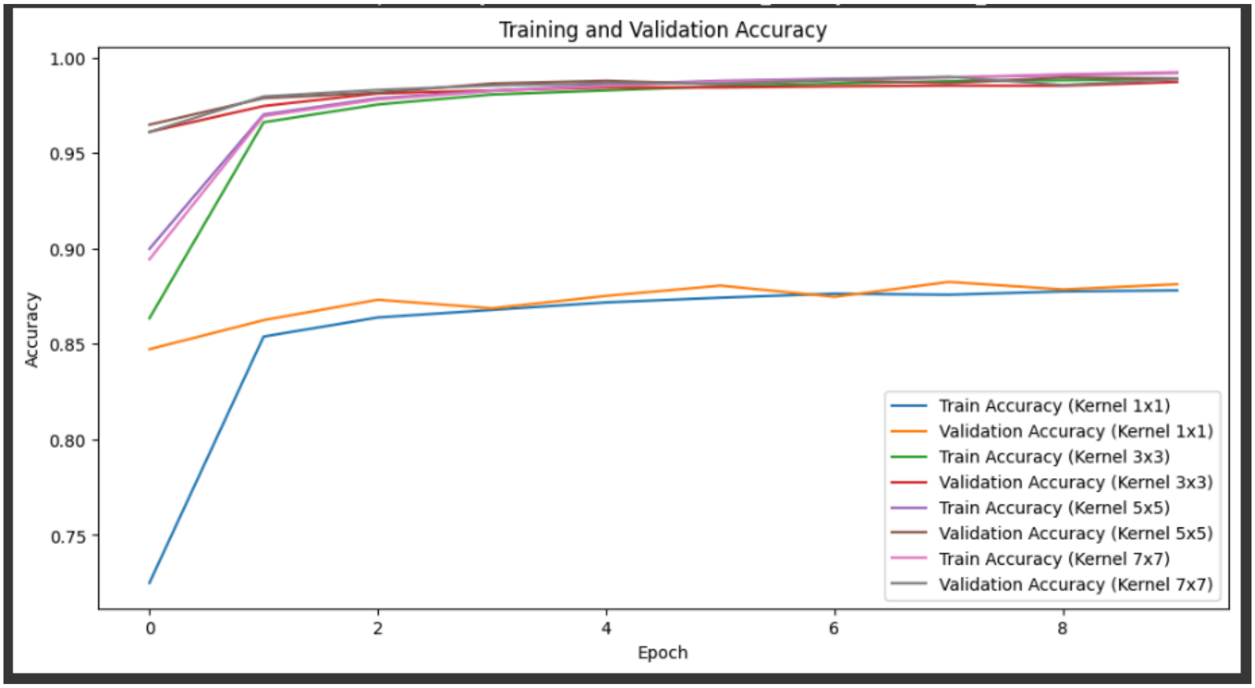
Prestin Bell
Patrick Hurley

Homework 6 (Implementing a CNN for the MNIST Dataset)

1. A larger kernel captures a larger field of view in the input. This could be beneficial for inputs where larger patterns are relevant but can be computationally more expensive to handle and keep and might need more data to prevent from overfitting the dataset. We can see that the higher kernel sizes lead to higher training and validation accuracy. It did not seem to affect the convergence though.

Suggested code may be subject to a license | codinghub.sellfy.store/p/designing-a-microscope-using-machine-learning/ | asifulhaque.com/tensorflow-basics-for-absolute-beginn

```python
# prompt: Train CNN models for the mnist dataset with 2 convolution layers and kernel sizes of 1x1, 3x3, 5x5, and 7x7 for

import tensorflow as tf
import matplotlib.pyplot as plt

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Preprocess the data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

# Define a function to create the CNN model with a specific kernel size
def create_cnn_model(kernel_size):
  model = tf.keras.models.Sequential([
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D((2, 2)),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.MaxPooling2D((2, 2)),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(10, activation='softmax')
  ])
  return model

# Define kernel sizes to test
kernel_sizes = [1, 3, 5, 7]

# Store training history for each model
histories = []

# Train models with different kernel sizes
for kernel_size in kernel_sizes:
  model = create_cnn_model(kernel_size)
  model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                loss='categorical_crossentropy',
                metrics=['accuracy'])

  history = model.fit(x_train, y_train, epochs=10, batch_size=256, validation_data=(x_test, y_test))
  histories.append(history)

# Plot the training and validation accuracy for each model
plt.figure(figsize=(12, 6))
for i, kernel_size in enumerate(kernel_sizes):
  plt.plot(histories[i].history['accuracy'], label=f'Train Accuracy (Kernel {kernel_size}x{kernel_size})')
  plt.plot(histories[i].history['val_accuracy'], label=f'Validation Accuracy (Kernel {kernel_size}x{kernel_size})')

plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



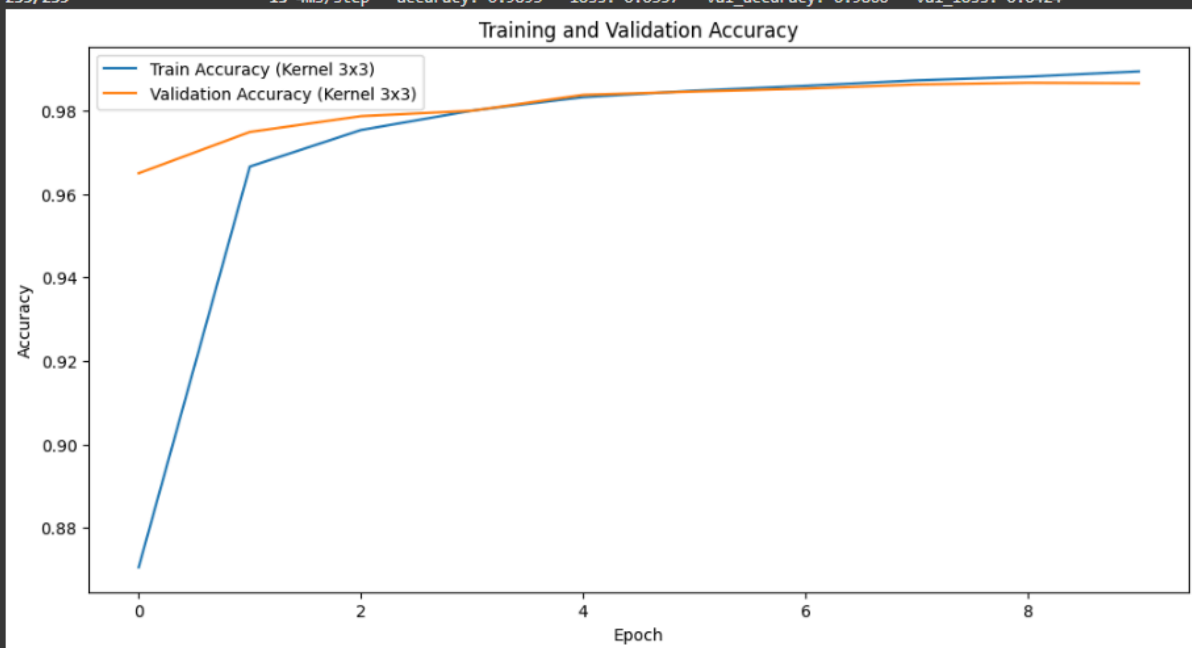
2. We use the same parameters as the previous question, for each number of convolution layers, and only use a kernel size of 3. Through our observations, we can see that 2 layers converged nicely. 6 layers converged really early and then split off right after. 12 layers also converged together, but it did not cleanly converge. the validation accuracy of 12 layers jumped above and below the training accuracy multiple times.

2 layers

Suggested code may be subject to a license | stackoverflow.com/questions/76006839/failed-to-convert-a-numpy-array-to-a-tensor-when-trying-to-train-the-image-mode | gndow

