Introduction

Classify fashion images on the MNIST data

TASK

Take the alternative version of the famous “MNIST dataset”, which consists of images of Zalando’s articles. Your task is to correctly classify the images into one of the ten categories, such as dress or shirt. The images are in exactly the same format as we saw for the handwritten digits: 28x28 pixel grayscale images. The task is to build deep neural network models to predict the items. You can use either sklearn or keras; to get the data, go to the corresponding Kaggle page or use the fashion_mnist.load_data() function from the keras.datasets module. Make sure you split the training set into two sets: one for training your models on and one for validation and model selection. You can work with a relatively small train set if you have computational problems.

Import libraries

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import fashion_mnist
from sklearn.model_selection import train_test_split
from keras.regularizers import l1
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, Input, Rescaling, BatchNormalization, GlobalAveragePooling
from keras.models import Sequential
from keras.utils import to_categorical
from keras.optimizers import Adam
from keras.callbacks import LearningRateScheduler
from keras.applications import VGG16, MobileNet
from tensorflow.image import resize
from keras.callbacks import EarlyStopping, ReduceLROnPlateau
from keras.applications.efficientnet import EfficientNetB0, preprocess_input
from skimage.transform import resize
from keras.applications.mobilenet import preprocess_input
import tensorflow as tf
from keras.applications import MobileNetV2

# ignore warnings
import warnings
warnings.filterwarnings('ignore')
```

```
d:\anaconda\Lib\site-packages\paramiko\transport.py:219: CryptographyDeprecationWarning: Blowfish has been deprecated
  "class": algorithms.Blowfish,
```

Question 1

What would be an appropriate metric to evaluate your models? Why?

Answer 1

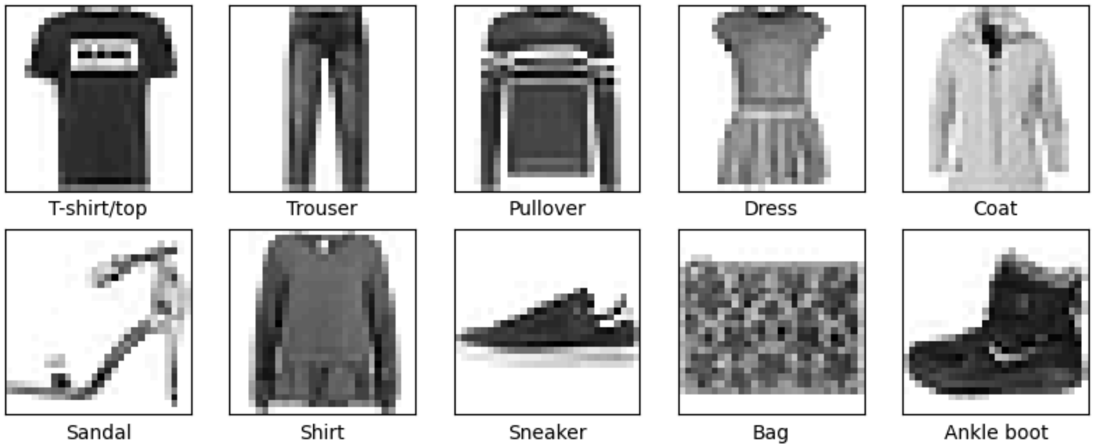
The most appropriate metric for evaluating models on the Fashion MNIST dataset is accuracy. Accuracy measures the percentage of correctly classified images across all categories (dress, t-shirt, etc.). Since the goal is to maximize the number of correctly identified fashion items, and we have a balanced dataset, accuracy provides a direct and intuitive evaluation of model performance.

Question 2

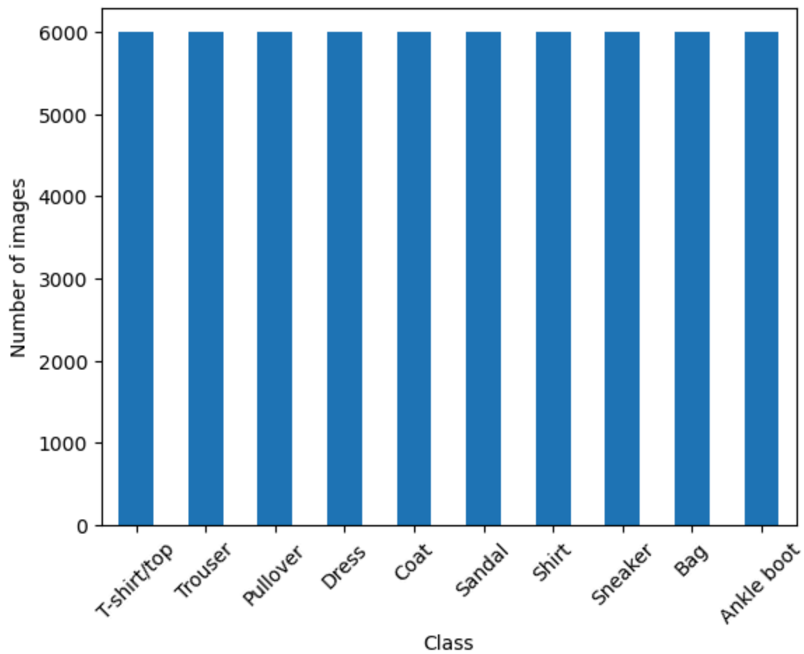
Get the data and show some example images from the data.

In [2]:
```
# define random seed for reproducibility
prng = np.random.RandomState(20240402)

# load the Fashion MNIST dataset
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

# define the class names of the Fashion MNIST dataset
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# plot an example image for each class in the Fashion MNIST dataset
plt.figure(figsize=(10, 10))
for i in range(10):
    plt.subplot(5, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
```

```
    plt.imshow(train_images[train_labels == i][0], cmap=plt.cm.binary)
    plt.xlabel(class_names[i])
```



In [3]:
```
# count the number of images in each class in the Fashion MNIST dataset
class_counts = pd.Series(train_labels).value_counts().sort_index()
plt.figure()
class_counts.plot(kind='bar')
plt.xlabel('Class')
plt.ylabel('Number of images')
plt.xticks(range(10), class_names, rotation=45)
plt.show()

# print the count of images in each class in the Fashion MNIST dataset in both test and train sets in a tabular format using class
class_counts = pd.DataFrame({'train': pd.Series(train_labels).value_counts().sort_index(),
                              'test': pd.Series(test_labels).value_counts().sort_index()})
class_counts.index = class_names
class_counts
```



Out[3]:

	train	test
T-shirt/top	6000	1000
Trouser	6000	1000
Pullover	6000	1000
Dress	6000	1000
Coat	6000	1000
Sandal	6000	1000
Shirt	6000	1000
Sneaker	6000	1000
Bag	6000	1000
Ankle boot	6000	1000

Answer 2

We loaded the Fashion MNIST dataset, defined class names for clarity, and plotted an example image for each of the ten classes to visually understand the types of items in the dataset. Additionally, we counted and visualized the number of images per class in both the training and test sets, revealing the distribution and ensuring a balanced dataset, which is crucial for training unbiased models.

Question 3

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 [4]:
```
# split the training set into a training set and a validation set
train_images, val_images, train_labels, val_labels = train_test_split(train_images, train_labels, test_size=0.2, random_state=prng
```

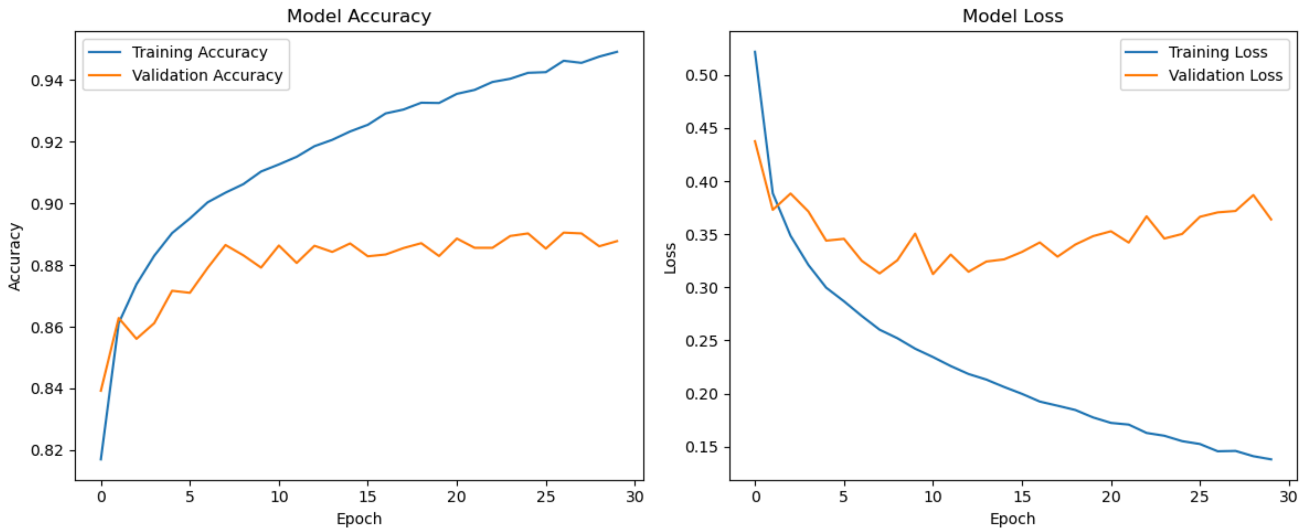
In [5]:
```
# convert labels to one-hot encoding
train_labels = to_categorical(train_labels)
val_labels = to_categorical(val_labels)
test_labels = to_categorical(test_labels)

# count the number of classes
num_classes = len(class_names)

# build the model with a single hidden layer with 128 neurons
model = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Rescaling(1./255), # normalization for getting values between 0 and 1
    Dense(128, activation='relu'),
    Dense(num_classes, activation='softmax')
])

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

# print the summary of the model
print(model.summary())

# train the model
history = model.fit(train_images, train_labels, epochs=30,
                    validation_data=(val_images, val_labels))

# define a function to plot the training history
def plot_training_history(history):
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

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

    plt.tight_layout()
```

```
    plt.show()

# plot the training history
plot_training_history(history)
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
rescaling (Rescaling)	(None, 784)	0
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 10)	1,290

```
 Total params: 101,770 (397.54 KB)
 Trainable params: 101,770 (397.54 KB)
 Non-trainable params: 0 (0.00 B)
None
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.7712 - loss: 0.6626 - val_accuracy: 0.8393 - val_loss: 0.4377
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8601 - loss: 0.3976 - val_accuracy: 0.8628 - val_loss: 0.3730
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8722 - loss: 0.3516 - val_accuracy: 0.8561 - val_loss: 0.3884
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8830 - loss: 0.3188 - val_accuracy: 0.8612 - val_loss: 0.3715
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8889 - loss: 0.3018 - val_accuracy: 0.8717 - val_loss: 0.3440
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8980 - loss: 0.2830 - val_accuracy: 0.8710 - val_loss: 0.3457
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.8998 - loss: 0.2717 - val_accuracy: 0.8791 - val_loss: 0.3250
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9043 - loss: 0.2597 - val_accuracy: 0.8865 - val_loss: 0.3131
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9076 - loss: 0.2467 - val_accuracy: 0.8832 - val_loss: 0.3255
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9114 - loss: 0.2405 - val_accuracy: 0.8792 - val_loss: 0.3506
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9123 - loss: 0.2371 - val_accuracy: 0.8863 - val_loss: 0.3125
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9173 - loss: 0.2239 - val_accuracy: 0.8807 - val_loss: 0.3309
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9200 - loss: 0.2136 - val_accuracy: 0.8863 - val_loss: 0.3147
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9223 - loss: 0.2106 - val_accuracy: 0.8842 - val_loss: 0.3243
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9239 - loss: 0.2048 - val_accuracy: 0.8870 - val_loss: 0.3264
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9272 - loss: 0.1958 - val_accuracy: 0.8828 - val_loss: 0.3334
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9280 - loss: 0.1962 - val_accuracy: 0.8834 - val_loss: 0.3423
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9295 - loss: 0.1912 - val_accuracy: 0.8855 - val_loss: 0.3289
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9343 - loss: 0.1800 - val_accuracy: 0.8871 - val_loss: 0.3403
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9351 - loss: 0.1743 - val_accuracy: 0.8829 - val_loss: 0.3482
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9369 - loss: 0.1693 - val_accuracy: 0.8886 - val_loss: 0.3528
Epoch 22/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9388 - loss: 0.1674 - val_accuracy: 0.8856 - val_loss: 0.3421
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9391 - loss: 0.1608 - val_accuracy: 0.8856 - val_loss: 0.3669
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9414 - loss: 0.1579 - val_accuracy: 0.8894 - val_loss: 0.3460
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9432 - loss: 0.1523 - val_accuracy: 0.8903 - val_loss: 0.3503
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9452 - loss: 0.1470 - val_accuracy: 0.8853 - val_loss: 0.3665
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9467 - loss: 0.1410 - val_accuracy: 0.8905 - val_loss: 0.3706
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9432 - loss: 0.1497 - val_accuracy: 0.8903 - val_loss: 0.3719
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9471 - loss: 0.1439 - val_accuracy: 0.8861 - val_loss: 0.3869
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9501 - loss: 0.1344 - val_accuracy: 0.8878 - val_loss: 0.3640
```



In [6]:
```
# print the highest validation accuracy and the epoch at which it was obtained
best_epoch = np.argmax(history.history['val_accuracy']) + 1
best_val_accuracy = history.history['val_accuracy'][best_epoch - 1]
print(f'Best validation accuracy: {best_val_accuracy:.4f} at epoch {best_epoch}')
# print the smallest validation loss and the epoch at which it was obtained
best_epoch_loss = np.argmin(history.history['val_loss']) + 1
best_val_loss = history.history['val_loss'][best_epoch_loss - 1]
print(f'Best validation loss: {best_val_loss:.4f} at epoch {best_epoch_loss}')
```

```
Best validation accuracy: 0.8905 at epoch 27
Best validation loss: 0.3125 at epoch 11
```

When training the model for 30 epochs, we observe that the accuracy on the training data consistently increases, suggesting the model is learning and adapting well to the training set. In contrast, the validation accuracy improves at the beginning but then stabilizes around 89% (epoch 27), showing limited gains from additional training past a certain point. The validation loss decreases initially but then starts to vary and generally trend upwards, which can indicate that the model is overfitting to the training data and not generalizing as effectively to new, unseen data. The divergence between training performance and validation performance suggests that improvements could be made, potentially by tuning the model or using regularization strategies to achieve better generalization.

Question 4

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.

Let's outline seven different models with varying architectures to experiment with:

- Model 1: Two hidden layers with increasing number of nodes.
- Model 2: Two hidden layers with decreasing number of nodes.
- Model 3: Two hidden layers with dropout regularization.
- Model 4: Three hidden layers
- Model 5: Two hidden layers with L1 regularization (LASSO)
- Model 6: Two hidden layers with L1 regularization and dropout
- Model 7: Three hidden layers with L1 regularization and dropout

Model 1: Two hidden layers with increasing number of nodes

In [7]:
```
# build and compile Model 1
model_1 = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Rescaling(1./255),
    Dense(256, activation='relu'),  # First hidden layer with more nodes
    Dense(512, activation='relu'),  # Second hidden layer with even more nodes
    Dense(num_classes, activation='softmax')  # Output layer with 10 classes
])

model_1.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# summarize Model 1
model_1.summary()

# train Model 1
history_1 = model_1.fit(
```

```python
    train_images, train_labels,
    epochs=30,
    validation_data=(val_images, val_labels)
)

# use the existing function to plot the training history for Model 1
plot_training_history(history_1)
```

Model: "sequential_1"

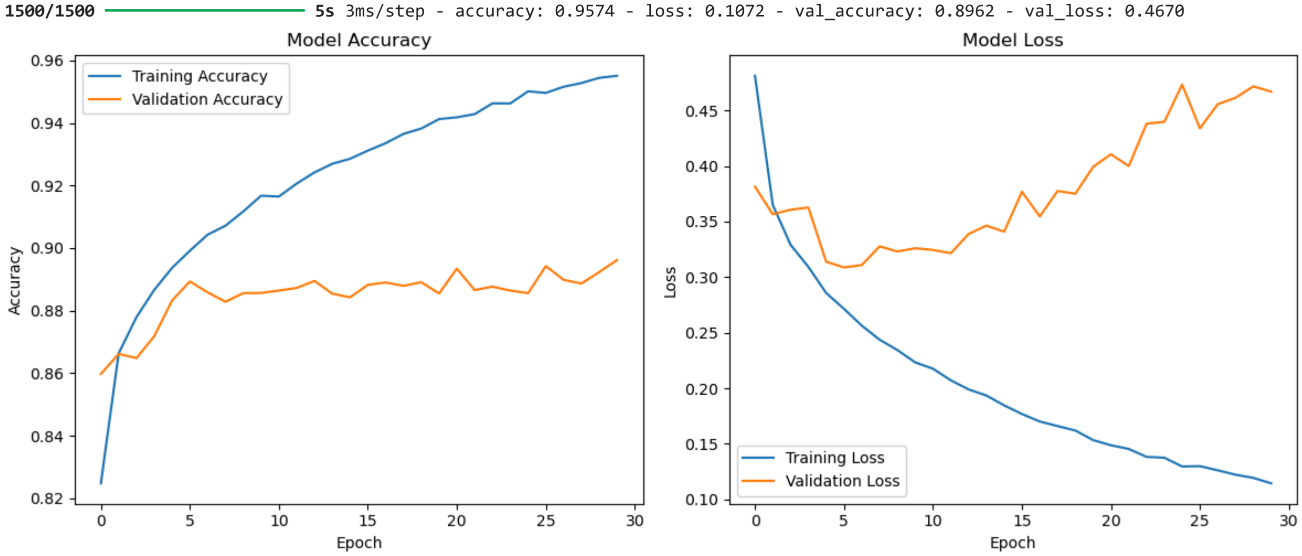
Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
rescaling_1 (Rescaling)	(None, 784)	0
dense_2 (Dense)	(None, 256)	200,960
dense_3 (Dense)	(None, 512)	131,584
dense_4 (Dense)	(None, 10)	5,130

