19N502-DEEP LEARNING

"Image Classification on Fashion MNIST Dataset Using Simple CNN and Pre-trained ResNet: Training and Prediction on External Images"

BACHELOR OF ENGINEERING
COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

SEPTEMBER 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PSG COLLEGE OF TECHNOLOGY
(Autonomous Institution)
COIMBATORE – 641 004

22N239 - PREETHI P M
22N245 - SAKTHI S

Problem statement:

The primary objective of this project is to create and evaluate two image classification models—one based on ResNet architecture and another using a simple CNN—to accurately classify images from the Fashion MNIST dataset. The effectiveness of each model will be compared based on accuracy, loss, and other relevant performance metrics. The task involves identifying 10 different classes of clothing items such as T-shirts, trousers, pullovers, dresses, etc.

Expected Output:

- The model should achieve at least **85% accuracy** on the test set. ● The model should output a **confusion matrix** to visualize how well the model performs across all clothing classes.
- Training and validation losses and accuracies should be plotted to track the model's learning curve.

FashionMNIST Dataset:

- Fashion MNIST contains 70,000 grayscale images of fashion items. ● The dataset is divided into 60,000 training images and 10,000 testing images.
- There are 10 different classes, each representing a different type of clothing item
- Each image is a 28x28 pixel grayscale image, providing a uniform input size for neural network models.
- Fashion MNIST is designed as a more challenging alternative to the original MNIST dataset (which consists of handwritten digits) and is often used for benchmarking machine learning algorithms.
- Although it is more complex than the original MNIST dataset, Fashion MNIST is still relatively simple compared to more sophisticated datasets like ImageNet, making it suitable for educational purposes and for testing new algorithms.

1.SIMPLE CNN

Improvements Over a Simple CNN

Several improvements are introduced over a basic CNN to enhance performance and training efficiency:

Batch Normalization helps in faster and more stable training by normalizing intermediate outputs.

Leaky ReLU activation addresses the vanishing gradient problem by allowing a small, non-zero gradient for negative inputs, improving the learning of the network.

Max Pooling reduces the spatial dimensions of feature maps, improving computational efficiency and helping the model generalize better.

Early Stopping prevents overfitting by stopping the training when validation performance ceases to improve.

Data Augmentation increases the dataset variability, helping the model generalize better by introducing new variations of training samples.

Hyperparameters

Number of Epochs determines how many full passes are made through the training dataset.

Batch Size controls how many images are processed at a time. Larger batch sizes reduce noise but may require more memory.

Learning Rate defines how much to adjust the model weights with each step of the optimizer.

Patience for Early Stopping is used to terminate training early if no improvement in validation loss is detected over a certain number of epochs.

Input to the Model

The input to the model is an image from the Fashion MNIST dataset, which has a size of 28x28 pixels and is grayscale (single channel).

Data augmentation

The image is passed through several data augmentation steps like random flipping and rotation before it enters the model. This helps generate more diverse training data and improves the model's generalization. Data augmentation is done.

Each image is also normalized to have a mean of 0.5 and a standard deviation of 0.5, which speeds up convergence and stabilizes training.

```
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

Model:

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=5, padding=2),
            nn.LeakyReLU(0.01),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=5, padding=2),
            nn.LeakyReLU(0.01),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.fc1 = nn.Linear(64*7*7, 1000)
        self.leaky_relu = nn.LeakyReLU(0.01)
        self.fc2 = nn.Linear(1000, 10)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.leaky_relu(self.fc1(out))
        out = self.fc2(out)
        return out
```


Convolutional Layer 1

The first layer is a convolutional layer that applies 32 filters (or kernels),

each of size 5x5, over the input image.Convolution extracts local features such as edges, textures, or simple patterns from the input image.The result of this convolution is a feature map, where each filter creates a unique map based on the learned features.

Batch Normalization

Batch normalization is applied to normalize the outputs of the first convolutional layer. This helps stabilize and accelerate the training process by ensuring the data is on a similar scale.

Activation Function: Leaky ReLU

After batch normalization, the output goes through the Leaky ReLU activation function.Unlike traditional ReLU, Leaky ReLU allows small negative values to pass through, which prevents the dying ReLU problem (where neurons stop learning).This introduces non-linearity into the model, allowing it to learn complex patterns from the data.