```
# prompt: Train CNN models for the mnist dataset with 2 convolution layers and kernel sizes of 1x1, 3x3, 5x5, and 7x7 for 1

import tensorflow as tf
import matplotlib.pyplot as plt

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Preprocess the data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

# Define a function to create the CNN model with a specific kernel size
def create_cnn_model(kernel_size):
  model = tf.keras.models.Sequential([
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D((2, 2)),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.MaxPooling2D((2, 2)),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(10, activation='softmax')
  ])
  return model

# Define kernel sizes to test
kernel_sizes = [3]

# Store training history for each model
histories = []

# Train models with different kernel sizes
for kernel_size in kernel_sizes:
  model = create_cnn_model(kernel_size)
  model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                loss='categorical_crossentropy',
                metrics=['accuracy'])

  history = model.fit(x_train, y_train, epochs=10, batch_size=256, validation_data=(x_test, y_test))
  histories.append(history)

# Plot the training and validation accuracy for each model
plt.figure(figsize=(12, 6))
for i, kernel_size in enumerate(kernel_sizes):
  plt.plot(histories[i].history['accuracy'], label=f'Train Accuracy (Kernel {kernel_size}x{kernel_size})')
  plt.plot(histories[i].history['val_accuracy'], label=f'Validation Accuracy (Kernel {kernel_size}x{kernel_size})')

plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

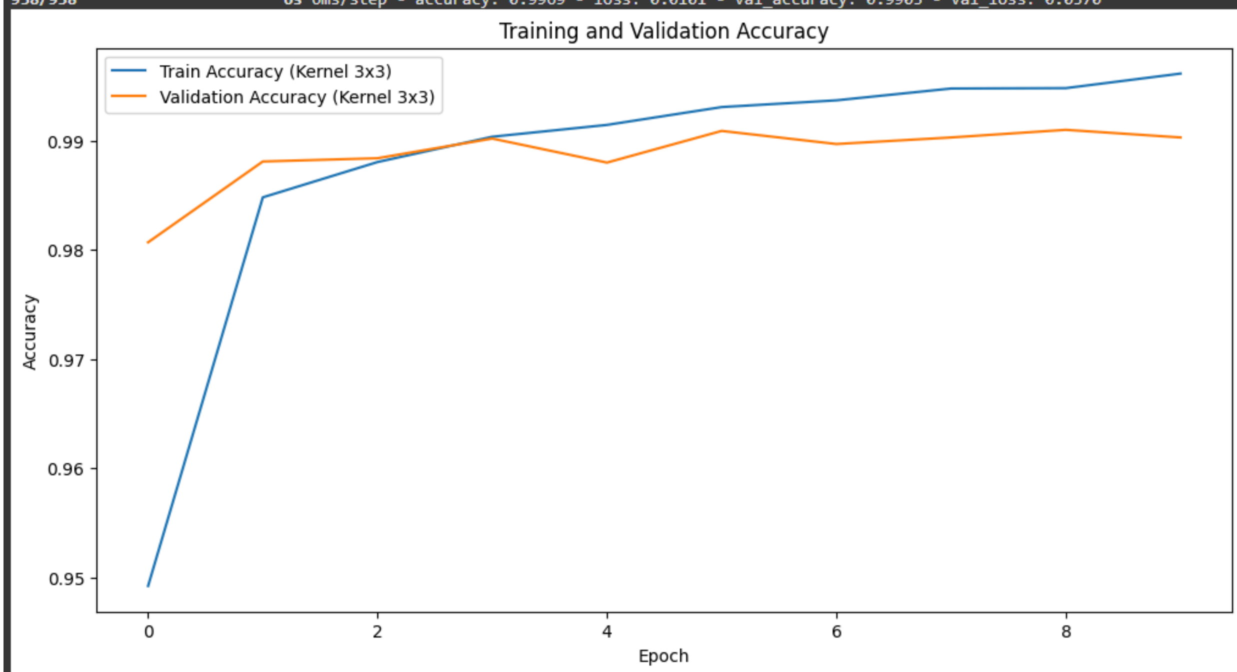
```
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.7354 - loss: 0.9788 - val_accuracy: 0.9650 - val_loss: 0.1242
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9644 - loss: 0.1237 - val_accuracy: 0.9749 - val_loss: 0.0830
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9740 - loss: 0.0866 - val_accuracy: 0.9787 - val_loss: 0.0672
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9801 - loss: 0.0668 - val_accuracy: 0.9800 - val_loss: 0.0636
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9829 - loss: 0.0583 - val_accuracy: 0.9838 - val_loss: 0.0504
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9851 - loss: 0.0513 - val_accuracy: 0.9846 - val_loss: 0.0462
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9865 - loss: 0.0450 - val_accuracy: 0.9853 - val_loss: 0.0471
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9870 - loss: 0.0432 - val_accuracy: 0.9863 - val_loss: 0.0398
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9878 - loss: 0.0397 - val_accuracy: 0.9867 - val_loss: 0.0383
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9893 - loss: 0.0357 - val_accuracy: 0.9866 - val_loss: 0.0424
```



