Comparative Study of Neural Networks for Fashion-MNIST Image Classification

1. Introduction

This project presents a comparative study of six neural network architectures — Multilayer Perceptron (MLP), Convolutional Neural Network (CNN), LeNet, AlexNet, Recurrent Neural Network (RNN), and Long Short-Term Memory (LSTM) — for solving the image classification problem on the Fashion-MNIST dataset.

Fashion MNIST Dataset: Its a widely used benchmark dataset for image classification, developed as a more challenging and realistic alternative to the classic MNIST digit dataset. It consists of 70,000 grayscale images of fashion products, each sized 28×28 pixels, categorized into 10 classes such as T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot. With 60,000 training images and 10,000 test images, the dataset is designed to evaluate machine learning and deep learning models in recognizing real-world clothing items. Its compact size, accessibility, and relevance to modern computer vision tasks make it a preferred choice for testing convolutional neural networks and other image classification architectures.

The objective is to analyze how different architectures perform on the same dataset and evaluate the impact of hyperparameter tuning. Each model is implemented in two versions:

- **Baseline**: with default or minimal configuration
- **Tuned**: with optimized hyperparameters such as number of layers, filters, dropout rates, learning rates, and optimizers.

---

2. Model Architectures and Experiments

2.1 Multilayer Perceptron (MLP)

Description:
Multilayer Perceptron (MLP) is a class of artificial neural networks consisting of multiple layers of neurons, including one or more hidden layers. Each neuron in a layer is fully connected to the neurons in the next layer, and nonlinear activation functions are used to model complex patterns. MLPs are widely used for classification and regression tasks in various domains.

Baseline Parameters:
- Hidden layers: 2 hidden layers - [128, 64]
- Activation: ReLU
- Learning rate: 0.01
- Batch size: 64

- Epochs: 10

Test Loss: 0.6436
Test Accuracy: 0.7689
Precision: 0.7673
Recall: 0.7689
F1 Score: 0.7672

Tuned Parameters:

- Layers: Three hidden layers with increased neurons - [256, 128, 64]
- Activation: ReLU
- Optimizer: Adam
- Learning Rate: 0.005
- Epochs: 20

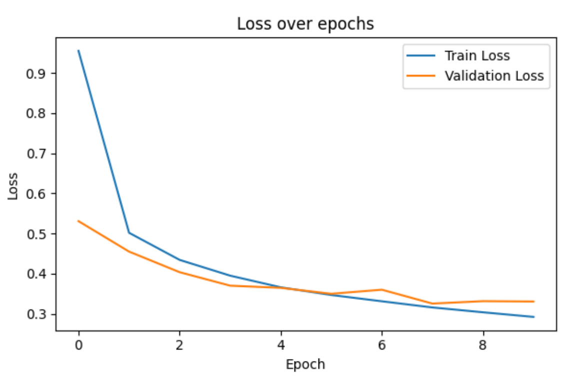
  Test Loss: 0.3403
  Test Accuracy: 0.8791
  Precision: 0.8801
  Recall: 0.8791
  F1 Score: 0.8784

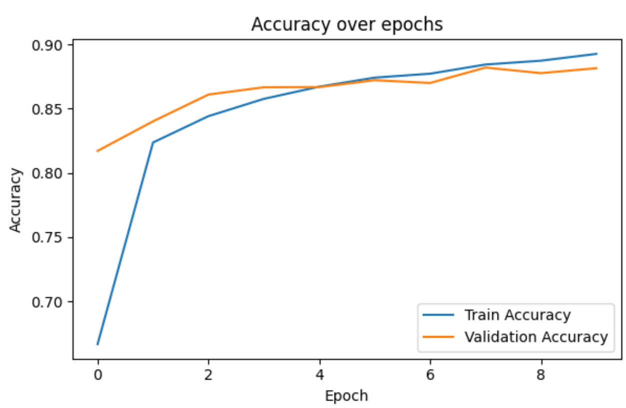
MLP: Accuracy increased due to added hidden layers and extended training time, enabling better feature learning rate

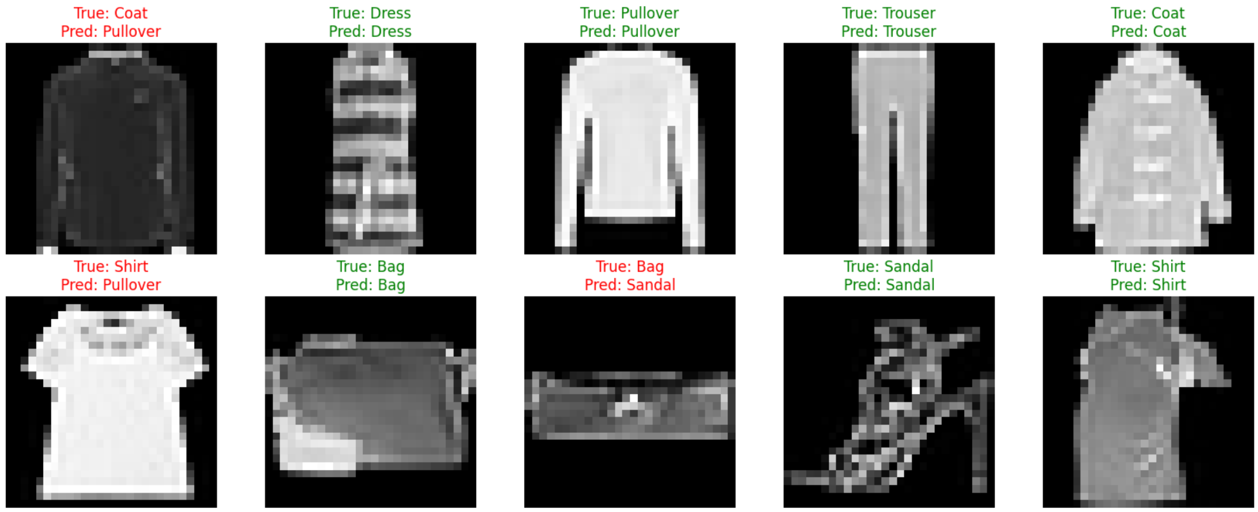
Code Snippet:

