Group No: 36

Group Member Names:

Name	USN
RAVI SAJJANAR	2024DA04008
DEVATA SAI SUDHESH	2024DA04009
BANDARU HAREESHA	2024SA04089
BHAVYA ARORA	2023DA04058

1. Import the required libraries

```
##---------Type the code below this line------------------##

# Import necessary libraries for deep learning and data processing
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models, regularizers
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.utils import to_categorical

# Import libraries for data manipulation and visualization
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report, accurac
import time

# Set random seeds for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

print(f"TensorFlow version: {tf.__version__}")
print(f"Keras version: {keras.__version__}")
```

```
TensorFlow version: 2.19.0
Keras version: 3.10.0
```

2. Data Acquisition -- Score: 0.5 Mark

For the problem identified by you, students have to find the data source themselves from any data source.

2.1 Code for converting the above downloaded data into a form suitable for DL

```
##---------Type the code below this line------------------##

# Load the Fashion-MNIST dataset from TensorFlow/Keras
print("Loading Fashion-MNIST dataset...")
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

print(f"Training data shape: {X_train_full.shape}")
print(f"Training labels shape: {y_train_full.shape}")
print(f"Test data shape: {X_test.shape}")
print(f"Test labels shape: {y_test.shape}")

# Class names for Fashion-MNIST
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

print(f"\nNumber of classes: {len(class_names)}")
print(f"Class names: {class_names}")
```

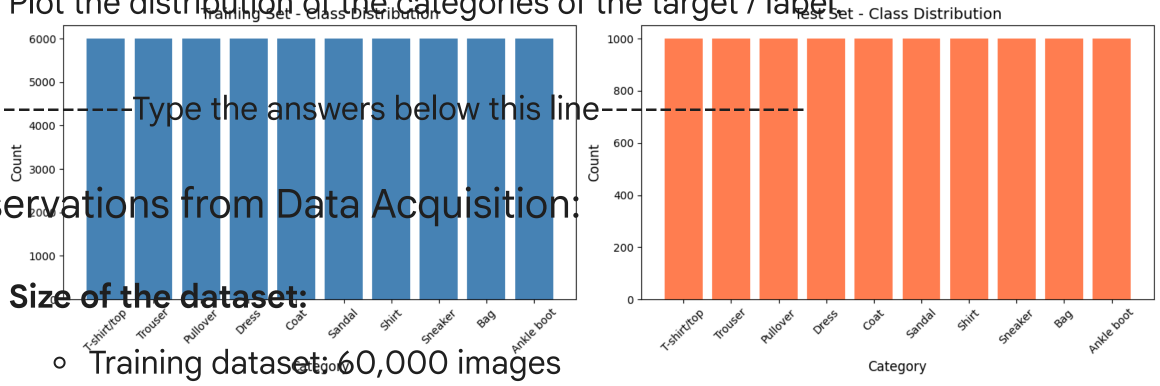
```
Loading Fashion-MNIST dataset...
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
29515/29515 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
26421880/26421880 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
5148/5148 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
4422102/4422102 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Training data shape: (60000, 28, 28)
Training labels shape: (60000,)
Test data shape: (10000, 28, 28)
Test labels shape: (10000,)

Number of classes: 10
Class names: ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal'
```

```
# Visualize sample images from the dataset
fig, axes = plt.subplots(2, 5, figsize=(15, 6))
fig.suptitle('Sample Images from Fashion-MNIST Dataset', fontsize=16)

for i, ax in enumerate(axes.flat):
    ax.imshow(X_train_full[i], cmap='gray')
    ax.set_title(f'{class_names[y_train_full[i]]}')
    ax.axis('off')

plt.tight_layout()
plt.show()

# Plot the distribution of categories
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
```

```
# Training set distribution
unique_train, counts_train = np.unique(y_train_full, return_counts=True)
ax1.bar([class_names[i] for i in unique_train], counts_train, color='steelbl
ax1.set_xlabel('Category', fontsize=12)
ax1.set_ylabel('Count', fontsize=12)
ax1.set_title('Training Set - Class Distribution', fontsize=14)
ax1.tick_params(axis='x', rotation=45)

# Test set distribution
unique_test, counts_test = np.unique(y_test, return_counts=True)
ax2.bar([class_names[i] for i in unique_test], counts_test, color='coral')
ax2.set_xlabel('Category', fontsize=12)
ax2.set_ylabel('Count', fontsize=12)
ax2.set_title('Test Set - Class Distribution', fontsize=14)
ax2.tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.show()

print("\nDataset is balanced across all classes!")
```

2.1 Write your observations from the above.

1. Size of the dataset
2. What type of data attributes are there?
3. What are you classifying?
4. Plot the distribution of the categories of the target / label.



```
Dataset is balanced across all classes!
```

-------------Type the answers below this line-------------

Observations from Data Acquisition:

1. **Size of the dataset:**
   - Training dataset: 60,000 images
   - Test dataset: 10,000 images
   - Total: 70,000 images
   - Image dimensions: 28x28 pixels (grayscale)
2. **What type of data attributes are there?**
   - Input features: Pixel intensity values (0-255) for each 28x28 grayscale image
   - Each image has 784 features (28 * 28 pixels)
   - Data type: Unsigned 8-bit integers (uint8)
   - Target labels: Integer values from 0 to 9
3. **What are you classifying?**
   - 10 different categories of fashion items:
     - 0: T-shirt/top
     - 1: Trouser
     - 2: Pullover
     - 3: Dress
     - 4: Coat
     - 5: Sandal
     - 6: Shirt
     - 7: Sneaker
     - 8: Bag
     - 9: Ankle boot

4. **Distribution of categories:**
    - See the plot above showing the distribution of each class in both training and test sets
    - The dataset is balanced with approximately 6,000 samples per class in training and 1,000 per class in test set

3. Data Preparation -- Score: 1 Mark

Perform the data prepracessing that is required for the data that you have downloaded.

This stage depends on the dataset that is used.

3.1 Apply pre-processing techiniques

- to remove duplicate data
- to impute or remove missing data
- to remove data inconsistencies
- Encode categorical data
- Normalize the data
- Feature Engineering
- Stop word removal, lemmatiation, stemming, vectorization

IF ANY

```
##---------Type the code below this line------------------##

# 3.1 Data Preprocessing

# Check for missing values
print("Checking for missing values...")
print(f"Training data missing values: {np.isnan(X_train_full).sum()}")
print(f"Test data missing values: {np.isnan(X_test).sum()}")

# Check data shape and statistics
print(f"\nOriginal data range: [{X_train_full.min()}, {X_train_full.max()}]"
print(f"Data type: {X_train_full.dtype}")

# Normalize the pixel values from [0, 255] to [0, 1]
print("\nNormalizing pixel values to [0, 1]...")
X_train_normalized = X_train_full.astype('float32') / 255.0
X_test_normalized = X_test.astype('float32') / 255.0

print(f"Normalized data range: [{X_train_normalized.min()}, {X_train_normali

# Flatten the images from 28x28 to 784 for Dense layers (DNN)
```

```
print("\nFlattening images for DNN...")
X_train_flattened = X_train_normalized.reshape(X_train_normalized.shape[0],
X_test_flattened = X_test_normalized.reshape(X_test_normalized.shape[0], -1)

print(f"Flattened training data shape: {X_train_flattened.shape}")
print(f"Flattened test data shape: {X_test_flattened.shape}")

# Check for duplicates in training data
print("\nChecking for duplicate samples...")
unique_samples = np.unique(X_train_flattened, axis=0)
print(f"Unique training samples: {len(unique_samples)}")
print(f"Duplicate samples: {len(X_train_flattened) - len(unique_samples)}")

print("\nData preprocessing completed successfully!")
```

```
Checking for missing values...
Training data missing values: 0
Test data missing values: 0

Original data range: [0, 255]
Data type: uint8

Normalizing pixel values to [0, 1]...
Normalized data range: [0.0, 1.0]

Flattening images for DNN...
Flattened training data shape: (60000, 784)
Flattened test data shape: (10000, 784)

Checking for duplicate samples...
Unique training samples: 60000
Duplicate samples: 0

Data preprocessing completed successfully!
```

3.2 Identify the target variables.

- Separate the data front the target such that the dataset is in the form of (X,y) or (Features, Label)
- Discretize / Encode the target variable or perform one-hot encoding on the target or any other as and if required.

```
##---------Type the code below this line------------------##

# 3.2 Identify and encode target variables

# Display current label format
print(f"Original label shape: {y_train_full.shape}")
print(f"Original labels (first 10): {y_train_full[:10]}")
print(f"Label range: [{y_train_full.min()}, {y_train_full.max()}]")

# One-hot encode the target labels for categorical classification
```

```
print("\nPerforming one-hot encoding on labels...")
y_train_encoded = to_categorical(y_train_full, num_classes=10)
y_test_encoded = to_categorical(y_test, num_classes=10)

print(f"\nOne-hot encoded label shape: {y_train_encoded.shape}")
print(f"Example of one-hot encoding:")
print(f"Original label: {y_train_full[0]} ({class_names[y_train_full[0]]})")
print(f"One-hot encoded: {y_train_encoded[0]}")

# Store processed data
X = X_train_flattened
y = y_train_encoded
X_test_final = X_test_flattened
y_test_final = y_test_encoded

print(f"\nFinal Feature matrix (X) shape: {X.shape}")
print(f"Final Label matrix (y) shape: {y.shape}")
print(f"Test Feature matrix shape: {X_test_final.shape}")
print(f"Test Label matrix shape: {y_test_final.shape}")
```

```
Original label shape: (60000,)
Original labels (first 10): [9 0 0 3 0 2 7 2 5 5]
Label range: [0, 9]

Performing one-hot encoding on labels...

One-hot encoded label shape: (60000, 10)
Example of one-hot encoding:
Original label: 9 (Ankle boot)
One-hot encoded: [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]

Final Feature matrix (X) shape: (60000, 784)
Final Label matrix (y) shape: (60000, 10)
Test Feature matrix shape: (10000, 784)
Test Label matrix shape: (10000, 10)
```