6 layers

```python
import tensorflow as tf
import matplotlib.pyplot as plt

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Preprocess the data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

# Define a function to create the CNN model with a specific kernel size
def create_cnn_model(kernel_size):
  model = tf.keras.models.Sequential([
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.MaxPooling2D((2, 2)),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(10, activation='softmax')
  ])
  return model

# Define kernel sizes to test
kernel_sizes = [3]

# Store training history for each model
histories = []

# Train models with different kernel sizes
for kernel_size in kernel_sizes:
  model = create_cnn_model(kernel_size)
  model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                loss='categorical_crossentropy',
                metrics=['accuracy'])

  history = model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test, y_test))
  histories.append(history)

# Plot the training and validation accuracy for each model
plt.figure(figsize=(12, 6))
for i, kernel_size in enumerate(kernel_sizes):
  plt.plot(histories[i].history['accuracy'], label=f'Train Accuracy (Kernel {kernel_size}x{kernel_size})')
  plt.plot(histories[i].history['val_accuracy'], label=f'Validation Accuracy (Kernel {kernel_size}x{kernel_size})')

plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
Epoch 1/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.8785 - loss: 0.3792 - val_accuracy: 0.9807 - val_loss: 0.0544
Epoch 2/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 6s 7ms/step - accuracy: 0.9839 - loss: 0.0532 - val_accuracy: 0.9881 - val_loss: 0.0382
Epoch 3/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.9883 - loss: 0.0371 - val_accuracy: 0.9884 - val_loss: 0.0346
Epoch 4/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.9905 - loss: 0.0300 - val_accuracy: 0.9902 - val_loss: 0.0306
Epoch 5/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.9917 - loss: 0.0239 - val_accuracy: 0.9880 - val_loss: 0.0339
Epoch 6/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.9935 - loss: 0.0202 - val_accuracy: 0.9909 - val_loss: 0.0315
Epoch 7/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.9944 - loss: 0.0174 - val_accuracy: 0.9897 - val_loss: 0.0312
Epoch 8/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 6s 6ms/step - accuracy: 0.9958 - loss: 0.0130 - val_accuracy: 0.9903 - val_loss: 0.0346
Epoch 9/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.9955 - loss: 0.0129 - val_accuracy: 0.9910 - val_loss: 0.0311
Epoch 10/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 6s 6ms/step - accuracy: 0.9969 - loss: 0.0101 - val_accuracy: 0.9903 - val_loss: 0.0370
```