Max Pooling Layer 1

Next, the feature map from the Leaky ReLU output is passed through a max pooling layer with a pool size of 2x2.Max pooling reduces the spatial dimensions of the feature map (by taking the maximum value in each 2x2 region), which reduces computational complexity and makes the model more efficient.This downsampling also helps the model become more translation invariant, meaning it can recognize patterns regardless of their location in the image.

Convolutional Layer 2

The output of the first max pooling layer is passed into the second convolutional layer, which uses 64 filters, also with a size of 5x5.Like the first convolutional layer, this layer extracts higher-level features from the downsampled feature maps, focusing on more complex structures or patterns.

Batch Normalization and Leaky ReLU

Similar to the first convolutional layer, batch normalization and Leaky ReLU are applied to the output of the second convolution.This continues to introduce non-linearity and stabilizes the training process.Batch normalization is applied after the convolutional layers to normalize the output of each layer within each mini-batch. This helps in stabilizing and accelerating training by reducing internal covariate shift, allowing the network to use higher learning rates and speeding up convergence.

Max Pooling Layer 2

The second set of feature maps is passed through another max pooling layer with a 2x2 pool size.This further reduces the spatial dimensions, allowing the model to focus on the most important features while keeping computations manageable.

Flattening the Output

After the second max pooling layer, the output is a set of reduced feature maps (of size 7x7 from each of the 64 filters).These feature maps are flattened into a single long vector, preparing the data to be passed into fully connected layers (dense layers).The flattened vector size is 64 * 7 * 7 = 3136.

Fully Connected Layer 1

- The flattened vector is passed through the first fully connected layer, which has 1000 neurons.Each neuron in this layer is connected to every output from the previous layer, allowing the model to combine features from different parts of the image to make more global inferences.

Leaky ReLU Activation

The output of the fully connected layer is passed through another Leaky ReLU activation function to introduce non-linearity.

Fully Connected Layer 2 (Output Layer)

The final layer is another fully connected layer with 10 output neurons, one for each class in the Fashion MNIST dataset.The model produces a logit (a raw score) for each class, indicating how likely the input image belongs to each class.

Softmax and Predictions

The raw logits are passed through the softmax function (implicitly handled during loss calculation) to convert them into probabilities. The class with the highest probability is selected as the predicted class for the image.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

Loss Calculation (Cross-Entropy Loss)

During training, the model's predictions (logits) are compared to the true labels using Cross-Entropy Loss.The **Cross Entropy Loss** is used as the loss function since this is a multi-class classification problem.

Backpropagation

Backpropagation is used to update the model's weights.The loss gradient is calculated for each weight, and the optimizer (Adam) adjusts the weights to minimize the loss**.Adam Optimizer** is chosen for its adaptive learning rate properties, which makes it efficient in handling noisy gradients and sparse gradients.

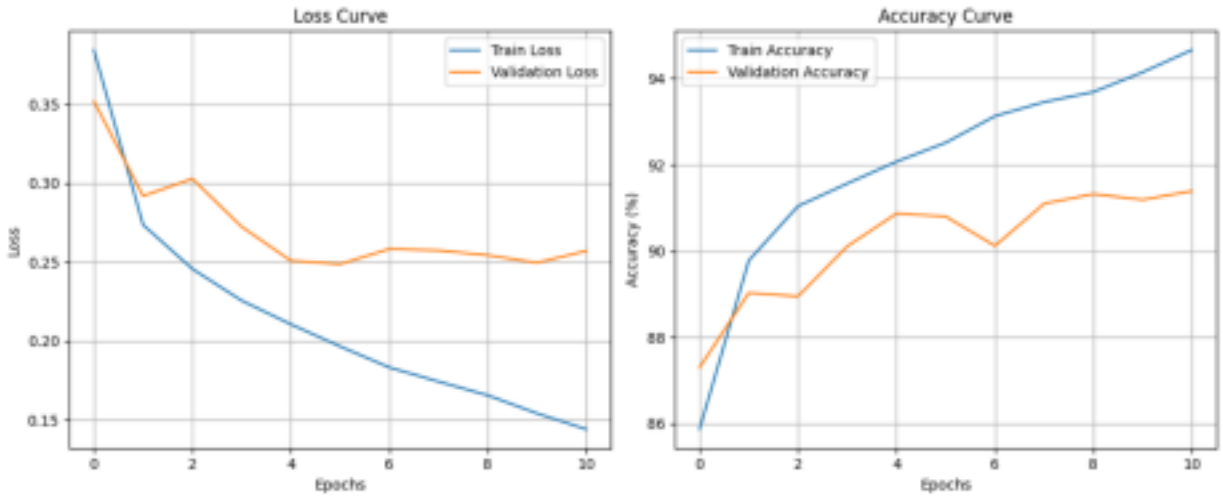
Early Stopping

```
if avg_val_loss < best_loss:
    best_loss = avg_val_loss
    epochs_without_improvement = 0
    torch.save(model.state_dict(), 'best_model.pth')
else:
    epochs_without_improvement += 1
    if epochs_without_improvement >= patience:
        print("Early stopping triggered. Stopping training.")
        break
```

