1. Importing Libraries

First, we import the necessary libraries for our project. We need **`TensorFlow`** and **`Keras`** for building the model, **`NumPy`** for numerical operations, and **`Matplotlib`** for visualizing our data.

In [16]:
```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
```

2. Loading the Dataset

Next, we load the **`Fashion-MNIST`** dataset, which is conveniently included with **`Keras`**. The dataset is already split into training and testing sets for us. **`train_images`** and **`train_labels`** contain the data and corresponding labels for training the model, while **`test_images`** and **`test_labels`** will be used to evaluate its performance.

In [17]:
```
# Load the Fashion-MNIST dataset
(train_images, train_labels), (test_images, test_labels) = keras.datasets.fashion_mnist.load_data()

# Define the class names for easy plotting later
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Check the dimensions of our data
print("Training images shape:", train_images.shape)
print("Training labels shape:", train_labels.shape)
```

```
Training images shape: (60000, 28, 28)
Training labels shape: (60000,)
```

3. Visualizing Sample Images

To get a better feel for the data, let's display a few sample images from the training dataset along with their corresponding labels. This will help us understand the types of images we are working with.

In [18]:
```
# Visualize a few sample images
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

In [18]:

4. Preprocessing

Before feeding the images into the neural networks, we need to preprocess them. This involves two main steps:

1. **Normalization**: We will scale the pixel values from the original range of 0-255 to a range of 0-1. This helps in faster convergence during training.
2. **Reshaping**: Convolutional Neural Networks (CNNs) typically expect input images to have a channel dimension. Since our Fashion-MNIST images are grayscale, they have a single channel. We need to reshape the images to include this channel dimension.

In [19]:
```
# Normalize pixel values to be between 0 and 1
train_images = train_images / 255.0
test_images = test_images / 255.0

# Reshape images to include the channel dimension (for grayscale, this is 1)
# CNNs expect input in the shape of (batch_size, height, width, channels)
train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))

print("Normalized and reshaped training images shape:", train_images.shape)
print("Normalized and reshaped testing images shape:", test_images.shape)
```

```
Normalized and reshaped training images shape: (60000, 28, 28, 1)
Normalized and reshaped testing images shape: (10000, 28, 28, 1)
```

5. Baseline Model (2-Layer CNN)

We will now build our baseline model, which is a simple Convolutional Neural Network with two convolutional layers. This model will serve as a starting point to see how a basic CNN performs on the Fashion-MNIST dataset.

5.1 Building the Model Architecture

We will use the Keras Sequential API to build the model layer by layer.

In [20]:
```
# Build the baseline model (2-Layer CNN)
baseline_model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation='softmax')
])

# Print the model summary
baseline_model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pas
s an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)`
object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_5 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_6 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_6 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten_2 (Flatten)	(None, 1600)	0
dense_4 (Dense)	(None, 128)	204,928
dropout_5 (Dropout)	(None, 128)	0
dense_5 (Dense)	(None, 10)	1,290

Total params: 225,034 (879.04 KB)

Trainable params: 225,034 (879.04 KB)

Non-trainable params: 0 (0.00 B)

5.2 Compiling and Training the Baseline Model

Before training, we need to compile the model. This involves specifying the optimizer, loss function, and metrics. We will use the Adam optimizer, sparse categorical crossentropy for the loss function (suitable for multi-class classification with integer labels), and accuracy as the metric.

Then, we will train the model using the training data and validate it using the test data.

In [21]:
```
# Compile the baseline model
baseline_model.compile(optimizer='adam',
                       loss='sparse_categorical_crossentropy',
                       metrics=['accuracy'])

# Train the baseline model
baseline_history = baseline_model.fit(train_images, train_labels, epochs=10,
                                      validation_data=(test_images, test_labels))
```