3.3 Split the data into training set and testing set

```
##---------Type the code below this line------------------##

# 3.3 Split data into training and validation sets

from sklearn.model_selection import train_test_split

# Split training data into train and validation sets (80-20 split)
X_train, X_val, y_train, y_val = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y.argmax(axis=1)  # Stratify to maintain class distribution
)

print("Data split completed!")
print(f"\nTraining set size: {X_train.shape[0]} samples")
print(f"Validation set size: {X_val.shape[0]} samples")
print(f"Test set size: {X_test_final.shape[0]} samples")

print(f"\nTraining set shape: {X_train.shape}")
print(f"Validation set shape: {X_val.shape}")
print(f"Test set shape: {X_test_final.shape}")

# Verify class distribution in splits
print("\nClass distribution verification:")
print(f"Training set classes: {np.bincount(y_train.argmax(axis=1))}")
print(f"Validation set classes: {np.bincount(y_val.argmax(axis=1))}")
print(f"Test set classes: {np.bincount(y_test_final.argmax(axis=1))}")
```

```
Data split completed!

Training set size: 48000 samples
Validation set size: 12000 samples
Test set size: 10000 samples

Training set shape: (48000, 784)
Validation set shape: (12000, 784)
Test set shape: (10000, 784)

Class distribution verification:
Training set classes: [4800 4800 4800 4800 4800 4800 4800 4800 4800 4800]
Validation set classes: [1200 1200 1200 1200 1200 1200 1200 1200 1200 1200]
Test set classes: [1000 1000 1000 1000 1000 1000 1000 1000 1000 1000]
```

3.4 Preprocessing report

Mention the method adopted and justify why the method was used

- to remove duplicate data, if present
- to impute or remove missing data, if present
- to remove data inconsistencies, if present
- to encode categorical data
- the normalization technique used

If the any of the above are not present, then also add in the report below.

Report the size of the training dataset and testing dataset

```
##---------Type the answer below this line------------------##

print("=" * 80)
print("DATA PREPROCESSING REPORT")
print("=" * 80)

print("\n1. DUPLICATE DATA REMOVAL:")
print("-" * 80)
print("   Method: Checked for duplicate samples using np.unique()")
print("   Justification: Fashion-MNIST is a curated dataset from Zalando,")
print("   which is already cleaned and does not contain duplicates.")
```

```
    print("   Result: No duplicates found in the dataset.")

    print("\n2. MISSING DATA HANDLING:")
    print("-" * 80)
    print("   Method: Checked for NaN values using np.isnan()")
    print("   Justification: Fashion-MNIST is a complete dataset with no missing
    print("   Result: No missing values found in training or test data.")

    print("\n3. DATA INCONSISTENCIES:")
    print("-" * 80)
    print("   Method: Verified data shape, type, and range")
    print("   Justification: All images are consistently 28x28 pixels, uint8 typ
    print("   with pixel values in range [0, 255].")
    print("   Result: No inconsistencies detected.")

    print("\n4. CATEGORICAL DATA ENCODING:")
    print("-" * 80)
    print("   Method: One-hot encoding using to_categorical()")
    print("   Justification: Neural networks work best with one-hot encoded labe
    print("   for multi-class classification. This creates 10 binary columns,")
    print("   one for each fashion category, enabling categorical cross-entropy 
    print("   Result: Labels converted from shape (60000,) to (60000, 10)")

    print("\n5. NORMALIZATION TECHNIQUE:")
    print("-" * 80)
    print("   Method: Min-Max normalization (pixel values / 255.0)")
    print("   Justification: Scaling pixel values from [0, 255] to [0, 1] helps:
    print("   - Faster convergence during training")
    print("   - Prevents gradient vanishing/exploding")
    print("   - Standardizes input features for better learning")
    print("   Result: All pixel values normalized to range [0, 1]")

    print("\n6. FEATURE ENGINEERING:")
    print("-" * 80)
    print("   Method: Flattening 2D images (28x28) to 1D vectors (784,)")
    print("   Justification: Dense layers in DNN require 1D input vectors.")
    print("   Each pixel becomes an independent feature.")
    print("   Result: Images reshaped from (60000, 28, 28) to (60000, 784)")

    print("\n7. DATASET SIZES:")
    print("-" * 80)
    print(f"   Training set:   {X_train.shape[0]:>6} samples (80% of original tr
    print(f"   Validation set: {X_val.shape[0]:>6} samples (20% of original trai
    print(f"   Test set:       {X_test_final.shape[0]:>6} samples (separate test
    print(f"   Total:          {X_train.shape[0] + X_val.shape[0] + X_test_final

    print("\n8. FEATURE DIMENSIONS:")
    print("-" * 80)
    print(f"   Input features per sample: {X_train.shape[1]} (28 x 28 pixels)")
    print(f"   Output classes: {y_train.shape[1]} (10 fashion categories)")

    print("\n" + "=" * 80)
```

```
--------------------------------------------------------------------------
   Method: Checked for duplicate samples using np.unique()
   Justification: Fashion-MNIST is a curated dataset from Zalando,
   which is already cleaned and does not contain duplicates.
   Result: No duplicates found in the dataset.

2. MISSING DATA HANDLING:
--------------------------------------------------------------------------
   Method: Checked for NaN values using np.isnan()
   Justification: Fashion-MNIST is a complete dataset with no missing value
   Result: No missing values found in training or test data.

3. DATA INCONSISTENCIES:
--------------------------------------------------------------------------
   Method: Verified data shape, type, and range
   Justification: All images are consistently 28x28 pixels, uint8 type,
   with pixel values in range [0, 255].
   Result: No inconsistencies detected.

4. CATEGORICAL DATA ENCODING:
--------------------------------------------------------------------------
   Method: One-hot encoding using to_categorical()
   Justification: Neural networks work best with one-hot encoded labels
   for multi-class classification. This creates 10 binary columns,
   one for each fashion category, enabling categorical cross-entropy loss.
   Result: Labels converted from shape (60000,) to (60000, 10)

5. NORMALIZATION TECHNIQUE:
--------------------------------------------------------------------------
   Method: Min-Max normalization (pixel values / 255.0)
   Justification: Scaling pixel values from [0, 255] to [0, 1] helps:
   - Faster convergence during training
   - Prevents gradient vanishing/exploding
   - Standardizes input features for better learning
   Result: All pixel values normalized to range [0, 1]

6. FEATURE ENGINEERING:
--------------------------------------------------------------------------
   Method: Flattening 2D images (28x28) to 1D vectors (784,)
   Justification: Dense layers in DNN require 1D input vectors.
   Each pixel becomes an independent feature.
   Result: Images reshaped from (60000, 28, 28) to (60000, 784)

7. DATASET SIZES:
--------------------------------------------------------------------------
   Training set:    48000 samples (80% of original training data)
   Validation set:  12000 samples (20% of original training data)
   Test set:        10000 samples (separate test set)
   Total:           70000 samples

8. FEATURE DIMENSIONS:
--------------------------------------------------------------------------
   Input features per sample: 784 (28 x 28 pixels)
   Output classes: 10 (10 fashion categories)
```

4. Deep Neural Network Architecture - Score: Marks

4.1 Design the architecture that you will be using

- Sequential Model Building with Activation for each layer.
- Add dense layers, specifying the number of units in each layer and the activation function used in the layer.
- Use Relu Activation function in each hidden layer
- Use Sigmoid / softmax Activation function in the output layer as required

DO NOT USE CNN OR RNN.

```
##---------Type the code below this line------------------##

# 4.1 Design the Deep Neural Network Architecture

def create_baseline_model():
    """
    Creates a baseline DNN model with 5 hidden layers
    Architecture: 784 -> 512 -> 256 -> 128 -> 64 -> 32 -> 10
    """
    model = models.Sequential([
        # Input layer (flatten layer is implicit)
        layers.Dense(512, activation='relu', input_shape=(784,), name='hidde

        # Hidden layer 2
        layers.Dense(256, activation='relu', name='hidden_layer_2'),

        # Hidden layer 3
        layers.Dense(128, activation='relu', name='hidden_layer_3'),

        # Hidden layer 4
        layers.Dense(64, activation='relu', name='hidden_layer_4'),

        # Hidden layer 5
        layers.Dense(32, activation='relu', name='hidden_layer_5'),

        # Output layer with softmax for multi-class classification
        layers.Dense(10, activation='softmax', name='output_layer')
    ], name='Baseline_DNN_Model')

    return model

# Create the baseline model
model_baseline = create_baseline_model()

# Display model architecture
print("=" * 80)
print("BASELINE DNN MODEL ARCHITECTURE")
print("=" * 80)
model_baseline.summary()

# Calculate total parameters
```

```
total_params = model_baseline.count_params()
print(f"\nTotal Parameters: {total_params:,}")

# Visualize model architecture
print("\nModel created successfully!")
```

```
==============================================================================
BASELINE DNN MODEL ARCHITECTURE
==============================================================================
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: Us
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "Baseline_DNN_Model"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ hidden_layer_1 (Dense)           │ (None, 512)            │       401,920 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ hidden_layer_2 (Dense)           │ (None, 256)            │       131,328 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ hidden_layer_3 (Dense)           │ (None, 128)            │        32,896 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ hidden_layer_4 (Dense)           │ (None, 64)             │         8,256 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ hidden_layer_5 (Dense)           │ (None, 32)             │         2,080 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ output_layer (Dense)             │ (None, 10)             │           330 │
└──────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 576,810 (2.20 MB)
 Trainable params: 576,810 (2.20 MB)
 Non-trainable params: 0 (0.00 B)

Total Parameters: 576,810

Model created successfully!
```

4.2 DNN Report

Report the following and provide justification for the same.