During each epoch, the model is evaluated on a validation set. If the validation loss does not improve for a certain number of epochs which is determined by patience, early stopping halts the training to prevent overfitting.

Testing the Model

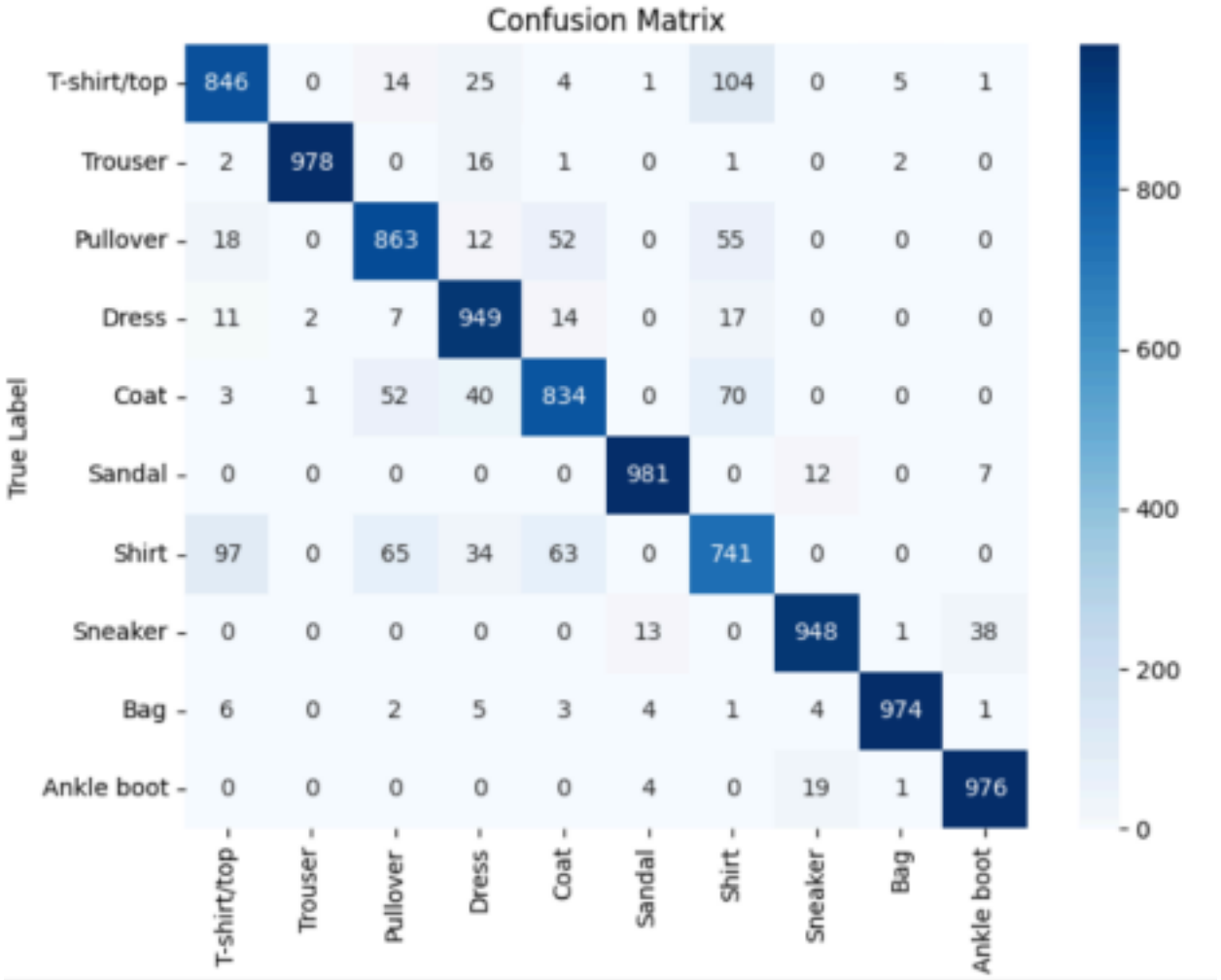
After training, the best-performing model (based on validation loss) is saved and used for testing on a separate test dataset.The model's predictions are compared to the ground truth, and the overall test accuracy is calculated.