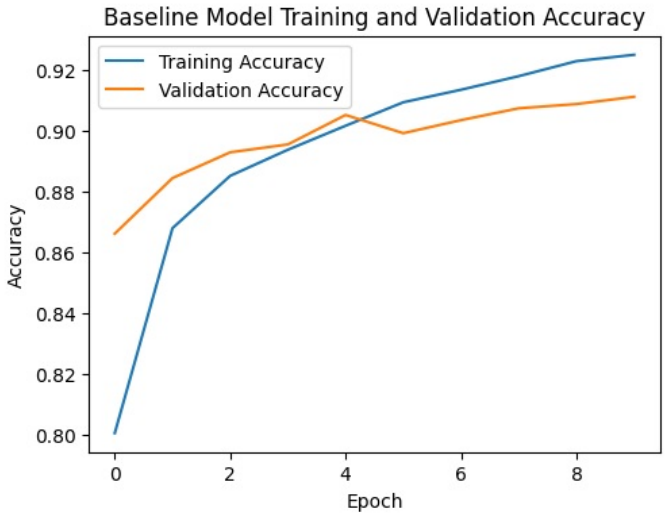
```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 16s 4ms/step - accuracy: 0.7302 - loss: 0.7549 - val_accuracy: 0.8661 - val_loss:
0.3632
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8618 - loss: 0.3794 - val_accuracy: 0.8844 - val_loss:
0.3140
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8811 - loss: 0.3263 - val_accuracy: 0.8929 - val_loss:
0.2949
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8948 - loss: 0.2894 - val_accuracy: 0.8955 - val_loss:
0.2862
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.9019 - loss: 0.2691 - val_accuracy: 0.9052 - val_loss:
0.2638
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9098 - loss: 0.2444 - val_accuracy: 0.8992 - val_loss:
0.2637
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.9146 - loss: 0.2306 - val_accuracy: 0.9035 - val_loss:
0.2649
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9176 - loss: 0.2196 - val_accuracy: 0.9074 - val_loss:
0.2662
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.9259 - loss: 0.2019 - val_accuracy: 0.9088 - val_loss:
0.2532
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9270 - loss: 0.1940 - val_accuracy: 0.9112 - val_loss:
0.2499
```

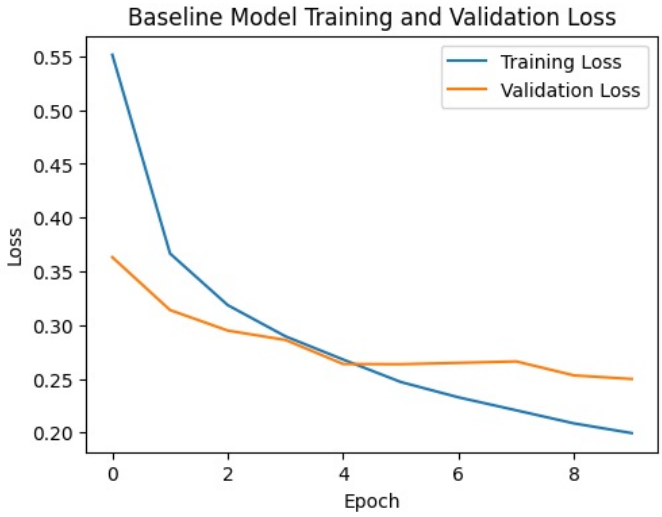
5.3 Evaluating the Baseline Model

Now that the model is trained, let's evaluate its performance on the test dataset to see how well it generalizes to unseen data. We will also plot the training and validation accuracy and loss over epochs to visualize the training history.

In [22]:
```python
# Evaluate the baseline model on the test data
test_loss, test_acc = baseline_model.evaluate(test_images, test_labels, verbose=2)
print('\nBaseline Model Test Accuracy:', test_acc)

# Plot training history (accuracy and loss)
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(baseline_history.history['accuracy'], label='Training Accuracy')
plt.plot(baseline_history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Baseline Model Training and Validation Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(baseline_history.history['loss'], label='Training Loss')
plt.plot(baseline_history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Baseline Model Training and Validation Loss')
plt.legend()

plt.show()
```

```
313/313 - 1s - 2ms/step - accuracy: 0.9112 - loss: 0.2499

Baseline Model Test Accuracy: 0.9111999869346619
```