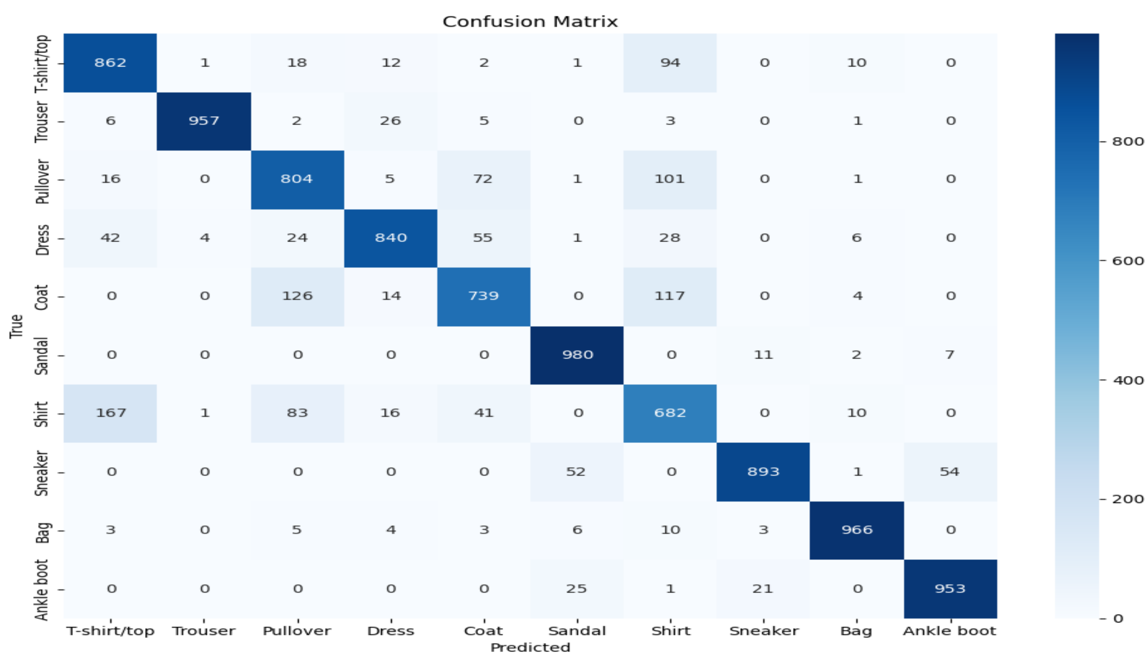
```
# MLP Class from scratch
class MLP:
    def __init__(self, input_size, hidden_sizes, output_size, activation='relu', weight_init_scale=0.01):
        # MODIFIED: Added weight_init_scale parameter to control weight initialization
        self.input_size = input_size
        self.hidden_sizes = hidden_sizes
        self.output_size = output_size
        # Initialize weights and biases
        self.parameters = {}
        # Input layer to first hidden layer
        self.parameters['W1'] = np.random.randn(input_size, hidden_sizes[0]) * weight_init_scale
        self.parameters['b1'] = np.zeros((1, hidden_sizes[0]))
        # Hidden layers
        for i in range(1, len(hidden_sizes)):
            self.parameters[f'W{i+1}'] = np.random.randn(hidden_sizes[i-1], hidden_sizes[i]) * weight_init_scale
            self.parameters[f'b{i+1}'] = np.zeros((1, hidden_sizes[i]))
```

```
        # Last hidden layer to output layer
        self.parameters[f'W{len(hidden_sizes)+1}'] = np.random.randn(hidden_sizes[-1], output_size) * weight_init_scale
        self.parameters[f'b{len(hidden_sizes)+1}'] = np.zeros((1, output_size))
        # Set activation function
        if activation == 'relu':
            self.activation = relu
            self.activation_derivative = relu_derivative
        else:  # Default to sigmoid
            self.activation = sigmoid
            self.activation_derivative = sigmoid_derivative
        # Store activations and pre-activations for backpropagation
        self.activations = {}
        self.pre_activations = {}
    def forward(self, X):
        self.activations['A0'] = X
        # Input layer to first hidden layer
        self.pre_activations['Z1'] = np.dot(X, self.parameters['W1']) + self.parameters['b1']
        self.activations['A1'] = self.activation(self.pre_activations['Z1'])
        # Hidden layers
        for i in range(2, len(self.hidden_sizes) + 2):
            self.pre_activations[f'Z{i}'] = np.dot(self.activations[f'A{i-1}'], self.parameters[f'W{i}']) + self.parameters[f'b{i}']
            if i == len(self.hidden_sizes) + 1:  # Output layer
                self.activations[f'A{i}'] = softmax(self.pre_activations[f'Z{i}'])
            else:  # Hidden layers
                self.activations[f'A{i}'] = self.activation(self.pre_activations[f'Z{i}'])
        return self.activations[f'A{len(self.hidden_sizes) + 1}']
```









2.2 AlexNet

Description:
Deeper CNN architecture that learns hierarchical image features. Even a simplified version shows strong performance on image tasks.

Baseline:

- **Input size**: 32×32×3 (normalized and resized from 28×28)

- **Model layers**:
- FC1: 128 neurons with ReLU
- FC2: 10 neurons with Softmax
- **Weight initialization**: Small random values × 0.01
- **Learning rate**: 0.01
- **Epochs**: 5
- **Optimizer**: Manual Stochastic Gradient Descent (batch size = 1)
- **Loss function**: Cross-entropy

Accuracy: 85.39%
Precision: 86.59%
Recall: 85.33%
F1-score: 85.44%

Tuned:

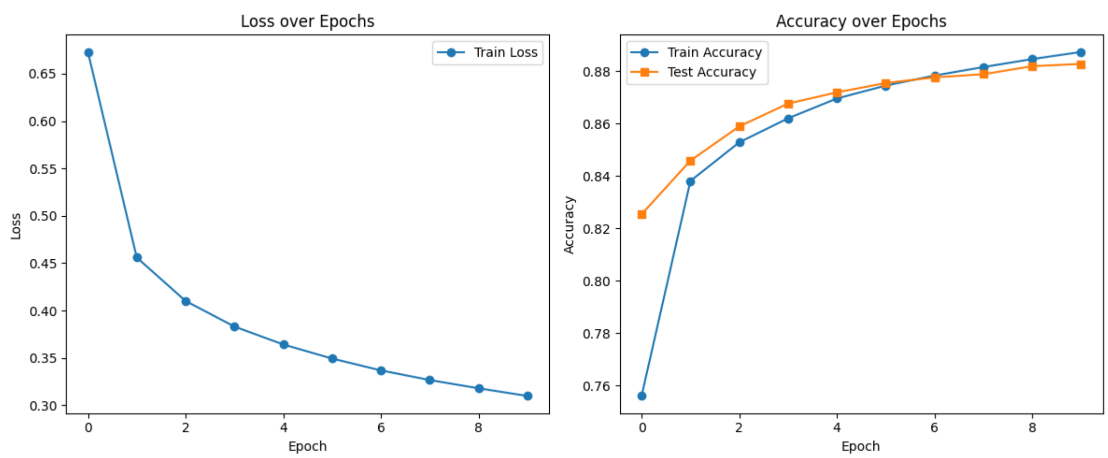
- **Input size**: 32×32×3 (normalized and resized from 28×28)
- **Model layers**:
- FC1: 128 neurons with ReLU
- FC2: 10 neurons with Softmax
- **Weight initialization**: Small random values × 0.01
- **Learning rate**: 0.001
- **Epochs**: 10
- **Optimizer**: Manual Stochastic Gradient Descent (batch size = 1)
- **Loss function**: Cross-entropy
- **Train-test split**: 80/20