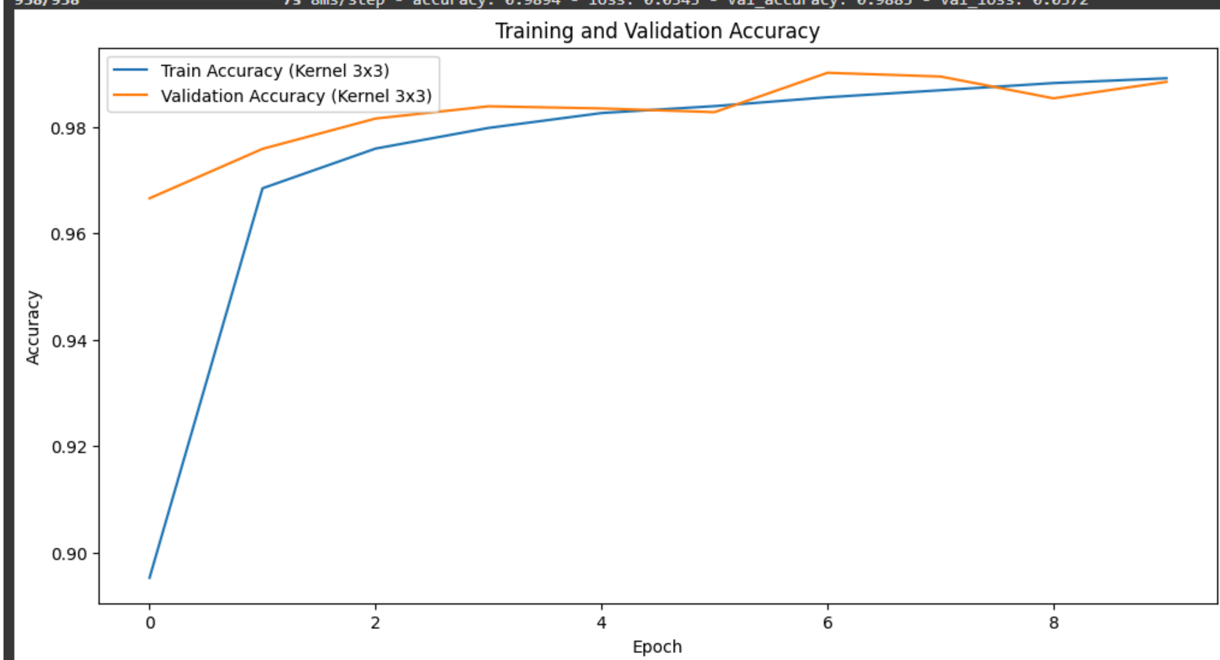


12 layers

Suggested code may be subject to a license | stackoverflow.com/questions/76006839/failed-to-convert-a-numpy-array-to-a-tensor-when-trying-to-train-the-image-mode | gndc

```
# prompt: Train CNN models for the mnist dataset with 2 convolution layers and kernel sizes of 1x1, 3x3, 5x5, and 7x7 for

import tensorflow as tf
import matplotlib.pyplot as plt

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Preprocess the data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

# Define a function to create the CNN model with a specific kernel size
def create_cnn_model(kernel_size):
  model = tf.keras.models.Sequential([
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.Conv2D(32, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.MaxPooling2D((2, 2)),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(10, activation='softmax')
  ])
  return model

# Define kernel sizes to test
kernel_sizes = [3]

# Store training history for each model
histories = []

# Train models with different kernel sizes
for kernel_size in kernel_sizes:
  model = create_cnn_model(kernel_size)
  model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                loss='categorical_crossentropy',
                metrics=['accuracy'])

  history = model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test, y_test))
  histories.append(history)

# Plot the training and validation accuracy for each model
plt.figure(figsize=(12, 6))
for i, kernel_size in enumerate(kernel_sizes):
  plt.plot(histories[i].history['accuracy'], label=f'Train Accuracy (Kernel {kernel_size}x{kernel_size})')
  plt.plot(histories[i].history['val_accuracy'], label=f'Validation Accuracy (Kernel {kernel_size}x{kernel_size})')
```

```
Epoch 1/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 16s 11ms/step - accuracy: 0.7853 - loss: 0.6472 - val_accuracy: 0.9666 - val_loss: 0.1051
Epoch 2/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 15s 8ms/step - accuracy: 0.9643 - loss: 0.1173 - val_accuracy: 0.9759 - val_loss: 0.0816
Epoch 3/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - accuracy: 0.9763 - loss: 0.0794 - val_accuracy: 0.9816 - val_loss: 0.0634
Epoch 4/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 11s 8ms/step - accuracy: 0.9796 - loss: 0.0678 - val_accuracy: 0.9839 - val_loss: 0.0507
Epoch 5/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 10s 8ms/step - accuracy: 0.9827 - loss: 0.0584 - val_accuracy: 0.9835 - val_loss: 0.0530
Epoch 6/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9836 - loss: 0.0524 - val_accuracy: 0.9828 - val_loss: 0.0534
Epoch 7/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 11s 8ms/step - accuracy: 0.9856 - loss: 0.0453 - val_accuracy: 0.9902 - val_loss: 0.0360
Epoch 8/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9872 - loss: 0.0406 - val_accuracy: 0.9895 - val_loss: 0.0370
Epoch 9/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - accuracy: 0.9884 - loss: 0.0382 - val_accuracy: 0.9854 - val_loss: 0.0480
Epoch 10/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - accuracy: 0.9894 - loss: 0.0343 - val_accuracy: 0.9885 - val_loss: 0.0372
```



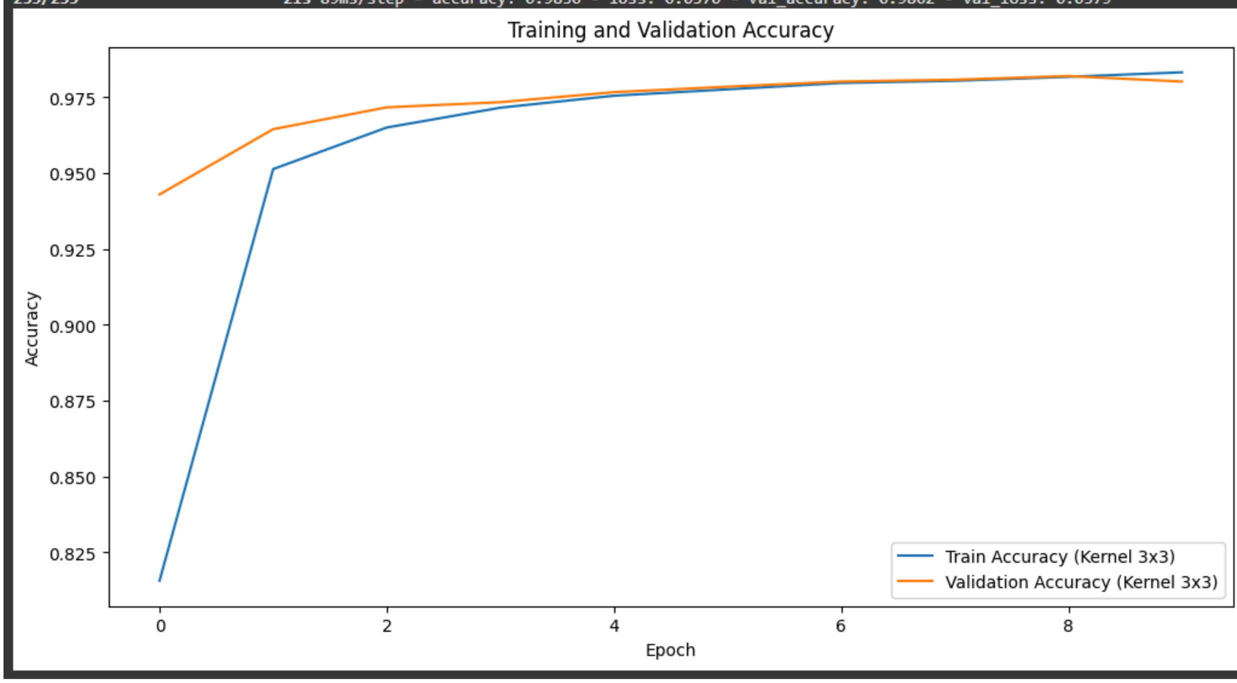
3. Below is a screen show of our architecture. We will first be using a filter of 16 then we will use 32, then 64, and finally 128.