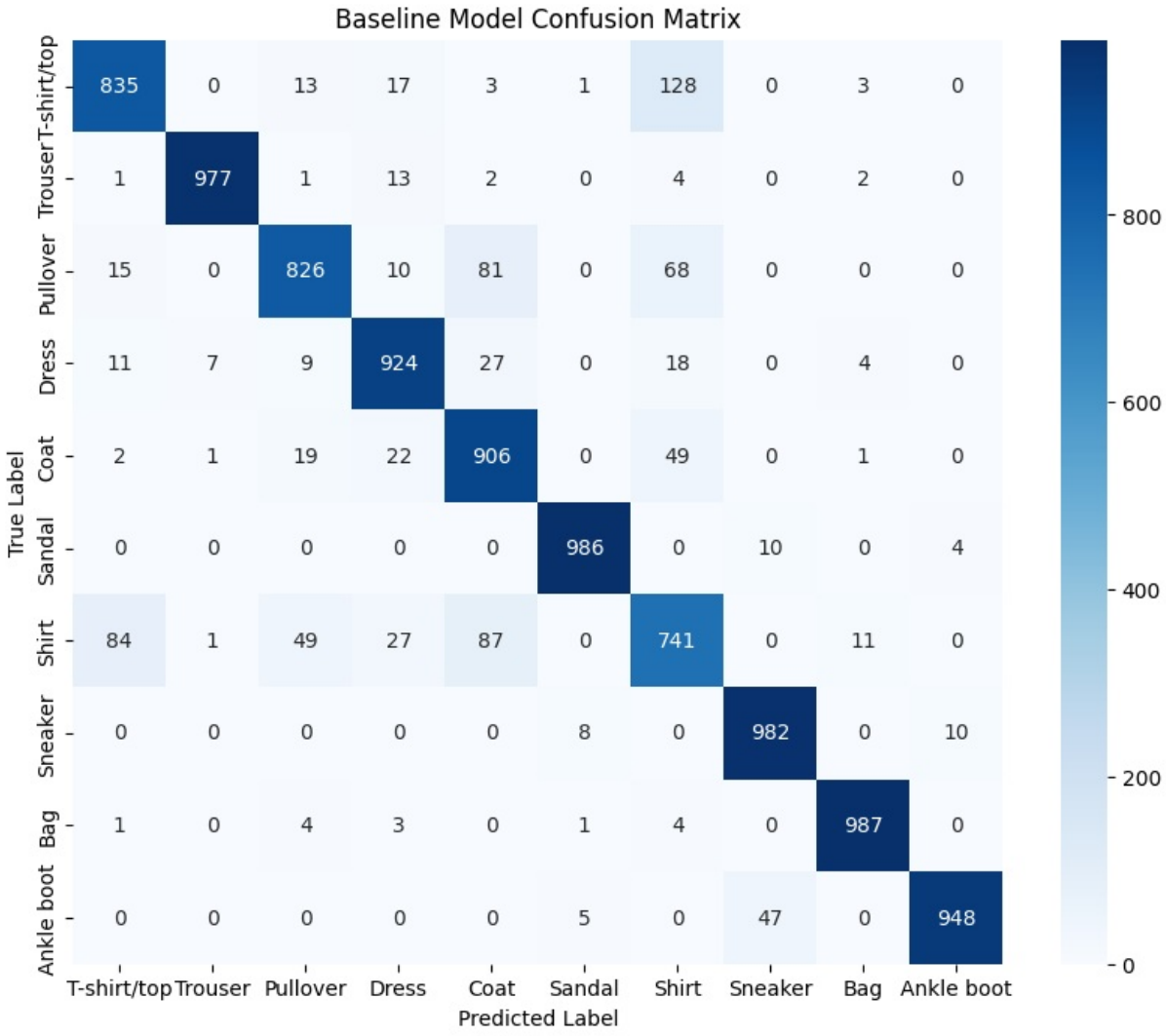


5.4 Confusion Matrix for the Baseline Model

A confusion matrix helps us understand the performance of our classification model by showing the counts of true positive, true negative, false positive, and false negative predictions.

In [23]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Get predictions for the test set
y_pred = np.argmax(baseline_model.predict(test_images), axis=-1)

# Compute the confusion matrix
cm = confusion_matrix(test_labels, y_pred)

# Plot the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Baseline Model Confusion Matrix')
plt.show()
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```



5.5 Saving and Loading the Baseline Model

It's a good practice to save your trained models so you can reuse them later without retraining. We will save the baseline model and then load it back to demonstrate how to use a saved model for predictions.

In [24]:
```
# Save the baseline model
baseline_model.save('fashion_baseline_model.h5')
print("Baseline model saved as fashion_baseline_model.h5")

# Load the saved model
loaded_model = keras.models.load_model('fashion_baseline_model.h5')
print("Baseline model loaded successfully")
```