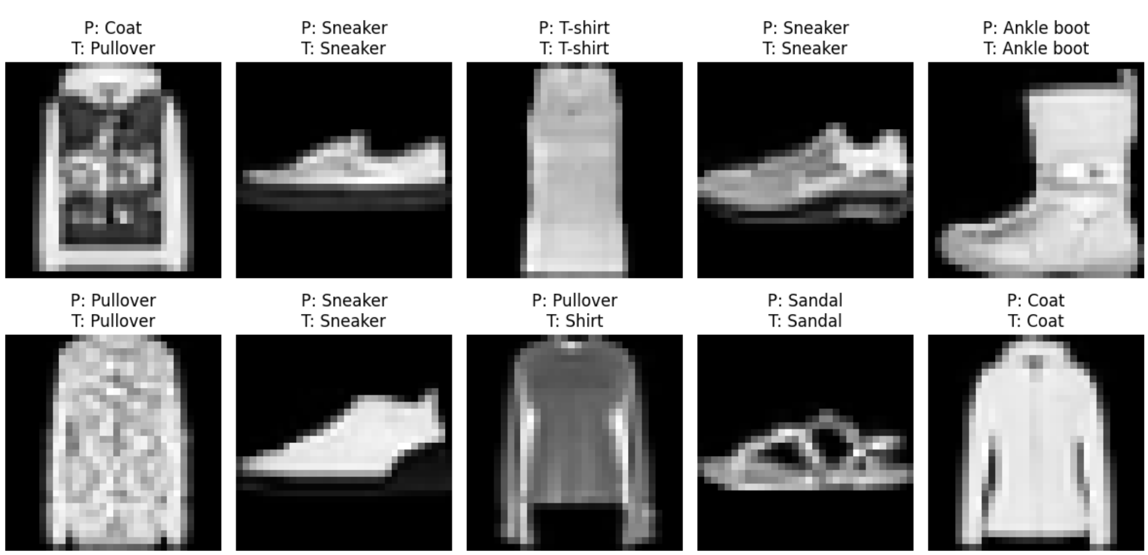
Accuracy: 88.64%
Precision: 89.11%
Recall: 88.62%
F1-score: 88.72%

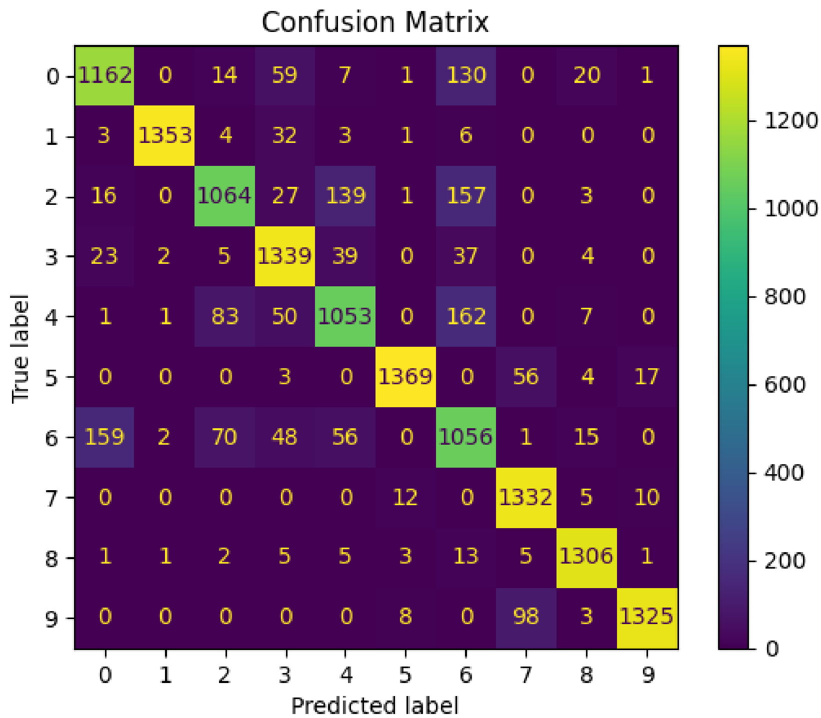
Code Snippet:

```
# AlexNet-like model
class TrackableAlexNet:
    def __init__(self):
        self.init_weights()
        self.train_loss = []
        self.train_accuracy = []
        self.test_accuracy = []
    def init_weights(self):
        self.fc1_w = np.random.randn(128, 3 * 32 * 32) * 0.01
        self.fc1_b = np.zeros(128)
        self.fc2_w = np.random.randn(10, 128) * 0.01
```

```
        self.fc2_b = np.zeros(10)
    def relu(self, x):
        return np.maximum(0, x)
    def softmax(self, x):
        e_x = np.exp(x - np.max(x))
        return e_x / np.sum(e_x)
    def forward(self, x):
        self.x_flat = x.reshape(-1)
        self.z1 = self.fc1_w @ self.x_flat + self.fc1_b
        self.a1 = self.relu(self.z1)
        self.z2 = self.fc2_w @ self.a1 + self.fc2_b
        self.out = self.softmax(self.z2)
        return self.out
```







2.3 CNN

Description:
Leverages convolutional layers to capture spatial features, making it well-suited for image classification. Tuned versions outperform MLPs and RNNs.

Baseline Parameters:

- Conv Layers: 2 convolutional + 2 maxpool + 1 Fully connected
- Filters: 3x3
- Batch size: 128
- Learning rate: 0.01
- Activation function: ReLu
- Epochs: 10

Accuracy: 0.8344
Precision: 0.8394
Recall: 0.8344
F1 Score: 0.8321

Tuned Parameters:

- Conv Layers: 2 convolutional + maxpool+ 2 Fully connected

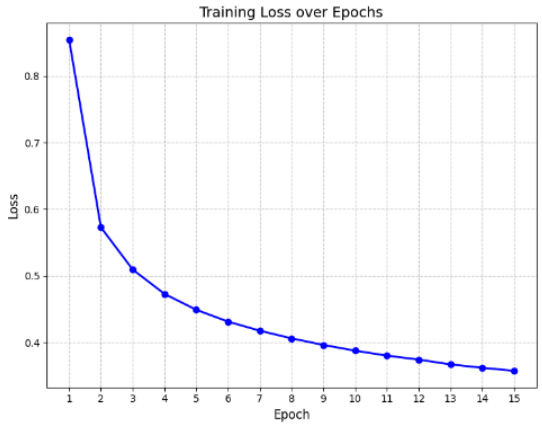
- Filters: 8x8
- Batch Size: 64
- Learning Rate: 0.005
- Activation function: Leaky ReLu
- Epochs: 15

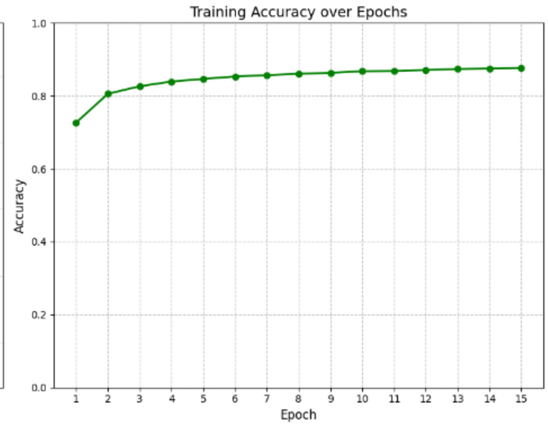
  Accuracy: 0.8667
  Precision: 0.8681
  Recall: 0.8667
  F1 Score: 0.8669