Explanation:
Models with more filters can capture more features but are computationally more demanding. The increase in model accuracy might diminish after a certain point, even with more filters. The first few filter runs, consisting of the lower number of filters, converged nicely. Then after a point, the number of filters seemed to be too much and started to give bad results.

```python
# prompt: Train CNN models for the mnist dataset with 2 convolution layers and kernel sizes of 1x1, 3x3, 5x5, and 7x7 for

import tensorflow as tf
import matplotlib.pyplot as plt

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Preprocess the data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)

# Define a function to create the CNN model with a specific kernel size
def create_cnn_model(kernel_size):
  model = tf.keras.models.Sequential([
      tf.keras.layers.Conv2D(16, (kernel_size, kernel_size), activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D((2, 2)),
      tf.keras.layers.Conv2D(16, (kernel_size, kernel_size), activation='relu'),
      tf.keras.layers.MaxPooling2D((2, 2)),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(10, activation='softmax')
  ])
  return model

# Define kernel sizes to test
kernel_sizes = [3]

# Store training history for each model
histories = []

# Train models with different kernel sizes
for kernel_size in kernel_sizes:
  model = create_cnn_model(kernel_size)
  model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

  history = model.fit(x_train, y_train, epochs=10, batch_size=256, validation_data=(x_test, y_test))
  histories.append(history)

# Plot the training and validation accuracy for each model
plt.figure(figsize=(12, 6))
for i, kernel_size in enumerate(kernel_sizes):
  plt.plot(histories[i].history['accuracy'], label=f'Train Accuracy (Kernel {kernel_size}x{kernel_size})')
  plt.plot(histories[i].history['val_accuracy'], label=f'Validation Accuracy (Kernel {kernel_size}x{kernel_size})')

plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

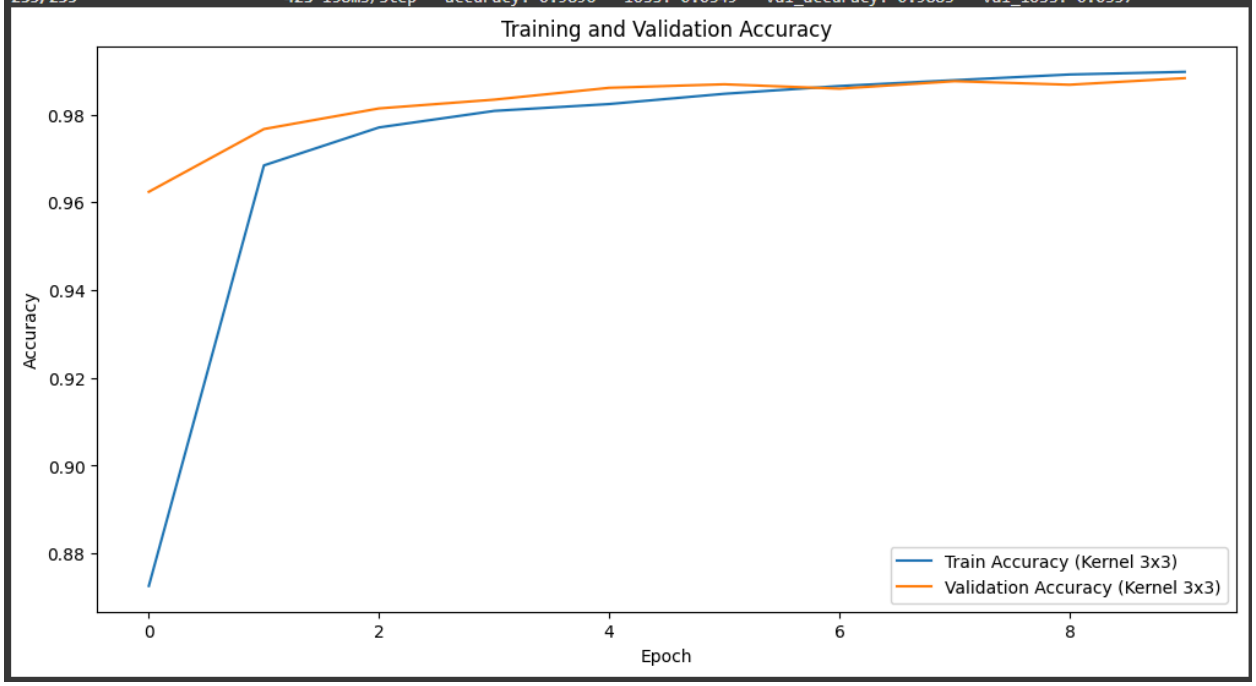
16 filter results