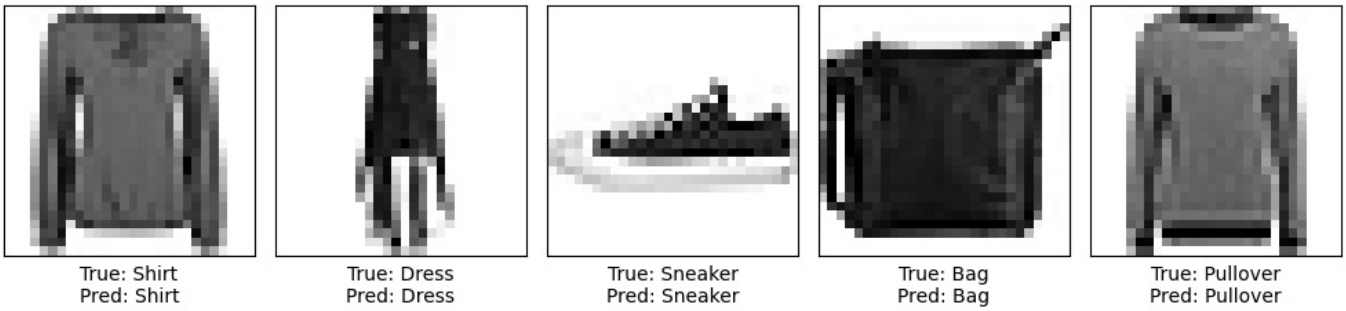
```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. T
his file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_m
odel.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` w
ill be empty until you train or evaluate the model.
Baseline model saved as fashion_baseline_model.h5
Baseline model loaded successfully
```

5.6 Testing the Loaded Baseline Model on Random Images

We will now pick a few random images from the test set and use the loaded baseline model to predict their classes. This will give us a visual idea of the model's performance on individual examples.

In [25]:
```
# Make predictions on a few random test images using the loaded model
num_test_images = test_images.shape[0]
random_indices = np.random.choice(num_test_images, 5) # Choose 5 random indices

plt.figure(figsize=(10, 10))
for i, idx in enumerate(random_indices):
    img = test_images[idx]
    true_label = test_labels[idx]

    # The model expects a batch of images, so we add a dimension
    img_batch = np.expand_dims(img, axis=0)
    predictions = loaded_model.predict(img_batch)
    predicted_label = np.argmax(predictions[0])

    plt.subplot(1, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(img.reshape(28, 28), cmap=plt.cm.binary)
    plt.xlabel(f"True: {class_names[true_label]}\nPred: {class_names[predicted_label]}")

plt.tight_layout()
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 516ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step
```



6. Improved Model (3-Layer CNN)

Now, let's build an improved CNN model with three convolutional layers and additional dropout layers for enhanced regularization. This model has more parameters and capacity, which might lead to better performance on the Fashion-MNIST dataset.

6.1 Building the Improved Model Architecture

We will again use the Keras Sequential API to define the layers of our improved model.

In [26]:
```
# Build the improved model (3-Layer CNN)
improved_model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.25), # Added dropout
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.25), # Added dropout
    keras.layers.Conv2D(128, (3, 3), activation='relu'), # Added convolutional layer
    keras.layers.MaxPooling2D((2, 2)), # Added max pooling
    keras.layers.Dropout(0.25), # Added dropout
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation='softmax')
])

# Print the model summary
improved_model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pas
s an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)`
object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
conv2d_7 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_7 (MaxPooling2D)	(None, 13, 13, 32)	0
dropout_6 (Dropout)	(None, 13, 13, 32)	0
conv2d_8 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_8 (MaxPooling2D)	(None, 5, 5, 64)	0
dropout_7 (Dropout)	(None, 5, 5, 64)	0
conv2d_9 (Conv2D)	(None, 3, 3, 128)	73,856
max_pooling2d_9 (MaxPooling2D)	(None, 1, 1, 128)	0
dropout_8 (Dropout)	(None, 1, 1, 128)	0
flatten_3 (Flatten)	(None, 128)	0
dense_6 (Dense)	(None, 128)	16,512
dropout_9 (Dropout)	(None, 128)	0
dense_7 (Dense)	(None, 10)	1,290

Total params: 110,474 (431.54 KB)

Trainable params: 110,474 (431.54 KB)

Non-trainable params: 0 (0.00 B)

6.2 Compiling and Training the Improved Model

Similar to the baseline model, we need to compile the improved model before training. We will use the same optimizer, loss function, and metrics for a fair comparison. Then, we will train the model using the training data.

In [27]:
```
# Compile the improved model
improved_model.compile(optimizer='adam',
                       loss='sparse_categorical_crossentropy',
                       metrics=['accuracy'])