Total params: 337,674 (1.29 MB)

Trainable params: 337,674 (1.29 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7794 - loss: 0.6059 - val_accuracy: 0.8597 - val_loss: 0.3815
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8649 - loss: 0.3675 - val_accuracy: 0.8662 - val_loss: 0.3566
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8774 - loss: 0.3324 - val_accuracy: 0.8648 - val_loss: 0.3607
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8851 - loss: 0.3119 - val_accuracy: 0.8718 - val_loss: 0.3627
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8942 - loss: 0.2815 - val_accuracy: 0.8832 - val_loss: 0.3139
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9009 - loss: 0.2674 - val_accuracy: 0.8893 - val_loss: 0.3088
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9052 - loss: 0.2545 - val_accuracy: 0.8858 - val_loss: 0.3108
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9079 - loss: 0.2419 - val_accuracy: 0.8828 - val_loss: 0.3277
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9128 - loss: 0.2276 - val_accuracy: 0.8856 - val_loss: 0.3231
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9194 - loss: 0.2164 - val_accuracy: 0.8857 - val_loss: 0.3260
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9171 - loss: 0.2129 - val_accuracy: 0.8864 - val_loss: 0.3245
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9222 - loss: 0.2037 - val_accuracy: 0.8873 - val_loss: 0.3216
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9263 - loss: 0.1921 - val_accuracy: 0.8895 - val_loss: 0.3388
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9293 - loss: 0.1858 - val_accuracy: 0.8854 - val_loss: 0.3464
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9296 - loss: 0.1816 - val_accuracy: 0.8842 - val_loss: 0.3410
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9340 - loss: 0.1690 - val_accuracy: 0.8882 - val_loss: 0.3769
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9356 - loss: 0.1654 - val_accuracy: 0.8890 - val_loss: 0.3547
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9381 - loss: 0.1610 - val_accuracy: 0.8879 - val_loss: 0.3776
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9377 - loss: 0.1591 - val_accuracy: 0.8891 - val_loss: 0.3751
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9428 - loss: 0.1480 - val_accuracy: 0.8855 - val_loss: 0.3993
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9420 - loss: 0.1460 - val_accuracy: 0.8934 - val_loss: 0.4106
Epoch 22/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9426 - loss: 0.1427 - val_accuracy: 0.8866 - val_loss: 0.4000
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9483 - loss: 0.1334 - val_accuracy: 0.8877 - val_loss: 0.4381
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9478 - loss: 0.1322 - val_accuracy: 0.8864 - val_loss: 0.4398
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9508 - loss: 0.1249 - val_accuracy: 0.8856 - val_loss: 0.4733
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9502 - loss: 0.1260 - val_accuracy: 0.8942 - val_loss: 0.4340
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9538 - loss: 0.1198 - val_accuracy: 0.8898 - val_loss: 0.4557
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9559 - loss: 0.1146 - val_accuracy: 0.8887 - val_loss: 0.4615
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9570 - loss: 0.1108 - val_accuracy: 0.8923 - val_loss: 0.4717
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9574 - loss: 0.1072 - val_accuracy: 0.8962 - val_loss: 0.4670
```



In [8]:
```python
# print the highest validation accuracy and the epoch at which it was obtained for Model 1
best_epoch_1 = np.argmax(history_1.history['val_accuracy']) + 1
best_val_accuracy_1 = history_1.history['val_accuracy'][best_epoch_1 - 1]
print(f'Best validation accuracy: {best_val_accuracy_1:.4f} at epoch {best_epoch_1}')
# print the smallest validation loss and the epoch at which it was obtained for Model 1
best_epoch_loss_1 = np.argmin(history_1.history['val_loss']) + 1
best_val_loss_1 = history_1.history['val_loss'][best_epoch_loss_1 - 1]
print(f'Smallest validation loss: {best_val_loss_1:.4f} at epoch {best_epoch_loss_1}')
```

```
Best validation accuracy: 0.8962 at epoch 30
Smallest validation loss: 0.3088 at epoch 6
```

For Model 1, the performance metrics indicate that as the model capacity increased with additional neurons, it could capture more complex patterns in the data, shown by the rising training accuracy. However, the validation accuracy stabilizes around epoch 30 (accuracy 0.8962), with no significant gains afterward, suggesting that beyond this point, additional training may lead to overfitting rather than improved generalization. This is confirmed by the validation loss, which reaches its lowest point at epoch 6 and then gradually increases, a good indicator that the model has started to overfit to the training data.

Model 2: Two hidden layers with decreasing number of nodes.

In [9]:
```python
# build Model 2 with two hidden layers that decrease in size
model_2 = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Rescaling(1./255),
    Dense(512, activation='relu'),  # First hidden layer with more nodes
    Dense(256, activation='relu'),  # Second hidden layer with fewer nodes
    Dense(num_classes, activation='softmax')
])

# compile Model 2
model_2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# summarize Model 2
model_2.summary()

# train Model 2
history_2 = model_2.fit(
    train_images, train_labels,
    epochs=30,
    validation_data=(val_images, val_labels)
)

# plot the training history for Model 2
plot_training_history(history_2)
```

Model: "sequential_2"

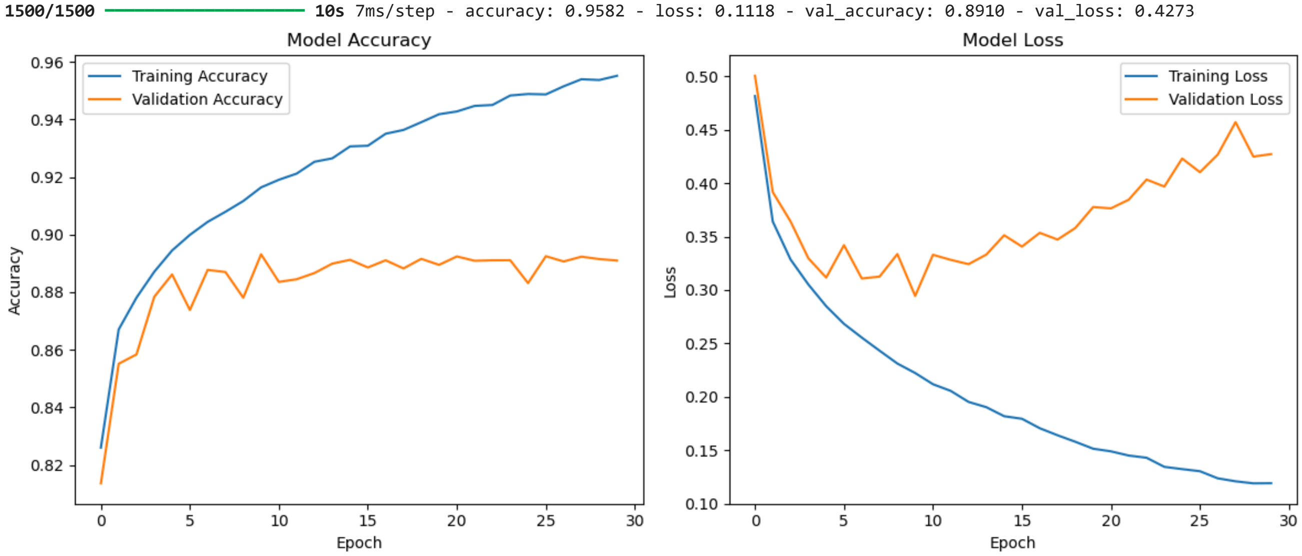
Layer (type)	Output Shape	Param #
flatten_2 (Flatten)	(None, 784)	0
rescaling_2 (Rescaling)	(None, 784)	0
dense_5 (Dense)	(None, 512)	401,920
dense_6 (Dense)	(None, 256)	131,328
dense_7 (Dense)	(None, 10)	2,570

Total params: 535,818 (2.04 MB)

Trainable params: 535,818 (2.04 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.7850 - loss: 0.6022 - val_accuracy: 0.8137 - val_loss: 0.5006
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8607 - loss: 0.3746 - val_accuracy: 0.8552 - val_loss: 0.3917
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8763 - loss: 0.3319 - val_accuracy: 0.8584 - val_loss: 0.3642
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8850 - loss: 0.3077 - val_accuracy: 0.8784 - val_loss: 0.3297
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8961 - loss: 0.2811 - val_accuracy: 0.8862 - val_loss: 0.3117
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8992 - loss: 0.2693 - val_accuracy: 0.8738 - val_loss: 0.3420
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9047 - loss: 0.2524 - val_accuracy: 0.8878 - val_loss: 0.3108
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9080 - loss: 0.2411 - val_accuracy: 0.8870 - val_loss: 0.3125
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9127 - loss: 0.2272 - val_accuracy: 0.8781 - val_loss: 0.3337
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9179 - loss: 0.2173 - val_accuracy: 0.8932 - val_loss: 0.2945
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9207 - loss: 0.2093 - val_accuracy: 0.8836 - val_loss: 0.3330
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9226 - loss: 0.2042 - val_accuracy: 0.8845 - val_loss: 0.3282
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.9262 - loss: 0.1920 - val_accuracy: 0.8867 - val_loss: 0.3242
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9283 - loss: 0.1854 - val_accuracy: 0.8899 - val_loss: 0.3332
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9324 - loss: 0.1779 - val_accuracy: 0.8913 - val_loss: 0.3513
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9300 - loss: 0.1820 - val_accuracy: 0.8886 - val_loss: 0.3407
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9373 - loss: 0.1651 - val_accuracy: 0.8911 - val_loss: 0.3535
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.9370 - loss: 0.1610 - val_accuracy: 0.8882 - val_loss: 0.3472
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9405 - loss: 0.1512 - val_accuracy: 0.8916 - val_loss: 0.3581
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9426 - loss: 0.1523 - val_accuracy: 0.8895 - val_loss: 0.3777
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9443 - loss: 0.1460 - val_accuracy: 0.8924 - val_loss: 0.3765
Epoch 22/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.9449 - loss: 0.1440 - val_accuracy: 0.8909 - val_loss: 0.3846
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9478 - loss: 0.1374 - val_accuracy: 0.8911 - val_loss: 0.4034
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9503 - loss: 0.1300 - val_accuracy: 0.8911 - val_loss: 0.3969
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.9508 - loss: 0.1261 - val_accuracy: 0.8832 - val_loss: 0.4231
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9494 - loss: 0.1273 - val_accuracy: 0.8925 - val_loss: 0.4103
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9545 - loss: 0.1178 - val_accuracy: 0.8907 - val_loss: 0.4268
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9560 - loss: 0.1154 - val_accuracy: 0.8923 - val_loss: 0.4572
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.9540 - loss: 0.1161 - val_accuracy: 0.8915 - val_loss: 0.4249
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.9582 - loss: 0.1118 - val_accuracy: 0.8910 - val_loss: 0.4273
```



In [10]:
```
# print the highest validation accuracy and the epoch at which it was obtained for Model 2
best_epoch_2 = np.argmax(history_2.history['val_accuracy']) + 1
best_val_accuracy_2 = history_2.history['val_accuracy'][best_epoch_2 - 1]
print(f'Best validation accuracy: {best_val_accuracy_2:.4f} at epoch {best_epoch_2}')
# print the smallest validation loss and the epoch at which it was obtained for Model 2
best_epoch_loss_2 = np.argmin(history_2.history['val_loss']) + 1
best_val_loss_2 = history_2.history['val_loss'][best_epoch_loss_2 - 1]
print(f'Smallest validation loss: {best_val_loss_2:.4f} at epoch {best_epoch_loss_2}')
```

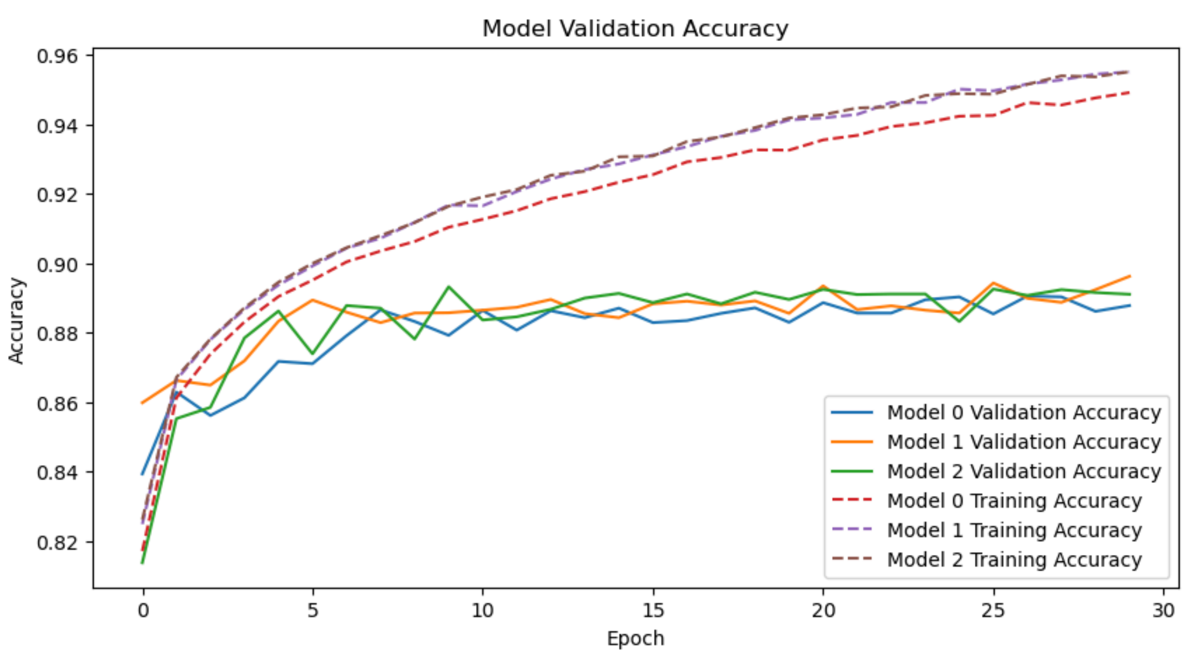
```
Best validation accuracy: 0.8932 at epoch 10
Smallest validation loss: 0.2945 at epoch 10
```

In [11]:
```
# put all the models in a dataframe
models = pd.DataFrame({
    'model': ['Model 0','Model 1', 'Model 2'],
    'val_accuracy': [best_val_accuracy, best_val_accuracy_1, np.max(history_2.history['val_accuracy'])],
    'val_loss': [best_val_loss, best_val_loss_1, np.min(history_2.history['val_loss'])],
    'best_epoch': [best_epoch, best_epoch_1, np.argmax(history_2.history['val_accuracy']) + 1]
})

# print the models dataframe
models
```

Out[11]:

	model	val_accuracy	val_loss	best_epoch
0	Model 0	0.890500	0.312480	27
1	Model 1	0.896167	0.308760	30
2	Model 2	0.893167	0.294455	10

In [12]:
```
# plot the validation accuracy of all models and training accuracy
plt.figure(figsize=(10, 5))
plt.plot(history.history['val_accuracy'], label='Model 0 Validation Accuracy')
plt.plot(history_1.history['val_accuracy'], label='Model 1 Validation Accuracy')
plt.plot(history_2.history['val_accuracy'], label='Model 2 Validation Accuracy')
plt.plot(history.history['accuracy'], label='Model 0 Training Accuracy', linestyle='--')
plt.plot(history_1.history['accuracy'], label='Model 1 Training Accuracy', linestyle='--')
plt.plot(history_2.history['accuracy'], label='Model 2 Training Accuracy', linestyle='--')
plt.title('Model Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



Model 2, which features a decreasing number of nodes in its hidden layers, achieved the highest validation accuracy at epoch 10 (0.8932) and the lowest validation loss at epoch 10 (0.2945). The model's validation accuracy shows a gradual improvement without drastic fluctuations, indicating good generalization. However, the increasing validation loss after the tenth epoch suggests that while the model is confident in its predictions, it may begin to overfit as training continues. This architecture shows a promising direction, but implementing early stopping or additional regularization might be beneficial to prevent overfitting and retain the best model performance.

Model 3: Two hidden layers with dropout regularization.

In [13]:

```
# build Model 3 with two hidden layers and dropout
model_3 = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Rescaling(1./255),
    Dense(512, activation='relu'),
    Dropout(0.5),  # dropout layer
    Dense(256, activation='relu'),
    Dropout(0.5),  # dropout layer
    Dense(num_classes, activation='softmax')
])

# compile Model 3
model_3.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# summarize Model 3
model_3.summary()

# train Model 3
history_3 = model_3.fit(
    train_images, train_labels,
    epochs=30,
    validation_data=(val_images, val_labels)
)