```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 22s 89ms/step - accuracy: 0.6348 - loss: 1.2274 - val_accuracy: 0.9430 - val_loss: 0.1991
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 21s 91ms/step - accuracy: 0.9447 - loss: 0.1899 - val_accuracy: 0.9645 - val_loss: 0.1216
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 23s 97ms/step - accuracy: 0.9629 - loss: 0.1271 - val_accuracy: 0.9717 - val_loss: 0.0934
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 40s 92ms/step - accuracy: 0.9700 - loss: 0.1019 - val_accuracy: 0.9734 - val_loss: 0.0825
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 42s 95ms/step - accuracy: 0.9751 - loss: 0.0847 - val_accuracy: 0.9767 - val_loss: 0.0732
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 46s 115ms/step - accuracy: 0.9769 - loss: 0.0775 - val_accuracy: 0.9785 - val_loss: 0.0690
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 36s 93ms/step - accuracy: 0.9793 - loss: 0.0689 - val_accuracy: 0.9802 - val_loss: 0.0607
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 23s 96ms/step - accuracy: 0.9797 - loss: 0.0660 - val_accuracy: 0.9808 - val_loss: 0.0601
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 42s 100ms/step - accuracy: 0.9808 - loss: 0.0611 - val_accuracy: 0.9820 - val_loss: 0.0553
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 21s 89ms/step - accuracy: 0.9830 - loss: 0.0570 - val_accuracy: 0.9802 - val_loss: 0.0579
```



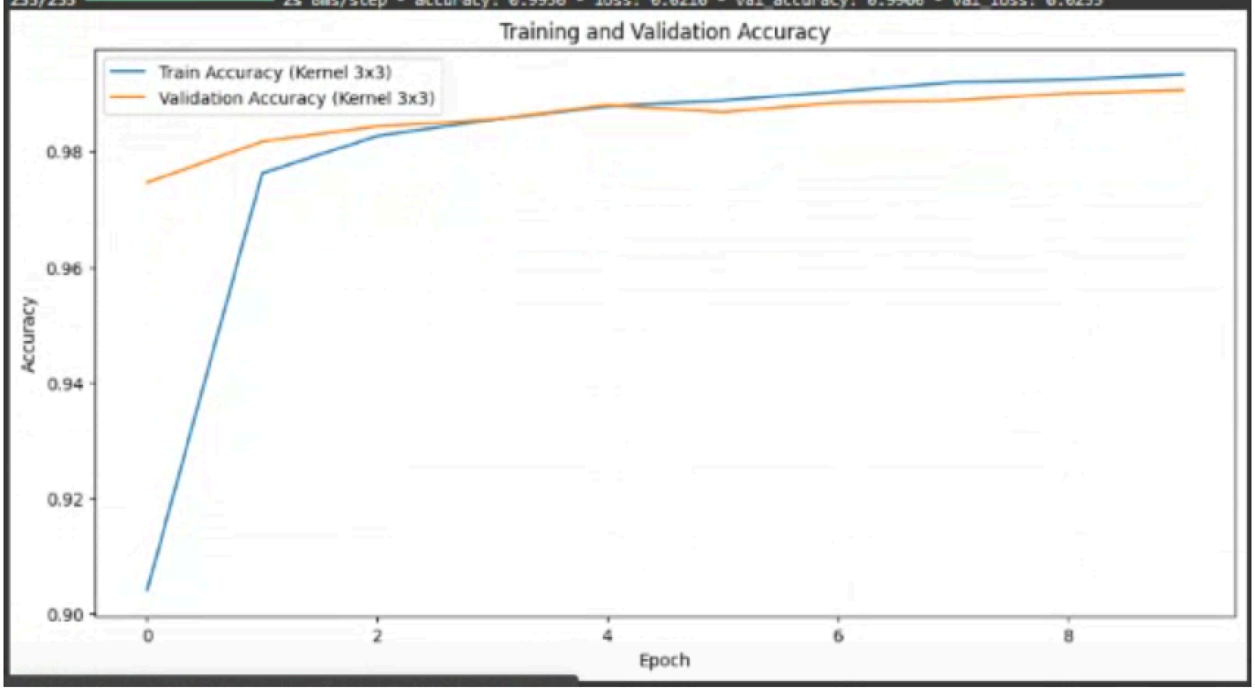
32 filter results

```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 38s 155ms/step - accuracy: 0.7339 - loss: 0.9817 - val_accuracy: 0.9624 - val_loss: 0.1260
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 39s 166ms/step - accuracy: 0.9656 - loss: 0.1171 - val_accuracy: 0.9767 - val_loss: 0.0762
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 40s 163ms/step - accuracy: 0.9769 - loss: 0.0775 - val_accuracy: 0.9814 - val_loss: 0.0614
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 42s 166ms/step - accuracy: 0.9797 - loss: 0.0660 - val_accuracy: 0.9834 - val_loss: 0.0521
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 39s 159ms/step - accuracy: 0.9816 - loss: 0.0558 - val_accuracy: 0.9861 - val_loss: 0.0468
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 38s 148ms/step - accuracy: 0.9842 - loss: 0.0503 - val_accuracy: 0.9869 - val_loss: 0.0428
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 43s 156ms/step - accuracy: 0.9869 - loss: 0.0440 - val_accuracy: 0.9859 - val_loss: 0.0433
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 35s 147ms/step - accuracy: 0.9872 - loss: 0.0415 - val_accuracy: 0.9876 - val_loss: 0.0387
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 42s 153ms/step - accuracy: 0.9892 - loss: 0.0348 - val_accuracy: 0.9868 - val_loss: 0.0399
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 42s 158ms/step - accuracy: 0.9896 - loss: 0.0349 - val_accuracy: 0.9883 - val_loss: 0.0357
```



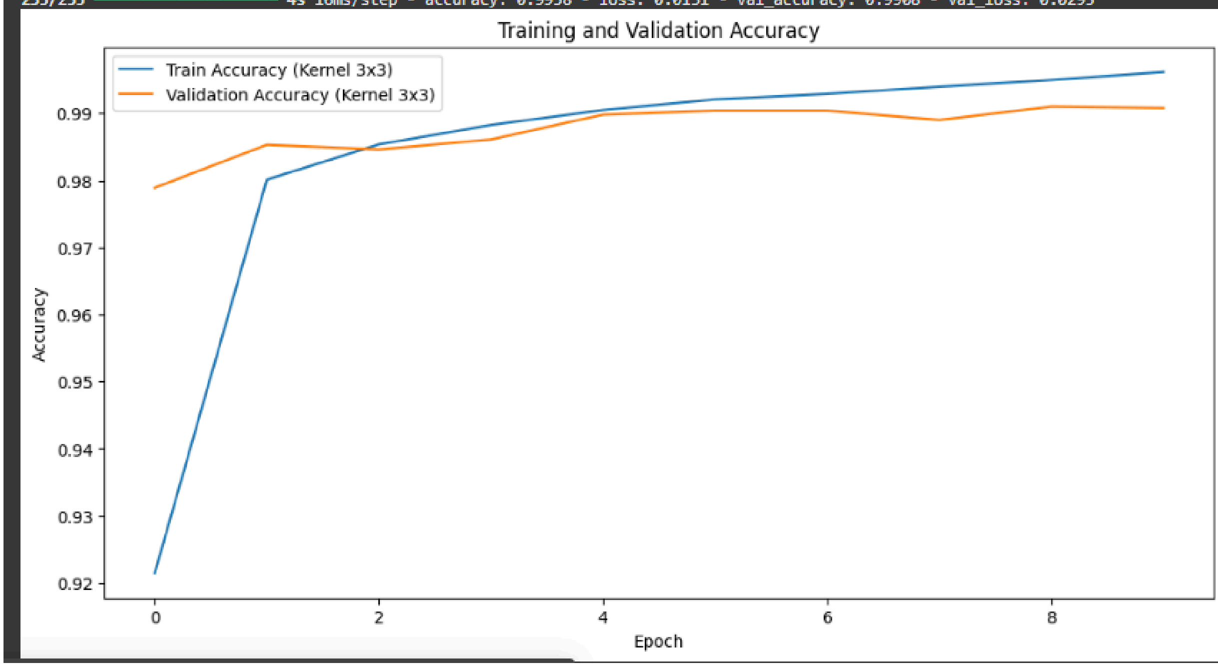
64 filter results