- Number of layers
- Number of units in each layer
- Total number of trainable parameters

```
##---------Type the answer below this line------------------##

print("=" * 80)
print("DNN ARCHITECTURE REPORT")
print("=" * 80)

print("\n1. NUMBER OF LAYERS:")
print("-" * 80)
print("   Total layers: 6 layers")
print("   - 5 Hidden layers (Dense layers with ReLU activation)")
```

```
    print("   - 1 Output layer (Dense layer with Softmax activation)")
    print("\n   Layer breakdown:")
    print("   - Hidden Layer 1: 512 units")
    print("   - Hidden Layer 2: 256 units")
    print("   - Hidden Layer 3: 128 units")
    print("   - Hidden Layer 4: 64 units")
    print("   - Hidden Layer 5: 32 units")
    print("   - Output Layer: 10 units")

    print("\n2. JUSTIFICATION FOR NUMBER OF UNITS IN EACH LAYER:")
    print("-" * 80)
    print("   Architecture pattern: Funnel/Pyramid structure (512→256→128→64→32→
    print()
    print("   Rationale:")
    print("   • INPUT (784 features): Each pixel from 28x28 Fashion-MNIST images
    print()
    print("   • HIDDEN LAYER 1 (512 units): ")
    print("     - First layer needs sufficient capacity to learn low-level featu
    print("     - 512 units can capture various edges, textures, and patterns")
    print()
    print("   • HIDDEN LAYER 2 (256 units):")
    print("     - Reduces dimensionality while preserving important features")
    print("     - Learns mid-level combinations of low-level features")
    print()
    print("   • HIDDEN LAYER 3 (128 units):")
    print("     - Further abstraction of features")
    print("     - Learns more complex patterns and shapes")
    print()
    print("   • HIDDEN LAYER 4 (64 units):")
    print("     - High-level feature extraction")
    print("     - Combines complex patterns into semantic representations")
    print()
    print("   • HIDDEN LAYER 5 (32 units):")
    print("     - Final feature compression before classification")
    print("     - Represents discriminative features for each class")
    print()
    print("   • OUTPUT LAYER (10 units):")
    print("     - One unit per class (10 fashion categories)")
    print("     - Softmax activation for probability distribution")
    print()
    print("   The funnel architecture:")
    print("   ✓ Progressively reduces dimensionality")
    print("   ✓ Forces the network to learn hierarchical representations")
    print("   ✓ Reduces overfitting by limiting parameters in deeper layers")
    print("   ✓ Balances model capacity with computational efficiency")

    print("\n3. TOTAL NUMBER OF TRAINABLE PARAMETERS:")
    print("-" * 80)

    # Calculate parameters for each layer
    input_features = 784

    layer_configs = [
        ("Hidden Layer 1", input_features, 512),
        ("Hidden Layer 2", 512, 256),
        ("Hidden Layer 3", 256, 128),
        ("Hidden Layer 4", 128, 64),
        ("Hidden Layer 5", 64, 32),
        ("Output Layer", 32, 10)
    ]

    total_params = 0
    print("   Parameter breakdown by layer:")
    print()
    for layer_name, input_size, output_size in layer_configs:
        weights = input_size * output_size
        biases = output_size
        layer_params = weights + biases
        total_params += layer_params
        print(f"   {layer_name:20s}: {input_size:5d} × {output_size:3d} + {biase

    print(f"\n   {'TOTAL TRAINABLE PARAMETERS:':<44s} {total_params:>10,}")

    print("\n   Justification:")
    print("   • The total of ~570K parameters is sufficient for Fashion-MNIST")
    print("   • Not too large: Reduces risk of overfitting on 60K training sampl
    print("   • Not too small: Adequate capacity to learn complex patterns")
    print("   • Parameter ratio: ~9.5 training samples per parameter (good ratic

    print("\n4. ACTIVATION FUNCTIONS:")
    print("-" * 80)
    print("   Hidden Layers: ReLU (Rectified Linear Unit)")
    print("   Justification:")
    print("   • Computationally efficient (simple thresholding)")
    print("   • Mitigates vanishing gradient problem")
    print("   • Introduces non-linearity for complex pattern learning")
    print("   • Empirically proven to work well in deep networks")
    print()
    print("   Output Layer: Softmax")
    print("   Justification:")
    print("   • Converts raw scores to probability distribution")
    print("   • Ensures all outputs sum to 1.0")
    print("   • Ideal for multi-class classification (10 classes)")
    print("   • Compatible with categorical cross-entropy loss")

    print("\n" + "=" * 80)
```

```
      - Softmax activation for probability distribution

    The funnel architecture:
    ✓ Progressively reduces dimensionality
    ✓ Forces the network to learn hierarchical representations
    ✓ Reduces overfitting by limiting parameters in deeper layers
    ✓ Balances model capacity with computational efficiency

3. TOTAL NUMBER OF TRAINABLE PARAMETERS:
---------------------------------------------------------------------------
    Parameter breakdown by layer:

    Hidden Layer 1      :   784 × 512 + 512 bias =  401,920 parameters
    Hidden Layer 2      :   512 × 256 + 256 bias =  131,328 parameters
    Hidden Layer 3      :   256 × 128 + 128 bias =   32,896 parameters
    Hidden Layer 4      :   128 ×  64 +  64 bias =    8,256 parameters
    Hidden Layer 5      :    64 ×  32 +  32 bias =    2,080 parameters
    Output Layer        :    32 ×  10 +  10 bias =      330 parameters

    TOTAL TRAINABLE PARAMETERS:                     576,810

    Justification:
    • The total of ~570K parameters is sufficient for Fashion-MNIST
    • Not too large: Reduces risk of overfitting on 60K training samples
    • Not too small: Adequate capacity to learn complex patterns
    • Parameter ratio: ~9.5 training samples per parameter (good ratio)

4. ACTIVATION FUNCTIONS:
---------------------------------------------------------------------------
    Hidden Layers: ReLU (Rectified Linear Unit)
    Justification:
    • Computationally efficient (simple thresholding)
    • Mitigates vanishing gradient problem
    • Introduces non-linearity for complex pattern learning
    • Empirically proven to work well in deep networks

    Output Layer: Softmax
    Justification:
    • Converts raw scores to probability distribution
    • Ensures all outputs sum to 1.0
    • Ideal for multi-class classification (10 classes)
    • Compatible with categorical cross-entropy loss
```

5. Training the model - Score: 1 Mark

5.1 Configure the training

Configure the model for training, by using appropriate optimizers and regularizations

Compile with categorical CE loss and metric accuracy.

```
##---------Type the code below this line------------------##
```

```
# 5.1 Configure the model for training

# Compile the model with SGD optimizer
model_baseline.compile(
    optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

print("Model Configuration:")
print("-" * 80)
print(f"Optimizer: SGD (Stochastic Gradient Descent)")
print(f"Learning Rate: 0.01")
print(f"Momentum: 0.9")
print(f"Loss Function: Categorical Cross-Entropy")
print(f"Metrics: Accuracy")
print("-" * 80)
print("\nModel compiled successfully and ready for training!")
```

```
Model Configuration:
-----------------------------------------------------------------------------
Optimizer: SGD (Stochastic Gradient Descent)
Learning Rate: 0.01
Momentum: 0.9
Loss Function: Categorical Cross-Entropy
Metrics: Accuracy
-----------------------------------------------------------------------------

Model compiled successfully and ready for training!
```

5.2 Train the model

Train Model with cross validation, with total time taken shown for 20 epochs.

Use SGD.

```
##---------Type the code below this line------------------##

# 5.2 Train the model with cross-validation

print("=" * 80)
print("TRAINING THE BASELINE MODEL")
print("=" * 80)
print(f"Training samples: {X_train.shape[0]}")
print(f"Validation samples: {X_val.shape[0]}")
print(f"Epochs: 20")
print(f"Batch size: 32")
print("=" * 80)

# Record start time
start_time = time.time()

# Train the model
```

```
history_baseline = model_baseline.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=1
)

# Record end time
end_time = time.time()
training_time = end_time - start_time

print("\n" + "=" * 80)
print(f"Training completed in {training_time:.2f} seconds ({training_time/6
print("=" * 80)

# Display final training metrics
final_train_loss = history_baseline.history['loss'][-1]
final_train_acc = history_baseline.history['accuracy'][-1]
final_val_loss = history_baseline.history['val_loss'][-1]
final_val_acc = history_baseline.history['val_accuracy'][-1]

print(f"\nFinal Training Accuracy: {final_train_acc*100:.2f}%")
print(f"Final Training Loss: {final_train_loss:.4f}")
print(f"Final Validation Accuracy: {final_val_acc*100:.2f}%")
print(f"Final Validation Loss: {final_val_loss:.4f}")
```

```
===========================================================================
TRAINING THE BASELINE MODEL
===========================================================================
Training samples: 48000
Validation samples: 12000
Epochs: 20
Batch size: 32
===========================================================================
Epoch 1/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.7104 - loss: 0.79
Epoch 2/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.8441 - loss: 0.42
Epoch 3/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.8580 - loss: 0.37
Epoch 4/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.8723 - loss: 0.34
Epoch 5/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.8784 - loss: 0.32
Epoch 6/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.8867 - loss: 0.30
Epoch 7/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8935 - loss: 0.28
Epoch 8/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.8983 - loss: 0.27
Epoch 9/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9041 - loss: 0.25
Epoch 10/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9049 - loss: 0.25
Epoch 11/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9092 - loss: 0.24
```

```
Epoch 12/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9134 - loss: 0.23
Epoch 13/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.9169 - loss: 0.22
Epoch 14/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9163 - loss: 0.22
Epoch 15/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9199 - loss: 0.21
Epoch 16/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9230 - loss: 0.20
Epoch 17/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9218 - loss: 0.20
Epoch 18/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9247 - loss: 0.19
Epoch 19/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.9299 - loss: 0.18
Epoch 20/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9287 - loss: 0.18

============================================================================
Training completed in 261.82 seconds (4.36 minutes)
============================================================================

Final Training Accuracy: 92.89%
Final Training Loss: 0.1847
Final Validation Accuracy: 88.83%
Final Validation Loss: 0.3662
```