# plot the training history for Model 3
plot_training_history(history_3)
```

Model: "sequential_3"

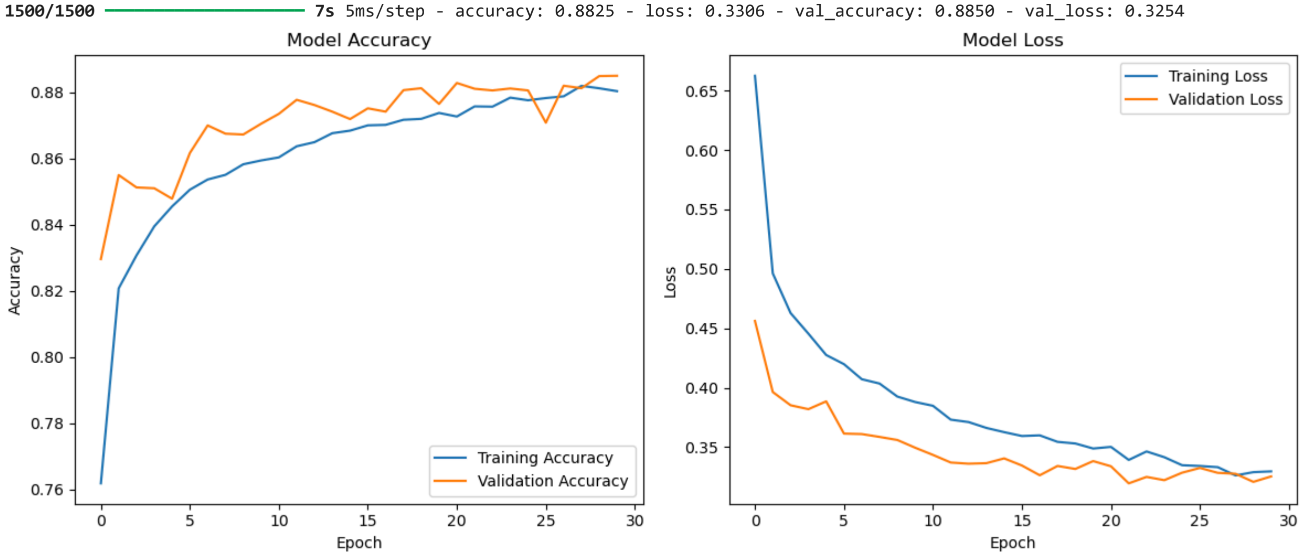
Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 784)	0
rescaling_3 (Rescaling)	(None, 784)	0
dense_8 (Dense)	(None, 512)	401,920
dropout (Dropout)	(None, 512)	0
dense_9 (Dense)	(None, 256)	131,328
dropout_1 (Dropout)	(None, 256)	0
dense_10 (Dense)	(None, 10)	2,570

Total params: 535,818 (2.04 MB)

Trainable params: 535,818 (2.04 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.6910 - loss: 0.8618 - val_accuracy: 0.8296 - val_loss: 0.4562
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.8168 - loss: 0.5081 - val_accuracy: 0.8550 - val_loss: 0.3964
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8307 - loss: 0.4654 - val_accuracy: 0.8512 - val_loss: 0.3853
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.8382 - loss: 0.4442 - val_accuracy: 0.8510 - val_loss: 0.3820
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8439 - loss: 0.4327 - val_accuracy: 0.8478 - val_loss: 0.3885
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8495 - loss: 0.4213 - val_accuracy: 0.8617 - val_loss: 0.3614
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8536 - loss: 0.4049 - val_accuracy: 0.8700 - val_loss: 0.3610
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8548 - loss: 0.4069 - val_accuracy: 0.8675 - val_loss: 0.3586
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8555 - loss: 0.3966 - val_accuracy: 0.8673 - val_loss: 0.3560
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8612 - loss: 0.3793 - val_accuracy: 0.8705 - val_loss: 0.3495
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8586 - loss: 0.3856 - val_accuracy: 0.8735 - val_loss: 0.3435
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8634 - loss: 0.3713 - val_accuracy: 0.8777 - val_loss: 0.3371
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8684 - loss: 0.3656 - val_accuracy: 0.8762 - val_loss: 0.3361
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8675 - loss: 0.3685 - val_accuracy: 0.8742 - val_loss: 0.3365
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8660 - loss: 0.3645 - val_accuracy: 0.8719 - val_loss: 0.3406
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8716 - loss: 0.3533 - val_accuracy: 0.8752 - val_loss: 0.3345
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8711 - loss: 0.3576 - val_accuracy: 0.8742 - val_loss: 0.3263
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8738 - loss: 0.3506 - val_accuracy: 0.8807 - val_loss: 0.3342
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8740 - loss: 0.3496 - val_accuracy: 0.8813 - val_loss: 0.3317
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8752 - loss: 0.3480 - val_accuracy: 0.8765 - val_loss: 0.3383
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8740 - loss: 0.3457 - val_accuracy: 0.8828 - val_loss: 0.3339
Epoch 22/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8752 - loss: 0.3372 - val_accuracy: 0.8811 - val_loss: 0.3196
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8752 - loss: 0.3477 - val_accuracy: 0.8806 - val_loss: 0.3250
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8784 - loss: 0.3416 - val_accuracy: 0.8812 - val_loss: 0.3222
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8750 - loss: 0.3413 - val_accuracy: 0.8806 - val_loss: 0.3285
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8791 - loss: 0.3280 - val_accuracy: 0.8708 - val_loss: 0.3325
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8804 - loss: 0.3302 - val_accuracy: 0.8820 - val_loss: 0.3284
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8838 - loss: 0.3226 - val_accuracy: 0.8813 - val_loss: 0.3276
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8816 - loss: 0.3284 - val_accuracy: 0.8849 - val_loss: 0.3208
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8825 - loss: 0.3306 - val_accuracy: 0.8850 - val_loss: 0.3254
```



In [14]:
```python
# print the highest validation accuracy and the epoch at which it was obtained for Model 3
best_epoch_3 = np.argmax(history_3.history['val_accuracy']) + 1
best_val_accuracy_3 = history_3.history['val_accuracy'][best_epoch_3 - 1]
print(f'Best validation accuracy: {best_val_accuracy_3:.4f} at epoch {best_epoch_3}')
# print the smallest validation loss and the epoch at which it was obtained for Model 3
best_epoch_loss_3 = np.argmin(history_3.history['val_loss']) + 1
best_val_loss_3 = history_3.history['val_loss'][best_epoch_loss_3 - 1]
print(f'Smallest validation loss: {best_val_loss_3:.4f} at epoch {best_epoch_loss_3}')
```

```
Best validation accuracy: 0.8850 at epoch 30
Smallest validation loss: 0.3196 at epoch 22
```

In [15]:
```python
# add Model 3 to the models dataframe
models = pd.concat([models, pd.DataFrame({
    'model': ['Model 3'],
    'val_accuracy': [best_val_accuracy_3],
    'val_loss': [best_val_loss_3],
    'best_epoch': [best_epoch_3]
})], ignore_index=True)

# print the models dataframe
models
```

Out[15]:

	model	val_accuracy	val_loss	best_epoch
0	Model 0	0.890500	0.312480	27
1	Model 1	0.896167	0.308760	30
2	Model 2	0.893167	0.294455	10
3	Model 3	0.885000	0.319558	30

For Model 3, we tried incorporating dropout to reduce overfitting, applying it after two dense layers with a substantial number of nodes. The training process showed a continuous increase in validation accuracy up to a peak at epoch 30, where it achieved the best accuracy and loss, suggesting that dropout helped maintain a balance between learning and overfitting. Despite this, the final accuracy wasn't as high as some of the other models we've experimented with, hinting that there might be a limit to the complexity this model can capture. This version has shown that while dropout is effective for generalization, we might need to adjust other parameters or add complexity to improve accuracy further.

Model 4: Three hidden layers

In [16]:
```python
# build Model 4 with three hidden layers in a funnel architecture
model_4 = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Rescaling(1./255),
    Dense(1024, activation='relu'),  # first hidden layer with 1024 neurons
    Dense(512, activation='relu'),    # less neurons
    Dense(256, activation='relu'),    # less than previous layers
    Dense(num_classes, activation='softmax')
])

# compile Model 4
model_4.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# summarize Model 4
model_4.summary()

# train Model 4
history_4 = model_4.fit(
    train_images, train_labels,
    epochs=30,
    validation_data=(val_images, val_labels)
)

# plot the training history for Model 4
plot_training_history(history_4)
```

Model: "sequential_4"

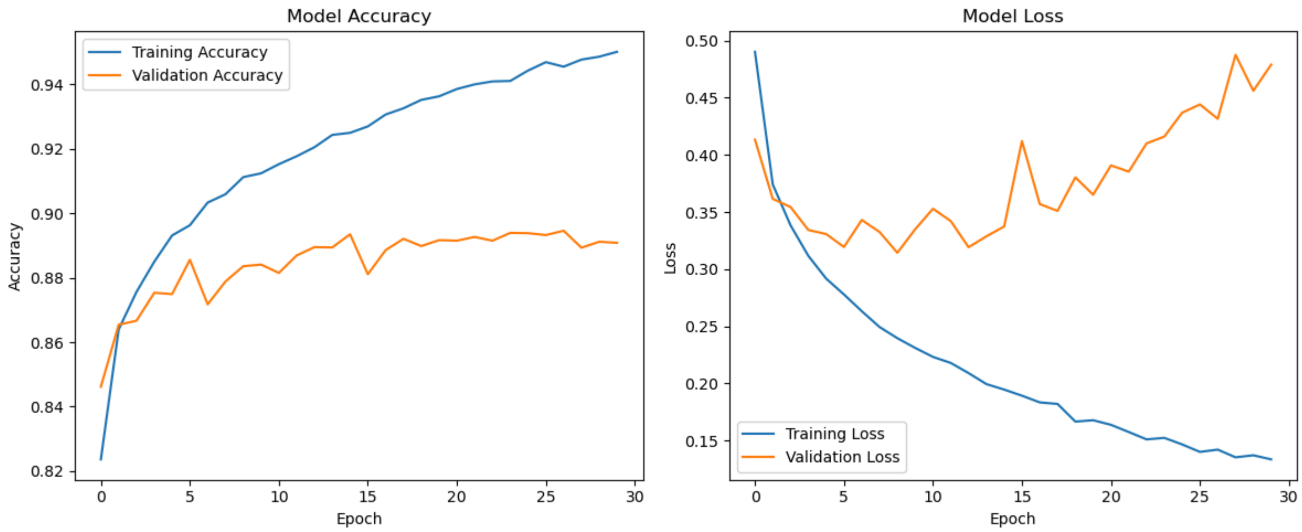
Layer (type)	Output Shape	Param #
flatten_4 (Flatten)	(None, 784)	0
rescaling_4 (Rescaling)	(None, 784)	0
dense_11 (Dense)	(None, 1024)	803,840
dense_12 (Dense)	(None, 512)	524,800
dense_13 (Dense)	(None, 256)	131,328
dense_14 (Dense)	(None, 10)	2,570

Total params: 1,462,538 (5.58 MB)

Trainable params: 1,462,538 (5.58 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 11ms/step - accuracy: 0.7805 - loss: 0.6082 - val_accuracy: 0.8462 - val_loss: 0.4134
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.8613 - loss: 0.3834 - val_accuracy: 0.8654 - val_loss: 0.3614
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8759 - loss: 0.3361 - val_accuracy: 0.8667 - val_loss: 0.3543
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 16s 11ms/step - accuracy: 0.8844 - loss: 0.3091 - val_accuracy: 0.8753 - val_loss: 0.3342
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.8938 - loss: 0.2912 - val_accuracy: 0.8749 - val_loss: 0.3307
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.8968 - loss: 0.2752 - val_accuracy: 0.8856 - val_loss: 0.3193
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.9041 - loss: 0.2608 - val_accuracy: 0.8717 - val_loss: 0.3430
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9058 - loss: 0.2450 - val_accuracy: 0.8788 - val_loss: 0.3325
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.9102 - loss: 0.2403 - val_accuracy: 0.8836 - val_loss: 0.3143
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.9144 - loss: 0.2251 - val_accuracy: 0.8841 - val_loss: 0.3349
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9189 - loss: 0.2139 - val_accuracy: 0.8815 - val_loss: 0.3528
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.9181 - loss: 0.2155 - val_accuracy: 0.8869 - val_loss: 0.3421
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9202 - loss: 0.2067 - val_accuracy: 0.8895 - val_loss: 0.3191
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9276 - loss: 0.1907 - val_accuracy: 0.8894 - val_loss: 0.3287
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9270 - loss: 0.1895 - val_accuracy: 0.8935 - val_loss: 0.3373
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9309 - loss: 0.1819 - val_accuracy: 0.8811 - val_loss: 0.4121
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.9315 - loss: 0.1830 - val_accuracy: 0.8886 - val_loss: 0.3570
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9333 - loss: 0.1769 - val_accuracy: 0.8921 - val_loss: 0.3509
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9372 - loss: 0.1605 - val_accuracy: 0.8898 - val_loss: 0.3803
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9327 - loss: 0.1737 - val_accuracy: 0.8917 - val_loss: 0.3651
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9414 - loss: 0.1555 - val_accuracy: 0.8915 - val_loss: 0.3908
Epoch 22/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 12ms/step - accuracy: 0.9398 - loss: 0.1564 - val_accuracy: 0.8927 - val_loss: 0.3854
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.9427 - loss: 0.1458 - val_accuracy: 0.8915 - val_loss: 0.4101
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 14ms/step - accuracy: 0.9434 - loss: 0.1446 - val_accuracy: 0.8939 - val_loss: 0.4161
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.9457 - loss: 0.1404 - val_accuracy: 0.8938 - val_loss: 0.4369
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.9487 - loss: 0.1345 - val_accuracy: 0.8932 - val_loss: 0.4442
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.9470 - loss: 0.1391 - val_accuracy: 0.8946 - val_loss: 0.4316
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.9517 - loss: 0.1264 - val_accuracy: 0.8893 - val_loss: 0.4876
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.9490 - loss: 0.1384 - val_accuracy: 0.8912 - val_loss: 0.4561
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.9504 - loss: 0.1295 - val_accuracy: 0.8908 - val_loss: 0.4790
```



In [17]:
```
# print the highest validation accuracy and the epoch at which it was obtained for Model 4
best_epoch_4 = np.argmax(history_4.history['val_accuracy']) + 1
best_val_accuracy_4 = history_4.history['val_accuracy'][best_epoch_4 - 1]
print(f'Best validation accuracy: {best_val_accuracy_4:.4f} at epoch {best_epoch_4}')
# print the smallest validation loss and the epoch at which it was obtained for Model 4
best_epoch_loss_4 = np.argmin(history_4.history['val_loss']) + 1
best_val_loss_4 = history_4.history['val_loss'][best_epoch_loss_4 - 1]
print(f'Smallest validation loss: {best_val_loss_4:.4f} at epoch {best_epoch_loss_4}')
```

```
Best validation accuracy: 0.8946 at epoch 27
Smallest validation loss: 0.3143 at epoch 9
```

In [18]:
```
# add Model 4 to the models dataframe
models = pd.concat([models, pd.DataFrame({
    'model': ['Model 4'],
    'val_accuracy': [best_val_accuracy_4],
    'val_loss': [best_val_loss_4],
    'best_epoch': [best_epoch_4]
})], ignore_index=True)

# print the models dataframe
models
```

Out[18]:

	model	val_accuracy	val_loss	best_epoch
0	Model 0	0.890500	0.312480	27
1	Model 1	0.896167	0.308760	30
2	Model 2	0.893167	0.294455	10
3	Model 3	0.885000	0.319558	30
4	Model 4	0.894583	0.314308	27

In Model 4, we introduced a three-layer funnel architecture, stepping down the number of neurons from 1024 to 256. This approach seeks to gradually compress the information the network learns, layer by layer. The model achieved its best validation accuracy and loss earlier in the training at epoch 27 and 9, respectively, suggesting that the added complexity of the model allowed it to learn efficiently. However, similar to previous models, the validation loss started to increase after reaching its minimum, which indicates the model began to overfit the training data as training continued.

Model 5: Two hidden layers with L1 regularization (LASSO)

In [19]:
```
# model 5: two hidden layers with L1 regularization
model_5 = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Rescaling(1./255),
    Dense(512, activation='relu', activity_regularizer=l1(0.0001)),  # LASSO regularization
    Dense(256, activation='relu', activity_regularizer=l1(0.0001)),
    Dense(num_classes, activation='softmax')
])

model_5.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# summarize Model 5
model_5.summary()