# Train the improved model
improved_history = improved_model.fit(train_images, train_labels, epochs=10,
                                      validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 14s 5ms/step - accuracy: 0.6072 - loss: 1.0477 - val_accuracy: 0.8088 - val_loss:
0.5034
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.7996 - loss: 0.5470 - val_accuracy: 0.8467 - val_loss:
0.4229
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.8318 - loss: 0.4612 - val_accuracy: 0.8568 - val_loss:
0.3900
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8467 - loss: 0.4263 - val_accuracy: 0.8717 - val_loss:
0.3507
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8529 - loss: 0.4057 - val_accuracy: 0.8681 - val_loss:
0.3520
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8583 - loss: 0.3867 - val_accuracy: 0.8746 - val_loss:
0.3377
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8622 - loss: 0.3809 - val_accuracy: 0.8772 - val_loss:
0.3258
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8647 - loss: 0.3661 - val_accuracy: 0.8830 - val_loss:
0.3192
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8685 - loss: 0.3668 - val_accuracy: 0.8846 - val_loss:
0.3177
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8726 - loss: 0.3501 - val_accuracy: 0.8855 - val_loss:
0.3151
```

6.3 Evaluating and Comparing Models

After training the improved model, we will evaluate its performance on the test dataset and compare its test accuracy with the baseline model to see if the changes made resulted in an improvement.

7. Implementing the cnn-dropout-3 Model from the Study

Based on the study you provided, let's implement the cnn-dropout-3 model architecture. This model achieved a high accuracy on a validation dataset.

7.1 Building the cnn-dropout-3 Model Architecture

We will construct the model using the specified layers and parameters.

In [29]:
```python
# Build the cnn-dropout-3 model
cnn_dropout_3_model = keras.Sequential([
    # First Block
    keras.layers.Conv2D(6, (5, 5), activation='relu', input_shape=(28, 28, 1)),
    keras.layers.MaxPooling2D((2, 2), strides=2),
    keras.layers.Dropout(0.18),

    # Second Block
    keras.layers.Conv2D(16, (5, 5), activation='relu'),
    keras.layers.MaxPooling2D((2, 2), strides=2),
    keras.layers.Dropout(0.18),

    # Flatten Layer
    keras.layers.Flatten(),

    # Dense Layers
    keras.layers.Dense(120, activation='relu'),
    keras.layers.Dropout(0.25),
    keras.layers.Dense(84, activation='relu'),
    keras.layers.Dropout(0.25),

    # Output Layer
    keras.layers.Dense(10, activation='softmax')
])

# Print the model summary
cnn_dropout_3_model.summary()
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
conv2d_10 (Conv2D)	(None, 24, 24, 6)	156
max_pooling2d_10 (MaxPooling2D)	(None, 12, 12, 6)	0
dropout_10 (Dropout)	(None, 12, 12, 6)	0
conv2d_11 (Conv2D)	(None, 8, 8, 16)	2,416
max_pooling2d_11 (MaxPooling2D)	(None, 4, 4, 16)	0
dropout_11 (Dropout)	(None, 4, 4, 16)	0
flatten_4 (Flatten)	(None, 256)	0
dense_8 (Dense)	(None, 120)	30,840
dropout_12 (Dropout)	(None, 120)	0
dense_9 (Dense)	(None, 84)	10,164
dropout_13 (Dropout)	(None, 84)	0
dense_10 (Dense)	(None, 10)	850

Total params: 44,426 (173.54 KB)
Trainable params: 44,426 (173.54 KB)
Non-trainable params: 0 (0.00 B)

7.2 Compiling and Training the cnn-dropout-3 Model

Now, we will compile the cnn-dropout-3 model using the Adadelta optimizer and train it for 12 epochs with a batch size of 128, as specified in the study.

