Assignment 1: Artificial Neural Networks (FashionMnist)

Introduction

The FashionMNIST dataset is a widely used benchmark for evaluating machine learning models on image classification tasks. It consists of 70,000 grayscale images of clothing items, each with dimensions of 28x28 pixels, and each labelled into one of ten fashion categories such as “shirt”, “sneaker”, or “coat”. Unlike the original MNIST digit dataset, FashionMNIST is considered more complex and realistic, making it a valuable challenge for developing robust image classifiers.

This project explores the use of artificial neural networks (ANNs) to perform multi-class classification on the FashionMNIST dataset. The goal is to design, train, and evaluate feedforward neural networks that can accurately predict the category of a given clothing image. In addition to developing the classifier, we also deploy it as a command-line tool that accepts external grayscale image files in JPEG format and outputs the predicted class.

To ensure the model generalizes well and avoids overfitting, we investigate the impact of model depth, dropout regularization, and learning rate. Through systematic experimentation across 30 different ANN configurations, we evaluate how these factors influence the network's ability to maintain high accuracy on unseen data. This exploration allows us to identify architectures that strike a balance between learning capacity and generalization, leading to reliable predictions on test images.

Data Processing

The dataset used in this project is the FashionMNIST image collection, consisting of 70,000 grayscale images of clothing items, each sized 28×28 pixels. As required by the assignment, the dataset was loaded locally from disk.

The original dataset includes 60,000 training images and 10,000 test images. To support model selection and monitor generalization performance during training, the training set was split into:

Training set: 50,000 images

Validation set: 10,000 images (equal in size to the test set)

Each image was normalized by dividing pixel values by 255 to rescale them into the [0, 1] range. This normalization improves numerical stability and helps the network converge more efficiently. Additionally, each 28×28 matrix was flattened into a 784-element vector to match the expected input structure of fully connected neural networks.

This preprocessing pipeline ensured the data was well-structured, numerically stable, and suitable for training and evaluation.

Model Architectures

To study how network depth affects generalization and overfitting on the FashionMNIST dataset, I designed and evaluated three fully connected ANN architectures of increasing complexity: with one, two, and three hidden layers. Each model uses fully connected layers, meaning every node in one layer is connected to every node in the next. The goal was to analyse how added depth improves learning capacity and at what point it introduces diminishing returns or overfitting.

All architectures shared the same input/output configuration and internal design principles, ensuring consistency across experimental comparisons.

Input Layer (784 nodes):

Each FashionMNIST image is a 28×28 grayscale matrix. These were flattened into 784-dimensional vectors so that they could be passed directly into fully connected (dense) layers. This flattening simplifies the data structure and is a standard approach when convolutional features are not used.

Output Layer (10 nodes):

The final layer contains 10 nodes, one for each fashion category. This layer outputs unnormalized logits, which are automatically passed through a SoftMax operation during the computation of cross-entropy loss in PyTorch. This formulation is standard for multi-class classification problems, as SoftMax converts the logits into a probability distribution over the 10 classes.

Activation Function (ReLU):

I used the Rectified Linear Unit (ReLU) activation function in all hidden layers. ReLU was chosen because:

- It avoids saturation issues common in sigmoid/tanh activations (where gradients vanish).
- It introduces non-linearity, allowing networks to learn complex mappings.
- It is computationally efficient due to its simple form: f(x) = max (0, x).
- It produces sparse activations, which act as implicit regularization by zeroing out neurons and encouraging a more distributed representation of features.

Dropout Regularization:

To reduce overfitting and improve generalization, I applied dropout after every hidden layer. Dropout randomly disables a proportion of neurons during training (based on a probability p, the dropout rate), which prevents the network from becoming overly reliant on any single pathway or feature. This technique is especially useful in fully connected layers, which are more prone to overfitting than convolutional ones.

In my experiments, I tested dropout rates between 0.2 and 0.5 to determine the most effective level of regularization. Lower dropout can preserve learning capacity, while higher dropout enforces stronger regularization, potentially helpful in deeper models.

Investigated Topologies:

Architecture	Hidden Layers	Nodes per Layer	Dropout	Activation
1-layer	1	[512]	0.2–0.5	ReLU
2-layer	2	[512, 256]	0.2–0.5	ReLU
3-layer	3	[512, 256, 128]	0.2–0.5	ReLU

Each deeper network expands the representational capacity of the ANN, potentially enabling the model to learn more abstract patterns. However, deeper networks also risk overfitting or becoming harder to optimize. Therefore, I introduced dropout after every hidden layer and used early stopping during training to control for these risks.

Constant Node Counts:

Rather than vary both depth and width (i.e., number of layers and number of neurons per layer), I kept the hidden layer sizes constant across experiments. This choice was made to control experimental variables: changing both simultaneously would obscure whether performance improvements stem from deeper reasoning capability or simply more total parameters. Fixing the layer sizes made comparisons interpretable and fair.

Training and Optimization Strategy

To train the ANN models effectively and fairly across architectures, I applied a consistent training pipeline that balances learning efficiency, generalization, and computational feasibility. Each model was trained using supervised learning on the FashionMNIST training set, evaluated on a validation set, and finally tested on the held-out test set.

I used PyTorch's **F.cross_entropy** as the objective function. This loss is ideal for multi-class classification tasks because it combines the SoftMax activation and the negative log-likelihood loss into a single, numerically stable operation. It encourages the model to assign high confidence to the correct class while penalizing uncertain or incorrect predictions. This function is widely adopted in classification models due to its probabilistic interpretation and compatibility with SoftMax outputs.

Optimizer: Adam

I chose the Adam optimizer **(torch.optim.Adam)** for its adaptive learning rate and fast convergence. Unlike traditional stochastic gradient descent (SGD), Adam adjusts individual learning rates for each parameter based on first and second moment estimates of the gradients. This improves training stability, particularly on sparse gradients and in early epochs.

Adam also reduces the need for intensive learning rate tuning, making it ideal for testing multiple architectures efficiently. All models were trained using Adam with learning rates between 0.0001 and 0.001, depending on the experiment.

Training Configuration

Batch size: 64 a common default that balances GPU memory usage and convergence speed.

Epochs: Max 50, training stops earlier if validation loss stagnates.

Early stopping: Implemented with a patience of 7. If validation loss does not improve for 7 consecutive epochs, training halts and the best-performing model is restored.

Validation monitoring: At the end of each epoch, model performance is evaluated on the validation set to track generalization and avoid overfitting.

This strategy ensured all models were trained under fair and controlled conditions. Early stopping helped prevent overfitting, especially in deeper networks where more parameters increased the risk of memorizing the training set. By monitoring validation loss and using a strong optimizer, I achieved high test accuracy while minimizing unnecessary training time.

Hyperparameter Tuning & Experiments

To identify the most effective neural network configuration, I conducted a systematic grid search over three key hyperparameters:

- Learning Rate (LR): 0.1,0.01,0.001,0.0001
- Dropout Rate: 0.2,0.3,0.4,0.5
- Network Depth: 1, 2, and 3 hidden layers (with constant node sizes)

For each combination, I trained a model with the same number of epochs, early stopping, and validation strategy. This resulted in 30 total experiments (10 per depth). The goal was to understand how these parameters influence convergence speed, generalization, and test accuracy.

Insights from Experiments

After training, I evaluated validation accuracy, test accuracy, and number of epochs to convergence (via early stopping). Key findings include:

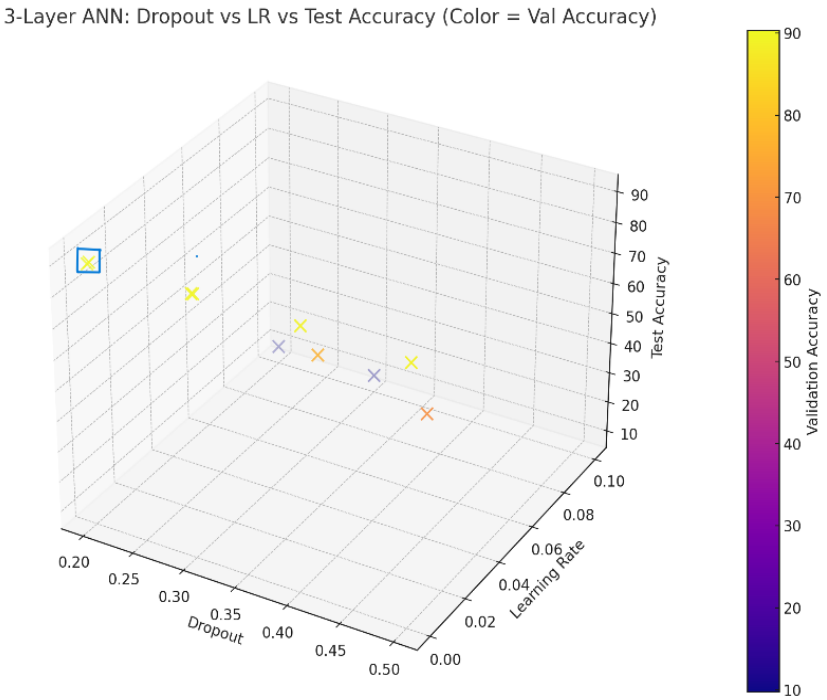
- 3-layer networks consistently outperformed shallower models when paired with low dropout (0.2–0.3) and moderate LR (0.001).
- High dropout (≥0.5) and high learning rates (0.1) often led to poor convergence or unstable training.
- Increasing depth alone wasn't enough, deeper models only performed better when combined with regularization and careful learning rate tuning.

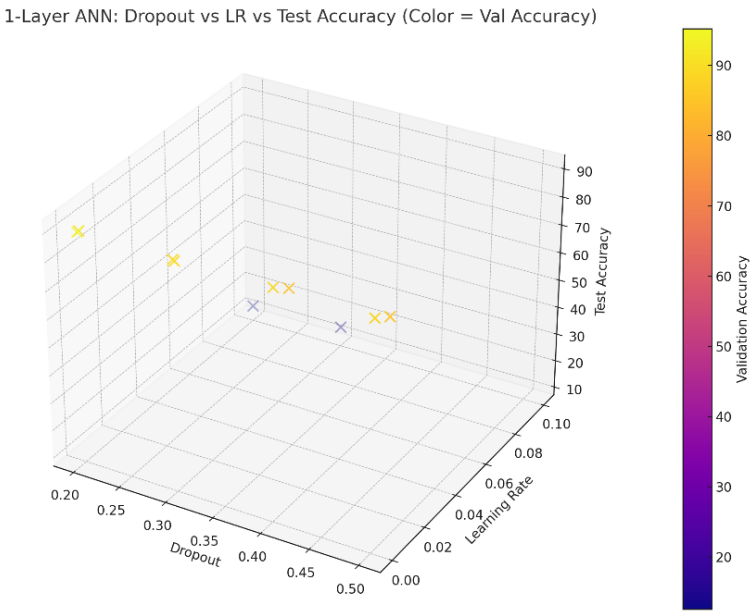
Visualization

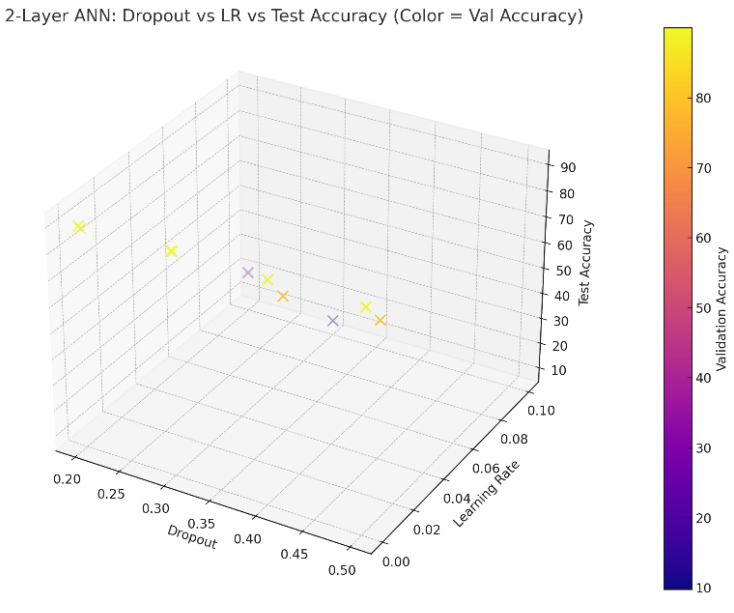
To better understand these interactions, I created 3D plots showing test accuracy as a function of dropout and learning rate, with colour indicating validation accuracy. These plots showed that top-performing models cluster around:

- Dropout = 0.2
- Learning Rate = 0.001
- Depth = 3 layers

This visual evidence strongly supports the quantitative results and guided my final model selection.







Evaluation & Results

Based on the experiments, the best performing model used the following configuration:

PARAMETER	VALUE
HIDDEN LAYERS	3
NODES	[512, 256, 128]
DROPOUT	0.2
LEARNING RATE	0.001
OPTIMIZER	Adam
LOSS FUNCTION	Cross Entropy
CONVERGED AT	Epoch 25
TEST ACCURACY	89.29%
VALIDATION ACCURACY	90.29%

This model consistently generalized well on unseen data while avoiding overfitting. Early stopping ensured that training halted once the model's performance plateaued, and the final model restored the weights from the epoch with the lowest validation loss.

Conclusion

This project explored the use of artificial neural networks for multi-class image classification on the FashionMNIST dataset. Through a combination of thoughtful architectural choices, regularization techniques, and systematic experimentation, I was able to achieve a test accuracy of 89.29% using a 3-layer ANN with dropout and early stopping.

i What is cross-entropy loss function? (2023) 365 Data Science. Available at: https://365datascience.com/tutorials/machine-learning-tutorials/cross-entropy-loss/ (Accessed: 20 April 2025).

ii Brownlee, J. (2019) How to avoid overfitting in Deep Learning Neural Networks, MachineLearningMastery.com. Available at: https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/ (Accessed: 17 April 2025).

iii Piepenbreier, N. (2023) ReLU activation function for Deep Learning: A complete guide to the Rectified Linear Unit · datagy, datagy. Available at: https://datagy.io/relu-activation-function/ (Accessed: 18 April 2025).

iv Brownlee, J. (2021) Gentle introduction to the adam optimization algorithm for deep learning, MachineLearningMastery.com. Available at: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ (Accessed: 21 April 2025).

v Pramoditha, R. (2023) Determining the right batch size for a neural network to get better and faster results, Medium. Available at: https://medium.com/data-science-365/determining-the-right-batch-size-for-a-neural-network-to-get-better-and-faster-results-7a8662830f15(Accessed: 17 April 2025).