Classify fashion images on the MNIST data

What would be an appropriate metric to evaluate your models? Why? (Hint: No code required.)

Since no specific cost is given, accuracy seems like the best metric to be used since we assume that all class are equally important. Accuracy is also intuitive to understand the performance of each of the models. Hence, in this assignment, we will choose accuracy as the mertric of choice.

Get the data and show some example images from the data.

In [1]:
```
%%capture
import warnings
warnings.filterwarnings('ignore')

import pandas as pd
import numpy as np
```

In [2]:
```
from keras.datasets import fashion_mnist

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
```

In [3]:
```
# Look at the dimensions
print(f"X_train: {X_train.shape}")
print(f"y_train: {y_train.shape}")
print(f"X_test:  {X_test.shape}")
print(f"y_test:  {y_test.shape}")
```

```
X_train: (60000, 28, 28)
y_train: (60000,)
X_test:  (10000, 28, 28)
y_test:  (10000,)
```

In [4]:
```
# Visualize some items in a grid
import matplotlib.pyplot as plt

fig, axs = plt.subplots(5, 5, figsize=(10,10))
for i, ax in enumerate(axs.flatten()):
    ax.imshow(X_train[i], cmap="binary")
    ax.axis("off")
    ax.set_title(f"Label: {y_train[i]}")
plt.tight_layout()
plt.show()
```


Label: 9
Label: 0
Label: 0
Label: 3
Label: 0
Label: 2
Label: 7
Label: 2
Label: 5
Label: 5
Label: 0
Label: 9
Label: 5
Label: 5
Label: 7
Label: 9
Label: 1
Label: 0
Label: 6
Label: 4
Label: 3
Label: 1
Label: 4
Label: 8
Label: 4

Train a simple fully connected single hidden layer network to predict the items. Remember to normalize the data similar to what we did in class. Make sure that you use enough epochs so that the validation error begins to level off - provide a plot of the training history.

In [5]:
```
from sklearn.model_selection import train_test_split

prng = np.random.RandomState(20240329)  # ensure we have the same split as in last class

# intentionally choose a small train set to decrease computational burden
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.3, random_state=prng)

print(f"X_train: {X_train.shape}")
print(f"y_train: {y_train.shape}")
print(f"X_val:  {X_val.shape}")
print(f"y_val:  {y_val.shape}")
print(f"X_test:  {X_test.shape}")
print(f"y_test:  {y_test.shape}")
```

```
X_train: (42000, 28, 28)
y_train: (42000,)
X_val:  (18000, 28, 28)
y_val:  (18000,)
X_test:  (10000, 28, 28)
y_test:  (10000,)
```

In [6]:
```
# Benchmark #1 (silly):

from sklearn.metrics import accuracy_score
from statistics import mode

most_frequent = mode(y_train)
print(f"Most frequent element is: {most_frequent}")
accuracy_most_frequent = accuracy_score(y_val, np.repeat(most_frequent, len(y_val)))
print(f"Accuracy for our no-brainer model: {round(accuracy_most_frequent, 4)}")
summary_df = pd.DataFrame({'Model': ['Benchmark'],
                           'Train accuracy': [round(accuracy_score(np.array([most_frequent] * len(y_train))
                           'Val accuracy': [round(accuracy_score(np.array([most_frequent] * len(y_val)), y_
                           'Test accuracy': [round(accuracy_score(np.array([most_frequent] * len(y_test)),
                          })
summary_df
```

```
Most frequent element is: 7
Accuracy for our no-brainer model: 0.0944
```

Out[6]:

	Model	Train accuracy	Val accuracy	Test accuracy
0	Benchmark	0.1024	0.0944	0.1

In [7]:
```
def update_summary(summary_df, model_name, train_score, val_score, test_score):
    if model_name not in summary_df.Model.values:
        summary_df.loc[len(summary_df.index)] = [model_name,
                                                 '{:.4f}'.format(train_score),
                                                 '{:.4f}'.format(val_score),
                                                 '{:.4f}'.format(test_score)]
    else:
        summary_df.loc[summary_df.Model == model_name] = [model_name,
                                                 '{:.4f}'.format(train_score),
                                                 '{:.4f}'.format(val_score),
                                                 '{:.4f}'.format(test_score)]
```

In [8]:
```
from sklearn.pipeline import Pipeline
# Benchmark #2 (RF):
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import FunctionTransformer

def flatten_data(X):
    return X.reshape(X.shape[0], -1)

transformer = FunctionTransformer(flatten_data)

rf_model = Pipeline(
    [("preprocess", transformer),
     ("rf", RandomForestClassifier(random_state = prng))
     ], verbose=True
)

rf_model.fit(X_train, y_train)
```

```
predictions_rf = rf_model.predict(X_val)
accuracy_score(y_val, predictions_rf)
```

```
[Pipeline] ........ (step 1 of 2) Processing preprocess, total=   0.0s
[Pipeline] ............... (step 2 of 2) Processing rf, total=  37.4s
```

Out[8]: 0.8811666666666667

In [9]:
```
update_summary(summary_df, 'Random Forest', accuracy_score(y_train, rf_model.predict(X_train)), accuracy_sc
summary_df
```

Out[9]:

	Model	Train accuracy	Val accuracy	Test accuracy
0	Benchmark	0.1024	0.0944	0.1
1	Random Forest	1.0000	0.8812	0.8699

In [10]:
```
from keras.utils import to_categorical

print(f"Dimension of y: {y_train.shape}")

# Convert target variables to categorical
num_classes = 10
y_sets = [y_train, y_test, y_val]
y_train, y_test, y_val = [to_categorical(y, num_classes=num_classes) for y in y_sets]
print(f"Dimension of y: {y_train.shape}")
```

```
Dimension of y: (42000,)
Dimension of y: (42000, 10)
```

In [11]:
```
from keras.models import Sequential
from keras.layers import Input, Flatten, Rescaling, Dense
from keras.utils import to_categorical

# Build the simple fully connected single hidden layer network model
model = Sequential([
    Input(shape=X_train.shape[1:]),
    Flatten(),
    Rescaling(1./255),
    Dense(255, activation='relu'),
    Dense(num_classes, activation='softmax')
])

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
```

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
rescaling (Rescaling)	(None, 784)	0
dense (Dense)	(None, 255)	200,175
dense_1 (Dense)	(None, 10)	2,560

Total params: 202,735 (791.93 KB)

Trainable params: 202,735 (791.93 KB)

Non-trainable params: 0 (0.00 B)

None

In [12]:
```
import keras

# Fit the model
keras.utils.set_random_seed(20240329)  # for reproducibility
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100, batch_size=512, verbose=0
```

In [13]:
```
# Evaluation of the model on the validation set
scores = model.evaluate(X_val, y_val)
print(f"Accuracy for keras single hidden layer: {round(scores[1], 4)}")
```

```
563/563 ━━━━━━━━━━━━━━━━━━━━ 0s 414us/step - accuracy: 0.8847 - loss: 0.5014
Accuracy for keras single hidden layer: 0.8867
```

Before building the simple fully connected single hidden layer network, we build a naive most frequent model and a Random Forest model as benchmarks.

The simple fully connected single hidden layer network consists of:

- Normalize layer to like we did in class
- A single hidden layer with 255 neurons
- An output layer with 10 neurons representing 10 classes and uses 'softmax' as activation function

In [14]:
```
def plot_history(fit_history):
    plt.plot(fit_history['accuracy'], label='Training Accuracy')
    plt.plot(fit_history['val_accuracy'], label='Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.legend()
    plt.show()

plot_history(history.history)
```

In [15]:
```
update_summary(summary_df, 'Single Hidden Layer', model.evaluate(X_train, y_train)[1], scores[1], model.eva
summary_df
```

```
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 1s 414us/step - accuracy: 0.9575 - loss: 0.1155
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 414us/step - accuracy: 0.8755 - loss: 0.5221
```

Out[15]:

	Model	Train accuracy	Val accuracy	Test accuracy
0	Benchmark	0.1024	0.0944	0.1
1	Random Forest	1.0000	0.8812	0.8699
2	Single Hidden Layer	0.9575	0.8867	0.8764

From the summary table, the simple fully connected single hidden layer network outperforms the benchmark and random forest models. Although on the train data, the simple networks does not generalize as perfect as the random forest, on validation and test data, it performs much better, meaning it avoid the overfitting problem that exist in the random forest model.

From the history graph, it seems the validation accuracy flattens out pretty quickly. We can probably stop the training earlier and still obtain the same performance on the validation and test data.

Experiment with different network architectures and settings (number of hidden layers, number of nodes, regularization, etc.). Train at least 3 models. Explain what you have tried and how it worked.

In [16]:
```
# Build the fully connected 3 hidden layers network model
hidden3_model = Sequential([
    Input(shape=X_train.shape[1:]),
    Flatten(),
    Rescaling(1./255),
    Dense(255, activation='relu'),
    Dense(1020, activation='relu'),
    Dense(510, activation='relu'),
    Dense(num_classes, activation='softmax')
])

# Compile the model
hidden3_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(hidden3_model.summary())
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
rescaling_1 (Rescaling)	(None, 784)	0
dense_2 (Dense)	(None, 255)	200,175
dense_3 (Dense)	(None, 1020)	261,120
dense_4 (Dense)	(None, 510)	520,710
dense_5 (Dense)	(None, 10)	5,110

Total params: 987,115 (3.77 MB)
Trainable params: 987,115 (3.77 MB)
Non-trainable params: 0 (0.00 B)

None

In [17]:
```
# Fit the model
keras.utils.set_random_seed(20240329)  # for reproducibility
hidden3_history = hidden3_model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100, batch_siz
```

In [18]:
```
# Evaluation of the model on the validation set
hidden3_scores = hidden3_model.evaluate(X_val, y_val)
print(f"Accuracy for keras 3 hidden layers: {round(hidden3_scores[1], 4)}")
```

```
563/563 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8911 - loss: 0.8222
Accuracy for keras 3 hidden layers: 0.8914
```

In [19]:
```
plot_history(hidden3_history.history)
```

In [20]:
```
update_summary(summary_df, '3 Hidden Layers', hidden3_model.evaluate(X_train, y_train)[1], hidden3_scores[1
summary_df
```

```
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9839 - loss: 0.0482
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8831 - loss: 0.9080
```

Out[20]:

	Model	Train accuracy	Val accuracy	Test accuracy
0	Benchmark	0.1024	0.0944	0.1
1	Random Forest	1.0000	0.8812	0.8699
2	Single Hidden Layer	0.9575	0.8867	0.8764
3	3 Hidden Layers	0.9834	0.8914	0.8855

In [21]:
```
from keras.layers import Dropout

# Build the regularized 3 hidden layers network model
reg_hidden3_model = Sequential([
    Input(shape=X_train.shape[1:]),
    Flatten(),
    Rescaling(1./255),
    Dense(255, activation='relu'),
    Dropout(0.2),
    Dense(1020, activation='relu'),
    Dropout(0.2),
    Dense(510, activation='relu'),
    Dense(num_classes, activation='softmax')
])

# Compile the model
reg_hidden3_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(reg_hidden3_model.summary())
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
flatten_2 (Flatten)	(None, 784)	0
rescaling_2 (Rescaling)	(None, 784)	0
dense_6 (Dense)	(None, 255)	200,175
dropout (Dropout)	(None, 255)	0
dense_7 (Dense)	(None, 1020)	261,120
dropout_1 (Dropout)	(None, 1020)	0
dense_8 (Dense)	(None, 510)	520,710
dense_9 (Dense)	(None, 10)	5,110

Total params: 987,115 (3.77 MB)
Trainable params: 987,115 (3.77 MB)
Non-trainable params: 0 (0.00 B)
None

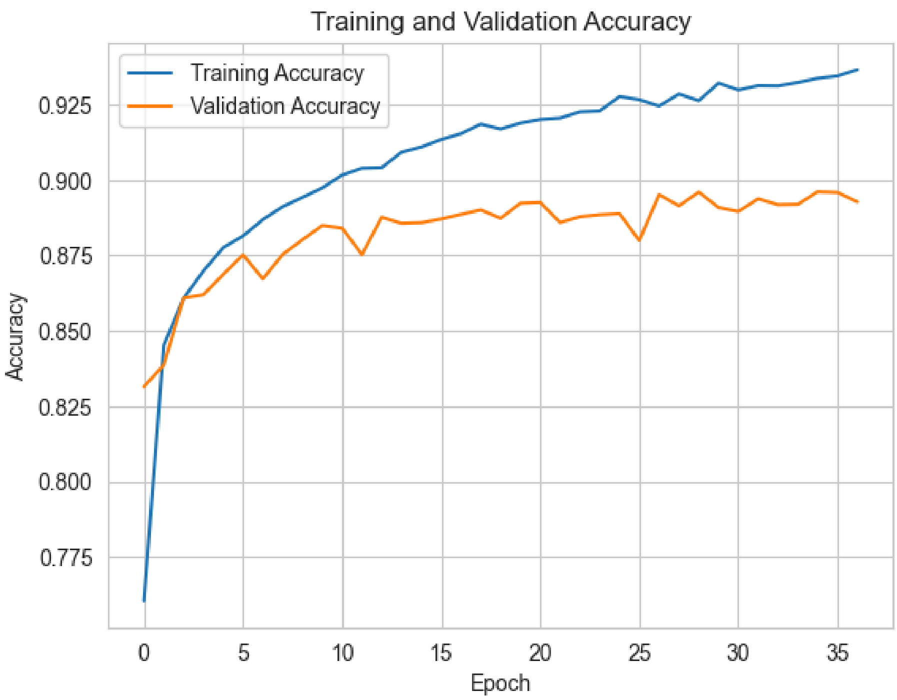
In [22]:
```
from keras.callbacks import EarlyStopping

# Fit the model with EarlyStopping
keras.utils.set_random_seed(20240329)  # for reproducibility
reg_hidden3_history = reg_hidden3_model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100, b
```

```
Epoch 37: early stopping
```

In [23]:
```
# Evaluation of the model on the validation set
reg_hidden3_model_scores = reg_hidden3_model.evaluate(X_val, y_val)
print(f"Accuracy for keras regularized 3 hidden layers: {round(hidden3_scores[1], 4)}")
```

```
563/563 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8910 - loss: 0.3622
Accuracy for keras regularized 3 hidden layers: 0.8914
```

In [24]:
```
plot_history(reg_hidden3_history.history)
```



In [25]:
```
update_summary(summary_df, 'Regularized 3 Hidden Layers', reg_hidden3_model.evaluate(X_train, y_train)[1],
summary_df
```

```
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9493 - loss: 0.1351
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8891 - loss: 0.3817
```

Out[25]:

	Model	Train accuracy	Val accuracy	Test accuracy
0	Benchmark	0.1024	0.0944	0.1
1	Random Forest	1.0000	0.8812	0.8699
2	Single Hidden Layer	0.9575	0.8867	0.8764
3	3 Hidden Layers	0.9834	0.8914	0.8855
4	Regularized 3 Hidden Layers	0.9473	0.8928	0.8878

In [26]:

```
# Build the regularized 5 hidden layers network model
reg_hidden5_model = Sequential([
    Input(shape=X_train.shape[1:]),
    Flatten(),
    Rescaling(1./255),
    Dense(255, activation='relu'),
    Dropout(0.2),
    Dense(510, activation='relu'),
    Dropout(0.2),
    Dense(2040, activation='relu'),
    Dropout(0.2),
    Dense(1020, activation='relu'),
    Dropout(0.2),
    Dense(255, activation='relu'),
    Dense(num_classes, activation='softmax')
])

# Compile the model
reg_hidden5_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(reg_hidden5_model.summary())
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 784)	0
rescaling_3 (Rescaling)	(None, 784)	0
dense_10 (Dense)	(None, 255)	200,175
dropout_2 (Dropout)	(None, 255)	0
dense_11 (Dense)	(None, 510)	130,560
dropout_3 (Dropout)	(None, 510)	0
dense_12 (Dense)	(None, 2040)	1,042,440
dropout_4 (Dropout)	(None, 2040)	0
dense_13 (Dense)	(None, 1020)	2,081,820
dropout_5 (Dropout)	(None, 1020)	0
dense_14 (Dense)	(None, 255)	260,355
dense_15 (Dense)	(None, 10)	2,560

Total params: 3,717,910 (14.18 MB)

Trainable params: 3,717,910 (14.18 MB)

Non-trainable params: 0 (0.00 B)

None

In [27]:

```
# Fit the model with EarlyStopping
keras.utils.set_random_seed(20240329)  # for reproducibility
reg_hidden5_history = reg_hidden5_model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100, b
```

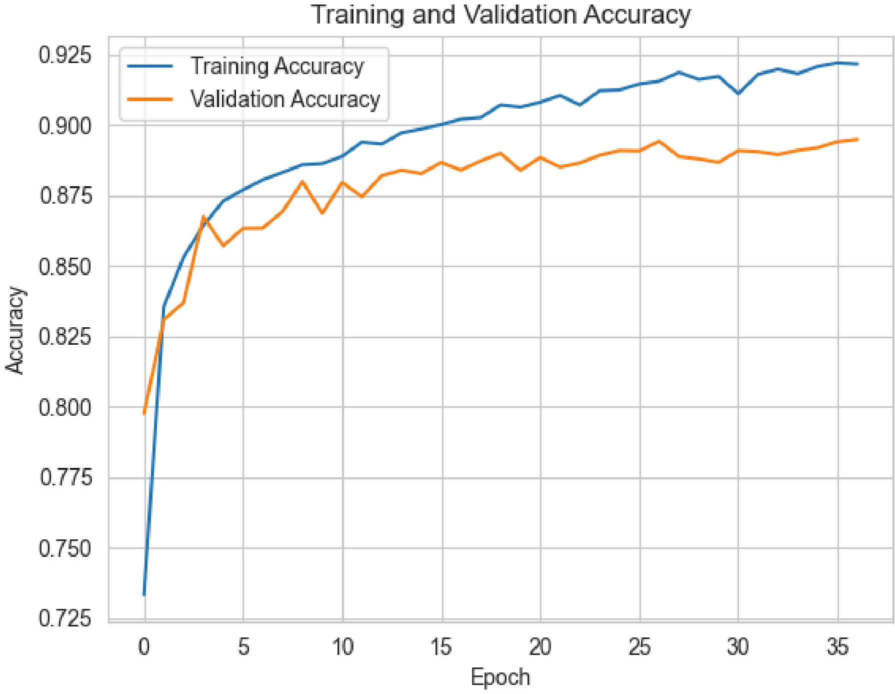
```
Epoch 37: early stopping
```

In [28]:

```
# Evaluation of the model on the validation set
reg_hidden5_model_scores = reg_hidden5_model.evaluate(X_val, y_val)
print(f"Accuracy for keras regularized 5 hidden layers: {round(reg_hidden5_model_scores[1], 4)}")
```

```
563/563 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8958 - loss: 0.3266
Accuracy for keras regularized 5 hidden layers: 0.8948
```

In [29]:
```
plot_history(reg_hidden5_history.history)
```



In [30]:
```
update_summary(summary_df, 'Regularized 5 Hidden Layers', reg_hidden5_model.evaluate(X_train, y_train)[1],
summary_df
```

```
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9390 - loss: 0.1569
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8902 - loss: 0.3446
```

Out[30]:

	Model	Train accuracy	Val accuracy	Test accuracy
0	Benchmark	0.1024	0.0944	0.1
1	Random Forest	1.0000	0.8812	0.8699
2	Single Hidden Layer	0.9575	0.8867	0.8764
3	3 Hidden Layers	0.9834	0.8914	0.8855
4	Regularized 3 Hidden Layers	0.9473	0.8928	0.8878
5	Regularized 5 Hidden Layers	0.9390	0.8948	0.8885

We try out 3 different network configurations as follows:

- A fully connected network with 3 hidden layers: a 255-neuron layer follows by a 1020-neuron follows by a 510-neuron layer.
- A regularized network with 3 hidden layers: a 255-neuron layer follows by a 1020-neuron follows by a 510-neuron layer. After each layer is a Dropout layer with 20% dropout rate.
- A regularized network with 5 hidden layers: a 255-neuron layer follows by a 510-neuron follows by a 2040-neuron layer follows by a 1020-neuron layer follows by a 510-neuron layer. After each layer is a Dropout layer with 20% dropout rate.

As expected, the more complicated the model, the slower the training becomes. However, the regularized networks do generalize the data better and perform better in the validation and test data.

The fully connected network with 3 hidden layers overfit the training data compared to the simple fully connected single hidden layer network and performs poorer in the validation and test data. To combat this, the regularized 3 hidden layers add the dropout at 20% rate to avoid the training overfitting. The regularized 3 hidden layer underfit the training but perform better in the validation and test data compared to the single hidden layer and 3 hidden layers network.

The regularized 5 hidden layers network out performs the regularized 3 hidden layers by a small amount, proving that more complicated network capture the data patterns better.

Both the regularized 5 hidden layers network and the regularized 3 hidden layers network utilized EarlyStopping with patience = 10 and min_delta = 0.001. Both stop earlier than the 100 epoch limit, cutting down training time but still perform relatively better than the no early stopping networks.

Try to improve the accuracy of your model by using convolution. Train at least two different models (you can vary the number of convolutional and pooling layers or whether you include a fully connected layer before the output, etc.).

In [31]:
```
from keras.layers import Conv2D, MaxPooling2D, Reshape

# Build the single cnn network model
cnn1_model = Sequential([
    Input(shape=X_train.shape[1:]),
    Reshape(target_shape=(X_train.shape[1], X_train.shape[2], 1)),  # explicitly state the 4th (channel) di
    Rescaling(1./255),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dropout(0.2),
    Dense(255, activation='relu'),
    Dropout(0.2),
    Dense(127, activation='relu'),
    Dropout(0.2),
    Dense(num_classes, activation='softmax')
])

# Compile the model
cnn1_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(cnn1_model.summary())
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
reshape (Reshape)	(None, 28, 28, 1)	0
rescaling_4 (Rescaling)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
flatten_4 (Flatten)	(None, 5408)	0
dropout_6 (Dropout)	(None, 5408)	0
dense_16 (Dense)	(None, 255)	1,379,295
dropout_7 (Dropout)	(None, 255)	0
dense_17 (Dense)	(None, 127)	32,512
dropout_8 (Dropout)	(None, 127)	0
dense_18 (Dense)	(None, 10)	1,280

Total params: 1,413,407 (5.39 MB)
Trainable params: 1,413,407 (5.39 MB)
Non-trainable params: 0 (0.00 B)
None

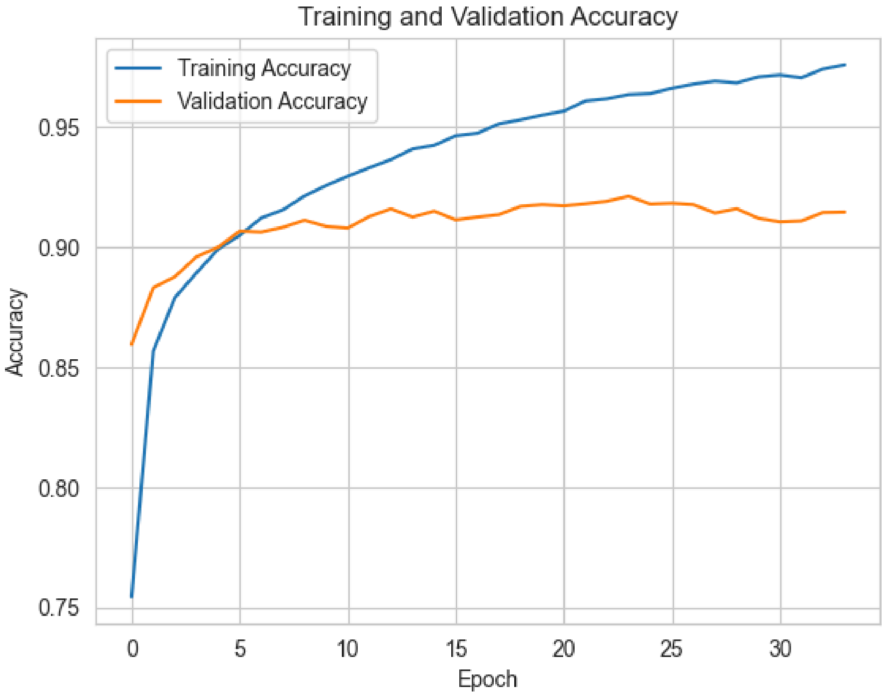
In [32]:
```
# Fit the model with EarlyStopping
keras.utils.set_random_seed(20240329)  # for reproducibility
cnn1_history = cnn1_model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100, batch_size=512,
```

```
Epoch 34: early stopping
```

In [33]:
```
# Evaluation of the model on the validation set
cnn1_model_scores = cnn1_model.evaluate(X_val, y_val)
print(f"Accuracy for keras single cnn layer: {round(cnn1_model_scores[1], 4)}")
```

```
563/563 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9148 - loss: 0.3235
Accuracy for keras single cnn layer: 0.9146
```

In [34]:
```
plot_history(cnn1_history.history)
```



In [35]:
```
update_summary(summary_df, 'Single CNN Layer', cnn1_model.evaluate(X_train, y_train)[1], cnn1_model_scores[
summary_df
```

```
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9849 - loss: 0.0428
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9077 - loss: 0.3588
```

Out[35]:

	Model	Train accuracy	Val accuracy	Test accuracy
0	Benchmark	0.1024	0.0944	0.1
1	Random Forest	1.0000	0.8812	0.8699
2	Single Hidden Layer	0.9575	0.8867	0.8764
3	3 Hidden Layers	0.9834	0.8914	0.8855
4	Regularized 3 Hidden Layers	0.9473	0.8928	0.8878
5	Regularized 5 Hidden Layers	0.9390	0.8948	0.8885
6	Single CNN Layer	0.9842	0.9146	0.9109

In [36]:
```
# Build the single cnn network model
cnn2_model = Sequential([
    Input(shape=X_train.shape[1:]),
    Reshape(target_shape=(X_train.shape[1], X_train.shape[2], 1)),  # explicitly state the 4th (channel) di
    Rescaling(1. / 255),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(32, (3, 3), activation='relu'),
    Conv2D(32, (3, 3), activation='relu'),
    # MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dropout(0.2),
    Dense(255, activation='relu'),
    Dropout(0.2),
    Dense(127, activation='relu'),
    Dropout(0.2),
    Dense(num_classes, activation='softmax')
])

# Compile the model
cnn2_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(cnn2_model.summary())
```

Model: "sequential_5"

Layer (type)	Output Shape	Param #
reshape_1 (Reshape)	(None, 28, 28, 1)	0
rescaling_5 (Rescaling)	(None, 28, 28, 1)	0
conv2d_1 (Conv2D)	(None, 26, 26, 64)	640
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 64)	0
conv2d_2 (Conv2D)	(None, 11, 11, 32)	18,464
conv2d_3 (Conv2D)	(None, 9, 9, 32)	9,248
flatten_5 (Flatten)	(None, 2592)	0
dropout_9 (Dropout)	(None, 2592)	0
dense_19 (Dense)	(None, 255)	661,215
dropout_10 (Dropout)	(None, 255)	0
dense_20 (Dense)	(None, 127)	32,512
dropout_11 (Dropout)	(None, 127)	0
dense_21 (Dense)	(None, 10)	1,280

Total params: 723,359 (2.76 MB)
Trainable params: 723,359 (2.76 MB)
Non-trainable params: 0 (0.00 B)
None

In [37]:
```
# Fit the model with EarlyStopping
keras.utils.set_random_seed(20240329)  # for reproducibility
cnn2_history = cnn2_model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100, batch_size=512,
    EarlyStopping(monitor='val_accuracy', patience=10, min_delta=0.001, verbose=1)], verbose=0)
```

```
Epoch 36: early stopping
```

In [38]:
```
# Evaluation of the model on the validation set
cnn2_model_scores = cnn2_model.evaluate(X_val, y_val)
print(f"Accuracy for multi cnn layers: {round(cnn2_model_scores[1], 4)}")
```

```
563/563 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - accuracy: 0.9156 - loss: 0.3100
Accuracy for multi cnn layers: 0.9174
```

In [39]:
```
plot_history(cnn2_history.history)
```

In [40]:
```
update_summary(summary_df, 'Multi CNN Layers', cnn2_model.evaluate(X_train, y_train)[1], cnn2_model_scores[
               cnn2_model.evaluate(X_test, y_test)[1])
```

```
summary_df
```

```
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.9892 - loss: 0.0362
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9163 - loss: 0.3207
```

Out[40]:

	Model	Train accuracy	Val accuracy	Test accuracy
0	Benchmark	0.1024	0.0944	0.1
1	Random Forest	1.0000	0.8812	0.8699
2	Single Hidden Layer	0.9575	0.8867	0.8764
3	3 Hidden Layers	0.9834	0.8914	0.8855
4	Regularized 3 Hidden Layers	0.9473	0.8928	0.8878
5	Regularized 5 Hidden Layers	0.9390	0.8948	0.8885
6	Single CNN Layer	0.9842	0.9146	0.9109
7	Multi CNN Layers	0.9885	0.9174	0.9158

We build 2 more networks featuring the convolutional layers:

- A single convolutional network: a convolutional 2D layer with 3x3 filter size follows by a max pooling layer with 2x2 pool size follows by 2 hidden layers size 255 and 127 respectively.
- A multi convolutional network: a convolutional 2D layer with 3x3 filter size follows by a max pooling layer with 2x2 pool size follows by 2 convolutional layers with 3x3 filter size follows by 2 hidden layers size 255 and 127 respectively.

Both convolutional networks feature early stopping and dropout layers after each hidden layer.

From the summary table, both convolutional networks performs even better than previous networks. Not only do they fit the training data better, they also raise the accuracy for both the validation and test data above 90%. It seems the convolutional layers help extracting more relevant features from the data, thus improve the classification accuracy. Among the 2 convolutional networks, the multi convolutional network does slightly better, suggesting that using more convolutional layers extracts even more details from the images. However, introducing more convolutional layers significantly increase the computational effort and training time. The pooling layers balance it out slightly by reducing the data resolution. However, it should be noted that more pooling layers might compress the images too much, causing the accuracy to fall. The balance between computational requirements and accuracy thus should be experimented and chosen carefully.

Try to use a pre-trained network to improve accuracy.

In [41]:
```
import tensorflow as tf

def preprocess_resnet(images):
    images = np.stack([images]*3, axis=-1) / 255.0  # Convert to 3 channels and normalize
    images = tf.image.resize(images, [32, 32])  # Resize images
    return images

# Load and preprocess data
X_train_resnet = preprocess_resnet(X_train)
X_val_resnet = preprocess_resnet(X_val)
X_test_resnet = preprocess_resnet(X_test)

X_train_resnet.shape
```

Out[41]:
```
TensorShape([42000, 32, 32, 3])
```

In [42]:
```
from keras.applications.efficientnet import EfficientNetB0
from keras.layers import GlobalAveragePooling2D
from keras.applications import ResNet50, ResNet101
from keras.models import Model


# Load pre-trained ResNet50 model without the top layers as we do not want to classify for 1000 classes but
base_model = ResNet50(include_top=False, weights='imagenet', input_shape=(32, 32, 3))
base_model.trainable = False

# Model definition
output = base_model.output
output = GlobalAveragePooling2D()(output)
output = Dense(256, activation="relu")(output)
output = Dense(10, activation="softmax")(output)
fine_tuned_model = Model(inputs=base_model.input, outputs=output)

# Compile the model
fine_tuned_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Model summary to check the architecture
print(fine_tuned_model.summary())
```

Model: "functional_7"

Layer (type)	Output Shape	Param #	Connected to
input_layer_6 (InputLayer)	(None, 32, 32, 3)	0	-
conv1_pad (ZeroPadding2D)	(None, 38, 38, 3)	0	input_layer_6[0]…
conv1_conv (Conv2D)	(None, 16, 16, 64)	9,472	conv1_pad[0][0]
conv1_bn (BatchNormalizatio…	(None, 16, 16, 64)	256	conv1_conv[0][0]
conv1_relu (Activation)	(None, 16, 16, 64)	0	conv1_bn[0][0]
pool1_pad (ZeroPadding2D)	(None, 18, 18, 64)	0	conv1_relu[0][0]
pool1_pool (MaxPooling2D)	(None, 8, 8, 64)	0	pool1_pad[0][0]
conv2_block1_1_conv (Conv2D)	(None, 8, 8, 64)	4,160	pool1_pool[0][0]
conv2_block1_1_bn (BatchNormalizatio…	(None, 8, 8, 64)	256	conv2_block1_1_c…
conv2_block1_1_relu (Activation)	(None, 8, 8, 64)	0	conv2_block1_1_b…
conv2_block1_2_conv (Conv2D)	(None, 8, 8, 64)	36,928	conv2_block1_1_r…
conv2_block1_2_bn (BatchNormalizatio…	(None, 8, 8, 64)	256	conv2_block1_2_c…
conv2_block1_2_relu (Activation)	(None, 8, 8, 64)	0	conv2_block1_2_b…
conv2_block1_0_conv (Conv2D)	(None, 8, 8, 256)	16,640	pool1_pool[0][0]
conv2_block1_3_conv (Conv2D)	(None, 8, 8, 256)	16,640	conv2_block1_2_r…
conv2_block1_0_bn (BatchNormalizatio…	(None, 8, 8, 256)	1,024	conv2_block1_0_c…
conv2_block1_3_bn (BatchNormalizatio…	(None, 8, 8, 256)	1,024	conv2_block1_3_c…
conv2_block1_add (Add)	(None, 8, 8, 256)	0	conv2_block1_0_b… conv2_block1_3_b…
conv2_block1_out (Activation)	(None, 8, 8, 256)	0	conv2_block1_add…
conv2_block2_1_conv (Conv2D)	(None, 8, 8, 64)	16,448	conv2_block1_out…
conv2_block2_1_bn (BatchNormalizatio…	(None, 8, 8, 64)	256	conv2_block2_1_c…
conv2_block2_1_relu (Activation)	(None, 8, 8, 64)	0	conv2_block2_1_b…
conv2_block2_2_conv (Conv2D)	(None, 8, 8, 64)	36,928	conv2_block2_1_r…
conv2_block2_2_bn (BatchNormalizatio…	(None, 8, 8, 64)	256	conv2_block2_2_c…
conv2_block2_2_relu (Activation)	(None, 8, 8, 64)	0	conv2_block2_2_b…
conv2_block2_3_conv (Conv2D)	(None, 8, 8, 256)	16,640	conv2_block2_2_r…
conv2_block2_3_bn (BatchNormalizatio…	(None, 8, 8, 256)	1,024	conv2_block2_3_c…

conv2_block2_add (Add)	(None, 8, 8, 256)	0	conv2_block1_out… conv2_block2_3_b…
conv2_block2_out (Activation)	(None, 8, 8, 256)	0	conv2_block2_add…
conv2_block3_1_conv (Conv2D)	(None, 8, 8, 64)	16,448	conv2_block2_out…
conv2_block3_1_bn (BatchNormalizatio…	(None, 8, 8, 64)	256	conv2_block3_1_c…
conv2_block3_1_relu (Activation)	(None, 8, 8, 64)	0	conv2_block3_1_b…
conv2_block3_2_conv (Conv2D)	(None, 8, 8, 64)	36,928	conv2_block3_1_r…
conv2_block3_2_bn (BatchNormalizatio…	(None, 8, 8, 64)	256	conv2_block3_2_c…
conv2_block3_2_relu (Activation)	(None, 8, 8, 64)	0	conv2_block3_2_b…
conv2_block3_3_conv (Conv2D)	(None, 8, 8, 256)	16,640	conv2_block3_2_r…
conv2_block3_3_bn (BatchNormalizatio…	(None, 8, 8, 256)	1,024	conv2_block3_3_c…
conv2_block3_add (Add)	(None, 8, 8, 256)	0	conv2_block2_out… conv2_block3_3_b…
conv2_block3_out (Activation)	(None, 8, 8, 256)	0	conv2_block3_add…
conv3_block1_1_conv (Conv2D)	(None, 4, 4, 128)	32,896	conv2_block3_out…
conv3_block1_1_bn (BatchNormalizatio…	(None, 4, 4, 128)	512	conv3_block1_1_c…
conv3_block1_1_relu (Activation)	(None, 4, 4, 128)	0	conv3_block1_1_b…
conv3_block1_2_conv (Conv2D)	(None, 4, 4, 128)	147,584	conv3_block1_1_r…
conv3_block1_2_bn (BatchNormalizatio…	(None, 4, 4, 128)	512	conv3_block1_2_c…
conv3_block1_2_relu (Activation)	(None, 4, 4, 128)	0	conv3_block1_2_b…
conv3_block1_0_conv (Conv2D)	(None, 4, 4, 512)	131,584	conv2_block3_out…
conv3_block1_3_conv (Conv2D)	(None, 4, 4, 512)	66,048	conv3_block1_2_r…
conv3_block1_0_bn (BatchNormalizatio…	(None, 4, 4, 512)	2,048	conv3_block1_0_c…
conv3_block1_3_bn (BatchNormalizatio…	(None, 4, 4, 512)	2,048	conv3_block1_3_c…
conv3_block1_add (Add)	(None, 4, 4, 512)	0	conv3_block1_0_b… conv3_block1_3_b…
conv3_block1_out (Activation)	(None, 4, 4, 512)	0	conv3_block1_add…
conv3_block2_1_conv (Conv2D)	(None, 4, 4, 128)	65,664	conv3_block1_out…
conv3_block2_1_bn (BatchNormalizatio…	(None, 4, 4, 128)	512	conv3_block2_1_c…
conv3_block2_1_relu (Activation)	(None, 4, 4, 128)	0	conv3_block2_1_b…
conv3_block2_2_conv (Conv2D)	(None, 4, 4, 128)	147,584	conv3_block2_1_r…

conv3_block2_2_bn (BatchNormalizatio…	(None, 4, 4, 128)	512	conv3_block2_2_c…
conv3_block2_2_relu (Activation)	(None, 4, 4, 128)	0	conv3_block2_2_b…
conv3_block2_3_conv (Conv2D)	(None, 4, 4, 512)	66,048	conv3_block2_2_r…
conv3_block2_3_bn (BatchNormalizatio…	(None, 4, 4, 512)	2,048	conv3_block2_3_c…
conv3_block2_add (Add)	(None, 4, 4, 512)	0	conv3_block1_out… conv3_block2_3_b…
conv3_block2_out (Activation)	(None, 4, 4, 512)	0	conv3_block2_add…
conv3_block3_1_conv (Conv2D)	(None, 4, 4, 128)	65,664	conv3_block2_out…
conv3_block3_1_bn (BatchNormalizatio…	(None, 4, 4, 128)	512	conv3_block3_1_c…
conv3_block3_1_relu (Activation)	(None, 4, 4, 128)	0	conv3_block3_1_b…
conv3_block3_2_conv (Conv2D)	(None, 4, 4, 128)	147,584	conv3_block3_1_r…
conv3_block3_2_bn (BatchNormalizatio…	(None, 4, 4, 128)	512	conv3_block3_2_c…
conv3_block3_2_relu (Activation)	(None, 4, 4, 128)	0	conv3_block3_2_b…
conv3_block3_3_conv (Conv2D)	(None, 4, 4, 512)	66,048	conv3_block3_2_r…
conv3_block3_3_bn (BatchNormalizatio…	(None, 4, 4, 512)	2,048	conv3_block3_3_c…
conv3_block3_add (Add)	(None, 4, 4, 512)	0	conv3_block2_out… conv3_block3_3_b…
conv3_block3_out (Activation)	(None, 4, 4, 512)	0	conv3_block3_add…
conv3_block4_1_conv (Conv2D)	(None, 4, 4, 128)	65,664	conv3_block3_out…
conv3_block4_1_bn (BatchNormalizatio…	(None, 4, 4, 128)	512	conv3_block4_1_c…
conv3_block4_1_relu (Activation)	(None, 4, 4, 128)	0	conv3_block4_1_b…
conv3_block4_2_conv (Conv2D)	(None, 4, 4, 128)	147,584	conv3_block4_1_r…
conv3_block4_2_bn (BatchNormalizatio…	(None, 4, 4, 128)	512	conv3_block4_2_c…
conv3_block4_2_relu (Activation)	(None, 4, 4, 128)	0	conv3_block4_2_b…
conv3_block4_3_conv (Conv2D)	(None, 4, 4, 512)	66,048	conv3_block4_2_r…
conv3_block4_3_bn (BatchNormalizatio…	(None, 4, 4, 512)	2,048	conv3_block4_3_c…
conv3_block4_add (Add)	(None, 4, 4, 512)	0	conv3_block3_out… conv3_block4_3_b…
conv3_block4_out (Activation)	(None, 4, 4, 512)	0	conv3_block4_add…
conv4_block1_1_conv (Conv2D)	(None, 2, 2, 256)	131,328	conv3_block4_out…
conv4_block1_1_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block1_1_c…

conv4_block1_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block1_1_b…
conv4_block1_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block1_1_r…
conv4_block1_2_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block1_2_c…
conv4_block1_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block1_2_b…
conv4_block1_0_conv (Conv2D)	(None, 2, 2, 1024)	525,312	conv3_block4_out…
conv4_block1_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block1_2_r…
conv4_block1_0_bn (BatchNormalizatio…	(None, 2, 2, 1024)	4,096	conv4_block1_0_c…
conv4_block1_3_bn (BatchNormalizatio…	(None, 2, 2, 1024)	4,096	conv4_block1_3_c…
conv4_block1_add (Add)	(None, 2, 2, 1024)	0	conv4_block1_0_b… conv4_block1_3_b…
conv4_block1_out (Activation)	(None, 2, 2, 1024)	0	conv4_block1_add…
conv4_block2_1_conv (Conv2D)	(None, 2, 2, 256)	262,400	conv4_block1_out…
conv4_block2_1_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block2_1_c…
conv4_block2_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block2_1_b…
conv4_block2_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block2_1_r…
conv4_block2_2_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block2_2_c…
conv4_block2_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block2_2_b…
conv4_block2_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block2_2_r…
conv4_block2_3_bn (BatchNormalizatio…	(None, 2, 2, 1024)	4,096	conv4_block2_3_c…
conv4_block2_add (Add)	(None, 2, 2, 1024)	0	conv4_block1_out… conv4_block2_3_b…
conv4_block2_out (Activation)	(None, 2, 2, 1024)	0	conv4_block2_add…
conv4_block3_1_conv (Conv2D)	(None, 2, 2, 256)	262,400	conv4_block2_out…
conv4_block3_1_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block3_1_c…
conv4_block3_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block3_1_b…
conv4_block3_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block3_1_r…
conv4_block3_2_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block3_2_c…
conv4_block3_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block3_2_b…
conv4_block3_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block3_2_r…
conv4_block3_3_bn (BatchNormalizatio…	(None, 2, 2, 1024)	4,096	conv4_block3_3_c…

conv4_block3_add (Add)	(None, 2, 2, 1024)	0	conv4_block2_out… conv4_block3_3_b…
conv4_block3_out (Activation)	(None, 2, 2, 1024)	0	conv4_block3_add…
conv4_block4_1_conv (Conv2D)	(None, 2, 2, 256)	262,400	conv4_block3_out…
conv4_block4_1_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block4_1_c…
conv4_block4_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block4_1_b…
conv4_block4_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block4_1_r…
conv4_block4_2_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block4_2_c…
conv4_block4_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block4_2_b…
conv4_block4_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block4_2_r…
conv4_block4_3_bn (BatchNormalizatio…	(None, 2, 2, 1024)	4,096	conv4_block4_3_c…
conv4_block4_add (Add)	(None, 2, 2, 1024)	0	conv4_block3_out… conv4_block4_3_b…
conv4_block4_out (Activation)	(None, 2, 2, 1024)	0	conv4_block4_add…
conv4_block5_1_conv (Conv2D)	(None, 2, 2, 256)	262,400	conv4_block4_out…
conv4_block5_1_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block5_1_c…
conv4_block5_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block5_1_b…
conv4_block5_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block5_1_r…
conv4_block5_2_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block5_2_c…
conv4_block5_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block5_2_b…
conv4_block5_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block5_2_r…
conv4_block5_3_bn (BatchNormalizatio…	(None, 2, 2, 1024)	4,096	conv4_block5_3_c…
conv4_block5_add (Add)	(None, 2, 2, 1024)	0	conv4_block4_out… conv4_block5_3_b…
conv4_block5_out (Activation)	(None, 2, 2, 1024)	0	conv4_block5_add…
conv4_block6_1_conv (Conv2D)	(None, 2, 2, 256)	262,400	conv4_block5_out…
conv4_block6_1_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block6_1_c…
conv4_block6_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block6_1_b…
conv4_block6_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block6_1_r…
conv4_block6_2_bn (BatchNormalizatio…	(None, 2, 2, 256)	1,024	conv4_block6_2_c…
conv4_block6_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block6_2_b…

conv4_block6_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block6_2_r…
conv4_block6_3_bn (BatchNormalizatio…	(None, 2, 2, 1024)	4,096	conv4_block6_3_c…
conv4_block6_add (Add)	(None, 2, 2, 1024)	0	conv4_block5_out… conv4_block6_3_b…
conv4_block6_out (Activation)	(None, 2, 2, 1024)	0	conv4_block6_add…
conv5_block1_1_conv (Conv2D)	(None, 1, 1, 512)	524,800	conv4_block6_out…
conv5_block1_1_bn (BatchNormalizatio…	(None, 1, 1, 512)	2,048	conv5_block1_1_c…
conv5_block1_1_relu (Activation)	(None, 1, 1, 512)	0	conv5_block1_1_b…
conv5_block1_2_conv (Conv2D)	(None, 1, 1, 512)	2,359,808	conv5_block1_1_r…
conv5_block1_2_bn (BatchNormalizatio…	(None, 1, 1, 512)	2,048	conv5_block1_2_c…
conv5_block1_2_relu (Activation)	(None, 1, 1, 512)	0	conv5_block1_2_b…
conv5_block1_0_conv (Conv2D)	(None, 1, 1, 2048)	2,099,200	conv4_block6_out…
conv5_block1_3_conv (Conv2D)	(None, 1, 1, 2048)	1,050,624	conv5_block1_2_r…
conv5_block1_0_bn (BatchNormalizatio…	(None, 1, 1, 2048)	8,192	conv5_block1_0_c…
conv5_block1_3_bn (BatchNormalizatio…	(None, 1, 1, 2048)	8,192	conv5_block1_3_c…
conv5_block1_add (Add)	(None, 1, 1, 2048)	0	conv5_block1_0_b… conv5_block1_3_b…
conv5_block1_out (Activation)	(None, 1, 1, 2048)	0	conv5_block1_add…
conv5_block2_1_conv (Conv2D)	(None, 1, 1, 512)	1,049,088	conv5_block1_out…
conv5_block2_1_bn (BatchNormalizatio…	(None, 1, 1, 512)	2,048	conv5_block2_1_c…
conv5_block2_1_relu (Activation)	(None, 1, 1, 512)	0	conv5_block2_1_b…
conv5_block2_2_conv (Conv2D)	(None, 1, 1, 512)	2,359,808	conv5_block2_1_r…
conv5_block2_2_bn (BatchNormalizatio…	(None, 1, 1, 512)	2,048	conv5_block2_2_c…
conv5_block2_2_relu (Activation)	(None, 1, 1, 512)	0	conv5_block2_2_b…
conv5_block2_3_conv (Conv2D)	(None, 1, 1, 2048)	1,050,624	conv5_block2_2_r…
conv5_block2_3_bn (BatchNormalizatio…	(None, 1, 1, 2048)	8,192	conv5_block2_3_c…
conv5_block2_add (Add)	(None, 1, 1, 2048)	0	conv5_block1_out… conv5_block2_3_b…
conv5_block2_out (Activation)	(None, 1, 1, 2048)	0	conv5_block2_add…
conv5_block3_1_conv (Conv2D)	(None, 1, 1, 512)	1,049,088	conv5_block2_out…
conv5_block3_1_bn (BatchNormalizatio…	(None, 1, 1, 512)	2,048	conv5_block3_1_c…

conv5_block3_1_relu (Activation)	(None, 1, 1, 512)	0	conv5_block3_1_b…
conv5_block3_2_conv (Conv2D)	(None, 1, 1, 512)	2,359,808	conv5_block3_1_r…
conv5_block3_2_bn (BatchNormalizatio…	(None, 1, 1, 512)	2,048	conv5_block3_2_c…
conv5_block3_2_relu (Activation)	(None, 1, 1, 512)	0	conv5_block3_2_b…
conv5_block3_3_conv (Conv2D)	(None, 1, 1, 2048)	1,050,624	conv5_block3_2_r…
conv5_block3_3_bn (BatchNormalizatio…	(None, 1, 1, 2048)	8,192	conv5_block3_3_c…
conv5_block3_add (Add)	(None, 1, 1, 2048)	0	conv5_block2_out… conv5_block3_3_b…
conv5_block3_out (Activation)	(None, 1, 1, 2048)	0	conv5_block3_add…
global_average_poo… (GlobalAveragePool…	(None, 2048)	0	conv5_block3_out…
dense_22 (Dense)	(None, 256)	524,544	global_average_p…
dense_23 (Dense)	(None, 10)	2,570	dense_22[0][0]

Total params: 24,114,826 (91.99 MB)
Trainable params: 527,114 (2.01 MB)
Non-trainable params: 23,587,712 (89.98 MB)
None

In [43]:
```
# Fit the model with EarlyStopping
keras.utils.set_random_seed(20240329)  # for reproducibility
fine_tune_history = fine_tuned_model.fit(X_train_resnet, y_train, validation_data=(X_val_resnet, y_val), ep
```

In [44]:
```
# Evaluation of the model on the validation set
fine_tuned_model_scores = fine_tuned_model.evaluate(X_val_resnet, y_val)
print(f"Accuracy for pretrained model: {round(fine_tuned_model_scores[1], 4)}")
```

```
563/563 ━━━━━━━━━━━━━━━━━━━━ 27s 48ms/step - accuracy: 0.7787 - loss: 0.5926
Accuracy for pretrained model: 0.7833
```

In [45]:
```
update_summary(summary_df, 'Pre-Trained Model', fine_tuned_model.evaluate(X_train_resnet, y_train)[1], fine
summary_df
```

```
1313/1313 ━━━━━━━━━━━━━━━━━━━━ 61s 47ms/step - accuracy: 0.7817 - loss: 0.5789
313/313 ━━━━━━━━━━━━━━━━━━━━ 15s 47ms/step - accuracy: 0.7717 - loss: 0.6013
```

Out[45]:

	Model	Train accuracy	Val accuracy	Test accuracy
0	Benchmark	0.1024	0.0944	0.1
1	Random Forest	1.0000	0.8812	0.8699
2	Single Hidden Layer	0.9575	0.8867	0.8764
3	3 Hidden Layers	0.9834	0.8914	0.8855
4	Regularized 3 Hidden Layers	0.9473	0.8928	0.8878
5	Regularized 5 Hidden Layers	0.9390	0.8948	0.8885
6	Single CNN Layer	0.9842	0.9146	0.9109
7	Multi CNN Layers	0.9885	0.9174	0.9158
8	Pre-Trained Model	0.7822	0.7833	0.7732

In order to improve the accuracy even further, we try to fine tune ResNet50, a pretrained network, to fit our fashion mnist data. The ResNet50 requires some data manipulation as it originally only accepts images with 3 color channels (RGB) and minimum image resolution of 32x32. The manipulation steps are as follows:

- Triplicate the fashion mnist image to obtain a 3-color-channel images.
- Rescale each channel by dividing by 255.
- Resize the 3-color-channel images to 32x32 (from 28x28).

The fine tuned network then consist of:

- The ResNet50 base network without the top layers.
- A global average pooling layer.
- A hidden layer with 256 neurons.
- An output layer with 10 neurons representing the 10 classes with a 'softmax' activation function.

After training with 10 epochs, unfortunately, the fine tuned network does not perform as well as any previous networks. It even does worse than the random forest classification. It is possible that the data manipulation negatively impacts the data quality or the fine tuned network requires some special layers to transfer learning properly from the pretrained network. Either of these possibility causes the classification accuracy falls greatly.

Select a final model and evaluate it on the test set. How does the test error compare to the validation error?

In [46]:
```
summary_df
```

Out[46]:

	Model	Train accuracy	Val accuracy	Test accuracy
0	Benchmark	0.1024	0.0944	0.1
1	Random Forest	1.0000	0.8812	0.8699
2	Single Hidden Layer	0.9575	0.8867	0.8764
3	3 Hidden Layers	0.9834	0.8914	0.8855
4	Regularized 3 Hidden Layers	0.9473	0.8928	0.8878
5	Regularized 5 Hidden Layers	0.9390	0.8948	0.8885
6	Single CNN Layer	0.9842	0.9146	0.9109
7	Multi CNN Layers	0.9885	0.9174	0.9158
8	Pre-Trained Model	0.7822	0.7833	0.7732

From the summary table, the model with the highest test accuracy is the multi convolutional layers network, which has a test accuracy of 0.9188. This model also shows high validation accuracy (0.9212), indicating good generalization from the training to the unseen data. The test error for this model is 1 - 0.9188 = 0.0812, and the validation error is 1 - 0.9212 = 0.0788. The very close performance on both the validation and test sets suggests that the model generalizes well, with only a slight increase in error from validation to test, indicating minimal overfitting.

In [46]:
```
```