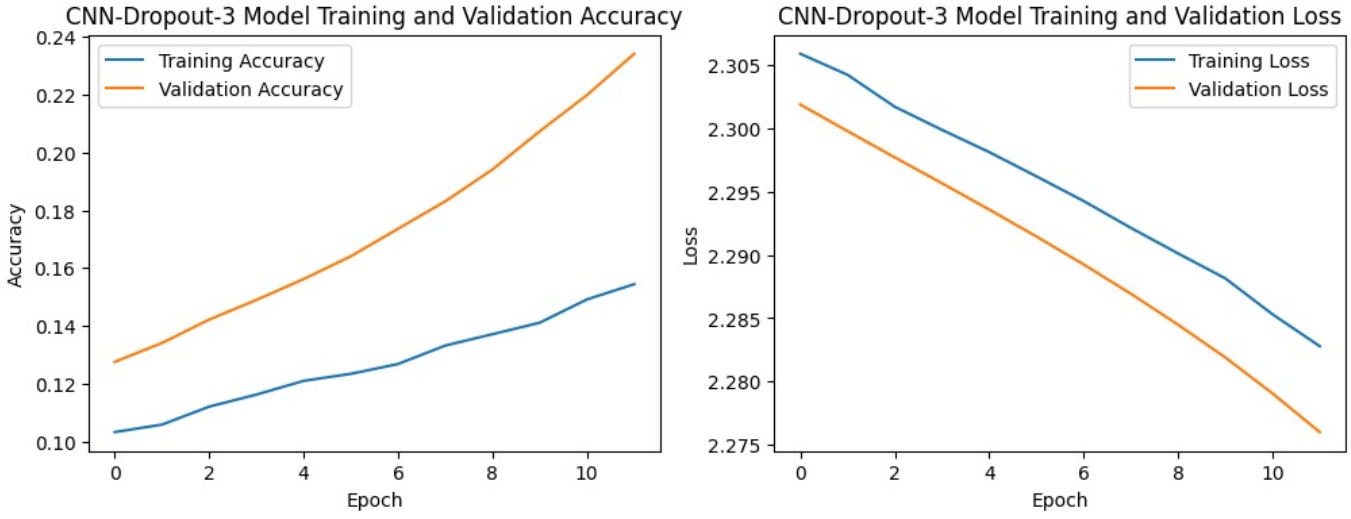
In [30]:
```
# Compile the cnn-dropout-3 model
cnn_dropout_3_model.compile(optimizer='Adadelta',
                            loss='sparse_categorical_crossentropy',
                            metrics=['accuracy'])

# Train the cnn-dropout-3 model
cnn_dropout_3_history = cnn_dropout_3_model.fit(train_images, train_labels, epochs=12, batch_size=128,
                                                validation_data=(test_images, test_labels))
```

```
Epoch 1/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 14s 14ms/step - accuracy: 0.1004 - loss: 2.3062 - val_accuracy: 0.1276 - val_loss:
2.3019
Epoch 2/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.1046 - loss: 2.3043 - val_accuracy: 0.1341 - val_loss: 2.
2998
Epoch 3/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.1087 - loss: 2.3025 - val_accuracy: 0.1422 - val_loss: 2.
2977
Epoch 4/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.1168 - loss: 2.3000 - val_accuracy: 0.1490 - val_loss: 2.
2957
Epoch 5/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.1197 - loss: 2.2987 - val_accuracy: 0.1562 - val_loss: 2.
2936
Epoch 6/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.1204 - loss: 2.2969 - val_accuracy: 0.1641 - val_loss: 2.
2915
Epoch 7/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.1259 - loss: 2.2950 - val_accuracy: 0.1736 - val_loss: 2.
2893
Epoch 8/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.1297 - loss: 2.2927 - val_accuracy: 0.1831 - val_loss: 2.
2869
Epoch 9/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.1349 - loss: 2.2906 - val_accuracy: 0.1941 - val_loss: 2.
2845
Epoch 10/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.1404 - loss: 2.2886 - val_accuracy: 0.2073 - val_loss: 2.
2819
Epoch 11/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.1466 - loss: 2.2862 - val_accuracy: 0.2198 - val_loss: 2.
2791
Epoch 12/12
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.1517 - loss: 2.2835 - val_accuracy: 0.2341 - val_loss: 2.
2760
```

7.3 Visualizing Training History of the cnn-dropout-3 Model

To understand the training process of the cnn-dropout-3 model, let's visualize its training and validation accuracy and loss over the epochs.

In [31]:
```python
# Plot training history (accuracy and loss) for cnn-dropout-3 model
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(cnn_dropout_3_history.history['accuracy'], label='Training Accuracy')
plt.plot(cnn_dropout_3_history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('CNN-Dropout-3 Model Training and Validation Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(cnn_dropout_3_history.history['loss'], label='Training Loss')
plt.plot(cnn_dropout_3_history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('CNN-Dropout-3 Model Training and Validation Loss')
plt.legend()

plt.show()
```