# train Model 5
```

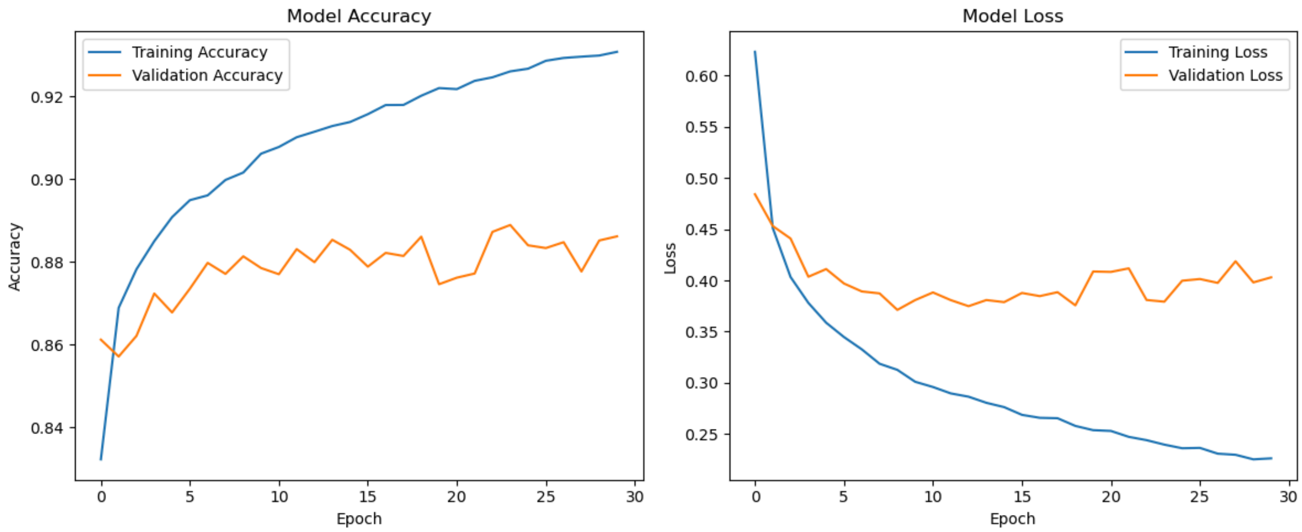
```python
history_5 = model_5.fit(train_images, train_labels, epochs=30, validation_data=(val_images, val_labels))

# plot the training history for Model 5
plot_training_history(history_5)
```

```
Model: "sequential_5"
```

Layer (type)	Output Shape	Param #
flatten_5 (Flatten)	(None, 784)	0
rescaling_5 (Rescaling)	(None, 784)	0
dense_15 (Dense)	(None, 512)	401,920
dense_16 (Dense)	(None, 256)	131,328
dense_17 (Dense)	(None, 10)	2,570

```
Total params: 535,818 (2.04 MB)
Trainable params: 535,818 (2.04 MB)
Non-trainable params: 0 (0.00 B)
```

```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.7932 - loss: 0.8039 - val_accuracy: 0.8612 - val_loss: 0.4842
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8682 - loss: 0.4619 - val_accuracy: 0.8571 - val_loss: 0.4532
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8803 - loss: 0.4036 - val_accuracy: 0.8621 - val_loss: 0.4409
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8864 - loss: 0.3787 - val_accuracy: 0.8723 - val_loss: 0.4036
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8917 - loss: 0.3547 - val_accuracy: 0.8677 - val_loss: 0.4111
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8942 - loss: 0.3449 - val_accuracy: 0.8735 - val_loss: 0.3970
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8945 - loss: 0.3337 - val_accuracy: 0.8798 - val_loss: 0.3893
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 6ms/step - accuracy: 0.8998 - loss: 0.3188 - val_accuracy: 0.8771 - val_loss: 0.3874
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9024 - loss: 0.3117 - val_accuracy: 0.8813 - val_loss: 0.3713
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9061 - loss: 0.3035 - val_accuracy: 0.8785 - val_loss: 0.3809
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9099 - loss: 0.2938 - val_accuracy: 0.8770 - val_loss: 0.3883
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9107 - loss: 0.2876 - val_accuracy: 0.8831 - val_loss: 0.3808
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9120 - loss: 0.2865 - val_accuracy: 0.8799 - val_loss: 0.3749
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9131 - loss: 0.2785 - val_accuracy: 0.8853 - val_loss: 0.3809
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9143 - loss: 0.2759 - val_accuracy: 0.8829 - val_loss: 0.3788
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.9177 - loss: 0.2648 - val_accuracy: 0.8788 - val_loss: 0.3878
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.9195 - loss: 0.2618 - val_accuracy: 0.8822 - val_loss: 0.3847
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.9199 - loss: 0.2603 - val_accuracy: 0.8814 - val_loss: 0.3886
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9216 - loss: 0.2528 - val_accuracy: 0.8861 - val_loss: 0.3757
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9221 - loss: 0.2546 - val_accuracy: 0.8746 - val_loss: 0.4088
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9235 - loss: 0.2477 - val_accuracy: 0.8762 - val_loss: 0.4084
Epoch 22/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9261 - loss: 0.2454 - val_accuracy: 0.8772 - val_loss: 0.4118
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9249 - loss: 0.2421 - val_accuracy: 0.8873 - val_loss: 0.3809
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9268 - loss: 0.2349 - val_accuracy: 0.8889 - val_loss: 0.3793
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9299 - loss: 0.2270 - val_accuracy: 0.8840 - val_loss: 0.3998
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9279 - loss: 0.2349 - val_accuracy: 0.8833 - val_loss: 0.4014
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9308 - loss: 0.2266 - val_accuracy: 0.8848 - val_loss: 0.3976
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9311 - loss: 0.2255 - val_accuracy: 0.8777 - val_loss: 0.4188
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9302 - loss: 0.2226 - val_accuracy: 0.8852 - val_loss: 0.3981
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9334 - loss: 0.2203 - val_accuracy: 0.8862 - val_loss: 0.4030
```



In [20]:
```
# print the highest validation accuracy and the epoch at which it was obtained for Model 5
best_epoch_5 = np.argmax(history_5.history['val_accuracy']) + 1
best_val_accuracy_5 = history_5.history['val_accuracy'][best_epoch_5 - 1]
print(f'Best validation accuracy: {best_val_accuracy_5:.4f} at epoch {best_epoch_5}')
# print the smallest validation loss and the epoch at which it was obtained for Model 5
best_epoch_loss_5 = np.argmin(history_5.history['val_loss']) + 1
best_val_loss_5 = history_5.history['val_loss'][best_epoch_loss_5 - 1]
print(f'Smallest validation loss: {best_val_loss_5:.4f} at epoch {best_epoch_loss_5}')
```

```
Best validation accuracy: 0.8889 at epoch 24
Smallest validation loss: 0.3713 at epoch 9
```

In [21]:
```
# add Model 5 to the models dataframe
models = pd.concat([models, pd.DataFrame({
    'model': ['Model 5'],
    'val_accuracy': [best_val_accuracy_5],
    'val_loss': [best_val_loss_5],
    'best_epoch': [best_epoch_5]
})], ignore_index=True)

models
```

Out[21]:

	model	val_accuracy	val_loss	best_epoch
0	Model 0	0.890500	0.312480	27
1	Model 1	0.896167	0.308760	30
2	Model 2	0.893167	0.294455	10
3	Model 3	0.885000	0.319558	30
4	Model 4	0.894583	0.314308	27
5	Model 5	0.888917	0.371258	24

For Model 5, we introduced L1 regularization. While this model's validation accuracy did not surpass previous models, peaking at 0.8867 at epoch 24, it maintained a competitive performance. The model achieved its lowest validation loss quite early at epoch 9, suggesting that regularization helped in preventing overfitting. However, similar to the other models, the validation loss increased as training proceeded, which could indicate a need for further regularization or a more sophisticated approach to prevent overfitting.

Model 6: Two hidden layers with L1 regularization and dropout

In [22]:
```
# model 6: two hidden layers with dropout and L1 regularization
model_6 = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Rescaling(1./255),
    Dense(512, activation='relu', activity_regularizer=l1(0.0001)),
    Dropout(0.5),
    Dense(256, activation='relu', activity_regularizer=l1(0.0001)),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])

model_6.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

model_6.summary()
```

```python
history_6 = model_6.fit(train_images, train_labels, epochs=30, validation_data=(val_images, val_labels))

plot_training_history(history_6)
```

Model: "sequential_6"

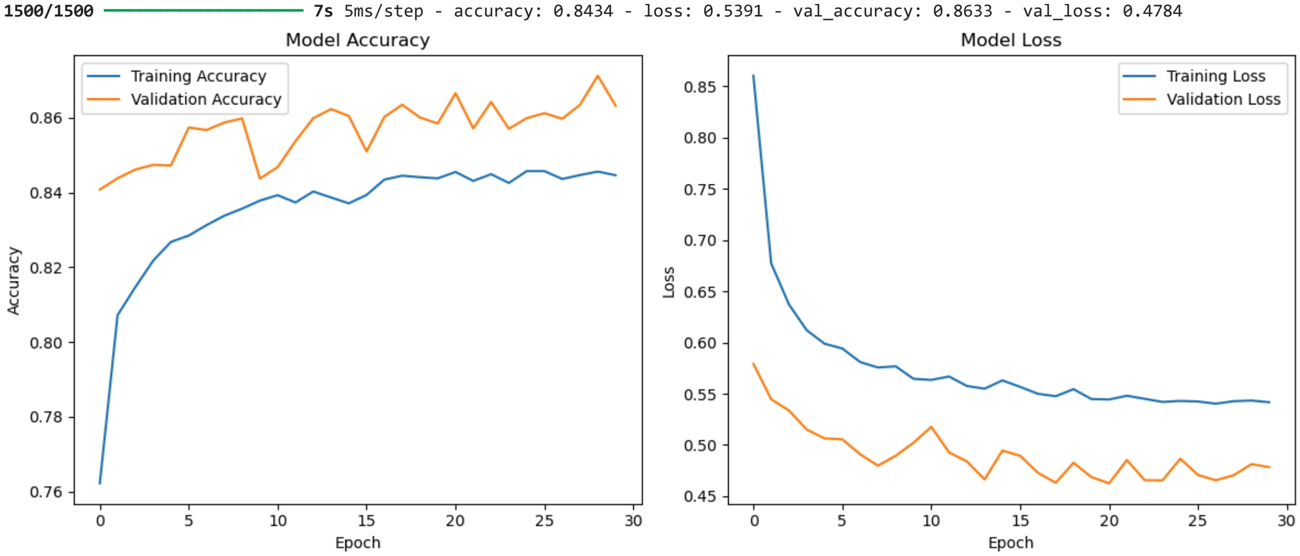
Layer (type)	Output Shape	Param #
flatten_6 (Flatten)	(None, 784)	0
rescaling_6 (Rescaling)	(None, 784)	0
dense_18 (Dense)	(None, 512)	401,920
dropout_2 (Dropout)	(None, 512)	0
dense_19 (Dense)	(None, 256)	131,328
dropout_3 (Dropout)	(None, 256)	0
dense_20 (Dense)	(None, 10)	2,570

Total params: 535,818 (2.04 MB)

Trainable params: 535,818 (2.04 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.7062 - loss: 1.0690 - val_accuracy: 0.8408 - val_loss: 0.5791
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8047 - loss: 0.6837 - val_accuracy: 0.8438 - val_loss: 0.5446
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8115 - loss: 0.6469 - val_accuracy: 0.8462 - val_loss: 0.5336
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8226 - loss: 0.6130 - val_accuracy: 0.8474 - val_loss: 0.5150
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8284 - loss: 0.5976 - val_accuracy: 0.8472 - val_loss: 0.5064
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8296 - loss: 0.5923 - val_accuracy: 0.8574 - val_loss: 0.5055
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8350 - loss: 0.5779 - val_accuracy: 0.8568 - val_loss: 0.4909
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8324 - loss: 0.5764 - val_accuracy: 0.8587 - val_loss: 0.4798
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8343 - loss: 0.5804 - val_accuracy: 0.8598 - val_loss: 0.4895
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8351 - loss: 0.5700 - val_accuracy: 0.8438 - val_loss: 0.5022
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.8381 - loss: 0.5681 - val_accuracy: 0.8468 - val_loss: 0.5177
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8356 - loss: 0.5700 - val_accuracy: 0.8537 - val_loss: 0.4928
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8410 - loss: 0.5569 - val_accuracy: 0.8599 - val_loss: 0.4839
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8373 - loss: 0.5569 - val_accuracy: 0.8623 - val_loss: 0.4664
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8364 - loss: 0.5575 - val_accuracy: 0.8605 - val_loss: 0.4946
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.8411 - loss: 0.5517 - val_accuracy: 0.8510 - val_loss: 0.4894
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8442 - loss: 0.5436 - val_accuracy: 0.8602 - val_loss: 0.4728
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8441 - loss: 0.5487 - val_accuracy: 0.8635 - val_loss: 0.4631
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8447 - loss: 0.5501 - val_accuracy: 0.8601 - val_loss: 0.4826
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8446 - loss: 0.5400 - val_accuracy: 0.8585 - val_loss: 0.4687
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8477 - loss: 0.5394 - val_accuracy: 0.8666 - val_loss: 0.4626
Epoch 22/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8407 - loss: 0.5521 - val_accuracy: 0.8572 - val_loss: 0.4853
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8459 - loss: 0.5428 - val_accuracy: 0.8643 - val_loss: 0.4656
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8403 - loss: 0.5465 - val_accuracy: 0.8571 - val_loss: 0.4654
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8420 - loss: 0.5521 - val_accuracy: 0.8599 - val_loss: 0.4866
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8460 - loss: 0.5411 - val_accuracy: 0.8612 - val_loss: 0.4707
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8426 - loss: 0.5416 - val_accuracy: 0.8597 - val_loss: 0.4656
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8404 - loss: 0.5498 - val_accuracy: 0.8635 - val_loss: 0.4704
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8481 - loss: 0.5368 - val_accuracy: 0.8712 - val_loss: 0.4813
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8434 - loss: 0.5391 - val_accuracy: 0.8633 - val_loss: 0.4784
```



In [23]:
```
# print the highest validation accuracy and the epoch at which it was obtained for Model 6
best_epoch_6 = np.argmax(history_6.history['val_accuracy']) + 1
best_val_accuracy_6 = history_6.history['val_accuracy'][best_epoch_6 - 1]
print(f'Best validation accuracy: {best_val_accuracy_6:.4f} at epoch {best_epoch_6}')
# print the smallest validation loss and the epoch at which it was obtained for Model 6
best_epoch_loss_6 = np.argmin(history_6.history['val_loss']) + 1
best_val_loss_6 = history_6.history['val_loss'][best_epoch_loss_6 - 1]
print(f'Smallest validation loss: {best_val_loss_6:.4f} at epoch {best_epoch_loss_6}')
```

```
Best validation accuracy: 0.8712 at epoch 29
Smallest validation loss: 0.4626 at epoch 21
```

In [24]:
```
# add Model 6 to the models dataframe
models = pd.concat([models, pd.DataFrame({
    'model': ['Model 6'],
    'val_accuracy': [best_val_accuracy_6],
    'val_loss': [best_val_loss_6],
    'best_epoch': [best_epoch_6]
})], ignore_index=True)

models
```

Out[24]:

	model	val_accuracy	val_loss	best_epoch
0	Model 0	0.890500	0.312480	27
1	Model 1	0.896167	0.308760	30
2	Model 2	0.893167	0.294455	10
3	Model 3	0.885000	0.319558	30
4	Model 4	0.894583	0.314308	27
5	Model 5	0.888917	0.371258	24
6	Model 6	0.871250	0.462555	29

For Model 6, we incorporated L1 regularization and dropout into the network, aiming to prevent overfitting by encouraging simpler models and reducing neuron co-dependency. This resulted in the highest validation accuracy at epoch 29, indicating a good fit at this point in the training. The model achieved its lowest validation loss at epoch 21, where the combined effects of L1 and dropout effectively minimized overfitting. However, this approach did seem to limit the model's overall accuracy potential compared to previous models without these regularization techniques.

Model 7: Three hidden layers with L1 regularization and dropout

In [25]:
```
# build model 7: three hidden layers with dropout and L1 regularization
model_7 = Sequential([
    Input(shape=(28, 28)),
    Flatten(),
    Rescaling(1./255),
    Dense(1024, activation='relu', activity_regularizer=l1(0.0001)),  # increased number of nodes
    Dropout(0.5),
    Dense(512, activation='relu', activity_regularizer=l1(0.0001)),
    Dropout(0.5),
    Dense(256, activation='relu', activity_regularizer=l1(0.0001)),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])

model_7.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

model_7.summary()

history_7 = model_7.fit(train_images, train_labels, epochs=30, validation_data=(val_images, val_labels))

plot_training_history(history_7)
```

Model: "sequential_7"

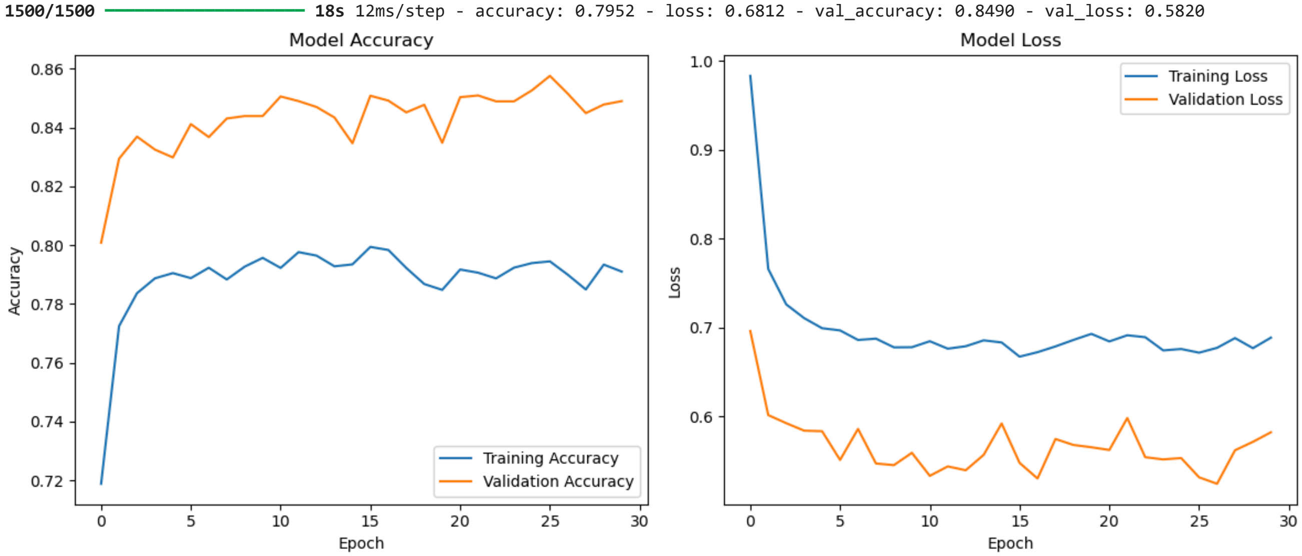
Layer (type)	Output Shape	Param #
flatten_7 (Flatten)	(None, 784)	0
rescaling_7 (Rescaling)	(None, 784)	0
dense_21 (Dense)	(None, 1024)	803,840
dropout_4 (Dropout)	(None, 1024)	0
dense_22 (Dense)	(None, 512)	524,800
dropout_5 (Dropout)	(None, 512)	0
dense_23 (Dense)	(None, 256)	131,328
dropout_6 (Dropout)	(None, 256)	0
dense_24 (Dense)	(None, 10)	2,570

Total params: 1,462,538 (5.58 MB)

Trainable params: 1,462,538 (5.58 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 12ms/step - accuracy: 0.6538 - loss: 1.2368 - val_accuracy: 0.8008 - val_loss: 0.6960
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.7687 - loss: 0.7835 - val_accuracy: 0.8294 - val_loss: 0.6013
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7852 - loss: 0.7253 - val_accuracy: 0.8369 - val_loss: 0.5922
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.7895 - loss: 0.7195 - val_accuracy: 0.8325 - val_loss: 0.5838
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7896 - loss: 0.7016 - val_accuracy: 0.8298 - val_loss: 0.5831
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.7898 - loss: 0.6937 - val_accuracy: 0.8412 - val_loss: 0.5508
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.7919 - loss: 0.6908 - val_accuracy: 0.8367 - val_loss: 0.5857
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.7919 - loss: 0.6887 - val_accuracy: 0.8431 - val_loss: 0.5468
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.7935 - loss: 0.6665 - val_accuracy: 0.8439 - val_loss: 0.5450
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7938 - loss: 0.6809 - val_accuracy: 0.8439 - val_loss: 0.5588
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7952 - loss: 0.6802 - val_accuracy: 0.8506 - val_loss: 0.5329
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8016 - loss: 0.6705 - val_accuracy: 0.8490 - val_loss: 0.5435
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8009 - loss: 0.6688 - val_accuracy: 0.8470 - val_loss: 0.5392
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7929 - loss: 0.6849 - val_accuracy: 0.8434 - val_loss: 0.5565
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.7943 - loss: 0.6780 - val_accuracy: 0.8347 - val_loss: 0.5919
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.7995 - loss: 0.6734 - val_accuracy: 0.8508 - val_loss: 0.5475
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8001 - loss: 0.6684 - val_accuracy: 0.8492 - val_loss: 0.5299
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7950 - loss: 0.6717 - val_accuracy: 0.8452 - val_loss: 0.5744
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7932 - loss: 0.6780 - val_accuracy: 0.8478 - val_loss: 0.5677
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7839 - loss: 0.6862 - val_accuracy: 0.8348 - val_loss: 0.5651
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7891 - loss: 0.6962 - val_accuracy: 0.8503 - val_loss: 0.5620
Epoch 22/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.7890 - loss: 0.6973 - val_accuracy: 0.8509 - val_loss: 0.5980
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 24s 16ms/step - accuracy: 0.7892 - loss: 0.6898 - val_accuracy: 0.8489 - val_loss: 0.5539
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 23s 16ms/step - accuracy: 0.7906 - loss: 0.6778 - val_accuracy: 0.8489 - val_loss: 0.5514
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 21s 14ms/step - accuracy: 0.7945 - loss: 0.6739 - val_accuracy: 0.8527 - val_loss: 0.5529
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7917 - loss: 0.6764 - val_accuracy: 0.8576 - val_loss: 0.5312
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7932 - loss: 0.6717 - val_accuracy: 0.8515 - val_loss: 0.5239
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7858 - loss: 0.6797 - val_accuracy: 0.8449 - val_loss: 0.5617
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7922 - loss: 0.6743 - val_accuracy: 0.8478 - val_loss: 0.5712
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7952 - loss: 0.6812 - val_accuracy: 0.8490 - val_loss: 0.5820
```



In [26]:
```
# print the highest validation accuracy and the epoch at which it was obtained for Model 7
best_epoch_7 = np.argmax(history_7.history['val_accuracy']) + 1
best_val_accuracy_7 = history_7.history['val_accuracy'][best_epoch_7 - 1]
print(f'Best validation accuracy: {best_val_accuracy_7:.4f} at epoch {best_epoch_7}')
# print the smallest validation loss and the epoch at which it was obtained for Model 7
best_epoch_loss_7 = np.argmin(history_7.history['val_loss']) + 1
best_val_loss_7 = history_7.history['val_loss'][best_epoch_loss_7 - 1]
print(f'Smallest validation loss: {best_val_loss_7:.4f} at epoch {best_epoch_loss_7}')
```

```
Best validation accuracy: 0.8576 at epoch 26
Smallest validation loss: 0.5239 at epoch 27
```

In [27]:
```
# add Model 7 to the models dataframe
models = pd.concat([models, pd.DataFrame({
    'model': ['Model 7'],
    'val_accuracy': [best_val_accuracy_7],
    'val_loss': [best_val_loss_7],
    'best_epoch': [best_epoch_7]
})], ignore_index=True)

# add new ranking columns based on validation accuracy
models['val_accuracy_rank'] = models['val_accuracy'].rank(ascending=False).astype(int)

models
```

Out[27]:

	model	val_accuracy	val_loss	best_epoch	val_accuracy_rank
0	Model 0	0.890500	0.312480	27	4
1	Model 1	0.896167	0.308760	30	1
2	Model 2	0.893167	0.294455	10	3
3	Model 3	0.885000	0.319558	30	6
4	Model 4	0.894583	0.314308	27	2
5	Model 5	0.888917	0.371258	24	5
6	Model 6	0.871250	0.462555	29	7
7	Model 7	0.857583	0.523946	26	8

For Model 7, we incorporated a complex architecture with three dense layers and combined L1 regularization with dropout in an attempt to manage overfitting while encouraging model compactness. However, the results showed that while the model's performance on the validation set improved gradually, it did not surpass the best models from our previous experiments. With a best validation accuracy of 85.76% at epoch 26 and a lowest validation loss occurring at epoch 27, it suggests that while regularization helps in combating overfitting, the model might be too complex or not have the right balance between regularization and capacity.

Throughout our experimentation, we built and evaluated seven different models with varying complexities and regularization techniques, aiming to optimize the validation accuracy while minimizing the loss. The Model 1 emerged as the top performer with the highest validation accuracy, indicating a suitable balance between model complexity and ability to generalize. It's evident that the addition of more layers and regularization did not linearly correlate with better performance, emphasizing the importance of finding the right architecture and hyperparameters for the dataset at hand. Overall, the experiments provided valuable insights into the trade-offs between model complexity, overfitting, and predictive power.

Question 5

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.).

To enhance the accuracy, we've decided to incorporate convolution into the next set of models:

- Model A: Basic Convolutional Model - Starts with a single convolutional layer, a max-pooling layer for spatial reduction, flattened into a dense layer, and then the output layer.
- Model B: Deep Convolutional Model - Features three convolutional layers
- Model C: Deeper Convolutional Neural Network with Adjusted Dropout Rates - This is a further adjustment of Model B
- Model D: Optimized deep convolutional network

Model A: Basic Convulutional Model

In [28]:
```
model_a = Sequential([
    Input(shape=(28, 28, 1)),
    Rescaling(1./255),
    # convolutional layer with 32 filters, a kernel size of 3, and ReLU activation
    Conv2D(32, kernel_size=(3, 3), activation='relu'),
```

```
    # max pooling layer with a pool size of 2
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])


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

model_a.summary()

# train the model
history_a = model_a.fit(
    train_images,
    train_labels,
    batch_size=64,
    epochs=20,
    validation_data=(val_images, val_labels)
)

# plot the training history
plot_training_history(history_a)
```

Model: "sequential_8"

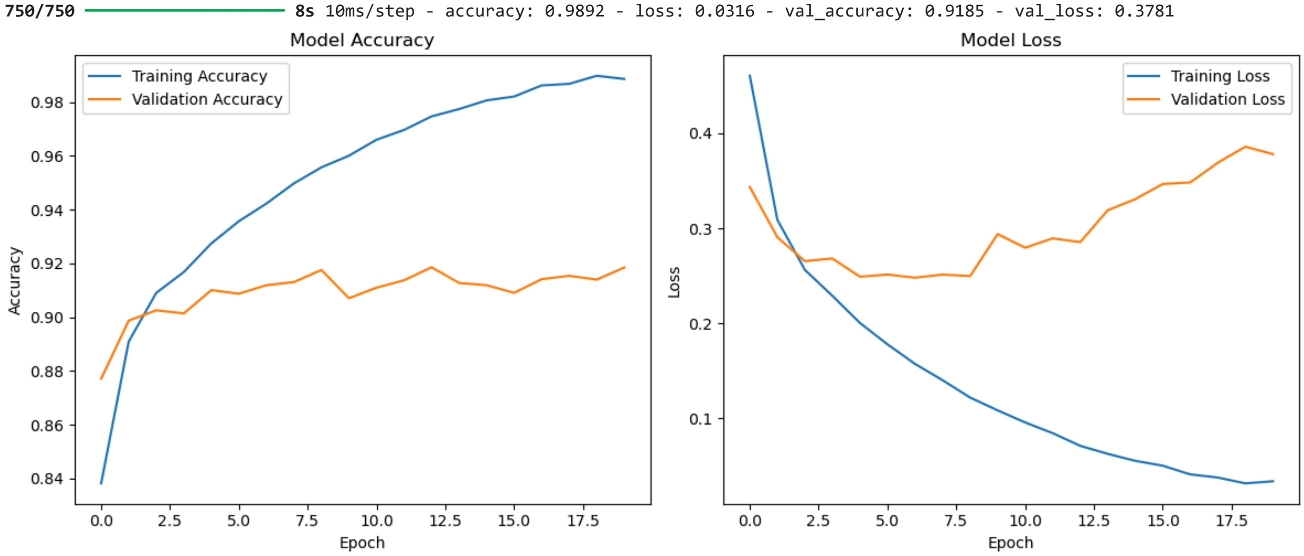
Layer (type)	Output Shape	Param #
rescaling_8 (Rescaling)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
flatten_8 (Flatten)	(None, 5408)	0
dense_25 (Dense)	(None, 128)	692,352
dense_26 (Dense)	(None, 10)	1,290

Total params: 693,962 (2.65 MB)

Trainable params: 693,962 (2.65 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 10s 11ms/step - accuracy: 0.7845 - loss: 0.6139 - val_accuracy: 0.8772 - val_loss: 0.3435
Epoch 2/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.8891 - loss: 0.3211 - val_accuracy: 0.8988 - val_loss: 0.2907
Epoch 3/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9090 - loss: 0.2561 - val_accuracy: 0.9026 - val_loss: 0.2655
Epoch 4/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9153 - loss: 0.2306 - val_accuracy: 0.9014 - val_loss: 0.2683
Epoch 5/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9305 - loss: 0.1968 - val_accuracy: 0.9101 - val_loss: 0.2491
Epoch 6/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9370 - loss: 0.1761 - val_accuracy: 0.9087 - val_loss: 0.2514
Epoch 7/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - accuracy: 0.9423 - loss: 0.1581 - val_accuracy: 0.9119 - val_loss: 0.2481
Epoch 8/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - accuracy: 0.9512 - loss: 0.1384 - val_accuracy: 0.9131 - val_loss: 0.2514
Epoch 9/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - accuracy: 0.9551 - loss: 0.1207 - val_accuracy: 0.9176 - val_loss: 0.2498
Epoch 10/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9621 - loss: 0.1033 - val_accuracy: 0.9071 - val_loss: 0.2940
Epoch 11/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9666 - loss: 0.0945 - val_accuracy: 0.9110 - val_loss: 0.2797
Epoch 12/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9722 - loss: 0.0785 - val_accuracy: 0.9137 - val_loss: 0.2895
Epoch 13/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9763 - loss: 0.0689 - val_accuracy: 0.9186 - val_loss: 0.2856
Epoch 14/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9782 - loss: 0.0602 - val_accuracy: 0.9128 - val_loss: 0.3191
Epoch 15/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9822 - loss: 0.0520 - val_accuracy: 0.9119 - val_loss: 0.3308
Epoch 16/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9839 - loss: 0.0456 - val_accuracy: 0.9091 - val_loss: 0.3467
Epoch 17/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9860 - loss: 0.0428 - val_accuracy: 0.9142 - val_loss: 0.3483
Epoch 18/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 7s 10ms/step - accuracy: 0.9882 - loss: 0.0353 - val_accuracy: 0.9154 - val_loss: 0.3691
Epoch 19/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - accuracy: 0.9907 - loss: 0.0281 - val_accuracy: 0.9140 - val_loss: 0.3859
Epoch 20/20
750/750 ━━━━━━━━━━━━━━━━━━━━ 8s 10ms/step - accuracy: 0.9892 - loss: 0.0316 - val_accuracy: 0.9185 - val_loss: 0.3781
```



In [29]:
```python
# print the highest validation accuracy and the epoch at which it was obtained for Model A
best_epoch_a = np.argmax(history_a.history['val_accuracy']) + 1
best_val_accuracy_a = history_a.history['val_accuracy'][best_epoch_a - 1]
print(f'Best validation accuracy: {best_val_accuracy_a:.4f} at epoch {best_epoch_a}')
# print the smallest validation loss and the epoch at which it was obtained for Model A
best_epoch_loss_a = np.argmin(history_a.history['val_loss']) + 1
best_val_loss_a = history_a.history['val_loss'][best_epoch_loss_a - 1]
print(f'Smallest validation loss: {best_val_loss_a:.4f} at epoch {best_epoch_loss_a}')
```

```
Best validation accuracy: 0.9186 at epoch 13
Smallest validation loss: 0.2481 at epoch 7
```

In [30]:
```python
# add Model A to the models dataframe
models = pd.concat([models, pd.DataFrame({
    'model': ['Model A'],
    'val_accuracy': [best_val_accuracy_a],
    'val_loss': [best_val_loss_a],
    'best_epoch': [best_epoch_a]
})], ignore_index=True)

# reset val_accuracy_rank based on validation accuracy
```

```
models['val_accuracy_rank'] = models['val_accuracy'].rank(ascending=False).astype(int)

models
```

Out[30]:

	model	val_accuracy	val_loss	best_epoch	val_accuracy_rank
0	Model 0	0.890500	0.312480	27	5
1	Model 1	0.896167	0.308760	30	2
2	Model 2	0.893167	0.294455	10	4
3	Model 3	0.885000	0.319558	30	7
4	Model 4	0.894583	0.314308	27	3
5	Model 5	0.888917	0.371258	24	6
6	Model 6	0.871250	0.462555	29	8
7	Model 7	0.857583	0.523946	26	9
8	Model A	0.918583	0.248096	13	1

For Model A, we implemented a convolutional neural network with one convolutional layer followed by a max-pooling layer to extract features from the images. This model achieved a notable improvement in validation accuracy (we got our best model at the moment), indicating the effectiveness of convolutional layers for image classification tasks. However, the model began to show signs of overfitting as the training loss continued to decrease while the validation loss increased after a certain number of epochs.

Model B: Deep Convolutional Model

In [31]:

```
# build Model B with three convolutional blocks
model_b = Sequential([
    Input(shape=(28, 28, 1)),
    Rescaling(1./255),

    # first Convolutional Block
    Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),  # dropout

    # second Convolutional Block
    Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    # third Convolutional Block
    Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.4),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

model_b.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

model_b.summary()


history_b = model_b.fit(
    train_images, train_labels,
    epochs=30,
    validation_data=(val_images, val_labels)
)

plot_training_history(history_b)
```

Model: "sequential_9"

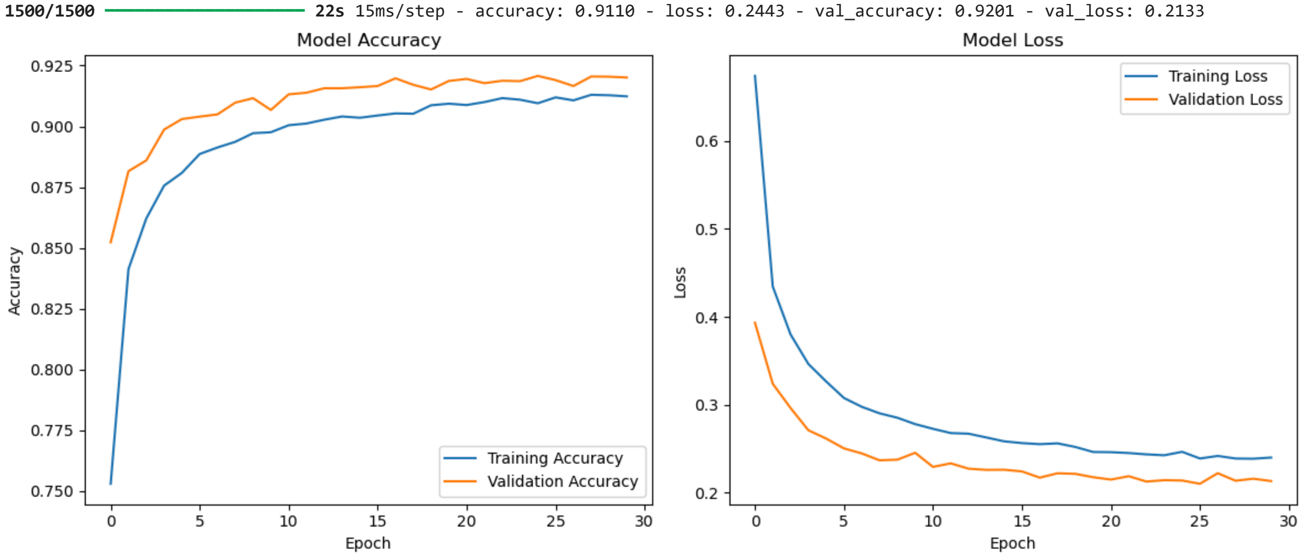
Layer (type)	Output Shape	Param #
rescaling_9 (Rescaling)	(None, 28, 28, 1)	0
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 32)	0
dropout_7 (Dropout)	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18,496
max_pooling2d_2 (MaxPooling2D)	(None, 7, 7, 64)	0
dropout_8 (Dropout)	(None, 7, 7, 64)	0
conv2d_3 (Conv2D)	(None, 7, 7, 128)	73,856
max_pooling2d_3 (MaxPooling2D)	(None, 3, 3, 128)	0
dropout_9 (Dropout)	(None, 3, 3, 128)	0
flatten_9 (Flatten)	(None, 1152)	0
dense_27 (Dense)	(None, 128)	147,584
dropout_10 (Dropout)	(None, 128)	0
dense_28 (Dense)	(None, 10)	1,290

Total params: 241,546 (943.54 KB)

Trainable params: 241,546 (943.54 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 23s 14ms/step - accuracy: 0.6439 - loss: 0.9648 - val_accuracy: 0.8523 - val_loss: 0.3933
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 21s 14ms/step - accuracy: 0.8357 - loss: 0.4531 - val_accuracy: 0.8816 - val_loss: 0.3239
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 21s 14ms/step - accuracy: 0.8583 - loss: 0.3868 - val_accuracy: 0.8860 - val_loss: 0.2964
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 15ms/step - accuracy: 0.8730 - loss: 0.3513 - val_accuracy: 0.8987 - val_loss: 0.2711
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 15ms/step - accuracy: 0.8813 - loss: 0.3324 - val_accuracy: 0.9030 - val_loss: 0.2616
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 41s 15ms/step - accuracy: 0.8884 - loss: 0.3107 - val_accuracy: 0.9040 - val_loss: 0.2505
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 26s 18ms/step - accuracy: 0.8916 - loss: 0.2974 - val_accuracy: 0.9049 - val_loss: 0.2448
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 25s 17ms/step - accuracy: 0.8941 - loss: 0.2899 - val_accuracy: 0.9097 - val_loss: 0.2369
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 27s 18ms/step - accuracy: 0.8975 - loss: 0.2816 - val_accuracy: 0.9116 - val_loss: 0.2377
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 26s 17ms/step - accuracy: 0.8978 - loss: 0.2744 - val_accuracy: 0.9068 - val_loss: 0.2455
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 28s 19ms/step - accuracy: 0.9027 - loss: 0.2662 - val_accuracy: 0.9132 - val_loss: 0.2294
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 15ms/step - accuracy: 0.9003 - loss: 0.2655 - val_accuracy: 0.9138 - val_loss: 0.2334
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 23s 15ms/step - accuracy: 0.9040 - loss: 0.2661 - val_accuracy: 0.9157 - val_loss: 0.2274
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 28s 19ms/step - accuracy: 0.9040 - loss: 0.2610 - val_accuracy: 0.9157 - val_loss: 0.2261
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 24s 16ms/step - accuracy: 0.9048 - loss: 0.2568 - val_accuracy: 0.9161 - val_loss: 0.2262
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 25s 17ms/step - accuracy: 0.9058 - loss: 0.2504 - val_accuracy: 0.9166 - val_loss: 0.2243
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 26s 17ms/step - accuracy: 0.9076 - loss: 0.2486 - val_accuracy: 0.9197 - val_loss: 0.2172
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 15ms/step - accuracy: 0.9058 - loss: 0.2521 - val_accuracy: 0.9171 - val_loss: 0.2222
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 23s 15ms/step - accuracy: 0.9114 - loss: 0.2475 - val_accuracy: 0.9152 - val_loss: 0.2215
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 14ms/step - accuracy: 0.9101 - loss: 0.2455 - val_accuracy: 0.9187 - val_loss: 0.2177
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 15ms/step - accuracy: 0.9102 - loss: 0.2389 - val_accuracy: 0.9195 - val_loss: 0.2150
Epoch 22/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 23s 15ms/step - accuracy: 0.9105 - loss: 0.2453 - val_accuracy: 0.9178 - val_loss: 0.2188
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 23s 15ms/step - accuracy: 0.9131 - loss: 0.2394 - val_accuracy: 0.9187 - val_loss: 0.2127
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 24s 16ms/step - accuracy: 0.9110 - loss: 0.2425 - val_accuracy: 0.9186 - val_loss: 0.2144
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 23s 15ms/step - accuracy: 0.9117 - loss: 0.2409 - val_accuracy: 0.9208 - val_loss: 0.2140
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 23s 16ms/step - accuracy: 0.9133 - loss: 0.2395 - val_accuracy: 0.9190 - val_loss: 0.2102
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 24s 16ms/step - accuracy: 0.9122 - loss: 0.2381 - val_accuracy: 0.9166 - val_loss: 0.2223
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 24s 16ms/step - accuracy: 0.9135 - loss: 0.2378 - val_accuracy: 0.9205 - val_loss: 0.2137
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 15ms/step - accuracy: 0.9132 - loss: 0.2373 - val_accuracy: 0.9204 - val_loss: 0.2160
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 22s 15ms/step - accuracy: 0.9110 - loss: 0.2443 - val_accuracy: 0.9201 - val_loss: 0.2133
```



In [32]:
```python
# print the highest validation accuracy and the epoch at which it was obtained for Model B
best_epoch_b = np.argmax(history_b.history['val_accuracy']) + 1
best_val_accuracy_b = history_b.history['val_accuracy'][best_epoch_b - 1]
print(f'Best validation accuracy: {best_val_accuracy_b:.4f} at epoch {best_epoch_b}')
# print the smallest validation loss and the epoch at which it was obtained for Model B
best_epoch_loss_b = np.argmin(history_b.history['val_loss']) + 1
best_val_loss_b = history_b.history['val_loss'][best_epoch_loss_b - 1]
print(f'Smallest validation loss: {best_val_loss_b:.4f} at epoch {best_epoch_loss_b}')
```

```
Best validation accuracy: 0.9208 at epoch 25
Smallest validation loss: 0.2102 at epoch 26
```

In [33]:
```python
# add Model B to the models dataframe
models = pd.concat([models, pd.DataFrame({
    'model': ['Model B'],
    'val_accuracy': [best_val_accuracy_b],
    'val_loss': [best_val_loss_b],
    'best_epoch': [best_epoch_b]
})], ignore_index=True)

models
```

Out[33]:

	model	val_accuracy	val_loss	best_epoch	val_accuracy_rank
0	Model 0	0.890500	0.312480	27	5.0
1	Model 1	0.896167	0.308760	30	2.0
2	Model 2	0.893167	0.294455	10	4.0
3	Model 3	0.885000	0.319558	30	7.0
4	Model 4	0.894583	0.314308	27	3.0
5	Model 5	0.888917	0.371258	24	6.0
6	Model 6	0.871250	0.462555	29	8.0
7	Model 7	0.857583	0.523946	26	9.0
8	Model A	0.918583	0.248096	13	1.0
9	Model B	0.920750	0.210236	25	NaN

Model B employed a more complex convolutional architecture, with deeper layering and added dropout to mitigate overfitting. This approach yielded a robust best validation accuracy of 92.08% at epoch 25, showcasing an improved performance over previous models. The smallest validation loss was notably low at 0.2102, reached at epoch 26, which indicates a consistent learning without significant overfitting. Considering this positive outcome, it's reasonable to explore further enhancements or to try a new variation in Model C.

Model C: Deep Convolutional Neural Network with Adjusted Dropout Rates

In [34]:
```python
# Model C: A deeper convolutional model with adjusted dropout rates
model_c = Sequential([
    Input(shape=(28, 28, 1)),
    Rescaling(1./255),

    # First Convolutional Block
    Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.2),  # slight dropout after initial feature extraction

    # Second Convolutional Block
    Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.3),  # increased dropout for more complex features

    # Third Convolutional Block
    Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.4),

    # Fourth Convolutional Block
    Conv2D(256, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.4),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

# compile Model C
model_c.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```python
# model Summary
model_c.summary()

# train Model C
history_c = model_c.fit(
    train_images, train_labels,
    epochs=30,
    validation_data=(val_images, val_labels)
)

# plot the training history for Model C
plot_training_history(history_c)
```

Model: "sequential_10"

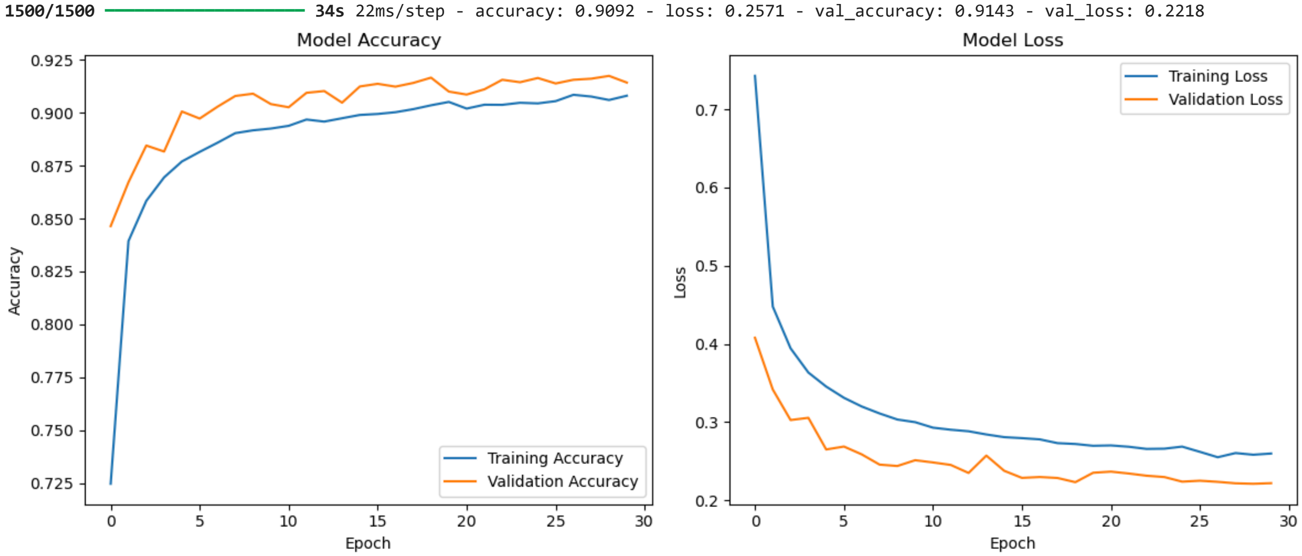
Layer (type)	Output Shape	Param #
rescaling_10 (Rescaling)	(None, 28, 28, 1)	0
conv2d_4 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_4 (MaxPooling2D)	(None, 14, 14, 32)	0
dropout_11 (Dropout)	(None, 14, 14, 32)	0
conv2d_5 (Conv2D)	(None, 14, 14, 64)	18,496
max_pooling2d_5 (MaxPooling2D)	(None, 7, 7, 64)	0
dropout_12 (Dropout)	(None, 7, 7, 64)	0
conv2d_6 (Conv2D)	(None, 7, 7, 128)	73,856
max_pooling2d_6 (MaxPooling2D)	(None, 3, 3, 128)	0
dropout_13 (Dropout)	(None, 3, 3, 128)	0
conv2d_7 (Conv2D)	(None, 3, 3, 256)	295,168
max_pooling2d_7 (MaxPooling2D)	(None, 1, 1, 256)	0
dropout_14 (Dropout)	(None, 1, 1, 256)	0
flatten_10 (Flatten)	(None, 256)	0
dense_29 (Dense)	(None, 128)	32,896
dropout_15 (Dropout)	(None, 128)	0
dense_30 (Dense)	(None, 10)	1,290

Total params: 422,026 (1.61 MB)

Trainable params: 422,026 (1.61 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 33s 21ms/step - accuracy: 0.5873 - loss: 1.0884 - val_accuracy: 0.8464 - val_loss: 0.4078
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - accuracy: 0.8332 - loss: 0.4668 - val_accuracy: 0.8672 - val_loss: 0.3417
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 31s 21ms/step - accuracy: 0.8551 - loss: 0.4004 - val_accuracy: 0.8845 - val_loss: 0.3026
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - accuracy: 0.8707 - loss: 0.3595 - val_accuracy: 0.8817 - val_loss: 0.3054
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - accuracy: 0.8777 - loss: 0.3428 - val_accuracy: 0.9006 - val_loss: 0.2649
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 31s 21ms/step - accuracy: 0.8793 - loss: 0.3353 - val_accuracy: 0.8972 - val_loss: 0.2687
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 33s 22ms/step - accuracy: 0.8840 - loss: 0.3220 - val_accuracy: 0.9028 - val_loss: 0.2586
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - accuracy: 0.8907 - loss: 0.3072 - val_accuracy: 0.9079 - val_loss: 0.2456
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 31s 21ms/step - accuracy: 0.8919 - loss: 0.2999 - val_accuracy: 0.9090 - val_loss: 0.2438
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 31s 21ms/step - accuracy: 0.8949 - loss: 0.2946 - val_accuracy: 0.9041 - val_loss: 0.2512
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 22ms/step - accuracy: 0.8960 - loss: 0.2868 - val_accuracy: 0.9026 - val_loss: 0.2484
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - accuracy: 0.8972 - loss: 0.2874 - val_accuracy: 0.9094 - val_loss: 0.2452
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 33s 22ms/step - accuracy: 0.8970 - loss: 0.2889 - val_accuracy: 0.9103 - val_loss: 0.2349
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 35s 23ms/step - accuracy: 0.8976 - loss: 0.2861 - val_accuracy: 0.9047 - val_loss: 0.2571
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 35s 24ms/step - accuracy: 0.8993 - loss: 0.2810 - val_accuracy: 0.9124 - val_loss: 0.2377
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 36s 24ms/step - accuracy: 0.9005 - loss: 0.2784 - val_accuracy: 0.9137 - val_loss: 0.2286
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 22ms/step - accuracy: 0.9008 - loss: 0.2740 - val_accuracy: 0.9123 - val_loss: 0.2297
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 22ms/step - accuracy: 0.9019 - loss: 0.2689 - val_accuracy: 0.9141 - val_loss: 0.2284
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 33s 22ms/step - accuracy: 0.9046 - loss: 0.2705 - val_accuracy: 0.9166 - val_loss: 0.2231
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 22ms/step - accuracy: 0.9043 - loss: 0.2697 - val_accuracy: 0.9100 - val_loss: 0.2352
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 22ms/step - accuracy: 0.9044 - loss: 0.2661 - val_accuracy: 0.9086 - val_loss: 0.2366
Epoch 22/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 36s 24ms/step - accuracy: 0.9072 - loss: 0.2643 - val_accuracy: 0.9111 - val_loss: 0.2342
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 33s 22ms/step - accuracy: 0.9052 - loss: 0.2633 - val_accuracy: 0.9156 - val_loss: 0.2313
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 33s 22ms/step - accuracy: 0.9043 - loss: 0.2663 - val_accuracy: 0.9144 - val_loss: 0.2297
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 33s 22ms/step - accuracy: 0.9051 - loss: 0.2652 - val_accuracy: 0.9164 - val_loss: 0.2238
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - accuracy: 0.9049 - loss: 0.2622 - val_accuracy: 0.9138 - val_loss: 0.2250
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 33s 22ms/step - accuracy: 0.9102 - loss: 0.2496 - val_accuracy: 0.9156 - val_loss: 0.2235
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 32s 21ms/step - accuracy: 0.9083 - loss: 0.2555 - val_accuracy: 0.9161 - val_loss: 0.2217
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 33s 22ms/step - accuracy: 0.9060 - loss: 0.2541 - val_accuracy: 0.9174 - val_loss: 0.2210
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 34s 22ms/step - accuracy: 0.9092 - loss: 0.2571 - val_accuracy: 0.9143 - val_loss: 0.2218
```



In [35]:
```python
# print the highest validation accuracy and the epoch at which it was obtained for Model C
best_epoch_c = np.argmax(history_c.history['val_accuracy']) + 1
best_val_accuracy_c = history_c.history['val_accuracy'][best_epoch_c - 1]
print(f'Best validation accuracy: {best_val_accuracy_c:.4f} at epoch {best_epoch_c}')
# print the smallest validation loss and the epoch at which it was obtained for Model C
best_epoch_loss_c = np.argmin(history_c.history['val_loss']) + 1
best_val_loss_c = history_c.history['val_loss'][best_epoch_loss_c - 1]
print(f'Smallest validation loss: {best_val_loss_c:.4f} at epoch {best_epoch_loss_c}')
```

```
Best validation accuracy: 0.9174 at epoch 29
Smallest validation loss: 0.2210 at epoch 29
```

In [36]:
```python
# add Model C to the models dataframe
models = pd.concat([models, pd.DataFrame({
    'model': ['Model C'],
    'val_accuracy': [best_val_accuracy_c],
    'val_loss': [best_val_loss_c],
    'best_epoch': [best_epoch_c]
})], ignore_index=True)

models
```

Out[36]:

	model	val_accuracy	val_loss	best_epoch	val_accuracy_rank
0	Model 0	0.890500	0.312480	27	5.0
1	Model 1	0.896167	0.308760	30	2.0
2	Model 2	0.893167	0.294455	10	4.0
3	Model 3	0.885000	0.319558	30	7.0
4	Model 4	0.894583	0.314308	27	3.0
5	Model 5	0.888917	0.371258	24	6.0
6	Model 6	0.871250	0.462555	29	8.0
7	Model 7	0.857583	0.523946	26	9.0
8	Model A	0.918583	0.248096	13	1.0
9	Model B	0.920750	0.210236	25	NaN
10	Model C	0.917417	0.221049	29	NaN

Adding more convulotional layers and adjusting dropout rates didn't help to improve the accuracy. Model B is still the best. Considering that we are going to focus more on optimization rather than on increasing the complexity of the model.

Model D: Optimized deep convolutional network

In [37]:
```python
# Learning Rate Schedule
def lr_schedule(epoch, lr):
    """Learning Rate Schedule

    Learning rate is scheduled to be reduced after 20, 30, 40 epochs.
    Called automatically every epoch as part of callbacks during training.
    """
    if epoch > 40:
        lr *= 0.5e-3
    elif epoch > 30:
        lr *= 1e-3
    elif epoch > 20:
        lr *= 1e-2
    elif epoch > 10:
        lr *= 1e-1
    print('Learning rate: ', lr)
    return lr

# Model D: optimized Deep Convolutional Network with Batch Normalization and Learning Rate Schedule
model_d = Sequential([
    Input(shape=(28, 28, 1)),
    Rescaling(1./255),

    # First Convolutional Block
    Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),
    BatchNormalization(), # batch normalization
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),

    # Second Convolutional Block
    Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
```

```
    # Third Convolutional Block
    Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.4),

    Flatten(),
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

# optimizer
optimizer = Adam(learning_rate=0.001)
model_d.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

model_d.summary()

# set up the learning rate schedule callback
lr_scheduler = LearningRateScheduler(lr_schedule)

# train Model D
history_d = model_d.fit(
    train_images, train_labels,
    epochs=30,
    validation_data=(val_images, val_labels),
    callbacks=[lr_scheduler]
)

# plot the training history for Model D
plot_training_history(history_d)
```

Model: "sequential_11"

Layer (type)	Output Shape	Param #
rescaling_11 (Rescaling)	(None, 28, 28, 1)	0
conv2d_8 (Conv2D)	(None, 28, 28, 32)	320
batch_normalization (BatchNormalization)	(None, 28, 28, 32)	128
max_pooling2d_8 (MaxPooling2D)	(None, 14, 14, 32)	0
dropout_16 (Dropout)	(None, 14, 14, 32)	0
conv2d_9 (Conv2D)	(None, 14, 14, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 14, 14, 64)	256
max_pooling2d_9 (MaxPooling2D)	(None, 7, 7, 64)	0
dropout_17 (Dropout)	(None, 7, 7, 64)	0
conv2d_10 (Conv2D)	(None, 7, 7, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 7, 7, 128)	512
max_pooling2d_10 (MaxPooling2D)	(None, 3, 3, 128)	0
dropout_18 (Dropout)	(None, 3, 3, 128)	0
flatten_11 (Flatten)	(None, 1152)	0
dense_31 (Dense)	(None, 128)	147,584
batch_normalization_3 (BatchNormalization)	(None, 128)	512
dropout_19 (Dropout)	(None, 128)	0
dense_32 (Dense)	(None, 10)	1,290

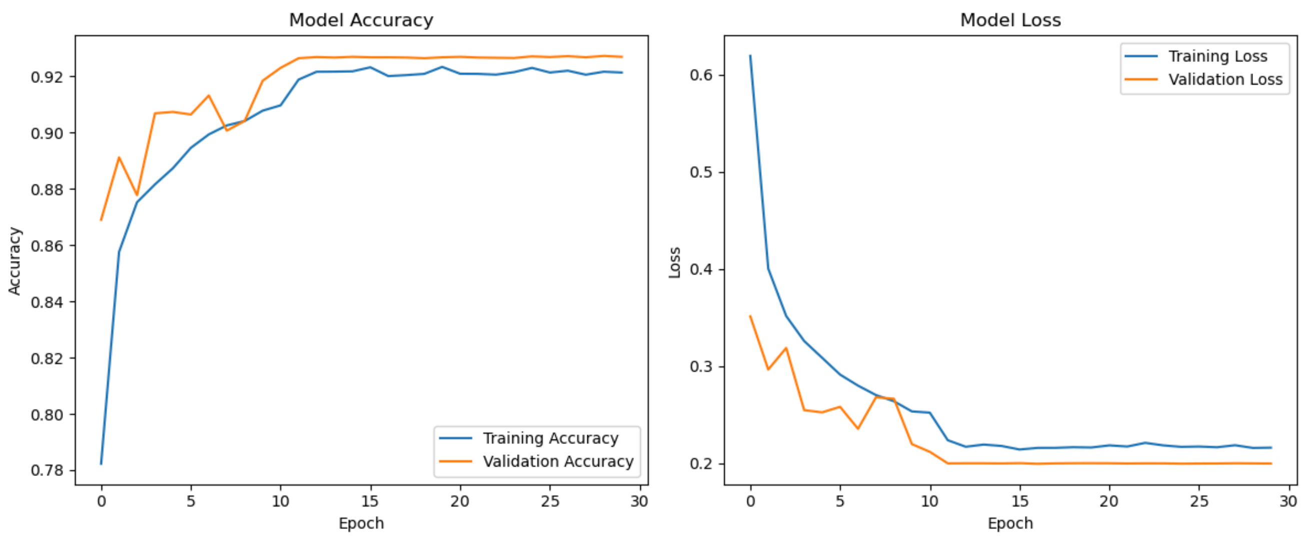
Total params: 242,954 (949.04 KB)

Trainable params: 242,250 (946.29 KB)

Non-trainable params: 704 (2.75 KB)

```
Learning rate:  0.0010000000474974513
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 40s 24ms/step - accuracy: 0.7034 - loss: 0.8737 - val_accuracy: 0.8690 - val_loss: 0.3510 - learnin
g_rate: 0.0010
Learning rate:  0.0010000000474974513
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 37s 25ms/step - accuracy: 0.8507 - loss: 0.4114 - val_accuracy: 0.8912 - val_loss: 0.2963 - learnin
g_rate: 0.0010
Learning rate:  0.0010000000474974513
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 36s 24ms/step - accuracy: 0.8743 - loss: 0.3544 - val_accuracy: 0.8777 - val_loss: 0.3185 - learnin
g_rate: 0.0010
Learning rate:  0.0010000000474974513
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 35s 23ms/step - accuracy: 0.8809 - loss: 0.3272 - val_accuracy: 0.9068 - val_loss: 0.2545 - learnin
g_rate: 0.0010
Learning rate:  0.0010000000474974513
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 35s 23ms/step - accuracy: 0.8878 - loss: 0.3038 - val_accuracy: 0.9073 - val_loss: 0.2523 - learnin
g_rate: 0.0010
Learning rate:  0.0010000000474974513
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 35s 24ms/step - accuracy: 0.8958 - loss: 0.2877 - val_accuracy: 0.9064 - val_loss: 0.2580 - learnin
g_rate: 0.0010
Learning rate:  0.0010000000474974513
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 35s 23ms/step - accuracy: 0.9001 - loss: 0.2778 - val_accuracy: 0.9132 - val_loss: 0.2353 - learnin
g_rate: 0.0010
Learning rate:  0.0010000000474974513
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 39s 26ms/step - accuracy: 0.9033 - loss: 0.2678 - val_accuracy: 0.9007 - val_loss: 0.2678 - learnin
g_rate: 0.0010
Learning rate:  0.0010000000474974513
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 40s 27ms/step - accuracy: 0.9048 - loss: 0.2591 - val_accuracy: 0.9042 - val_loss: 0.2663 - learnin
g_rate: 0.0010
Learning rate:  0.0010000000474974513
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 48s 32ms/step - accuracy: 0.9069 - loss: 0.2530 - val_accuracy: 0.9184 - val_loss: 0.2196 - learnin
g_rate: 0.0010
Learning rate:  0.0010000000474974513
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 41s 28ms/step - accuracy: 0.9091 - loss: 0.2511 - val_accuracy: 0.9230 - val_loss: 0.2116 - learnin
g_rate: 0.0010
Learning rate:  0.00010000000474974513
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 38s 25ms/step - accuracy: 0.9172 - loss: 0.2285 - val_accuracy: 0.9264 - val_loss: 0.1998 - learnin
g_rate: 1.0000e-04
Learning rate:  1.0000000474974514e-05
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 41s 27ms/step - accuracy: 0.9204 - loss: 0.2185 - val_accuracy: 0.9268 - val_loss: 0.1999 - learnin
g_rate: 1.0000e-05
Learning rate:  1.0000000656873453e-06
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 41s 27ms/step - accuracy: 0.9217 - loss: 0.2170 - val_accuracy: 0.9267 - val_loss: 0.1999 - learnin
g_rate: 1.0000e-06
Learning rate:  1.0000001111620805e-07
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 39s 26ms/step - accuracy: 0.9225 - loss: 0.2174 - val_accuracy: 0.9269 - val_loss: 0.1997 - learnin
g_rate: 1.0000e-07
Learning rate:  1.000000082740371e-08
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 44s 29ms/step - accuracy: 0.9243 - loss: 0.2126 - val_accuracy: 0.9268 - val_loss: 0.2000 - learnin
g_rate: 1.0000e-08
Learning rate:  1.000000082740371e-09
Epoch 17/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 38s 25ms/step - accuracy: 0.9212 - loss: 0.2140 - val_accuracy: 0.9268 - val_loss: 0.1993 - learnin
g_rate: 1.0000e-09
Learning rate:  1.000000082740371e-10
Epoch 18/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 38s 25ms/step - accuracy: 0.9187 - loss: 0.2148 - val_accuracy: 0.9267 - val_loss: 0.1998 - learnin
g_rate: 1.0000e-10
Learning rate:  1.000000082740371e-11
Epoch 19/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 35s 24ms/step - accuracy: 0.9210 - loss: 0.2157 - val_accuracy: 0.9264 - val_loss: 0.1999 - learnin
g_rate: 1.0000e-11
Learning rate:  1.000000082740371e-12
Epoch 20/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 35s 23ms/step - accuracy: 0.9236 - loss: 0.2176 - val_accuracy: 0.9268 - val_loss: 0.1999 - learnin
g_rate: 1.0000e-12
Learning rate:  1.0000001044244145e-13
Epoch 21/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 39s 26ms/step - accuracy: 0.9206 - loss: 0.2161 - val_accuracy: 0.9269 - val_loss: 0.1999 - learnin
g_rate: 1.0000e-13
Learning rate:  1.0000001179769416e-15
Epoch 22/30
```

```
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 38s 25ms/step - accuracy: 0.9222 - loss: 0.2152 - val_accuracy: 0.9267 - val_loss: 0.1997 - learnin
g_rate: 1.0000e-15
Learning rate:  1.0000001095066121e-17
Epoch 23/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 38s 25ms/step - accuracy: 0.9200 - loss: 0.2266 - val_accuracy: 0.9266 - val_loss: 0.1998 - learnin
g_rate: 1.0000e-17
Learning rate:  1.0000001492112816e-19
Epoch 24/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 38s 25ms/step - accuracy: 0.9220 - loss: 0.2184 - val_accuracy: 0.9265 - val_loss: 0.1998 - learnin
g_rate: 1.0000e-19
Learning rate:  1.0000001621359786e-21
Epoch 25/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 38s 25ms/step - accuracy: 0.9226 - loss: 0.2140 - val_accuracy: 0.9271 - val_loss: 0.1995 - learnin
g_rate: 1.0000e-21
Learning rate:  1.0000001702139143e-23
Epoch 26/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 40s 27ms/step - accuracy: 0.9230 - loss: 0.2162 - val_accuracy: 0.9268 - val_loss: 0.1996 - learnin
g_rate: 1.0000e-23
Learning rate:  1.0000001575921398e-25
Epoch 27/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 38s 25ms/step - accuracy: 0.9215 - loss: 0.2176 - val_accuracy: 0.9272 - val_loss: 0.1997 - learnin
g_rate: 1.0000e-25
Learning rate:  1.000000142800998e-27
Epoch 28/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 37s 24ms/step - accuracy: 0.9199 - loss: 0.2207 - val_accuracy: 0.9268 - val_loss: 0.1999 - learnin
g_rate: 1.0000e-27
Learning rate:  1.0000001235416983e-29
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 36s 24ms/step - accuracy: 0.9206 - loss: 0.2195 - val_accuracy: 0.9273 - val_loss: 0.1998 - learnin
g_rate: 1.0000e-29
Learning rate:  1.0000001536343538e-31
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 38s 25ms/step - accuracy: 0.9217 - loss: 0.2159 - val_accuracy: 0.9269 - val_loss: 0.1996 - learnin
g_rate: 1.0000e-31
```



In [38]:
```python
# print the highest validation accuracy and the epoch at which it was obtained for Model D
best_epoch_d = np.argmax(history_d.history['val_accuracy']) + 1
best_val_accuracy_d = history_d.history['val_accuracy'][best_epoch_d - 1]
print(f'Best validation accuracy: {best_val_accuracy_d:.4f} at epoch {best_epoch_d}')
# print the smallest validation loss and the epoch at which it was obtained for Model D
best_epoch_loss_d = np.argmin(history_d.history['val_loss']) + 1
best_val_loss_d = history_d.history['val_loss'][best_epoch_loss_d - 1]
print(f'Smallest validation loss: {best_val_loss_d:.4f} at epoch {best_epoch_loss_d}')
```

```
Best validation accuracy: 0.9273 at epoch 29
Smallest validation loss: 0.1993 at epoch 17
```

In [39]:
```python
# add Model D to the models dataframe
models = pd.concat([models, pd.DataFrame({
    'model': ['Model D'],
    'val_accuracy': [best_val_accuracy_d],
    'val_loss': [best_val_loss_d],
    'best_epoch': [best_epoch_d]
})], ignore_index=True)

models
```

Out[39]:

	model	val_accuracy	val_loss	best_epoch	val_accuracy_rank
0	Model 0	0.890500	0.312480	27	5.0
1	Model 1	0.896167	0.308760	30	2.0
2	Model 2	0.893167	0.294455	10	4.0
3	Model 3	0.885000	0.319558	30	7.0
4	Model 4	0.894583	0.314308	27	3.0
5	Model 5	0.888917	0.371258	24	6.0
6	Model 6	0.871250	0.462555	29	8.0
7	Model 7	0.857583	0.523946	26	9.0
8	Model A	0.918583	0.248096	13	1.0
9	Model B	0.920750	0.210236	25	NaN
10	Model C	0.917417	0.221049	29	NaN
11	Model D	0.927250	0.199344	29	NaN

In [40]:

```
# reset the models dataframe val_accuracy_rank column based on the new validation accuracy values
models['val_accuracy_rank'] = models['val_accuracy'].rank(ascending=False).astype(int)

models
```

Out[40]:

	model	val_accuracy	val_loss	best_epoch	val_accuracy_rank
0	Model 0	0.890500	0.312480	27	8
1	Model 1	0.896167	0.308760	30	5
2	Model 2	0.893167	0.294455	10	7
3	Model 3	0.885000	0.319558	30	10
4	Model 4	0.894583	0.314308	27	6
5	Model 5	0.888917	0.371258	24	9
6	Model 6	0.871250	0.462555	29	11
7	Model 7	0.857583	0.523946	26	12
8	Model A	0.918583	0.248096	13	3
9	Model B	0.920750	0.210236	25	2
10	Model C	0.917417	0.221049	29	4
11	Model D	0.927250	0.199344	29	1

Model D with 3 convolutional layers, pooling, batch normalization and learning rate scheduler performed really well, we got highest accuracy so far 92.7% at epoch 29.

Across models A to D, we've explored a variety of convolutional neural network architectures, progressively building complexity and incorporating techniques like dropout and batch normalization to combat overfitting. Model D shows the best performance in both validation accuracy and loss, suggesting that the combination of deep convolutional layers and additional regularization, along with learning rate scheduling, can get us better accuracy than previous models.

Question 6

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

In [41]:

```
# define a function to pad the images
def pad(images):
    if images.ndim == 3:
        images = images[..., tf.newaxis]
    padding = [[0, 0], [2, 2], [2, 2], [0, 0]]
    images_padded = tf.pad(images, paddings=padding, mode='CONSTANT', constant_values=0)
    return images_padded

# convert to rgb
def convert_to_rgb(images):
    images_rgb = tf.repeat(images, 3, axis=-1)
    return images_rgb

train_images_padded = pad(train_images)
val_images_padded = pad(val_images)

train_images_rgb = convert_to_rgb(train_images_padded)
```

```python
val_images_rgb = convert_to_rgb(val_images_padded)

# load MobileNetV2
MobileNetV2_base = MobileNetV2(weights='imagenet', include_top=False, input_shape=(32, 32, 3), alpha=1.0)

# freeze
MobileNetV2_base.trainable = False

# build our custom model
custom_model = Sequential([
    MobileNetV2_base, # MobileNetV2 base
    GlobalAveragePooling2D(),
    BatchNormalization(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])

custom_model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

print(custom_model.summary())

# callbacks
callbacks_list = [
    EarlyStopping(monitor='val_accuracy', patience=3, verbose=1),
    ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=2, verbose=1)
]

# train the custom model
history_custom_model = custom_model.fit(
    train_images_rgb, train_labels,
    epochs=30,
    validation_data=(val_images_rgb, val_labels),
    callbacks=callbacks_list  # using callbacks for early stopping and the learning rate reduction
)

# plot the training history for the custom model
plot_training_history(history_custom_model)
```

Model: "sequential_12"

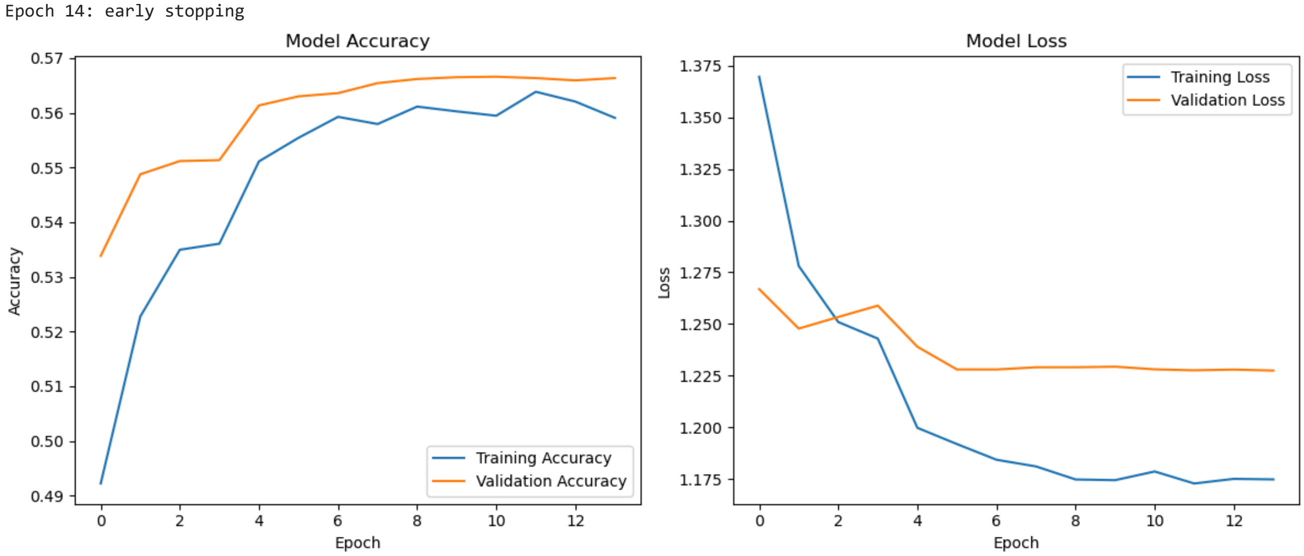
Layer (type)	Output Shape	Param #
mobilenetv2_1.00_224 (Functional)	?	2,257,984
global_average_pooling2d (GlobalAveragePooling2D)	?	0 (unbuilt)
batch_normalization_4 (BatchNormalization)	?	0 (unbuilt)
dense_33 (Dense)	?	0 (unbuilt)
dropout_20 (Dropout)	?	0
dense_34 (Dense)	?	0 (unbuilt)

Total params: 2,257,984 (8.61 MB)

Trainable params: 0 (0.00 B)

Non-trainable params: 2,257,984 (8.61 MB)

```
None
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 54s 32ms/step - accuracy: 0.4626 - loss: 1.4663 - val_accuracy: 0.5338 - val_loss: 1.2669 - learnin
g_rate: 0.0010
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 48s 32ms/step - accuracy: 0.5201 - loss: 1.2823 - val_accuracy: 0.5487 - val_loss: 1.2478 - learnin
g_rate: 0.0010
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 48s 32ms/step - accuracy: 0.5332 - loss: 1.2642 - val_accuracy: 0.5512 - val_loss: 1.2534 - learnin
g_rate: 0.0010
Epoch 4/30
1499/1500 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.5343 - loss: 1.2429
Epoch 4: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 50s 33ms/step - accuracy: 0.5343 - loss: 1.2429 - val_accuracy: 0.5513 - val_loss: 1.2589 - learnin
g_rate: 0.0010
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 49s 33ms/step - accuracy: 0.5490 - loss: 1.2043 - val_accuracy: 0.5613 - val_loss: 1.2390 - learnin
g_rate: 1.0000e-04
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 51s 34ms/step - accuracy: 0.5538 - loss: 1.1953 - val_accuracy: 0.5630 - val_loss: 1.2280 - learnin
g_rate: 1.0000e-04
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 49s 33ms/step - accuracy: 0.5579 - loss: 1.1852 - val_accuracy: 0.5636 - val_loss: 1.2280 - learnin
g_rate: 1.0000e-04
Epoch 8/30
1499/1500 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.5603 - loss: 1.1779
Epoch 8: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 51s 34ms/step - accuracy: 0.5603 - loss: 1.1779 - val_accuracy: 0.5654 - val_loss: 1.2290 - learnin
g_rate: 1.0000e-04
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 50s 33ms/step - accuracy: 0.5614 - loss: 1.1795 - val_accuracy: 0.5662 - val_loss: 1.2291 - learnin
g_rate: 1.0000e-05
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.5598 - loss: 1.1780
Epoch 10: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 50s 33ms/step - accuracy: 0.5598 - loss: 1.1780 - val_accuracy: 0.5665 - val_loss: 1.2293 - learnin
g_rate: 1.0000e-05
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 49s 33ms/step - accuracy: 0.5594 - loss: 1.1763 - val_accuracy: 0.5666 - val_loss: 1.2281 - learnin
g_rate: 1.0000e-06
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 51s 34ms/step - accuracy: 0.5674 - loss: 1.1695 - val_accuracy: 0.5663 - val_loss: 1.2276 - learnin
g_rate: 1.0000e-06
Epoch 13/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 50s 33ms/step - accuracy: 0.5590 - loss: 1.1773 - val_accuracy: 0.5659 - val_loss: 1.2279 - learnin
g_rate: 1.0000e-06
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 51s 34ms/step - accuracy: 0.5575 - loss: 1.1770 - val_accuracy: 0.5663 - val_loss: 1.2275 - learnin
g_rate: 1.0000e-06
Epoch 14: early stopping
```



In [42]:

```python
# print the highest validation accuracy and the epoch at which it was obtained for the custom model
best_epoch_custom_model = np.argmax(history_custom_model.history['val_accuracy']) + 1
best_val_accuracy_custom_model = history_custom_model.history['val_accuracy'][best_epoch_custom_model - 1]
print(f'Best validation accuracy: {best_val_accuracy_custom_model:.4f} at epoch {best_epoch_custom_model}')
# print the smallest validation loss and the epoch at which it was obtained for the custom model
best_epoch_loss_custom_model = np.argmin(history_custom_model.history['val_loss']) + 1
best_val_loss_custom_model = history_custom_model.history['val_loss'][best_epoch_loss_custom_model - 1]
print(f'Smallest validation loss: {best_val_loss_custom_model:.4f} at epoch {best_epoch_loss_custom_model}')
```

```
Best validation accuracy: 0.5666 at epoch 11
Smallest validation loss: 1.2275 at epoch 14
```

In [43]:
```
# add the custom model to the models dataframe
models = pd.concat([models, pd.DataFrame({
    'model': ['Custom Model'],
    'val_accuracy': [best_val_accuracy_custom_model],
    'val_loss': [best_val_loss_custom_model],
    'best_epoch': [best_epoch_custom_model]
})], ignore_index=True)

# reset the models dataframe val_accuracy_rank column based on the new validation accuracy values
models['val_accuracy_rank'] = models['val_accuracy'].rank(ascending=False).astype(int)

models
```

Out[43]:

	model	val_accuracy	val_loss	best_epoch	val_accuracy_rank
0	Model 0	0.890500	0.312480	27	8
1	Model 1	0.896167	0.308760	30	5
2	Model 2	0.893167	0.294455	10	7
3	Model 3	0.885000	0.319558	30	10
4	Model 4	0.894583	0.314308	27	6
5	Model 5	0.888917	0.371258	24	9
6	Model 6	0.871250	0.462555	29	11
7	Model 7	0.857583	0.523946	26	12
8	Model A	0.918583	0.248096	13	3
9	Model B	0.920750	0.210236	25	2
10	Model C	0.917417	0.221049	29	4
11	Model D	0.927250	0.199344	29	1
12	Custom Model	0.566583	1.227457	11	13

The Custom Model, incorporating MobileNetV2 as a feature extractor, shows pretty low accuracy for both training and validation. The learning rate adjustments and early stopping helped us in avoiding overfitting, but the overall performance indicates that this approach may not be the best fit for such small, grayscale images. There is a chance that other pretrained models would perform better. We have tried to use VGG16 and EfficientNetB0 models, but bumped into issues with having not enough memory on the machine.

Question 7

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

ANSWER We'll select the final model based on the best validation accuracy achieved in previous steps. From our models dataframe Model D had the highest validation accuracy, we would evaluate this model on the test set to compare its performance against the validation error.

In [44]:
```
# evaluate the model on the test set
test_loss, test_accuracy = model_d.evaluate(test_images, test_labels)

print(f"Test Accuracy: {test_accuracy:.4f}")
print(f"Test Loss: {test_loss:.4f}")

# compare the test accuracy and loss with the validation accuracy and loss from Model D
print(f"Validation Accuracy for Model D: {best_val_accuracy_d:.4f}")
print(f"Validation Loss for Model D: {best_val_loss_d:.4f}")
```

```
 14/313 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9260 - loss: 0.2401  313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy:
0.9224 - loss: 0.2152
Test Accuracy: 0.9229
Test Loss: 0.2109
Validation Accuracy for Model D: 0.9273
Validation Loss for Model D: 0.1993
```

Model D, when evaluated on the test set, achieved an accuracy of 92.29%, with a corresponding loss of 0.2109. Compared to the validation results, the test accuracy is slightly lower by 0.44 percentage points, and the test loss is slightly higher by about 0.0116. This slight decrease in accuracy and increase in loss on the test set, indicates that Model D is generalizing well to unseen data, with only a minor drop in performance. This performance gap is acceptable and indicates that the model has not overfitted to the validation data and maintains its predictive power on new, unseen data. Let's calculate the accuracy for other models as well.

In [45]:
```
# calculate test accuracy and loss for all models in the models dataframe
test_accuracies = []
test_losses = []

# all models list
all_models = [model, model_1, model_2, model_3, model_4, model_5, model_6, model_7, model_a, model_b, model_c, model_d, custom_mod
```

```python
for model in all_models:
    if model == custom_model:
        test_images_padded = pad(test_images)
        test_images_rgb = convert_to_rgb(test_images_padded)
        test_loss, test_accuracy = model.evaluate(test_images_rgb, test_labels)
    else:
        test_loss, test_accuracy = model.evaluate(test_images, test_labels)
    test_accuracies.append(test_accuracy)
    test_losses.append(test_loss)

# add the test accuracy and loss to the models dataframe
models['test_accuracy'] = test_accuracies
models['test_loss'] = test_losses
# rank the models based on test accuracy
models['test_accuracy_rank'] = models['test_accuracy'].rank(ascending=False).astype(int)
# reset the val_accuracy_rank column based on the new test accuracy values
models['val_accuracy_rank'] = models['val_accuracy'].rank(ascending=False).astype(int)

models
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - accuracy: 0.8834 - loss: 0.3888
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8897 - loss: 0.5258
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8917 - loss: 0.4902
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8780 - loss: 0.3471
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8868 - loss: 0.5575
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8785 - loss: 0.4275
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8589 - loss: 0.4932
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8409 - loss: 0.5956
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9120 - loss: 0.4319
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9194 - loss: 0.2252
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9139 - loss: 0.2458
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9224 - loss: 0.2152
313/313 ━━━━━━━━━━━━━━━━━━━━ 9s 23ms/step - accuracy: 0.5670 - loss: 1.2315
```

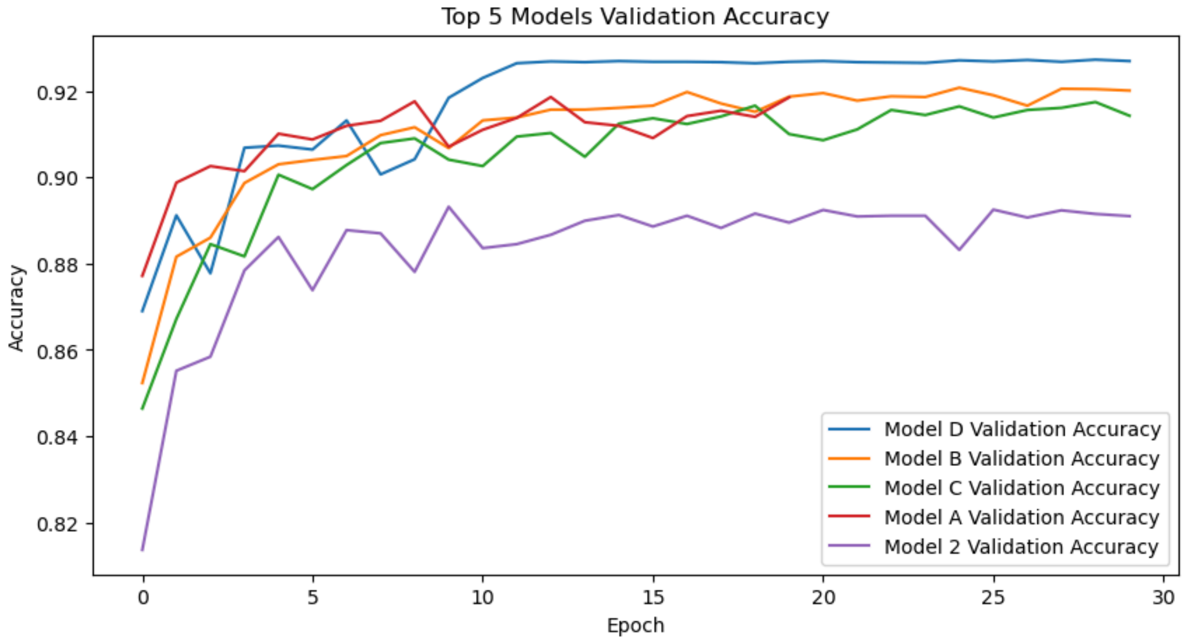
Out[45]:

	model	val_accuracy	val_loss	best_epoch	val_accuracy_rank	test_accuracy	test_loss	test_accuracy_rank
0	Model 0	0.890500	0.312480	27	8	0.8848	0.391090	8
1	Model 1	0.896167	0.308760	30	5	0.8898	0.518272	6
2	Model 2	0.893167	0.294455	10	7	0.8914	0.481396	5
3	Model 3	0.885000	0.319558	30	10	0.8801	0.346686	9
4	Model 4	0.894583	0.314308	27	6	0.8879	0.535407	7
5	Model 5	0.888917	0.371258	24	9	0.8793	0.430896	10
6	Model 6	0.871250	0.462555	29	11	0.8575	0.493536	11
7	Model 7	0.857583	0.523946	26	12	0.8387	0.597240	12
8	Model A	0.918583	0.248096	13	3	0.9157	0.406704	3
9	Model B	0.920750	0.210236	25	2	0.9186	0.224004	2
10	Model C	0.917417	0.221049	29	4	0.9147	0.238689	4
11	Model D	0.927250	0.199344	29	1	0.9229	0.210937	1
12	Custom Model	0.566583	1.227457	11	13	0.5708	1.214219	13

As it was expected Model D outperforms others with the highest test accuracy of 92.29% and a closely matching test loss (0.210937) to the validation loss, indicating consistency between validation and test performance. The results also demonstrate that while Model B ranks second in test accuracy with tiny difference suggesting that it also generalizes well. Comparing convolutional neural networks with others we can also notice that all 4 models performed better than others.

In [48]:

```python
# plot top 5 models historical accuracy
plt.figure(figsize=(10, 5))
plt.plot(history_d.history['val_accuracy'], label='Model D Validation Accuracy')
plt.plot(history_b.history['val_accuracy'], label='Model B Validation Accuracy')
plt.plot(history_c.history['val_accuracy'], label='Model C Validation Accuracy')
plt.plot(history_a.history['val_accuracy'], label='Model A Validation Accuracy')
plt.plot(history_2.history['val_accuracy'], label='Model 2 Validation Accuracy')
plt.title('Top 5 Models Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



In [49]:
```python
# plot bar chart of test accuracy for all models
plt.figure(figsize=(10, 5))
models.set_index('model')['test_accuracy'].plot(kind='bar')
plt.ylabel('Test Accuracy')
plt.title('Test Accuracy for all models')
# add labels
for i, v in enumerate(models['test_accuracy']):
    plt.text(i, v + 0.01, f'{v:.4f}', ha='center')
plt.xticks(rotation=45)
plt.show()
```