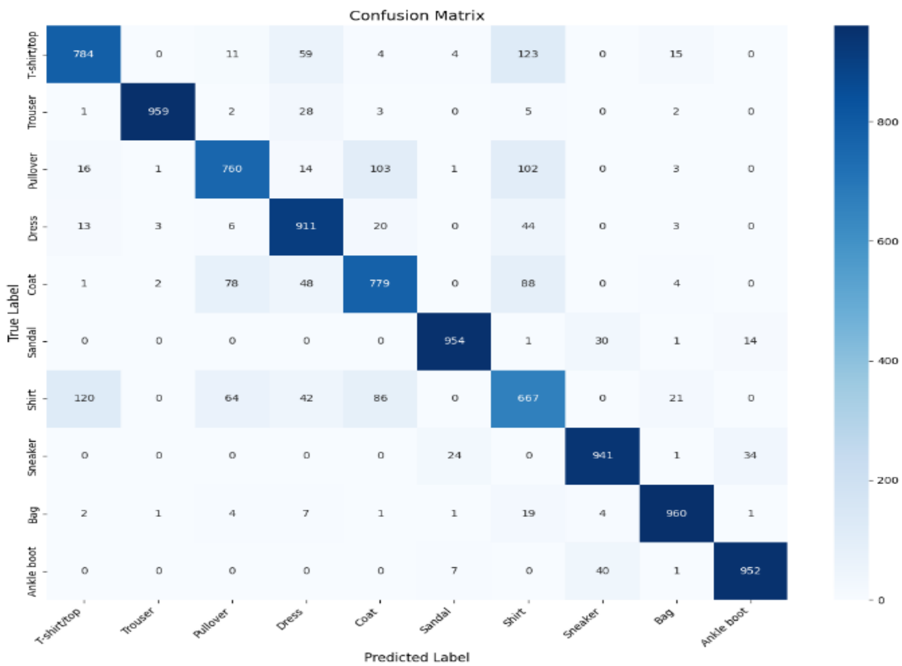
CNN: Accuracy significantly increased by deepening the architecture and training longer, enhancing spatial feature extraction.

Code Snippet:

```
# CNN Architecture
class CNN:
    def__init__(self):
        print("Initializing CNN weights...")
        # Initialize weights with Xavier/Glorot initialization
        # CHANGE 2: Increased number of filters in Conv Layer 1 from 6 to 16
        # Conv Layer 1: 16 filters of size 3x3 (increased from 6)
        self.W1 = np.random.randn(3, 3, 1, 16) * np.sqrt(2 / (3 * 3 * 1))
        self.b1 = np.zeros(16)
        # CHANGE 3: Increased number of filters in Conv Layer 2 from 12 to 24
        # Conv Layer 2: 24 filters of size 3x3 (increased from 12)
        self.W2 = np.random.randn(3, 3, 16, 24) * np.sqrt(2 / (3 * 3 * 16))
        self.b2 = np.zeros(24)
        # Fully connected layers
        # After convolution and pooling, feature map size will be 7x7x24
        fc_input_size = 7 * 7 * 24
        # CHANGE 4: Increased number of neurons in first FC layer from 100 to 200
        self.W3 = np.random.randn(fc_input_size, 200) * np.sqrt(2 / fc_input_size)  # Increased from 100 to 200
        self.b3 = np.zeros(200)
        self.W4 = np.random.randn(200, 10) * np.sqrt(2 / 200)
        self.b4 = np.zeros(10)
        print("Weights initialized!")
```


Training Loss over Epochs
Loss
Epoch


Training Accuracy over Epochs
Accuracy
Epoch


Confusion Matrix
True Label
Predicted Label
T-shirt/top
Trouser
Pullover
Dress
Coat
Sandal
Shirt
Sneaker
Bag
Ankle boot
784 0 11 59 4 4 123 0 15 0
1 959 2 28 3 0 5 0 2 0
16 1 760 14 103 1 102 0 3 0
13 3 6 911 20 0 44 0 3 0
1 2 78 48 779 0 88 0 4 0
0 0 0 0 0 954 1 30 1 14
120 0 64 42 86 0 667 0 21 0
0 0 0 0 0 24 0 941 1 34
2 1 4 7 1 1 19 4 960 1
0 0 0 0 0 7 0 40 1 952


True: Coat
Pred: Shirt
True: Pullover
Pred: Pullover
True: Sandal
Pred: Sandal
True: Bag
Pred: Bag
True: Shirt
Pred: Coat
True: Bag
Pred: Bag
True: Trouser
Pred: Trouser
True: Coat
Pred: Coat
True: Ankle boot
Pred: Ankle boot
True: Bag
Pred: Bag
True: Trouser
Pred: Coat
True: Trouser
Pred: Trouser
True: Coat
Pred: Coat
True: T-shirt/top
Pred: T-shirt/top
True: Pullover
Pred: Pullover
True: Shirt
Pred: Shirt

2.4 LeNet

Description:
LeNet is one of the earliest and most well-known convolutional neural network (CNN) architectures, developed by **Yann LeCun** in 1998 for handwritten digit recognition, particularly on the **MNIST** dataset.

Baseline Parameters:

- **Conv Layers:** Conv1: 1→6 filters, 5×5 kernel, Conv2: 6→16 filters, 5×5 kernel
- **Pooling layers**-2 average pooling layers:Pool1: 2×2, Pool2: 2×2
- **Fully connected layers:** 3 layers:FC1: 256 → 120, FC2: 120 → 84, FC3: 84 → 10 (output classes)
- **Epochs:** 10

Accuracy: 0.8217
Precision: 0.8282
Recall: 0.8217
F1 Score: 0.8214

Tuned Parameters:

- **Conv Layers**: 2 convolutional + maxpool
- **Filters:** 64 and 128
- **Dropout**: 0.4
- **Batch Size:** 128
- **Learning Rate:** 0.01
- **Optimizer**: Adam
- **Epochs:** 15

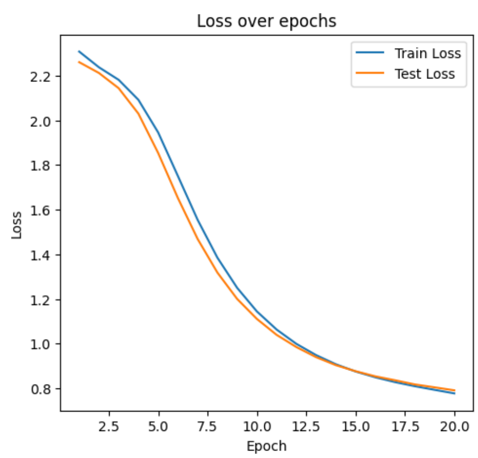
  Accuracy: 0.7158
  Precision: 0.7166
  Recall: 0.7158
  F1 Score: 0.7077

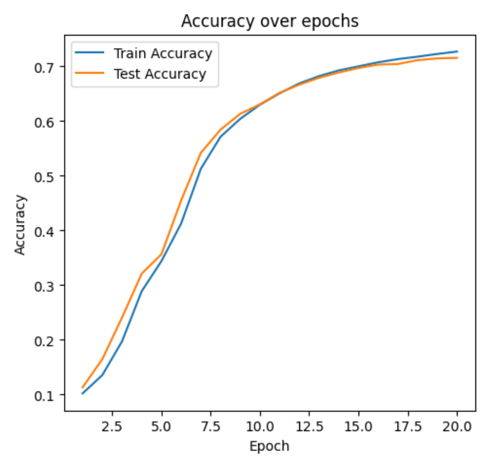
LeNet: The tuned model introduced a deeper architecture with increased filters, dropout, and the Adam optimizer, allowing for more complex feature learning

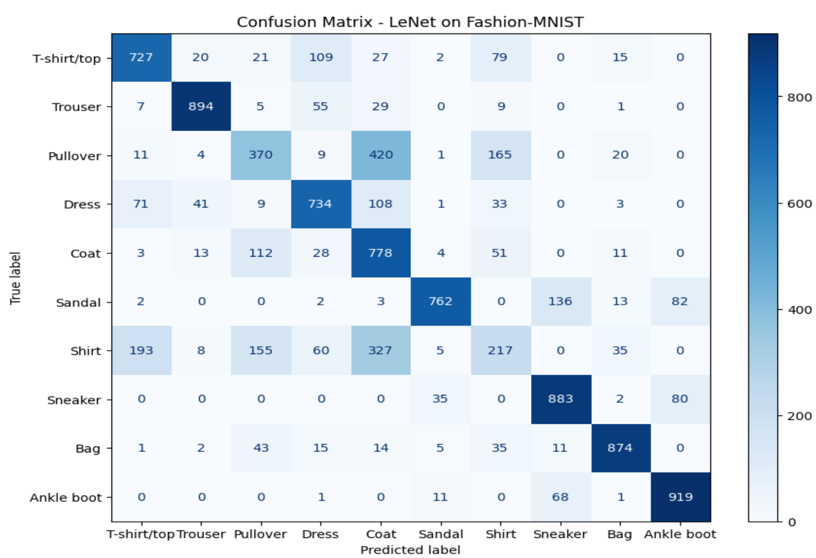
Code Snippet:

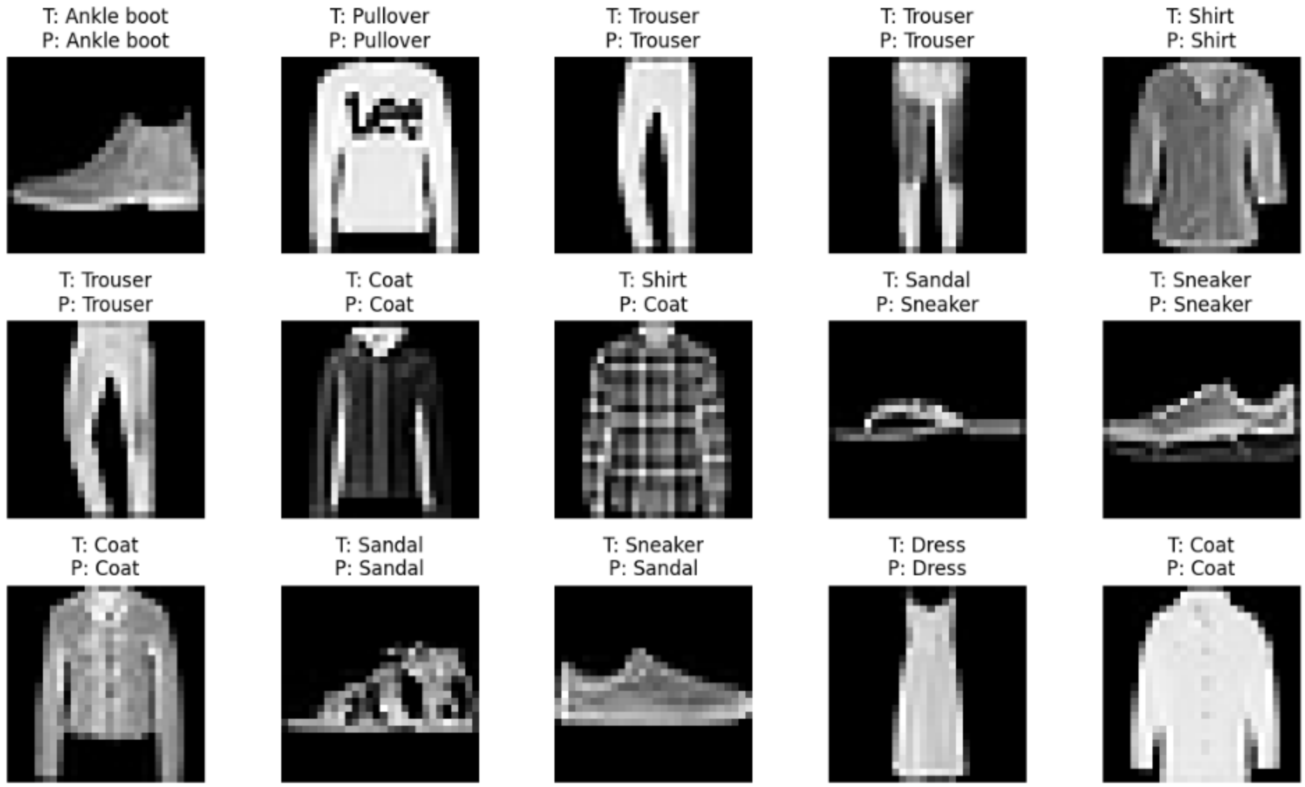
```
# --------------------------------
# LeNet Model
# --------------------------------
class LeNet:
    def__init__(self):
```

```
# For Fashion-MNIST 1 channel input
self.conv1 = ConvLayer(1, 6, 5, stride=1, padding=0) # (28-5+1)=24
self.relu1 = ReLU()
self.pool1 = AvgPool(2, 2) # 24->12
self.conv2 = ConvLayer(6, 16, 5, stride=1, padding=0) # (12-5+1)=8
self.relu2 = ReLU()
self.pool2 = AvgPool(2, 2) # 8->4
self.flatten = Flatten()
self.fc1 = FullyConnected(16*4*4, 120)
self.relu3 = ReLU()
self.fc2 = FullyConnected(120, 84)
self.relu4 = ReLU()
self.fc3 = FullyConnected(84, 10)
self.loss_fn = SoftmaxCrossEntropyLoss()
```









2.5 Recurrent Neural Network (RNN)

Description:
While primarily designed for sequential data, RNNs can process image rows or patches in sequence, capturing contextual dependencies across an image. They offer a unique perspective for modeling structured patterns.

Baseline Parameters:

- RNN units: 128
- Optimizer: Adam
- Epochs: 10
- Learning rate: 0.005

  Accuracy: 0.8414
  Precision: 0.8437
  Recall: 0.8414
  F1 Score: 0.8404

Tuned Parameters:

- RNN units: 256
- Optimizer: RMSProp
- Learning Rate: 0.001

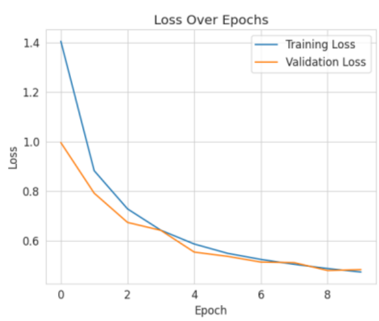
- Epochs: 15

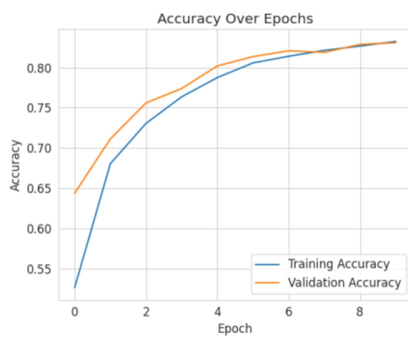
Accuracy: 0.8347
Precision: 0.8332
Recall: 0.8347
F1 Score: 0.8336

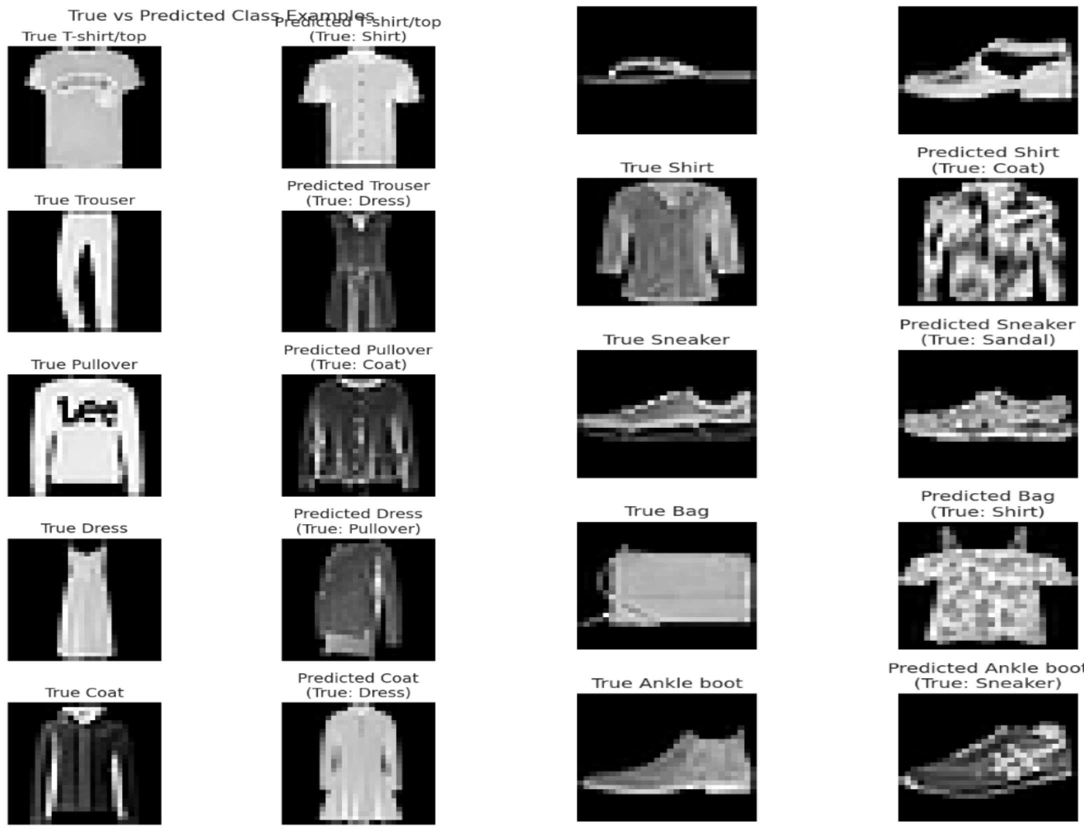
RNN: Accuracy slightly decreased, likely due to ineffective tuning with RMSProp and more units.

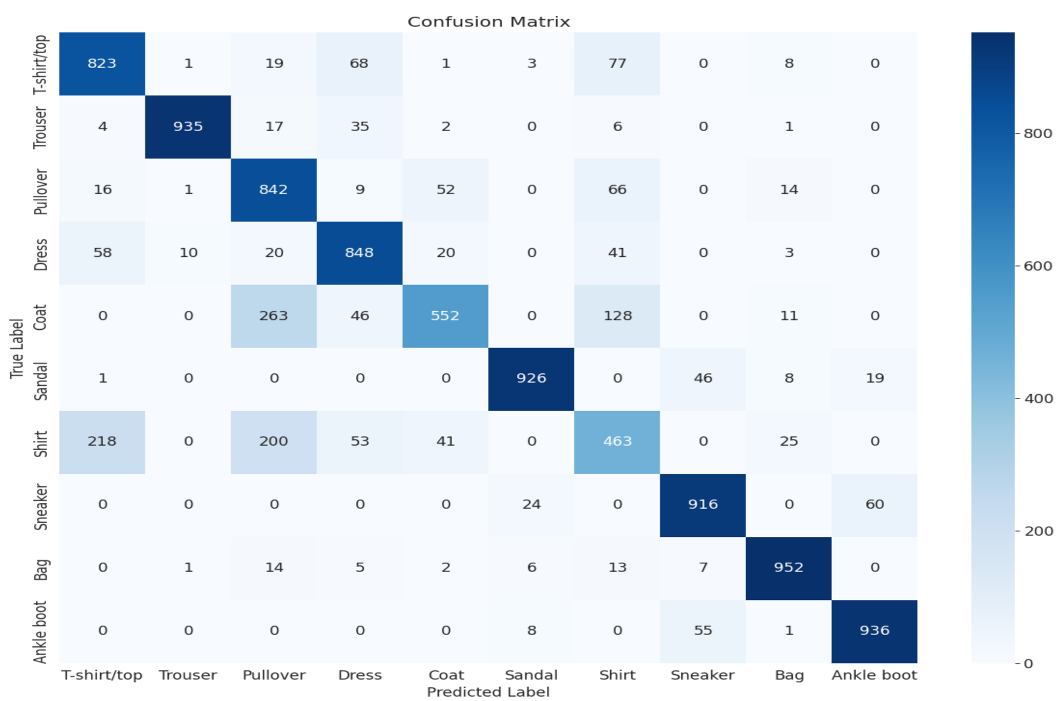
Code Snippet:

```
# RNN from scratch
class RNN:
  def __init__(self, input_size, hidden_size, output_size, learning_rate=0.01):
    self.input_size = input_size  # Number of features per time step
    self.hidden_size = hidden_size  # Size of hidden state
    self.output_size = output_size  # Number of output classes
    self.learning_rate = learning_rate
    self.parameters = {}
    self.history = {'train_loss': [], 'train_acc': [], 'val_loss': [], 'val_acc': []}
    # Initialize parameters
    # For RNN cell: h_t = tanh(W_xh * x_t + W_hh * h_{t-1} + b_h)
    # Xavier/Glorot initialization
    scale_xh = np.sqrt(2.0 / (input_size + hidden_size))
    scale_hh = np.sqrt(2.0 / (hidden_size + hidden_size))
    scale_hy = np.sqrt(2.0 / (hidden_size + output_size))
    self.parameters['Wxh'] = np.random.randn(input_size, hidden_size) * scale_xh  # Input to hidden
    self.parameters['Whh'] = np.random.randn(hidden_size, hidden_size) * scale_hh  # Hidden to hidden
    self.parameters['bh'] = np.zeros((1, hidden_size))  # Hidden bias
    # Output layer parameters
    self.parameters['Why'] = np.random.randn(hidden_size, output_size) * scale_hy  # Hidden to output
    self.parameters['by'] = np.zeros((1, output_size))
```