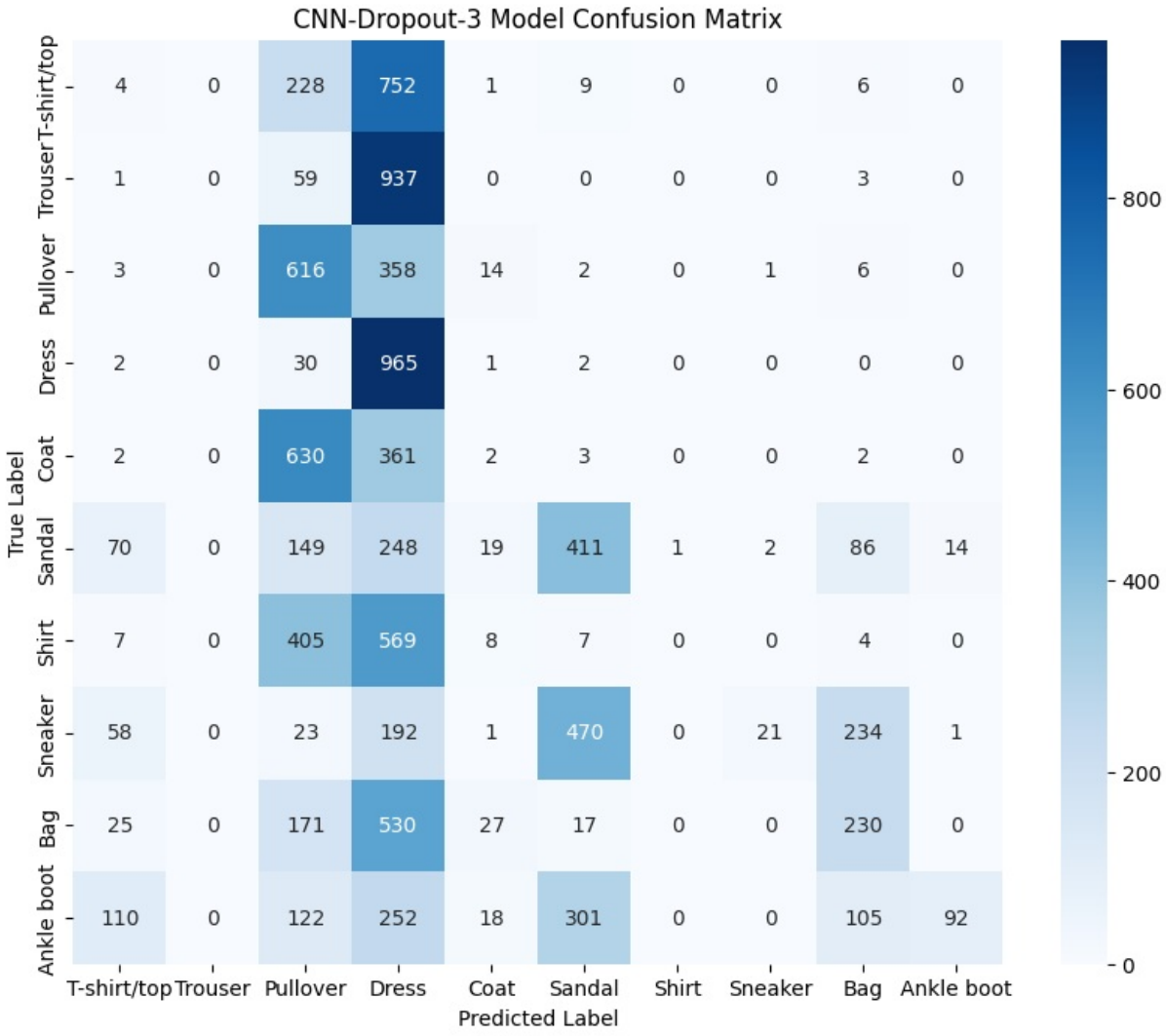


7.4 Confusion Matrix for the cnn-dropout-3 Model

Finally, let's generate a confusion matrix for the cnn-dropout-3 model to analyze its performance on each class.

In [32]:
```
# Get predictions for the test set using the cnn-dropout-3 model
y_pred_cnn_dropout_3 = np.argmax(cnn_dropout_3_model.predict(test_images), axis=-1)

# Compute the confusion matrix for the cnn-dropout-3 model
cm_cnn_dropout_3 = confusion_matrix(test_labels, y_pred_cnn_dropout_3)

# Plot the confusion matrix for the cnn-dropout-3 model
plt.figure(figsize=(10, 8))
sns.heatmap(cm_cnn_dropout_3, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_nam
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('CNN-Dropout-3 Model Confusion Matrix')
plt.show()
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step
```



In [32]:
```
```

8. Detailed Model Comparison

Let's perform a detailed comparison of the three models we built: the Baseline Model, the Improved Model, and the cnn-dropout-3 Model. We will compare their architectures, the number of trainable parameters, and their final test accuracies.