```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 16ms/step - accuracy: 0.7854 - loss: 0.7735 - val_accuracy: 0.9746 - val_loss: 0.0855
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9731 - loss: 0.0936 - val_accuracy: 0.9817 - val_loss: 0.0604
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9822 - loss: 0.0592 - val_accuracy: 0.9843 - val_loss: 0.0464
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.9859 - loss: 0.0489 - val_accuracy: 0.9855 - val_loss: 0.0399
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9878 - loss: 0.0394 - val_accuracy: 0.9880 - val_loss: 0.0365
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9884 - loss: 0.0361 - val_accuracy: 0.9868 - val_loss: 0.0373
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9902 - loss: 0.0305 - val_accuracy: 0.9884 - val_loss: 0.0332
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9921 - loss: 0.0263 - val_accuracy: 0.9888 - val_loss: 0.0324
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9925 - loss: 0.0244 - val_accuracy: 0.9900 - val_loss: 0.0286
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9936 - loss: 0.0210 - val_accuracy: 0.9906 - val_loss: 0.0255
```



128 filter results

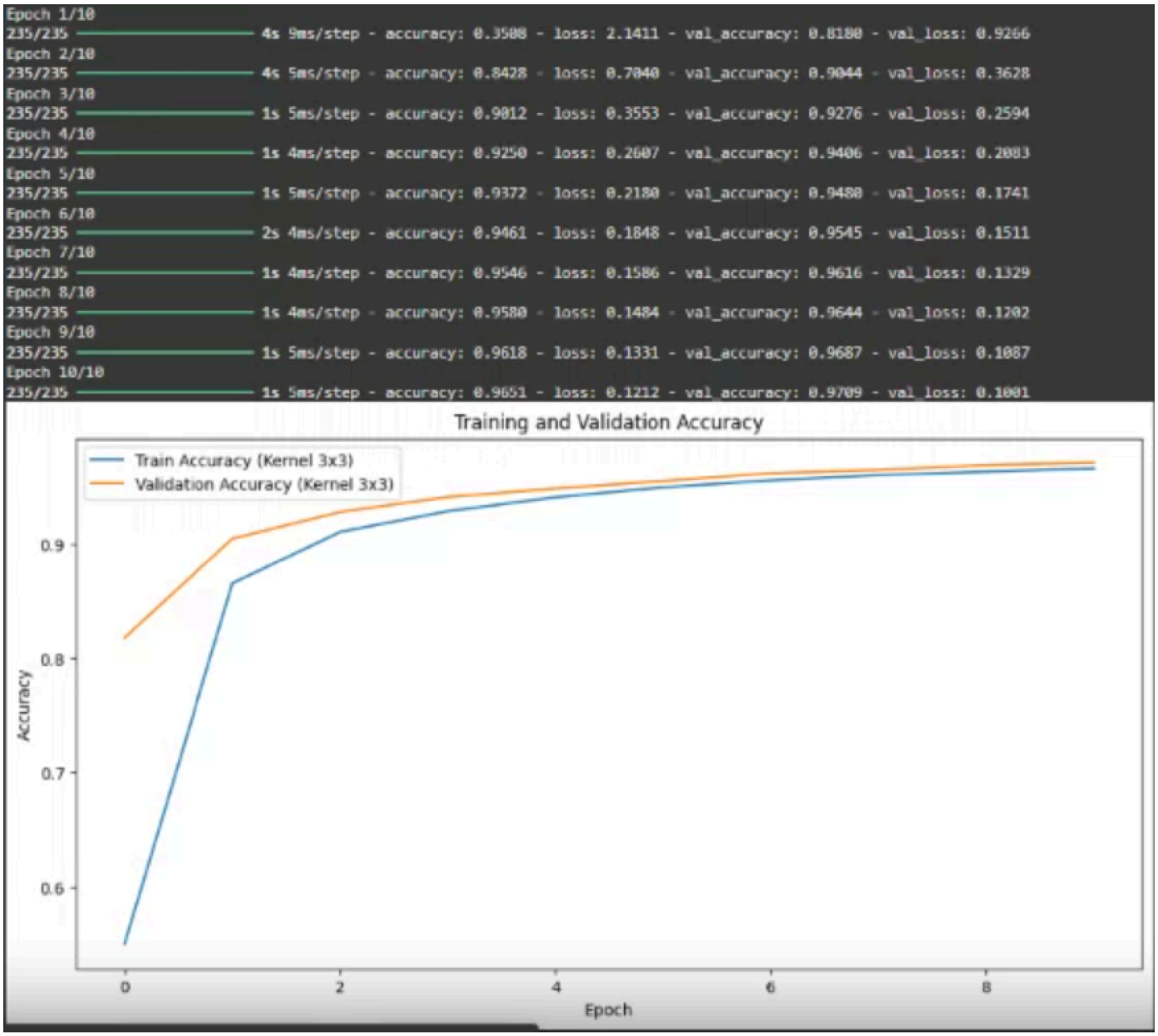
```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 25ms/step - accuracy: 0.8151 - loss: 0.6314 - val_accuracy: 0.9789 - val_loss: 0.0671
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 7s 15ms/step - accuracy: 0.9779 - loss: 0.0722 - val_accuracy: 0.9853 - val_loss: 0.0444
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - accuracy: 0.9855 - loss: 0.0488 - val_accuracy: 0.9846 - val_loss: 0.0453
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 15ms/step - accuracy: 0.9885 - loss: 0.0384 - val_accuracy: 0.9861 - val_loss: 0.0437
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 14ms/step - accuracy: 0.9894 - loss: 0.0362 - val_accuracy: 0.9898 - val_loss: 0.0313
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - accuracy: 0.9929 - loss: 0.0235 - val_accuracy: 0.9904 - val_loss: 0.0308
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 15ms/step - accuracy: 0.9936 - loss: 0.0215 - val_accuracy: 0.9904 - val_loss: 0.0274
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.9947 - loss: 0.0180 - val_accuracy: 0.9890 - val_loss: 0.0344
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 16ms/step - accuracy: 0.9958 - loss: 0.0146 - val_accuracy: 0.9910 - val_loss: 0.0308
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.9958 - loss: 0.0131 - val_accuracy: 0.9908 - val_loss: 0.0293
```



4. below is a screenshot of the model we used to try the different learning rates. We use kernel sizes of 3, filter size of 32, batch size of 256, and 10 epochs.

Observation: The learning rate controls the step size during optimization. A good learning rate allows the model to converge efficiently. As is shown in a learning rate of 0.0001, it looks like the learning rate is too low and it won't even converge at all.

The first learning rate we used was 0.0001



The second learning rate we used is 0.001

```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 15ms/step - accuracy: 0.7321 - loss: 0.9862 - val_accuracy: 0.9635 - val_loss: 0.1276
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9632 - loss: 0.1250 - val_accuracy: 0.9740 - val_loss: 0.0851
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9750 - loss: 0.0848 - val_accuracy: 0.9798 - val_loss: 0.0627
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9791 - loss: 0.0669 - val_accuracy: 0.9814 - val_loss: 0.0564
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9830 - loss: 0.0550 - val_accuracy: 0.9842 - val_loss: 0.0477
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9843 - loss: 0.0517 - val_accuracy: 0.9849 - val_loss: 0.0478
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9865 - loss: 0.0459 - val_accuracy: 0.9859 - val_loss: 0.0408
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9869 - loss: 0.0420 - val_accuracy: 0.9869 - val_loss: 0.0388
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9882 - loss: 0.0385 - val_accuracy: 0.9859 - val_loss: 0.0406
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9902 - loss: 0.0342 - val_accuracy: 0.9883 - val_loss: 0.0389
```

The third learning rate we used is 0.01

```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.8756 - loss: 0.3838 - val_accuracy: 0.9842 - val_loss: 0.0446
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9841 - loss: 0.0526 - val_accuracy: 0.9882 - val_loss: 0.0378
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9888 - loss: 0.0366 - val_accuracy: 0.9891 - val_loss: 0.0343
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9892 - loss: 0.0336 - val_accuracy: 0.9899 - val_loss: 0.0333
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9917 - loss: 0.0263 - val_accuracy: 0.9874 - val_loss: 0.0397
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9928 - loss: 0.0216 - val_accuracy: 0.9856 - val_loss: 0.0485
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9936 - loss: 0.0206 - val_accuracy: 0.9882 - val_loss: 0.0379
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9948 - loss: 0.0151 - val_accuracy: 0.9874 - val_loss: 0.0439
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9944 - loss: 0.0182 - val_accuracy: 0.9872 - val_loss: 0.0451
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9950 - loss: 0.0147 - val_accuracy: 0.9862 - val_loss: 0.0509
```

The fourth learning rate we used is 0.1

```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.7362 - loss: 2.3470 - val_accuracy: 0.9201 - val_loss: 0.2524
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9240 - loss: 0.2533 - val_accuracy: 0.9280 - val_loss: 0.2501
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9262 - loss: 0.2436 - val_accuracy: 0.9353 - val_loss: 0.2066
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9274 - loss: 0.2379 - val_accuracy: 0.9144 - val_loss: 0.2674
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9411 - loss: 0.2002 - val_accuracy: 0.9447 - val_loss: 0.1815
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9386 - loss: 0.2040 - val_accuracy: 0.9460 - val_loss: 0.1815
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9405 - loss: 0.2052 - val_accuracy: 0.9554 - val_loss: 0.1496
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9430 - loss: 0.1932 - val_accuracy: 0.9484 - val_loss: 0.1824
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9461 - loss: 0.1816 - val_accuracy: 0.9279 - val_loss: 0.2183
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9443 - loss: 0.1883 - val_accuracy: 0.9509 - val_loss: 0.1584
```