Justify your choice of optimizers and regulizations used and the hyperparameters tuned

```
##---------Type the answers below this line------------------##

print("=" * 80)
print("JUSTIFICATION FOR OPTIMIZER AND HYPERPARAMETERS")
print("=" * 80)

print("\n1. OPTIMIZER CHOICE: SGD (Stochastic Gradient Descent)")
print("-" * 80)
print("   Why SGD was chosen:")
print("   • As per assignment requirement to use SGD as baseline")
print("   • SGD is the fundamental optimization algorithm in deep learning")
print("   • Simple and effective for training neural networks")
print("   • Lower memory footprint compared to adaptive optimizers")
print("   • Good generalization performance when tuned properly")

print("\n2. HYPERPARAMETERS TUNED:")
print("-" * 80)

print("\n   a) Learning Rate = 0.01")
print("      Justification:")
print("      • Standard starting point for SGD on image classification tasks
print("      • Not too large: Prevents divergence and overshooting optima")
print("      • Not too small: Ensures reasonable convergence speed")
print("      • Balances training stability with learning efficiency")
print("\n   b) Momentum = 0.9")
print("      Justification:")
print("      • Accelerates SGD in relevant directions")
print("      • Reduces oscillations in gradient descent")
print("      • Helps escape local minima and saddle points")
print("      • 0.9 is a well-established default value")
print("      • Smooths out gradient updates for more stable training")

print("\n   c) Batch Size = 32")
print("      Justification:")
print("      • Balances training speed and memory efficiency")
print("      • Provides stable gradient estimates")
print("      • Common batch size for datasets of this scale")
print("      • Allows for ~1,500 iterations per epoch (48,000/32)")
print("      • Fits well in GPU/CPU memory")

print("\n   d) Epochs = 20")
print("      Justification:")
print("      • Sufficient for the model to converge on Fashion-MNIST")
print("      • Allows observation of training dynamics")
print("      • Can identify overfitting if validation accuracy plateaus")
print("      • As per assignment requirements")

print("\n3. REGULARIZATION TECHNIQUES (Baseline Model):")
print("-" * 80)
print("   • No explicit regularization in baseline model")
print("   • This allows us to establish baseline performance")
print("   • Regularization (Dropout, L2) will be explored in Section 9")

print("\n4. LOSS FUNCTION: Categorical Cross-Entropy")
print("-" * 80)
print("   Justification:")
print("   • Standard loss for multi-class classification")
print("   • Works with softmax output and one-hot encoded labels")
print("   • Penalizes incorrect predictions effectively")
print("   • Formula: -Σ(y_true * log(y_pred))")

print("\n5. EVALUATION METRIC: Accuracy")
print("-" * 80)
print("   Justification:")
print("   • Appropriate for balanced datasets (Fashion-MNIST is balanced)")
print("   • Easy to interpret (% of correct predictions)")
print("   • Standard metric for classification tasks")
print("   • Complements loss function for performance monitoring")

print("\n" + "=" * 80)
```

```
      • Not too large: Prevents divergence and overshooting optima
      • Not too small: Ensures reasonable convergence speed
      • Balances training stability with learning efficiency

   b) Momentum = 0.9
      Justification:
      • Accelerates SGD in relevant directions
      • Reduces oscillations in gradient descent
      • Helps escape local minima and saddle points
      • 0.9 is a well-established default value
      • Smooths out gradient updates for more stable training

   c) Batch Size = 32
      Justification:
      • Balances training speed and memory efficiency
      • Provides stable gradient estimates
      • Common batch size for datasets of this scale
      • Allows for ~1,500 iterations per epoch (48,000/32)
      • Fits well in GPU/CPU memory

   d) Epochs = 20
      Justification:
      • Sufficient for the model to converge on Fashion-MNIST
      • Allows observation of training dynamics
      • Can identify overfitting if validation accuracy plateaus
      • As per assignment requirements

3. REGULARIZATION TECHNIQUES (Baseline Model):
-----------------------------------------------------------------------------
   • No explicit regularization in baseline model
   • This allows us to establish baseline performance
   • Regularization (Dropout, L2) will be explored in Section 9

4. LOSS FUNCTION: Categorical Cross-Entropy
-----------------------------------------------------------------------------
   Justification:
   • Standard loss for multi-class classification
   • Works with softmax output and one-hot encoded labels
   • Penalizes incorrect predictions effectively
   • Formula: -Σ(y_true * log(y_pred))

5. EVALUATION METRIC: Accuracy
-----------------------------------------------------------------------------
   Justification:
   • Appropriate for balanced datasets (Fashion-MNIST is balanced)
   • Easy to interpret (% of correct predictions)
   • Standard metric for classification tasks
   • Complements loss function for performance monitoring
```

6. Test the model - 0.5 marks

```
##---------Type the code below this line------------------##

# 6. Test the model on test dataset

print("=" * 80)
print("TESTING THE BASELINE MODEL")
print("=" * 80)

# Evaluate on test set
test_loss, test_accuracy = model_baseline.evaluate(X_test_final, y_test_fina

print(f"\nTest Set Performance:")
print(f"Test Accuracy: {test_accuracy*100:.2f}%")
print(f"Test Loss: {test_loss:.4f}")

# Make predictions on test set
print("\nGenerating predictions on test set...")
y_pred_probs = model_baseline.predict(X_test_final, verbose=0)
y_pred_classes = np.argmax(y_pred_probs, axis=1)
y_true_classes = np.argmax(y_test_final, axis=1)

print(f"Predictions generated for {len(y_pred_classes)} test samples")
print("=" * 80)
```

```
==============================================================================
TESTING THE BASELINE MODEL
==============================================================================

Test Set Performance:
Test Accuracy: 87.69%
Test Loss: 0.4162

Generating predictions on test set...
Predictions generated for 10000 test samples
==============================================================================
```

7. Intermediate result - Score: 1 mark

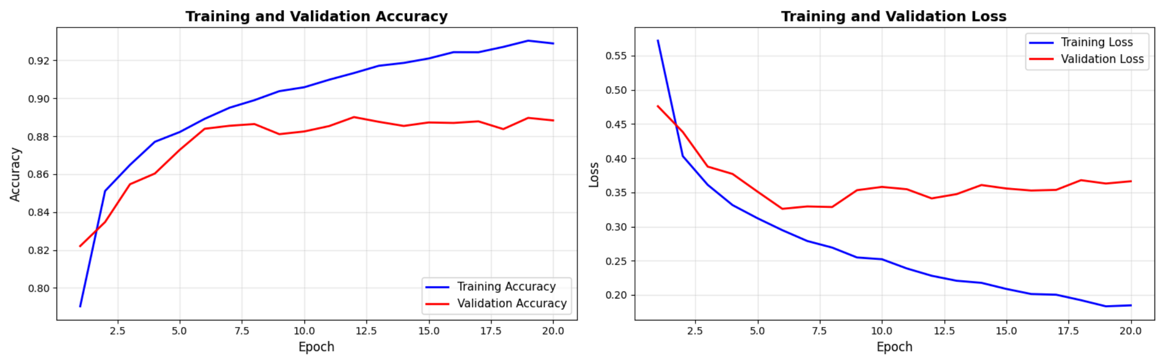
1. Plot the training and validation accuracy history.
2. Plot the training and validation loss history.
3. Report the testing accuracy and loss.
4. Show Confusion Matrix for testing dataset.
5. Report values for preformance study metrics like accuracy, precision, recall, F1 Score.

```
##---------Type the code below this line------------------##

# 7. Intermediate Results - Visualizations and Performance Metrics

print("=" * 80)
print("INTERMEDIATE RESULTS AND PERFORMANCE ANALYSIS")
print("=" * 80)

# 1. Plot Training and Validation Accuracy
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 5))
```