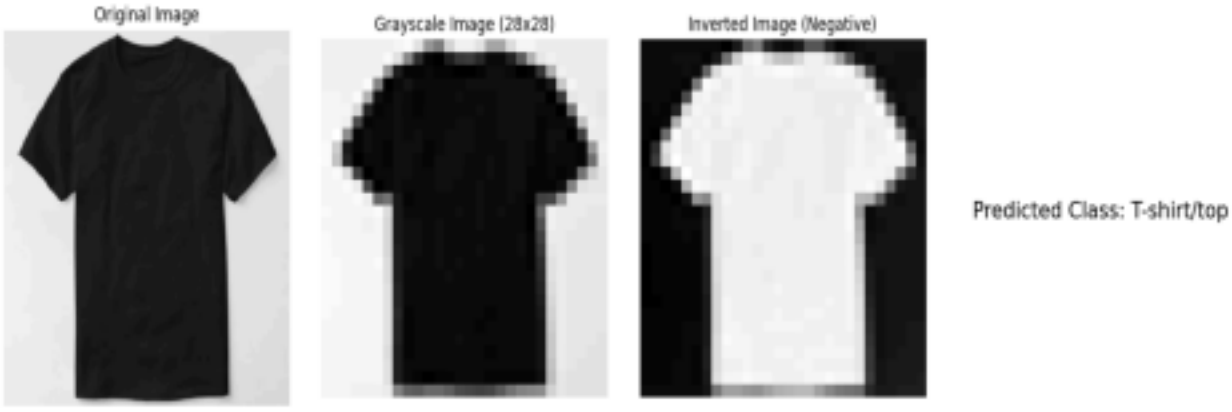
7

Confusion Matrix

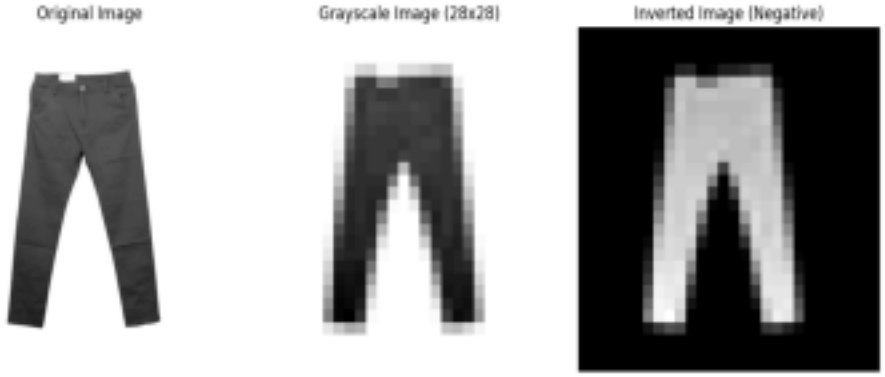
A confusion matrix is generated to evaluate how well the model performs across each class.It provides a detailed view of where the model misclassified, showing which classes are confused with others.


Confusion Matrix
True Label
T-shirt/top
Trouser
Pullover
Dress
Coat
Sandal
Shirt
Sneaker
Bag
Ankle boot
846 0 14 25 4 1 104 0 5 1
2 978 0 16 1 0 1 0 2 0
18 0 863 12 52 0 55 0 0 0
11 2 7 949 14 0 17 0 0 0
3 1 52 40 834 0 70 0 0 0
0 0 0 0 0 981 0 12 0 7
97 0 65 34 63 0 741 0 0 0
0 0 0 0 0 13 0 948 1 38
6 0 2 5 3 4 1 4 974 1
0 0 0 0 0 4 0 19 1 976
800
600
400
200
0

8


Original Image
Grayscale Image (28x28)
Inverted Image (Negative)
Predicted Class: T-shirt/top