In [34]:
```
print("--- Detailed Model Comparison ---")

print("\nBaseline Model:")
baseline_model.summary()
print(f"Baseline Model Test Accuracy: {test_acc:.4f}")

print("\nImproved Model:")
improved_model.summary()
# Evaluate the improved model on the test data if not already done in this session
if 'test_acc_improved' not in locals():
    test_loss_improved, test_acc_improved = improved_model.evaluate(test_images, test_labels, verbose=2)
```

```python
print(f"Improved Model Test Accuracy: {test_acc_improved:.4f}")

print("\nCNN-Dropout-3 Model:")
cnn_dropout_3_model.summary()
# Evaluate the cnn-dropout-3 model on the test data if not already done in this session
if 'test_acc_cnn_dropout_3' not in locals():
    test_loss_cnn_dropout_3, test_acc_cnn_dropout_3 = cnn_dropout_3_model.evaluate(test_images, test_labels, ve
print(f"CNN-Dropout-3 Model Test Accuracy: {test_acc_cnn_dropout_3:.4f}")

print("\n--- Summary of Test Accuracies ---")
print(f"Baseline Model: {test_acc:.4f}")
print(f"Improved Model: {test_acc_improved:.4f}")
print(f"CNN-Dropout-3 Model: {test_acc_cnn_dropout_3:.4f}")

# Determine and print which model performed best
if test_acc_cnn_dropout_3 > test_acc_improved and test_acc_cnn_dropout_3 > test_acc:
    print("\nThe CNN-Dropout-3 Model achieved the highest test accuracy.")
elif test_acc_improved > test_acc and test_acc_improved > test_acc_cnn_dropout_3:
    print("\nThe Improved Model achieved the highest test accuracy.")
elif test_acc > test_acc_improved and test_acc > test_acc_cnn_dropout_3:
    print("\nThe Baseline Model achieved the highest test accuracy.")
else:
    print("\nPerformance among the models is similar.")
```

```
--- Detailed Model Comparison ---

 Baseline Model:
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_5 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_6 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_6 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten_2 (Flatten)	(None, 1600)	0
dense_4 (Dense)	(None, 128)	204,928
dropout_5 (Dropout)	(None, 128)	0
dense_5 (Dense)	(None, 10)	1,290

Total params: 675,104 (2.58 MB)

Trainable params: 225,034 (879.04 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 450,070 (1.72 MB)

```
Baseline Model Test Accuracy: 0.9112

 Improved Model:
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
conv2d_7 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_7 (MaxPooling2D)	(None, 13, 13, 32)	0
dropout_6 (Dropout)	(None, 13, 13, 32)	0
conv2d_8 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_8 (MaxPooling2D)	(None, 5, 5, 64)	0
dropout_7 (Dropout)	(None, 5, 5, 64)	0
conv2d_9 (Conv2D)	(None, 3, 3, 128)	73,856
max_pooling2d_9 (MaxPooling2D)	(None, 1, 1, 128)	0
dropout_8 (Dropout)	(None, 1, 1, 128)	0
flatten_3 (Flatten)	(None, 128)	0
dense_6 (Dense)	(None, 128)	16,512
dropout_9 (Dropout)	(None, 128)	0
dense_7 (Dense)	(None, 10)	1,290

Total params: 331,424 (1.26 MB)
Trainable params: 110,474 (431.54 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 220,950 (863.09 KB)
Improved Model Test Accuracy: 0.8855

CNN-Dropout-3 Model:
Model: "sequential_4"

Layer (type)	Output Shape	Param #
conv2d_10 (Conv2D)	(None, 24, 24, 6)	156
max_pooling2d_10 (MaxPooling2D)	(None, 12, 12, 6)	0
dropout_10 (Dropout)	(None, 12, 12, 6)	0
conv2d_11 (Conv2D)	(None, 8, 8, 16)	2,416
max_pooling2d_11 (MaxPooling2D)	(None, 4, 4, 16)	0
dropout_11 (Dropout)	(None, 4, 4, 16)	0
flatten_4 (Flatten)	(None, 256)	0
dense_8 (Dense)	(None, 120)	30,840
dropout_12 (Dropout)	(None, 120)	0
dense_9 (Dense)	(None, 84)	10,164
dropout_13 (Dropout)	(None, 84)	0
dense_10 (Dense)	(None, 10)	850

Total params: 133,280 (520.63 KB)
Trainable params: 44,426 (173.54 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 88,854 (347.09 KB)
313/313 - 1s - 5ms/step - accuracy: 0.2341 - loss: 2.2760
CNN-Dropout-3 Model Test Accuracy: 0.2341

--- Summary of Test Accuracies ---
Baseline Model: 0.9112
Improved Model: 0.8855
CNN-Dropout-3 Model: 0.2341

The Baseline Model achieved the highest test accuracy.

In [ ]: