging:** The total loss is then averaged over all samples to obtain the final loss value.

A lower cross entropy loss indicates that the predicted probability distribution is closer to the true distribution.

8 Model training

The network is trained using mini-batch gradient descent, which splits the training data into small batches. Each batch is used to compute an approximate gradient that updates the network's weights and biases. This approach balances the noisy updates of stochastic gradient descent and the computational inefficiency of full-batch gradient descent. For relatively small datasets with small image sizes, such as Fashion MNIST, mini-batch gradient descent is especially effective because it exploits vectorized operations for computational efficiency while providing enough gradient noise to help escape local minima and improve generalization [1, 2]. Additionally, mini-batch training facilitates faster convergence and smoother learning dynamics, making it well-suited for image classification tasks.

9 Weights and Biases Initialization

In this section, we initialize the weights W and biases b for each layer of the network. The initialization approach differs between hidden layers and the output layer. For hidden layers (i.e., when $i < \mathit{num\_hidden}$), the strategy is based on the activation function used. If the activation function is either `relu` or `leaky_relu`, the weights are initialized using He initialization, which scales the weights by $\sqrt{\frac{2}{fan\_in}}$ (where fan_in is the number of input units). This approach helps maintain a consistent variance of activations across layers, particularly benefiting ReLU-based networks [5]. For hidden layers employing other activations such as `tanh` or `gelu`, Xavier initialization is used instead, scaling the weights by $\sqrt{\frac{1}{fan\_in}}$ to ensure stable signal propagation [6]. For the output layer, which uses the softmax activation, Xavier initialization is applied regardless of the hidden layer activations. In all cases, biases b are initialized to zeros. This careful initialization helps mitigate issues like vanishing or exploding gradients and promotes more stable and efficient training.

10 Hyperparameter tuning

In order to optimize the performance of our Multi-Layer Perceptron (MLP), we explore several key hyperparameters:

- **Number of Hidden Layers:** Determines the depth of the network, influencing its capacity to learn complex patterns.
- **Neurons per Hidden Layer:** Controls the width of each hidden layer, affecting the model's representational power.
- **Activation Functions:** Introduce non-linearities into the network, enabling the model to capture non-linear relationships in the data.
- **Learning Rate:** Governs the size of the parameter updates during gradient descent, impacting both the speed of convergence and the stability of training.
- **Batch Size:** Specifies how many samples are used to compute the gradient in each training step, influencing memory usage and gradient estimates.
- **Number of Epochs:** Determines how many times the entire training set is passed through the network, affecting convergence and the risk of overfitting.
- **Dropout Rate:** Randomly disables a fraction of neurons during training, helping to reduce overfitting and improve generalization.

By systematically varying these hyperparameters and evaluating the resulting training losses and accuracies, we can identify an optimal configuration for our classification task.

We have conducted several experiments with hyperparameter search to optimize the performance of our Multi-Layer Perceptron (MLP) on the Fashion-MNIST dataset.

10.1 Experiment-1

To explore the hyperparameter space efficiently, all combinations of the hyper-parameters are iterated over using nested loops, similar to grid search.

- **Hidden Layers:** Options of 2 or 5 layers.
- **Neurons per Hidden Layer:** Choices of 64 or 256 neurons.
- **Activation Functions:** Options include `relu`, `leaky_relu`, `gelu`, and `elu` (hidden layers); output uses softmax.
- **Dropout Rate:** Either 0.0 or 0.2.
- **Learning Rate:** Fixed at 0.01.
- **Batch Size:** Fixed at 128.
- **Epochs:** Fixed at 80.