Predicted class for /content/pant.webp: Trouser



Predicted Class: Trouser

Predicted class for /content/aboot.webp: Ankle boot

Predicted Class: Ankle boot

2.RESNET

Data Preprocessing:

```
transform = transforms.Compose([
    transforms.Lambda(lambda x: x.convert("RGB")),
    transforms.Resize((32, 32)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])
```

- Convert grayscale images to RGB to fit the ResNet18 model, which is pre-trained on color images.
- Resize the images to 32x32 pixels, as required by ResNet18. ● Apply data augmentation techniques such as random horizontal flip, rotation, and color jitter to prevent overfitting.
- Normalize the images to have a mean and standard deviation of 0.5 for each channel.

Model Architecture:

```
# Load the pre-trained ResNet18 model
resnet_model = models.resnet18(pretrained=True)

# Unfreeze the last few layers for fine-tuning
for param in resnet_model.parameters():
    param.requires_grad = False

# Unfreeze the last residual block and fully connected layer
for param in resnet_model.layer4.parameters():
    param.requires_grad = True

# Replace the final fully connected layer with a new one for 10 classes
resnet_model.fc = nn.Linear(resnet_model.fc.in_features, num_classes)
resnet_model = resnet_model.to(device)
```

- **Pre-trained ResNet18:** The model starts with ResNet18 pre-trained on ImageNet.

- **Fine-tuning:** Freeze most layers, but unfreeze the last residual block (`layer4`) and the fully connected layer for fine-tuning.
- The final fully connected layer is modified to output 10 classes (matching the number of Fashion MNIST categories).
- **Parameter Freezing:** All parameters are initially frozen to retain the learned features from the pre-trained model.
- **Selective Unfreezing:** The last residual block and fully connected layer are unfreezed, allowing fine-tuning for the new dataset.
- **Final Layer Replacement:** The fully connected layer is replaced with a new one for 10 output classes, enabling the model to classify the new data effectively.

Training Setup:

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(resnet_model.parameters(), lr=learning_rate, weight_decay=0.0001)
```

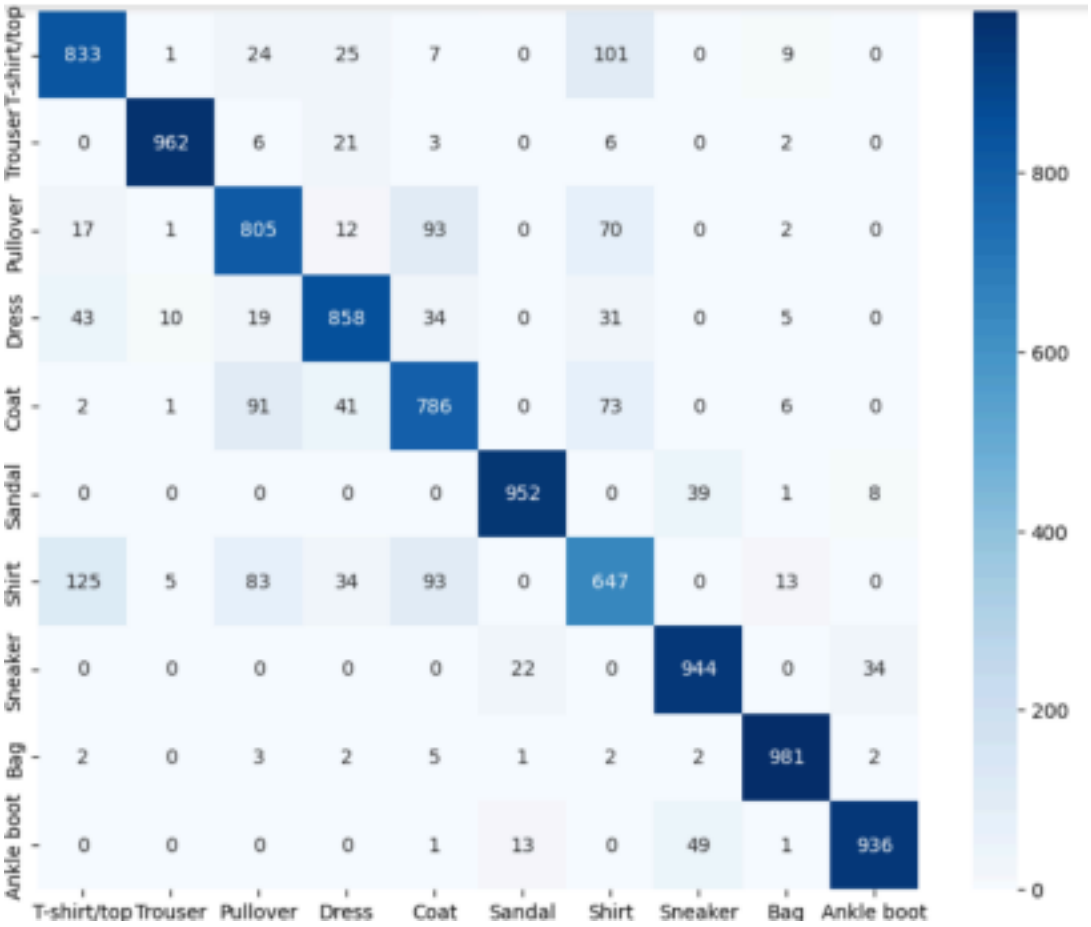
- Use *Cross-Entropy Loss* for multi-class classification.
- Adam optimizer is used, with a lower learning rate (0.0001) for fine-tuning and weight decay (L2 regularization).
- A learning rate scheduler reduces the learning rate every 5 epochs to refine the learning process.
- **Pretrained ResNet18**: The decision to use a pretrained ResNet18 is based on its strong ability to capture features in image classification tasks.
- **Fine-tuning**: Only the last block of layers (layer4) and the fully connected layer are fine-tuned to retain general-purpose features while adapting the network to the Fashion MNIST dataset.
- **Learning Rate and Scheduler**: A low learning rate of 0.0001 is used for fine-tuning, and a learning rate scheduler steps down the learning rate every 5 epochs to further refine model performance.

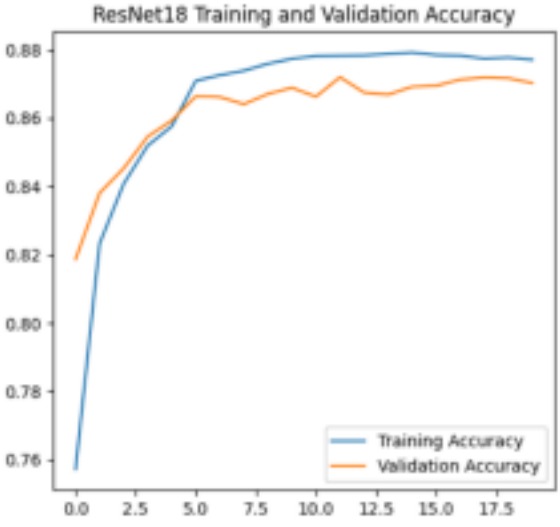
Training and Evaluation

Training Process: The model is trained for 20 epochs, and both training/validation loss and accuracy are tracked throughout.

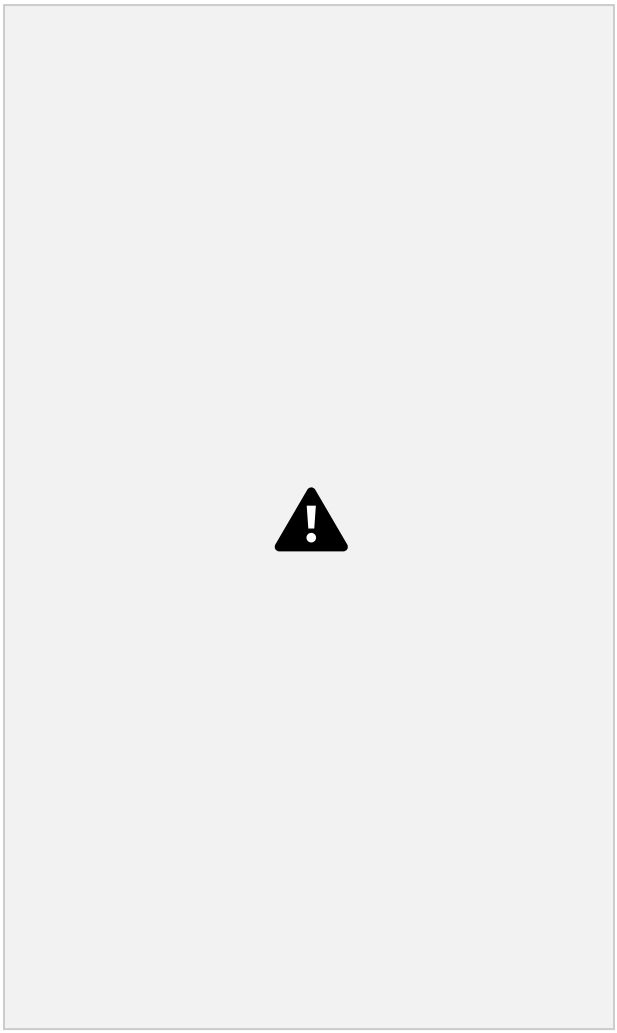