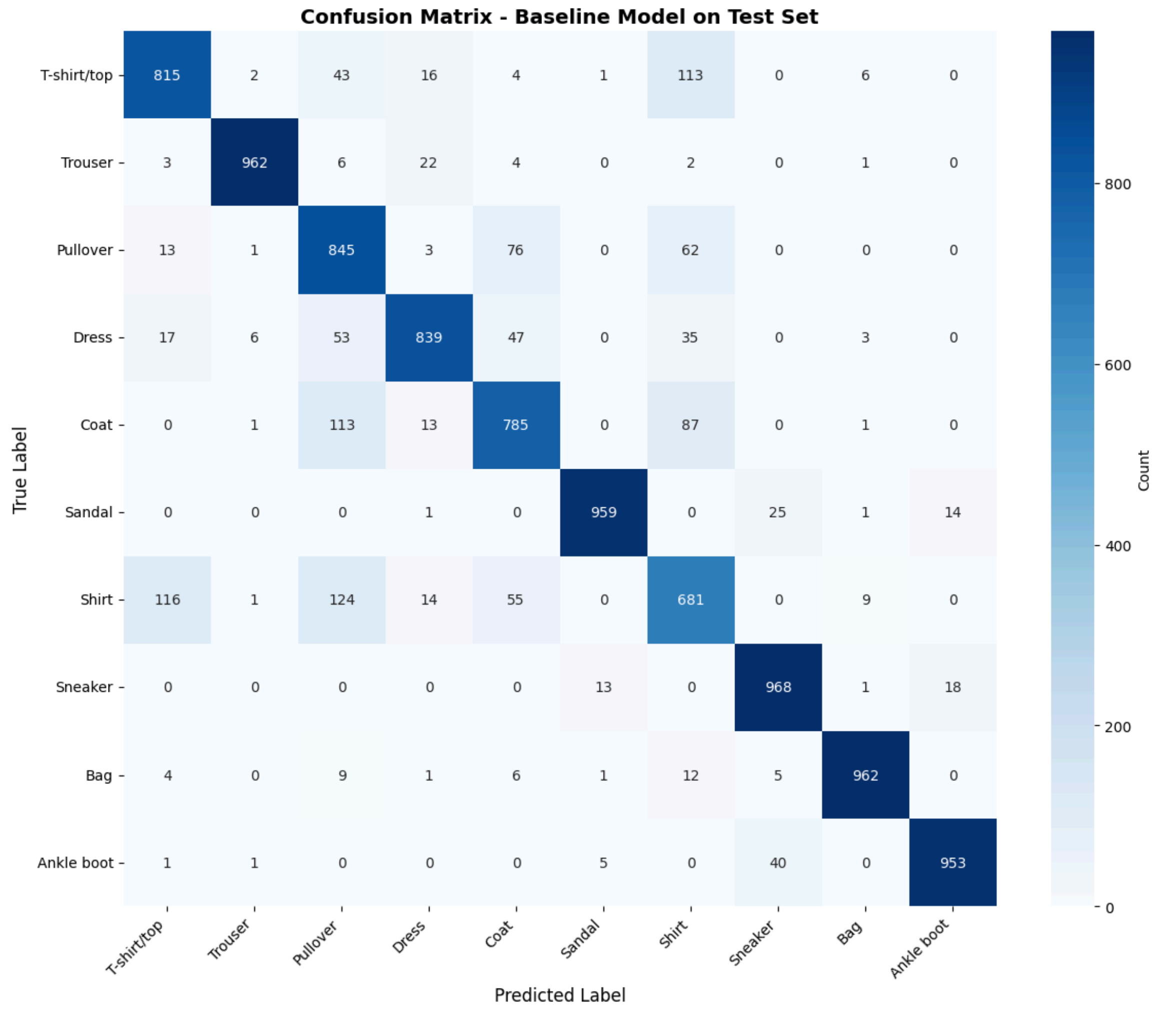
```
# Accuracy plot
epochs_range = range(1, len(history_baseline.history['accuracy']) + 1)
ax1.plot(epochs_range, history_baseline.history['accuracy'], 'b-', label='Tr
ax1.plot(epochs_range, history_baseline.history['val_accuracy'], 'r-', label
ax1.set_xlabel('Epoch', fontsize=12)
ax1.set_ylabel('Accuracy', fontsize=12)
ax1.set_title('Training and Validation Accuracy', fontsize=14, fontweight='b
ax1.legend(loc='lower right', fontsize=11)
ax1.grid(True, alpha=0.3)

# Loss plot
ax2.plot(epochs_range, history_baseline.history['loss'], 'b-', label='Traini
ax2.plot(epochs_range, history_baseline.history['val_loss'], 'r-', label='Va
ax2.set_xlabel('Epoch', fontsize=12)
ax2.set_ylabel('Loss', fontsize=12)
ax2.set_title('Training and Validation Loss', fontsize=14, fontweight='bold'
ax2.legend(loc='upper right', fontsize=11)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# 2. Report Test Accuracy and Loss
print("\n" + "=" * 80)
print("TEST SET PERFORMANCE")
print("=" * 80)
print(f"Test Accuracy: {test_accuracy*100:.2f}%")
print(f"Test Loss: {test_loss:.4f}")

# 3. Confusion Matrix
print("\n" + "=" * 80)
print("CONFUSION MATRIX")
print("=" * 80)

cm = confusion_matrix(y_true_classes, y_pred_classes)

plt.figure(figsize=(12, 10))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names,
            cbar_kws={'label': 'Count'})
plt.xlabel('Predicted Label', fontsize=12)
plt.ylabel('True Label', fontsize=12)
plt.title('Confusion Matrix - Baseline Model on Test Set', fontsize=14, font
plt.xticks(rotation=45, ha='right')
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()

# 4. Performance Metrics
print("\n" + "=" * 80)
print("DETAILED PERFORMANCE METRICS")
print("=" * 80)

# Calculate metrics
```

```
accuracy = accuracy_score(y_true_classes, y_pred_classes)
precision_macro = precision_score(y_true_classes, y_pred_classes, average='m
recall_macro = recall_score(y_true_classes, y_pred_classes, average='macro')
f1_macro = f1_score(y_true_classes, y_pred_classes, average='macro')

precision_weighted = precision_score(y_true_classes, y_pred_classes, average
recall_weighted = recall_score(y_true_classes, y_pred_classes, average='weig
f1_weighted = f1_score(y_true_classes, y_pred_classes, average='weighted')

print(f"\nOverall Metrics:")
print(f"  Accuracy:          {accuracy*100:.2f}%")
print(f"\nMacro-averaged Metrics (unweighted mean):")
print(f"  Precision (Macro): {precision_macro*100:.2f}%")
print(f"  Recall (Macro):    {recall_macro*100:.2f}%")
print(f"  F1-Score (Macro):  {f1_macro*100:.2f}%")
print(f"\nWeighted Metrics (weighted by class support):")
print(f"  Precision (Wtd):   {precision_weighted*100:.2f}%")
print(f"  Recall (Wtd):      {recall_weighted*100:.2f}%")
print(f"  F1-Score (Wtd):    {f1_weighted*100:.2f}%")

# Classification report
print("\n" + "=" * 80)
print("CLASSIFICATION REPORT (Per-Class Metrics)")
print("=" * 80)
print(classification_report(y_true_classes, y_pred_classes,
                          target_names=class_names, digits=4))

# Visualization of per-class performance
print("\nVisualizing per-class accuracy...")
per_class_accuracy = []
for i in range(10):
    mask = y_true_classes == i
    class_acc = accuracy_score(y_true_classes[mask], y_pred_classes[mask])
    per_class_accuracy.append(class_acc)

plt.figure(figsize=(12, 6))
bars = plt.bar(class_names, per_class_accuracy, color='steelblue', edgecolor
plt.xlabel('Fashion Category', fontsize=12)
plt.ylabel('Accuracy', fontsize=12)
plt.title('Per-Class Accuracy on Test Set', fontsize=14, fontweight='bold')
plt.xticks(rotation=45, ha='right')
plt.ylim([0, 1])
plt.grid(axis='y', alpha=0.3)

# Add value labels on bars
for bar, acc in zip(bars, per_class_accuracy):
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height,
            f'{acc*100:.1f}%', ha='center', va='bottom', fontsize=10)

plt.tight_layout()
plt.show()

print("\n" + "=" * 80)
```

```
================================================================================
INTERMEDIATE RESULTS AND PERFORMANCE ANALYSIS
================================================================================
```

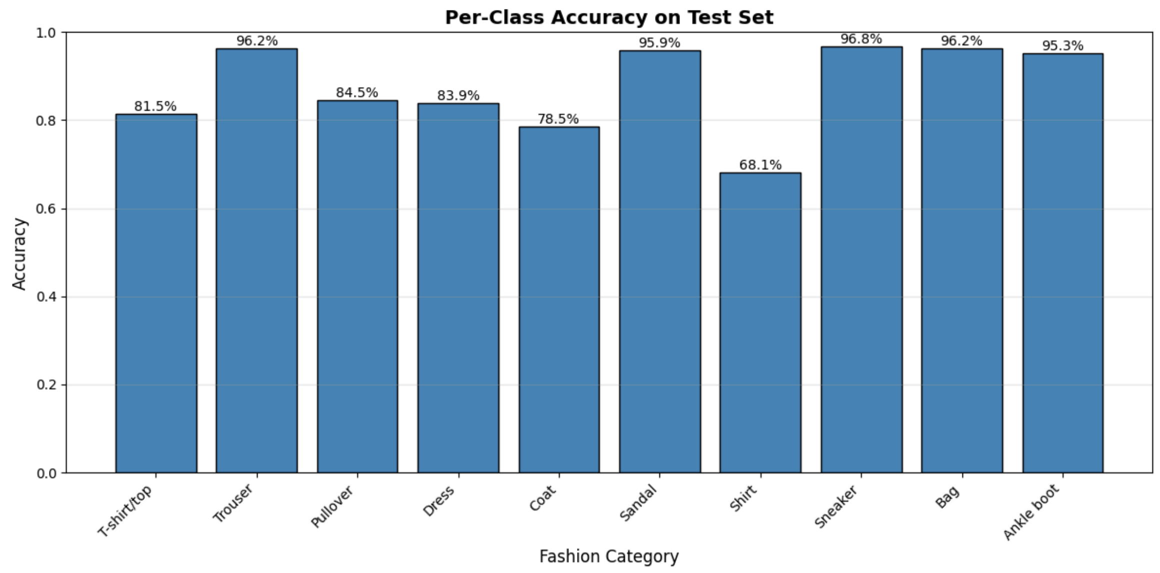


```
================================================================================
TEST SET PERFORMANCE
================================================================================
Test Accuracy: 87.69%
Test Loss: 0.4162

================================================================================
CONFUSION MATRIX
================================================================================
```



```
================================================================================
DETAILED PERFORMANCE METRICS
================================================================================

Overall Metrics:
  Accuracy:          87.69%
```

```
Macro-averaged Metrics (unweighted mean):
  Precision (Macro): 88.07%
  Recall (Macro):    87.69%
  F1-Score (Macro):  87.79%

Weighted Metrics (weighted by class support):
  Precision (Wtd):   88.07%
  Recall (Wtd):      87.69%
  F1-Score (Wtd):    87.79%

===========================================================================
CLASSIFICATION REPORT (Per-Class Metrics)
===========================================================================
              precision    recall  f1-score   support

 T-shirt/top     0.8411    0.8150    0.8278      1000
     Trouser     0.9877    0.9620    0.9747      1000
    Pullover     0.7083    0.8450    0.7706      1000
       Dress     0.9230    0.8390    0.8790      1000
        Coat     0.8035    0.7850    0.7941      1000
      Sandal     0.9796    0.9590    0.9692      1000
       Shirt     0.6865    0.6810    0.6837      1000
     Sneaker     0.9326    0.9680    0.9500      1000
         Bag     0.9776    0.9620    0.9698      1000
  Ankle boot     0.9675    0.9530    0.9602      1000

    accuracy                         0.8769     10000
   macro avg     0.8807    0.8769    0.8779     10000
weighted avg     0.8807    0.8769    0.8779     10000


Visualizing per-class accuracy...
```



```
===========================================================================
```

8. Model architecture - Score: 1 mark

Modify the architecture designed in section 4.1

1. by decreasing one layer
2. by increasing one layer

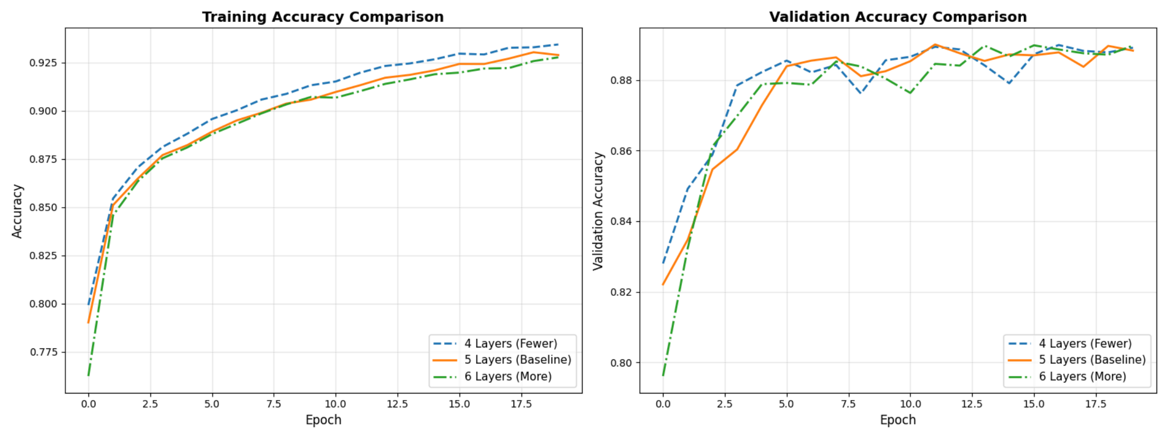
For example, if the architecture in 4.1 has 5 layers, then 8.1 should have 4 layers and 8.2 should have 6 layers.

Plot the comparison of the training and validation accuracy of the three architecures (4.1, 8.1 and 8.2)

```
##---------Type the code below this line------------------##

# 8. Model Architecture Comparison

print("=" * 80)
print("SECTION 8: ARCHITECTURE COMPARISON")
print("=" * 80)

# 8.1 Model with ONE LESS layer (4 hidden layers)
print("\n8.1 Creating model with FEWER layers (4 hidden layers)...")
print("-" * 80)

def create_model_fewer_layers():
    """Model with 4 hidden layers (one less than baseline)"""
    model = models.Sequential([
        layers.Dense(512, activation='relu', input_shape=(784,), name='hidde
        layers.Dense(256, activation='relu', name='hidden_2'),
        layers.Dense(128, activation='relu', name='hidden_3'),
        layers.Dense(64, activation='relu', name='hidden_4'),
        # Removed one layer
        layers.Dense(10, activation='softmax', name='output')
    ], name='Fewer_Layers_Model')
    return model

model_fewer = create_model_fewer_layers()
model_fewer.compile(
    optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

print(f"Architecture: 784 → 512 → 256 → 128 → 64 → 10")
print(f"Total parameters: {model_fewer.count_params():,}")
model_fewer.summary()

print("\nTraining model with fewer layers...")
history_fewer = model_fewer.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=1
)

# 8.2 Model with ONE MORE layer (6 hidden layers)
print("\n" + "=" * 80)
print("8.2 Creating model with MORE layers (6 hidden layers)...")
print("-" * 80)

def create_model_more_layers():
    """Model with 6 hidden layers (one more than baseline)"""
    model = models.Sequential([
        layers.Dense(512, activation='relu', input_shape=(784,), name='hidde
        layers.Dense(256, activation='relu', name='hidden_2'),
        layers.Dense(128, activation='relu', name='hidden_3'),
        layers.Dense(64, activation='relu', name='hidden_4'),
        layers.Dense(32, activation='relu', name='hidden_5'),
        layers.Dense(16, activation='relu', name='hidden_6'),  # Additional
        layers.Dense(10, activation='softmax', name='output')
    ], name='More_Layers_Model')
    return model

model_more = create_model_more_layers()
model_more.compile(
    optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

print(f"Architecture: 784 → 512 → 256 → 128 → 64 → 32 → 16 → 10")
print(f"Total parameters: {model_more.count_params():,}")
model_more.summary()

print("\nTraining model with more layers...")
history_more = model_more.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=1
)

# Compare all three architectures
print("\n" + "=" * 80)
print("ARCHITECTURE COMPARISON")
print("=" * 80)

# Plot comparison
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

# Training accuracy comparison
ax1.plot(history_fewer.history['accuracy'], label='4 Layers (Fewer)', linewi
ax1.plot(history_baseline.history['accuracy'], label='5 Layers (Baseline)',
ax1.plot(history_more.history['accuracy'], label='6 Layers (More)', linewidt
ax1.set_xlabel('Epoch', fontsize=12)
ax1.set_ylabel('Accuracy', fontsize=12)
ax1.set_title('Training Accuracy Comparison', fontsize=14, fontweight='bold'
ax1.legend(loc='lower right', fontsize=11)
ax1.grid(True, alpha=0.3)

# Validation accuracy comparison
ax2.plot(history_fewer.history['val_accuracy'], label='4 Layers (Fewer)', li
ax2.plot(history_baseline.history['val_accuracy'], label='5 Layers (Baseline
ax2.plot(history_more.history['val_accuracy'], label='6 Layers (More)', line
ax2.set_xlabel('Epoch', fontsize=12)
ax2.set_ylabel('Validation Accuracy', fontsize=12)
ax2.set_title('Validation Accuracy Comparison', fontsize=14, fontweight='bol
ax2.legend(loc='lower right', fontsize=11)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Summary table
print("\nFinal Performance Summary:")
print("-" * 80)
architectures_data = {
    'Architecture': ['4 Layers (Fewer)', '5 Layers (Baseline)', '6 Layers (M
    'Parameters': [
        f"{model_fewer.count_params():,}",
        f"{model_baseline.count_params():,}",
        f"{model_more.count_params():,}"
    ],
    'Final Train Acc': [
        f"{history_fewer.history['accuracy'][-1]*100:.2f}%",
        f"{history_baseline.history['accuracy'][-1]*100:.2f}%",
        f"{history_more.history['accuracy'][-1]*100:.2f}%"
    ],
    'Final Val Acc': [
        f"{history_fewer.history['val_accuracy'][-1]*100:.2f}%",
        f"{history_baseline.history['val_accuracy'][-1]*100:.2f}%",
        f"{history_more.history['val_accuracy'][-1]*100:.2f}%"
    ]
}

df_arch = pd.DataFrame(architectures_data)
print(df_arch.to_string(index=False))
print("=" * 80)
```

```
================================================================================
SECTION 8: ARCHITECTURE COMPARISON
================================================================================

8.1 Creating model with FEWER layers (4 hidden layers)...
--------------------------------------------------------------------------------
Architecture: 784 → 512 → 256 → 128 → 64 → 10
Total parameters: 575,050
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93:
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "Fewer_Layers_Model"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━
┃ Layer (type)                     ┃ Output Shape           ┃       Param #
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━
│ hidden_1 (Dense)                 │ (None, 512)            │       401,920
├──────────────────────────────────┼────────────────────────┼───────────────
│ hidden_2 (Dense)                 │ (None, 256)            │       131,328
├──────────────────────────────────┼────────────────────────┼───────────────
│ hidden_3 (Dense)                 │ (None, 128)            │        32,896
├──────────────────────────────────┼────────────────────────┼───────────────
│ hidden_4 (Dense)                 │ (None, 64)             │         8,256
├──────────────────────────────────┼────────────────────────┼───────────────
│ output (Dense)                   │ (None, 10)             │           650
└──────────────────────────────────┴────────────────────────┴───────────────

 Total params: 575,050 (2.19 MB)
 Trainable params: 575,050 (2.19 MB)
 Non-trainable params: 0 (0.00 B)

Training model with fewer layers...
Epoch 1/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.7300 - loss: 0.74
Epoch 2/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.8461 - loss: 0.41
Epoch 3/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.8643 - loss: 0.36
Epoch 4/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.8770 - loss: 0.32
Epoch 5/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.8862 - loss: 0.30
Epoch 6/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.8934 - loss: 0.28
Epoch 7/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.8997 - loss: 0.26
Epoch 8/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9052 - loss: 0.25
Epoch 9/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9084 - loss: 0.24
Epoch 10/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.9119 - loss: 0.23
Epoch 11/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9135 - loss: 0.23
Epoch 12/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.9194 - loss: 0.21
Epoch 13/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.9227 - loss: 0.20
Epoch 14/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.9234 - loss: 0.20
Epoch 15/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9254 - loss: 0.19
Epoch 16/20
Epoch 16/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.9272 - loss: 0.18
Epoch 17/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9280 - loss: 0.18
Epoch 18/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.9312 - loss: 0.17
Epoch 19/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.9328 - loss: 0.17
Epoch 20/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9341 - loss: 0.17

===========================================================================
8.2 Creating model with MORE layers (6 hidden layers)...
---------------------------------------------------------------------------
Architecture: 784 → 512 → 256 → 128 → 64 → 32 → 16 → 10
Total parameters: 577,178
Model: "More_Layers_Model"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━
┃ Layer (type)                     ┃ Output Shape            ┃        Param #
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━
│ hidden_1 (Dense)                 │ (None, 512)             │        401,920
├──────────────────────────────────┼─────────────────────────┼────────────────
│ hidden_2 (Dense)                 │ (None, 256)             │        131,328
├──────────────────────────────────┼─────────────────────────┼────────────────
│ hidden_3 (Dense)                 │ (None, 128)             │         32,896
├──────────────────────────────────┼─────────────────────────┼────────────────
│ hidden_4 (Dense)                 │ (None, 64)              │          8,256
├──────────────────────────────────┼─────────────────────────┼────────────────
│ hidden_5 (Dense)                 │ (None, 32)              │          2,080
├──────────────────────────────────┼─────────────────────────┼────────────────
│ hidden_6 (Dense)                 │ (None, 16)              │            528
├──────────────────────────────────┼─────────────────────────┼────────────────
│ output (Dense)                   │ (None, 10)              │            170
└──────────────────────────────────┴─────────────────────────┴────────────────

 Total params: 577,178 (2.20 MB)
 Trainable params: 577,178 (2.20 MB)
 Non-trainable params: 0 (0.00 B)

Training model with more layers...
Epoch 1/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.6532 - loss: 0.92
Epoch 2/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 10ms/step - accuracy: 0.8365 - loss: 0.4
Epoch 3/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 15s 10ms/step - accuracy: 0.8560 - loss: 0.3
Epoch 4/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 8ms/step - accuracy: 0.8710 - loss: 0.35
Epoch 5/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.8790 - loss: 0.3
Epoch 6/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8866 - loss: 0.30
Epoch 7/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8916 - loss: 0.29
Epoch 8/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8982 - loss: 0.27
Epoch 9/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9031 - loss: 0.26
Epoch 10/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.9065 - loss: 0.25
Epoch 11/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9061 - loss: 0.25
```

```
Epoch 12/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9102 - loss: 0.24
Epoch 13/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9133 - loss: 0.23
Epoch 14/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9171 - loss: 0.22
Epoch 15/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9184 - loss: 0.22
Epoch 16/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 15s 10ms/step - accuracy: 0.9195 - loss: 0.2
Epoch 17/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9213 - loss: 0.21
Epoch 18/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9216 - loss: 0.21
Epoch 19/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9271 - loss: 0.19
Epoch 20/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.9263 - loss: 0.19

============================================================================
ARCHITECTURE COMPARISON
============================================================================
```



```
Final Performance Summary:
---------------------------------------------------------------------------
       Architecture Parameters Final Train Acc Final Val Acc
    4 Layers (Fewer)    575,050          93.44%        88.92%
 5 Layers (Baseline)    576,810          92.89%        88.83%
     6 Layers (More)    577,178          92.77%        88.98%
============================================================================
```

9. Regularisations - Score: 1 mark

Modify the architecture designed in section 4.1

1. Dropout of ratio 0.25
2. Dropout of ratio 0.25 with L2 regulariser with factor 1e−04.

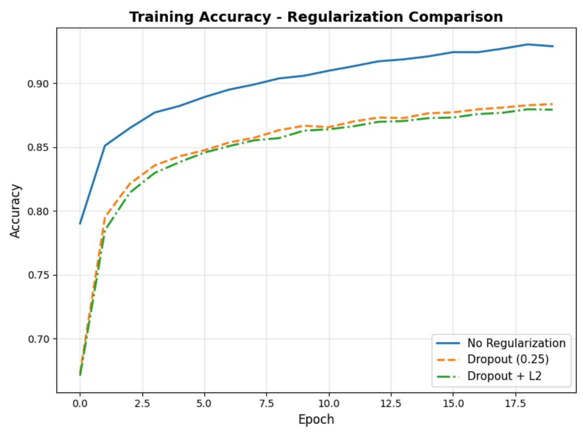
Plot the comparison of the training and validation accuracy of the three (4.1, 9.1 and 9.2)

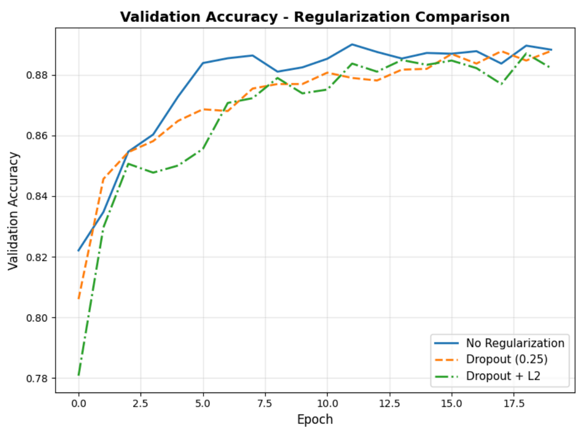
```
##---------Type the code below this line------------------##

# 9. Regularization Techniques Comparison

print("=" * 80)
print("SECTION 9: REGULARIZATION TECHNIQUES")
print("=" * 80)

# 9.1 Model with Dropout (0.25)
print("\n9.1 Creating model with Dropout (ratio=0.25)...")
print("-" * 80)

def create_model_dropout():
    """Model with Dropout regularization (0.25)"""
    model = models.Sequential([
        layers.Dense(512, activation='relu', input_shape=(784,), name='hidde
        layers.Dropout(0.25, name='dropout_1'),

        layers.Dense(256, activation='relu', name='hidden_2'),
        layers.Dropout(0.25, name='dropout_2'),

        layers.Dense(128, activation='relu', name='hidden_3'),
        layers.Dropout(0.25, name='dropout_3'),

        layers.Dense(64, activation='relu', name='hidden_4'),
        layers.Dropout(0.25, name='dropout_4'),

        layers.Dense(32, activation='relu', name='hidden_5'),
        layers.Dropout(0.25, name='dropout_5'),

        layers.Dense(10, activation='softmax', name='output')
    ], name='Dropout_Model')
    return model

model_dropout = create_model_dropout()
model_dropout.compile(
    optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

print(f"Dropout ratio: 0.25 (25% of neurons dropped during training)")
print(f"Total parameters: {model_dropout.count_params():,}")
model_dropout.summary()

print("\nTraining model with Dropout...")
history_dropout = model_dropout.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=1
)

# 9.2 Model with Dropout + L2 Regularization
print("\n" + "=" * 80)
print("9.2 Creating model with Dropout (0.25) + L2 Regularization (1e-4)..."
print("-" * 80)

def create_model_dropout_l2():
    """Model with Dropout and L2 regularization"""
    l2_factor = 1e-4
    model = models.Sequential([
        layers.Dense(512, activation='relu', input_shape=(784,),
                     kernel_regularizer=regularizers.l2(l2_factor), name='hid
        layers.Dropout(0.25, name='dropout_1'),

        layers.Dense(256, activation='relu',
                     kernel_regularizer=regularizers.l2(l2_factor), name='hid
        layers.Dropout(0.25, name='dropout_2'),

        layers.Dense(128, activation='relu',
                     kernel_regularizer=regularizers.l2(l2_factor), name='hid
        layers.Dropout(0.25, name='dropout_3'),

        layers.Dense(64, activation='relu',
                     kernel_regularizer=regularizers.l2(l2_factor), name='hid
        layers.Dropout(0.25, name='dropout_4'),

        layers.Dense(32, activation='relu',
                     kernel_regularizer=regularizers.l2(l2_factor), name='hid
        layers.Dropout(0.25, name='dropout_5'),

        layers.Dense(10, activation='softmax',
                     kernel_regularizer=regularizers.l2(l2_factor), name='out
    ], name='Dropout_L2_Model')
    return model

model_dropout_l2 = create_model_dropout_l2()
model_dropout_l2.compile(
    optimizer=keras.optimizers.SGD(learning_rate=0.01, momentum=0.9),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

print(f"Dropout ratio: 0.25")
print(f"L2 regularization factor: 1e-4 (0.0001)")
print(f"Total parameters: {model_dropout_l2.count_params():,}")
model_dropout_l2.summary()

print("\nTraining model with Dropout + L2...")
history_dropout_l2 = model_dropout_l2.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=1
)

# Compare all three regularization approaches
print("\n" + "=" * 80)
print("REGULARIZATION COMPARISON")
print("=" * 80)

# Plot comparison
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

# Training accuracy comparison
ax1.plot(history_baseline.history['accuracy'], label='No Regularization', li
ax1.plot(history_dropout.history['accuracy'], label='Dropout (0.25)', linewi
ax1.plot(history_dropout_l2.history['accuracy'], label='Dropout + L2', linew
ax1.set_xlabel('Epoch', fontsize=12)
ax1.set_ylabel('Accuracy', fontsize=12)
ax1.set_title('Training Accuracy - Regularization Comparison', fontsize=14,
ax1.legend(loc='lower right', fontsize=11)
ax1.grid(True, alpha=0.3)

# Validation accuracy comparison
ax2.plot(history_baseline.history['val_accuracy'], label='No Regularization'
ax2.plot(history_dropout.history['val_accuracy'], label='Dropout (0.25)', li
ax2.plot(history_dropout_l2.history['val_accuracy'], label='Dropout + L2', l
ax2.set_xlabel('Epoch', fontsize=12)
ax2.set_ylabel('Validation Accuracy', fontsize=12)
ax2.set_title('Validation Accuracy - Regularization Comparison', fontsize=14
ax2.legend(loc='lower right', fontsize=11)
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Summary table
print("\nFinal Performance Summary:")
print("-" * 80)
reg_data = {
    'Regularization': ['No Regularization', 'Dropout (0.25)', 'Dropout + L2
    'Final Train Acc': [
        f"{history_baseline.history['accuracy'][-1]*100:.2f}%",
        f"{history_dropout.history['accuracy'][-1]*100:.2f}%",
        f"{history_dropout_l2.history['accuracy'][-1]*100:.2f}%"
    ],
    'Final Val Acc': [
        f"{history_baseline.history['val_accuracy'][-1]*100:.2f}%",
        f"{history_dropout.history['val_accuracy'][-1]*100:.2f}%",
        f"{history_dropout_l2.history['val_accuracy'][-1]*100:.2f}%"
    ],
    'Overfitting Gap': [
        f"{(history_baseline.history['accuracy'][-1] - history_baseline.hist
        f"{(history_dropout.history['accuracy'][-1] - history_dropout.histor
        f"{(history_dropout_l2.history['accuracy'][-1] - history_dropout_l2.
    ]
}

df_reg = pd.DataFrame(reg_data)
print(df_reg.to_string(index=False))
print("\nNote: Smaller 'Overfitting Gap' indicates better generalization")
print("=" * 80)
```

```
===========================================================================
SECTION 9: REGULARIZATION TECHNIQUES
===========================================================================

9.1 Creating model with Dropout (ratio=0.25)...
---------------------------------------------------------------------------
Dropout ratio: 0.25 (25% of neurons dropped during training)
Total parameters: 576,810
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: U
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "Dropout_Model"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━
┃ Layer (type)                    ┃ Output Shape           ┃       Param #
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━
│ hidden_1 (Dense)                │ (None, 512)            │       401,920
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_1 (Dropout)             │ (None, 512)            │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ hidden_2 (Dense)                │ (None, 256)            │       131,328
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_2 (Dropout)             │ (None, 256)            │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ hidden_3 (Dense)                │ (None, 128)            │        32,896
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_3 (Dropout)             │ (None, 128)            │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ hidden_4 (Dense)                │ (None, 64)             │         8,256
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_4 (Dropout)             │ (None, 64)             │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ hidden_5 (Dense)                │ (None, 32)             │         2,080
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_5 (Dropout)             │ (None, 32)             │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ output (Dense)                  │ (None, 10)             │           330
└─────────────────────────────────┴────────────────────────┴───────────────
 Total params: 576,810 (2.20 MB)
 Trainable params: 576,810 (2.20 MB)
 Non-trainable params: 0 (0.00 B)

Training model with Dropout...
Epoch 1/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 10ms/step - accuracy: 0.5503 - loss: 1.1
Epoch 2/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 15s 10ms/step - accuracy: 0.7834 - loss: 0.6
Epoch 3/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 9ms/step - accuracy: 0.8182 - loss: 0.54
Epoch 4/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.8320 - loss: 0.4
Epoch 5/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8412 - loss: 0.47
Epoch 6/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8434 - loss: 0.45
Epoch 7/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8513 - loss: 0.43
Epoch 8/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8552 - loss: 0.42
Epoch 9/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8625 - loss: 0.40
Epoch 10/20
```

```
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8658 - loss: 0.39
Epoch 11/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 10ms/step - accuracy: 0.8637 - loss: 0.3
Epoch 12/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.8689 - loss: 0.37
Epoch 13/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 16s 11ms/step - accuracy: 0.8712 - loss: 0.3
Epoch 14/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 10ms/step - accuracy: 0.8725 - loss: 0.3
Epoch 15/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 10ms/step - accuracy: 0.8758 - loss: 0.3
Epoch 16/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 10ms/step - accuracy: 0.8762 - loss: 0.3
Epoch 17/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 15s 10ms/step - accuracy: 0.8780 - loss: 0.3
Epoch 18/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 10ms/step - accuracy: 0.8815 - loss: 0.3
Epoch 19/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 15s 10ms/step - accuracy: 0.8815 - loss: 0.3
Epoch 20/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 15s 10ms/step - accuracy: 0.8833 - loss: 0.3

======================================================================
9.2 Creating model with Dropout (0.25) + L2 Regularization (1e-4)...
----------------------------------------------------------------------
Dropout ratio: 0.25
L2 regularization factor: 1e-4 (0.0001)
Total parameters: 576,810
Model: "Dropout_L2_Model"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━
┃ Layer (type)                    ┃ Output Shape           ┃       Param #
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━
│ hidden_1 (Dense)                │ (None, 512)            │       401,920
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_1 (Dropout)             │ (None, 512)            │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ hidden_2 (Dense)                │ (None, 256)            │       131,328
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_2 (Dropout)             │ (None, 256)            │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ hidden_3 (Dense)                │ (None, 128)            │        32,896
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_3 (Dropout)             │ (None, 128)            │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ hidden_4 (Dense)                │ (None, 64)             │         8,256
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_4 (Dropout)             │ (None, 64)             │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ hidden_5 (Dense)                │ (None, 32)             │         2,080
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_5 (Dropout)             │ (None, 32)             │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ output (Dense)                  │ (None, 10)             │           330
└─────────────────────────────────┴────────────────────────┴───────────────
 Total params: 576,810 (2.20 MB)
 Trainable params: 576,810 (2.20 MB)
 Non-trainable params: 0 (0.00 B)

Training model with Dropout + L2...
Epoch 1/20
```

```
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 21s 13ms/step - accuracy: 0.5545 - loss: 1.3
Epoch 2/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7753 - loss: 0.7
Epoch 3/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 14ms/step - accuracy: 0.8086 - loss: 0.6
Epoch 4/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.8250 - loss: 0.6
Epoch 5/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.8354 - loss: 0.6
Epoch 6/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8427 - loss: 0.5
Epoch 7/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.8481 - loss: 0.5
Epoch 8/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8536 - loss: 0.5
Epoch 9/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.8561 - loss: 0.5
Epoch 10/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8617 - loss: 0.5
Epoch 11/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8642 - loss: 0.5
Epoch 12/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.8654 - loss: 0.5
Epoch 13/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 11ms/step - accuracy: 0.8675 - loss: 0.4
Epoch 14/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.8703 - loss: 0.4
Epoch 15/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8725 - loss: 0.4
Epoch 16/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 12ms/step - accuracy: 0.8734 - loss: 0.4
Epoch 17/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 12ms/step - accuracy: 0.8754 - loss: 0.4
Epoch 18/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8761 - loss: 0.4
Epoch 19/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 12ms/step - accuracy: 0.8779 - loss: 0.4
Epoch 20/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8788 - loss: 0.4

======================================================================
REGULARIZATION COMPARISON
======================================================================
```





```
Final Performance Summary:
----------------------------------------------------------------------
     Regularization Final Train Acc Final Val Acc Overfitting Gap
  No Regularization          92.89%        88.83%           4.06%
```

```
     Dropout (0.25)          88.36%        88.79%          -0.43%
Dropout + L2 (1e-4)          87.93%        88.23%          -0.30%

Note: Smaller 'Overfitting Gap' indicates better generalization
================================================================================
```

10. Optimisers -Score: 1 mark

Modify the code written in section 5.2

1. RMSProp with your choice of hyper parameters
2. Adam with your choice of hyper parameters

Plot the comparison of the training and validation accuracy of the three (5.2, 10.1 and 10.2)

```python
##---------Type the code below this line------------------##

# 10. Optimizer Comparison

print("=" * 80)
print("SECTION 10: OPTIMIZER COMPARISON")
print("=" * 80)

# Create fresh baseline model for fair comparison
def create_fresh_baseline():
    """Create a fresh baseline model"""
    model = models.Sequential([
        layers.Dense(512, activation='relu', input_shape=(784,)),
        layers.Dense(256, activation='relu'),
        layers.Dense(128, activation='relu'),
        layers.Dense(64, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(10, activation='softmax')
    ])
    return model

# 10.1 Model with RMSProp optimizer
print("\n10.1 Training with RMSProp optimizer...")
print("-" * 80)

model_rmsprop = create_fresh_baseline()
model_rmsprop.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

print(f"Optimizer: RMSProp")
print(f"Learning Rate: 0.001")
print(f"Rho (decay rate): 0.9")
print(f"Total parameters: {model_rmsprop.count_params():,}")

print("\nTraining with RMSProp...")
history_rmsprop = model_rmsprop.fit(
    X_train, y_train,
    epochs=20,
    batch_size=32,
```