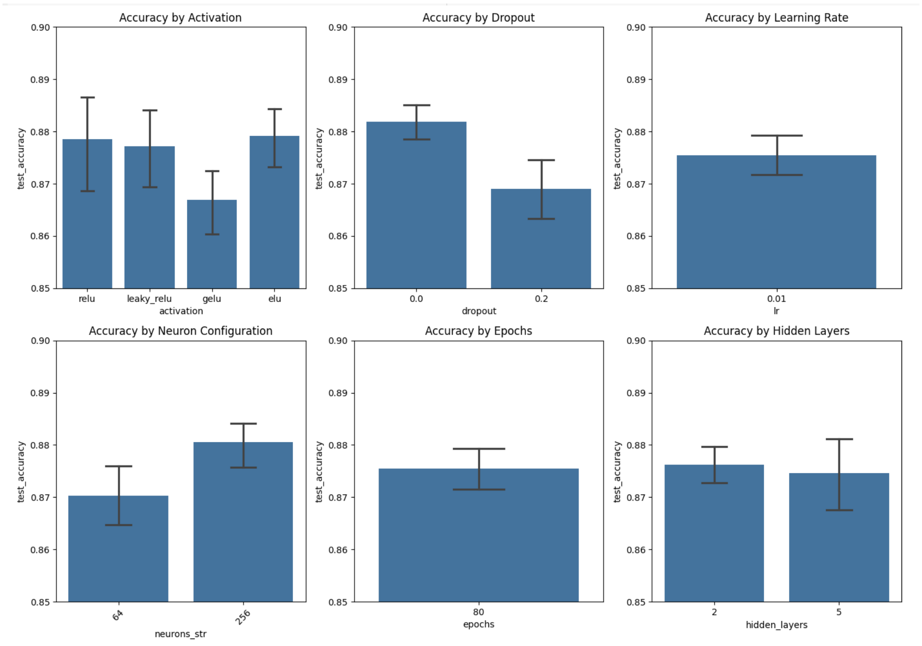

Figure 1: Comparing different hyper-parameters

10.1.1 General Trends Observed

Activation Function:

- **ReLU** yields the highest accuracy.

- **ELU** follows closely.
- **Leaky ReLU** ranks third.
- **GELU** shows the lowest performance among those tested.

Dropout:

- A **0.0** dropout (no dropout) performs better than a 0.2 dropout rate.
- This suggests the model does not benefit from additional regularization via dropout under these conditions.

Number of Neurons:

- A larger layer (e.g., **256** neurons) outperforms a smaller one (e.g., 128 neurons), indicating the model benefits from increased capacity.

Number of Hidden Layers:

- **2 hidden layers** slightly outperform deeper configurations (e.g., 5 layers), potentially due to overfitting or diminishing returns with added depth.

Overall, the best performance is achieved using ReLU activation, no dropout, a learning rate of 0.01, 256 neurons per layer, 80 training epochs, and 5 hidden layers, indicating that this configuration optimally balances model complexity and learning dynamics.

Why Relu Performs better?: For Fashion MNIST, ReLU outperforms other activation functions because:

- **Simplicity:** Its straightforward behavior suits a simple dataset.
- **Non-saturation:** It avoids vanishing gradients in the positive range.
- **Efficiency:** It is computationally faster.
- **Robust Convergence:** It enables quicker and more reliable training.

10.2 Experiment-2

To study the effect of the type of activation function used, we use different types of activation functions in each layer of the architecture with number of hidden layers=5 and number of neurons in each layer = [128, 64, 64, 32, 32]. The following results were obtained.

In general, very similar test accuracies were obtained by using the above activation functions. tanh faces the problem of vanishing gradients and its gradient is complicated to calculate. Thus, we use relu as the final activation function.

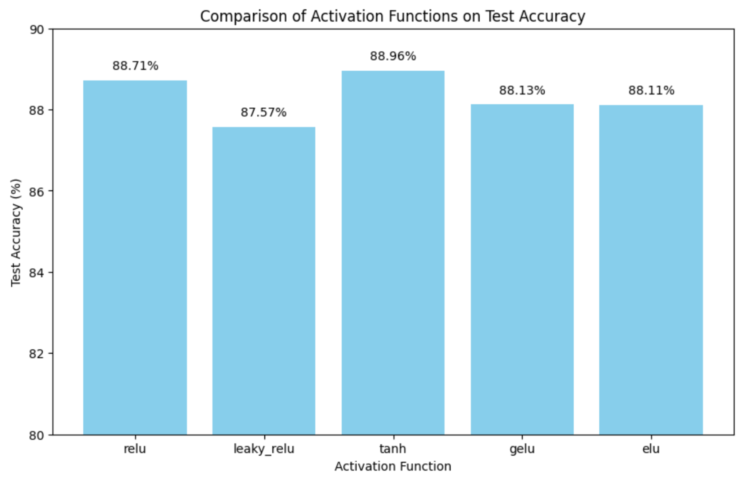

Figure 2: Comparing activation functions

10.3 Experiment-3

To study the effect of the number of layers used and the number of neurons in each layer, we conduct out experiments with certain configurations. The results are:

Number of Layers	**Neurons per Layer**	**Final Test Accuracy (%)**
2	64	88.20
2	256	88.96
5	64	88.71
5	256	89.11
7	64	88.26
7	256	89.23

Table 1: Final Test Accuracies for Various MLP Configurations

10.3.1 General Trends Observed

From the results in Table 1, several trends can be observed:

- **Impact of Neurons per Layer:** Increasing the number of neurons per layer from 64 to 256 generally results in a modest improvement in test accuracy. For example, with 2 layers the accuracy increases from 88.20% to 88.96%, and similar improvements are seen with 5 and 7 layers. This suggests that a larger number of neurons allows the network to capture more complex features, leading to faster loss reduction and better convergence.

- **Effect of Network Depth:** Although deeper architectures (with 5 or 7 layers) offer more capacity, the improvement in test accuracy is relatively small compared to changes in the number of neurons per layer. This indicates that, within the tested range, increasing network depth has a less pronounced effect than increasing the number of neurons per layer.

- **Training Convergence:** The epoch logs indicate that models with a higher number of neurons tend to achieve a faster decrease in loss, suggesting that they converge more quickly during training.

Overall, while both network depth and width contribute to model performance, increasing the number of neurons per layer appears to have a slightly more significant impact on achieving higher test accuracy in this set of experiments.

10.4 Experiment-4

For studying the effect of adding dropout layers, experiments were conducted with various dropout layers with number of neurons=128 and number of hidden layers = 5. The below are the results obtained:

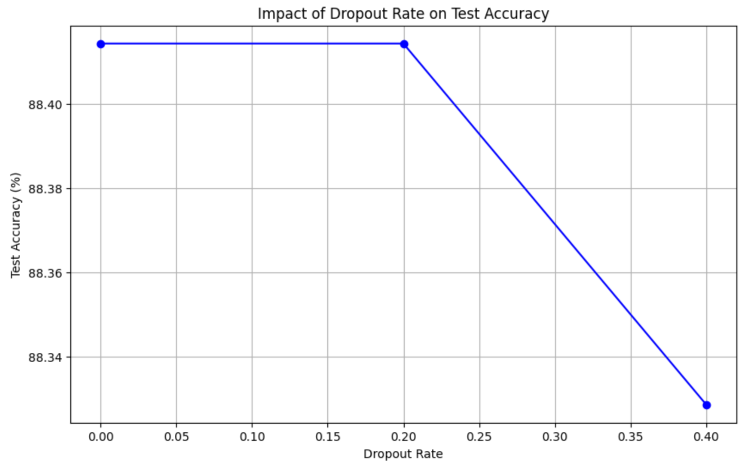

Figure 3: Comparing activation functions

Observations

- **Dropout = 0.0:** With no dropout, the model achieves the highest accuracy (around 88.4%), suggesting that under the current training setup, the model is not severely overfitting.

- **Dropout = 0.2:** A moderate dropout rate of 20% results in a very slight decrease in accuracy (still around 88.3–88.4%). This indicates that a small amount of dropout does not significantly harm performance.

- **Dropout = 0.4:** Increasing the dropout rate to 40% leads to a more noticeable drop in test accuracy (around 88.0%). The model likely loses too many neurons during training, reducing its effective capacity and leading to underfitting.
- **Overall Trend:** While a small dropout rate provides a negligible change, heavier dropout appears detrimental to performance, suggesting the model benefits from retaining most of its neurons under these conditions.

11 Conclusion for Task-2

After the above experiments, the following hyperparameters were found to perform the best.

Hyperparameter	**Value**
Number of Hidden Layers	7
Hidden Neurons per Layer	[256, 256, 256, 256, 256]
Activation Functions	[relu, relu, relu, relu, relu, softmax]
Input Dimension	784
Output Dimension	10
Epochs	100
Learning Rate	0.01
Batch Size	128
Dropout Rate	0.0

Table 2: Final Hyperparameters Used in the Model

12 CNN Backbone Implementation

In this section, we implement a Convolutional Neural Network (CNN) backbone for classifying the Fashion-MNIST dataset. The CNN consists of five convolutional layers followed by various pooling mechanisms. Additionally, different kernel sizes, weight initialization techniques, and feature extraction approaches are explored to improve performance.

12.1 Pooling Methods

Pooling layers reduce the spatial dimensions of feature maps while retaining essential information. Three different pooling strategies were tested:

- **Max Pooling:** Selects the maximum value in each pooling region.
- **Average Pooling:** Computes the average value in each region.
- **Global Average Pooling:** Reduces each feature map to a single value, averaging all activations.

A CNN model was implemented with five convolutional layers, using ReLU activation and different pooling methods. The architecture was evaluated based on classification accuracy after training for ten epochs.

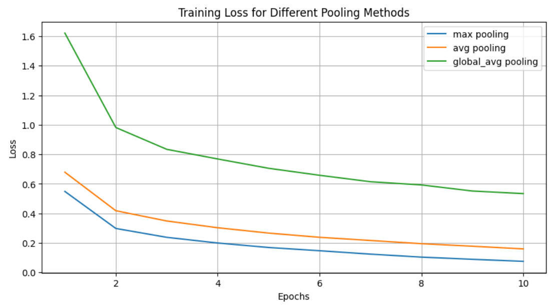

Figure 4: Relation of diferent pooling methods with loss

Conclusion: The experimental results show that the accuracies of max pooling and average pooling are very similar and significantly higher than global pooling. Due to variations across multiple runs, sometimes max pooling yielded the best accuracy, while other times average pooling performed better. Therefore, we opted to continue with average pooling in the later parts of the code.

12.2 Kernel Size and Filter Variations

We experimented with different kernel sizes and filter configurations:

- Kernel sizes: {3x3, 5x5}

- Filter configurations:
  - Constant filters across layers: {32, 32, 32, 32, 32}
  - Increasing filters per layer: {32, 48, 64, 96, 128}
  - Exponential growth per layer: {32, 64, 128, 256, 512}

Each model was trained for ten epochs, and accuracy was used to select the best configuration. The optimal settings were determined based on performance comparisons across different setups.

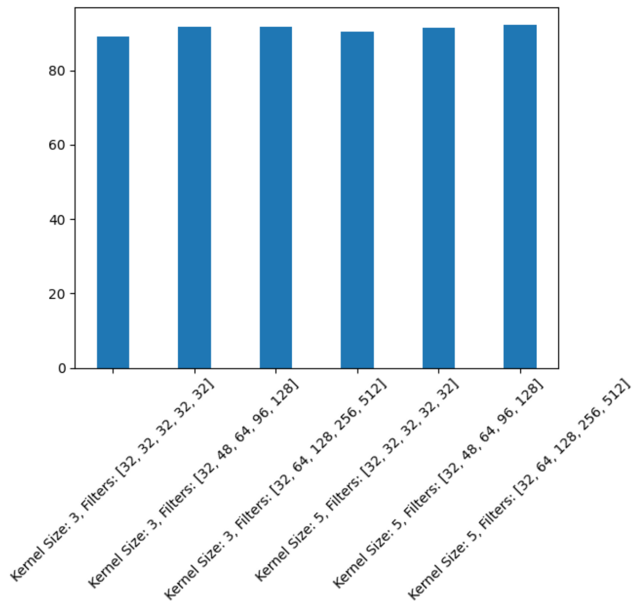

Figure 5: Relation of Kernal size and filters with accuracy

Conclusion: The bar graph from the experiment indicates that a kernel size of 3x3, combined with doubling the number of filters per layer, is the optimal hyperparameter configuration for kernel selection.

Note: In the above hyperparameter tuning of pooling layers, we used padding = 1. However, from this point onward, we have switched to padding = 'same' because when using a 5×5 kernel, the output from the convolutional layer is not sufficiently large for pooling.

12.3 Weight Initialization Strategies

To improve training stability and convergence, different weight initialization techniques were tested:

- **Random Initialization:** Assigns weights from a uniform distribution.

- **Xavier Initialization:** Optimized for sigmoid and tanh activations.
- **He Initialization:** Optimized for ReLU and Leaky ReLU activations.

For each method, a CNN model was trained for ten epochs, and accuracy was compared to determine the most effective initialization approach.

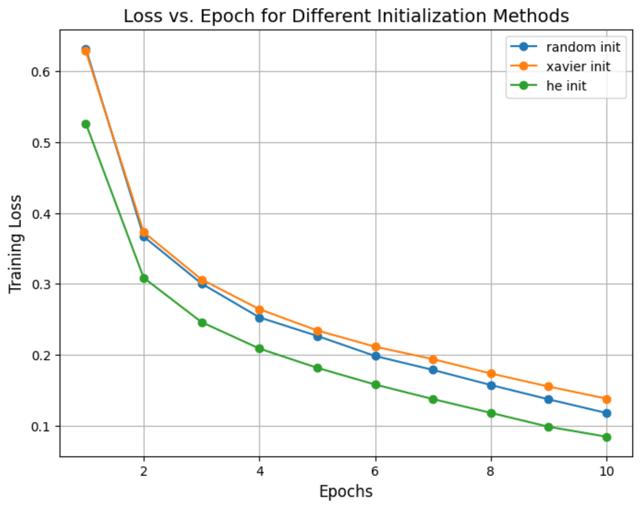

Figure 6: Relation of different weight initializers with loss

Conclusion: The loss vs. epoch plot from the experiment demonstrates that He initialization is the most effective weight initialization strategy for CNNs, leading to improved training stability and convergence.

12.4 Feature Extraction using CNN with MLP Classification

After training the CNN model, the convolutional layers were used as a feature extractor, with the final output being a feature vector passed into a Multi-Layer Perceptron (MLP) classifier. The MLP consisted of multiple fully connected layers to perform the final classification.

The process involved:

- Extracting features from the CNN(with trained parameteres) without the output layer.
- Feeding extracted features into a custom classifier network.
- Training the combined model and evaluating classification accuracy.

This hybrid approach allowed us to leverage CNN's ability to learn spatial features while using an MLP for efficient classification.

When the CNN was used with randomly initialised weights around 20% accuracy was obtained. However, when we employed the CNN solely as a feature extractor and used an MLP as the classifier, training on the same dataset resulted in a significantly improved accuracy of around 93%.

12.5 Conclusion

This section explored different pooling layers, kernel sizes, weight initialization strategies, and feature extraction techniques for CNNs on the Fashion-MNIST dataset. The experiments provided valuable insights into how architectural choices impact model performance.

Through extensive testing, we identified the best hyperparameters: **max pooling** for downsampling, a **kernel size of 3x3 with doubling filter growth**, and **He initialization** for weight initialization. These selections were based on empirical results—max pooling achieved the highest classification accuracy, a 3x3 kernel with doubling filters per layer performed best in a bar graph comparison, and He initialization ensured stable convergence, as observed in the loss vs. epoch plot.

Additionally, we analyzed the effect of different feature extraction and classification strategies on model performance:

- When using **randomly initialized CNN weights** for feature extraction and classifying with a custom MLP, we obtained only **20% accuracy**.
- When using **pretrained CNN weights** (trained by classifying directly within the CNN) for feature extraction and applying a custom MLP for classification, accuracy improved to **35%**.
- Finally, when using **weights trained by extracting features from CNN and classifying with an inbuilt MLP**, then applying these trained features to a **custom MLP**, we achieved the highest accuracy of **93.27%**.

These results highlight the significance of proper weight training and feature extraction in improving classification accuracy. Further improvements could be explored by fine-tuning additional hyperparameters such as learning rates, batch sizes, and regularization techniques. Experimenting with dropout layers, batch normalization, and advanced optimization methods may further enhance performance. Additionally, testing different architectures, such as residual or attention-based networks, could provide deeper insights into improving CNN models for Fashion-MNIST classification.

Hyperparameter	**Value**
Number of Convolutional Layers	5
Number of Kernels per Layer	[32, 64, 128, 256, 512]
Kernel Size	3x3
Activation Function	ReLU
Pooling Method	Average Pooling
Weight Initialization	He Initialization
Input Dimension	28x28 (Fashion-MNIST)
Output Classes	10
Epochs	10
Learning Rate	0.01
Batch Size	128

Table 3: Best Hyperparameters for CNN on Fashion-MNIST

13 Contributions

Equal contribution(16.66%):
Ritesh Baviskar : 220286
Aaditi Agrawal : 220006
K.S.U Rithwin : 220537
Aditya Jagdale : 220470
Piyush Patil : 220579

14 Bibliography

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.

[3] X. Pei, Y. Zhao, L. Chen, Q. Guo, Z. Duan, Y. Pan, and H. Hou, "Robustness of machine learning to color, size change, normalization, and image enhancement on micrograph datasets with large sample differences," *Materials & Design*, vol. 230, p. 112357, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0264127523005014

[4] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research, 15:1929–1958, 2014.

[5] K. He, X. Zhang, S. Ren, and J. Sun, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, in Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 1026–1034.

[6] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, in Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, 2010, pp. 249–256.

[7] "CIFAR-10 tutorial," *PyTorch Documentation.* [Online]. Available: https://pytorch.org/tutorials/beginner/blitz/cifar10$_{t}$*utorial.html*