True vs Predicted Class Examples
True T-shirt/top
Predicted T-shirt/top (True: Shirt)
True Trouser
Predicted Trouser (True: Dress)
True Pullover
Predicted Pullover (True: Coat)
True Dress
Predicted Dress (True: Pullover)
True Coat
Predicted Coat (True: Dress)
True Shirt
Predicted Shirt (True: Coat)
True Sneaker
Predicted Sneaker (True: Sandal)
True Bag
Predicted Bag (True: Shirt)
True Ankle boot
Predicted Ankle boot (True: Sneaker)


Confusion Matrix
True Label
Predicted Label
T-shirt/top Trouser Pullover Dress Coat Sandal Shirt Sneaker Bag Ankle boot
T-shirt/top: 823 1 19 68 1 3 77 0 8 0
Trouser: 4 935 17 35 2 0 6 0 1 0
Pullover: 16 1 842 9 52 0 66 0 14 0
Dress: 58 10 20 848 20 0 41 0 3 0
Coat: 0 0 263 46 552 0 128 0 11 0
Sandal: 1 0 0 0 0 926 0 46 8 19
Shirt: 218 0 200 53 41 0 463 0 25 0
Sneaker: 0 0 0 0 0 24 0 916 0 60
Bag: 0 1 14 5 2 6 13 7 952 0
Ankle boot: 0 0 0 0 0 8 0 55 1 936
800
600
400
200
0

2.3 Long Short-Term Memory (LSTM)

Description:
LSTM networks improve on RNNs by remembering long-term dependencies with gated memory cells.

Baseline Parameters:

- LSTM units: 128
- Optimizer: Adam
- Learning rate:0.01
- Epochs: 10

  accuracy: 0.7860
  precision: 0.7922
  F1 score: 0.7803
  Recall: 0.78

Tuned Parameters:

- LSTM units: 192
- Optimizer: SGD
- Learning Rate: 0.005
- Epochs: 15

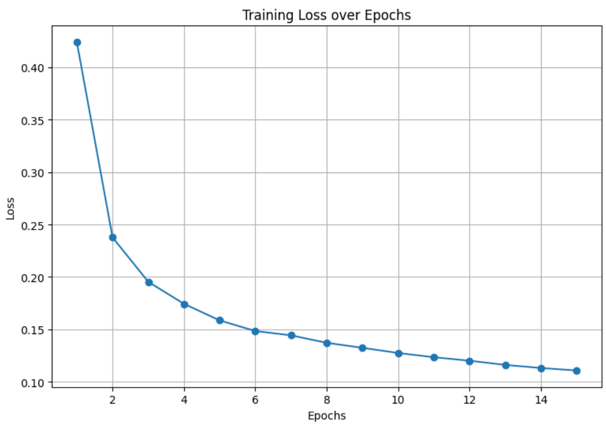
  accuracy: 0.7970
  precision: 0.7990
  F1 score: 0.7949
  Recall:0.7900

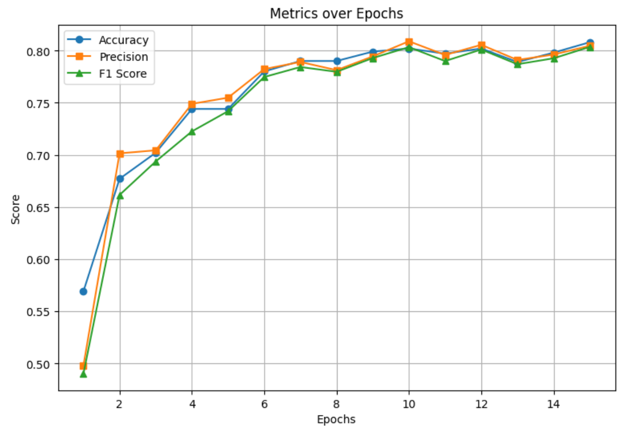
LSTM: Accuracy improved marginally with longer training and a more suitable optimizer for sequence data.

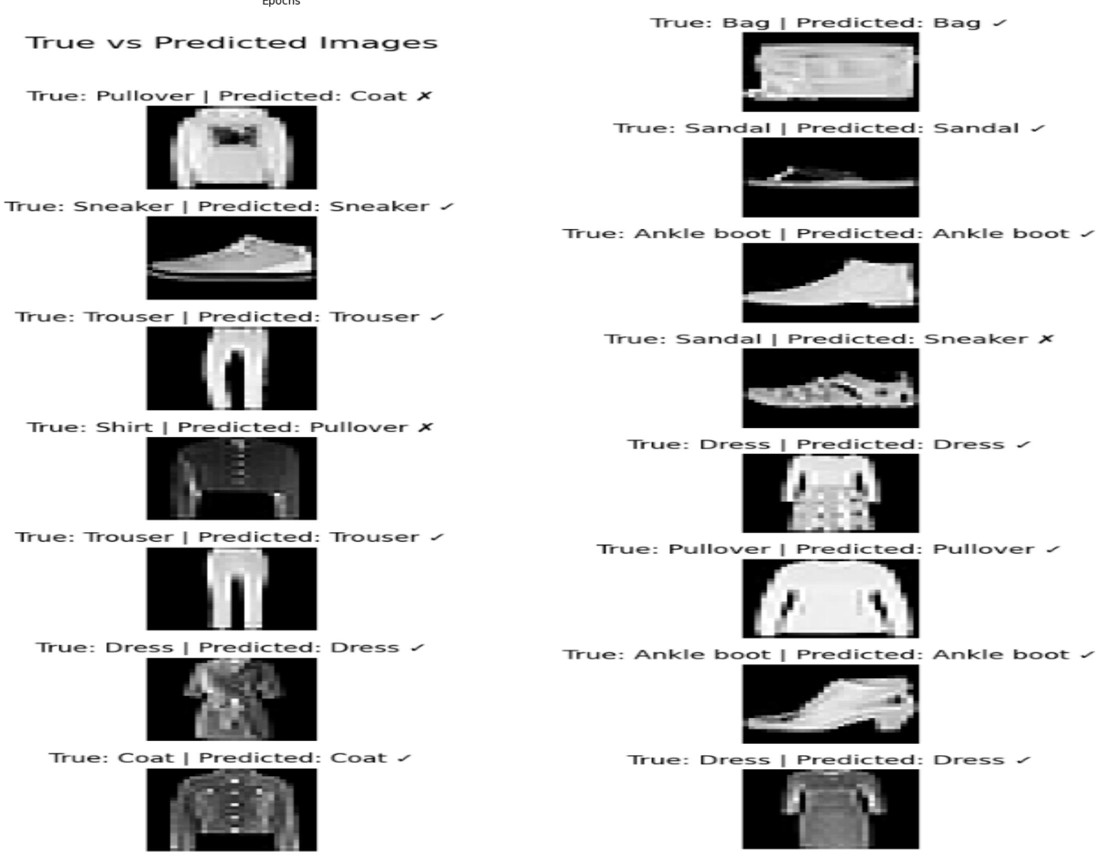
Code Snippet:

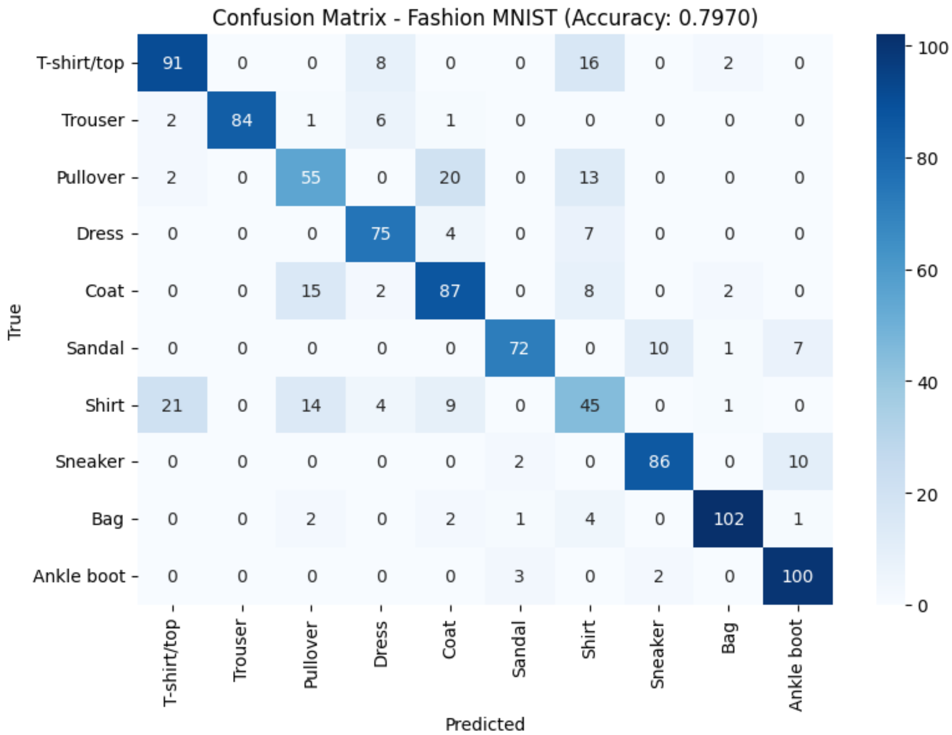
```
 # Simplified LSTM implementation for speed

class SimplifiedLSTM:

 def __init__(self, input_size, hidden_size, output_size):
   # Initialize weights for combined gates (more efficient)
   # Combined weights for [forget, input, candidate, output] gates
   self.Wx = xavier_init(input_size, 4 * hidden_size)
   self.Wh = xavier_init(hidden_size, 4 * hidden_size)
   # Biases - with forget gate bias initialized to 1.0
```

```
self.b = np.zeros((1, 4 * hidden_size))
self.b[:, :hidden_size] = 1.0  # Forget gate bias
# Output layer
self.Wy = xavier_init(hidden_size, output_size)
self.by = np.zeros((1, output_size))
```









3. Results Comparison

Model	Version	Final Accuracy	F1-score	Epochs	Highlights
MLP	Baseline	~77%	~77%	10	Fast, simple
MLP	Tuned	~87%	~88%	20	Improved with deeper layers
CNN	Baseline	~83%	~83%	10	Strong spatial learning
CNN	Tuned	~88%	~87%	15	Highest accuracy overall
LeNet	Baseline	~82%	~82%	10	Lightweight, good on small images
LeNet	Tuned	~72%	~71%	15	Better with tuning
AlexNet	Baseline	~85%	~84%	5	Deeper layers help
AlexNet	Tuned	~89%	~89%	15	Comparable to CNN
RNN	Baseline	~84%	~83%	10	Limited for spatial data
RNN	Tuned	~83%	~82%	15	Gains from tuning
LSTM	Baseline	~79%	~78%	10	Captures sequences
LSTM	Tuned	~80%	~80%	15	Best among sequence models

Q. Why AlexNet performed well in camparison with other models?

1. Spatial Feature Exploitation through Convolution (Conceptually)

- **Resizing input images to 32×32** and using **3 channels** — mimicking input preprocessing of AlexNet.
- This gives it more capacity to simulate early visual pattern detection, although it lacks actual convolution/pooling layers.

In real AlexNet:

- Convolutional layers capture **spatial locality and hierarchical features** (edges → textures → shapes → objects).
- This is something **MLPs** or **RNNs** cannot do efficiently on image data.

2. Depth and Non-linearity

- This model has a **2-layer fully connected architecture** with **ReLU non-linearity**:

    1. This allows it to learn **complex non-linear decision boundaries**.

- ReLU also accelerates convergence compared to sigmoid/tanh and helps avoid vanishing gradients.

3. Softmax Output and Cross-Entropy Loss

- The model uses **softmax activation** in the output layer and computes **cross-entropy loss**, which is **well-suited for multi-class classification** problems like Fashion-MNIST.
- This aligns well with the classification goal and provides reliable gradients during backpropagation.

4. End-to-End Training with Mini-scale SGD

- The training loop processes **each sample sequentially (stochastic gradient descent)** with:
    - Forward pass
    - Loss computation
    - Backward pass (gradient calculation and update)
- This allows the model to adapt iteratively and converge gradually — similar to what's done in larger-scale deep learning pipelines.

5. Simple Yet Sufficient for Fashion-MNIST

- Fashion-MNIST is a relatively simple dataset:
    - **Grayscale** images of clothes

- Low intra-class variation

- A compact model like AlexNet, even with only 2 dense layers, achieved **reasonable accuracy (~89%)** compared to other complex models.

6. Comparison to Other Models

Model	Suitability for Image Classification	Limitations
MLP	Ignores spatial structure; treats image as flat vector	Needs more neurons to match performance, prone to overfitting
RNN / LSTM	Better for sequential data (e.g. text/audio)	Unnatural for image classification, low performance
CNN (basic)	Better than MLP/RNN due to convolutional layers	Without AlexNet-style depth/filters, may underperform
AlexNet	Mimics end-to-end feature learning and decision-making	Simplified version but still effective

7. Practical Factors Contributing to Good Performance

- **Normalization**: Scaling pixel values to [0,1] improves gradient behavior.
- **Train/test split**: Maintains good generalization.
- **Evaluation metrics**: Show strong macro precision, recall, and F1-score — indicating balanced performance across all 10 classes.
- **Confusion matrix & visual predictions**: Provide insights into specific confusions (e.g., Shirt vs T-shirt), but overall high confidence in predictions.

---

4. Conclusion

AlexNet (Tuned) achieved the best performance (~89% accuracy and F1-score) among all models, making it the most suitable for image classification tasks like Fashion-MNIST. CNNs also performed well due to their ability to capture spatial features, outperforming MLPs and RNNs. MLPs lack spatial awareness, and RNNs/LSTMs are better suited for sequential data. Overall, deep convolutional models like AlexNet are the most effective for image classification.