Confusion Matrix: A confusion matrix is plotted to analyze model performance for each class.

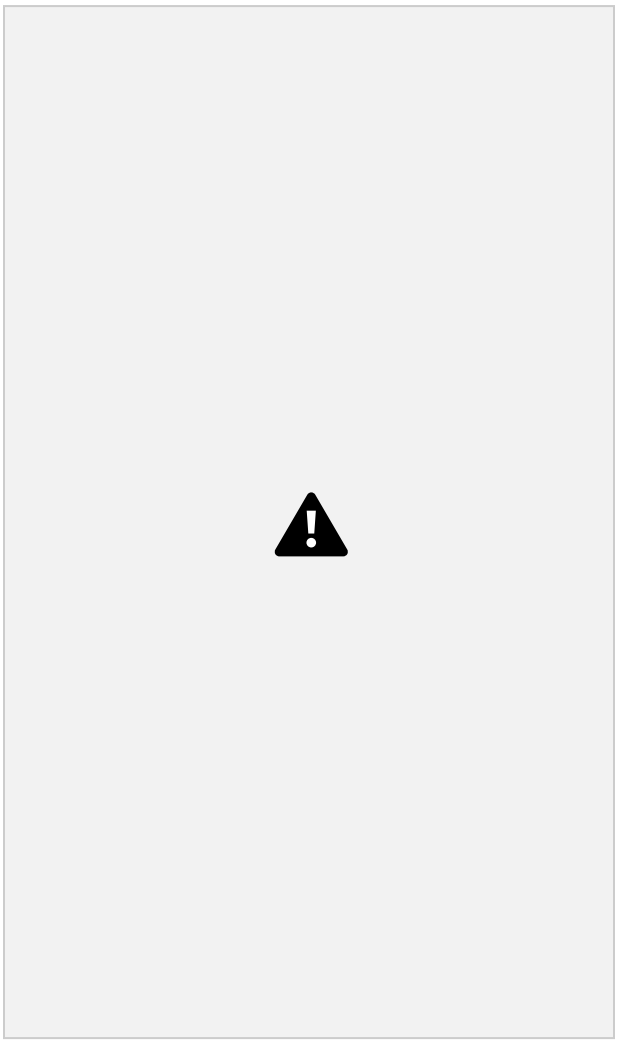
Accuracy and Loss: Both accuracy and loss plots are generated to monitor overfitting and performance.


T-shirt/top
Trouser
Pullover
Dress
Coat
Sandal
Shirt
Sneaker
Bag
Ankle boot
833 1 24 25 7 0 101 0 9 0
0 962 6 21 3 0 6 0 2 0
17 1 805 12 93 0 70 0 2 0
43 10 19 858 34 0 31 0 5 0
2 1 91 41 786 0 73 0 6 0
0 0 0 0 0 952 0 39 1 8
125 5 83 34 93 0 647 0 13 0
0 0 0 0 0 22 0 944 0 34
2 0 3 2 5 1 2 2 981 2
0 0 0 0 1 13 0 49 1 936
800
600
400
200
0


ResNet18 Training and Validation Accuracy
Training Accuracy
Validation Accuracy
0.88
0.86
0.84
0.82
0.80
0.78
0.76
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5


ResNet18 Training and Validation Loss
Training Loss
Validation Loss
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5





Results and Error Analysis

- The model achieved **90.90%** on the test set while using simple CNN and **87.04%** using ResNet18.
- The **confusion matrix** reveals that the model performs well in most cases but struggles slightly between similar clothing categories, such as shirts and T-shirts.

Conclusion:

Fashion MNIST dataset has been classified using simple CNN and ResNet accurately.