Submitted By: Mrinal Bhan (DSAI : 211020428)

Task 1. To implement a simple feed-forward neural network using keras library. The neural network should have the following specifications:

1. Input Layer: The input layer should match the dimension of your dataset.
2. Hidden Layers: Include at least one hidden layer with a reasonable number of neurons.
3. Output Layer: Design the output layer based on your chosen problem (classification or regression). Train the neural network on a dataset of your choice. You can use publicly available datasets or create a synthetic dataset.

```
import tensorflow as tf
from tensorflow.keras.layers import Flatten, Dense # For miscellaneous functions
from tensorflow.keras import utils # For datasets
from tensorflow.keras.datasets import mnist # For math functions and array
import numpy as np
import matplotlib.pyplot as plt
```

```
#Loading the MNIST dataset
(train_X, train_Y), (test_X, test_Y) = mnist.load_data()
train_Y_categorical = utils.to_categorical(train_Y)
test_Y_categorical = utils.to_categorical(test_Y)

print("Training data shape: ", train_X.shape)
print("Training labels shape: ", train_Y.shape)
print("Test data shape: ", test_X.shape)
print("Test labels shape: ", test_Y.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/
11490434/11490434 [==============================] - 0s 0us/step
Training data shape:  (60000, 28, 28)
Training labels shape:  (60000,)
Test data shape:  (10000, 28, 28)
Test labels shape:  (10000,)
```

```
features = train_X.shape[1]
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)), #Input layer with 28 neurons i.e
     tf.keras.layers.Dense(16, activation='relu'),  #Hidden layer with 16 neurons
    tf.keras.layers.Dense(10, activation='softmax')
])
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
```

```
 flatten_5 (Flatten)         (None, 784)               0

 dense_9 (Dense)             (None, 16)                12560

 dense_10 (Dense)            (None, 10)                170

=================================================================
Total params: 12,730
Trainable params: 12,730
Non-trainable params: 0
_________________________________________________________________
```

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy
```

```
history = model.fit(train_X, train_Y_categorical, epochs=10, validation_split=0.33)
```

```
Epoch 1/10
1257/1257 [==============================] - 5s 4ms/step - loss: 0.4947 - accuracy
Epoch 2/10
1257/1257 [==============================] - 4s 3ms/step - loss: 0.4495 - accuracy
Epoch 3/10
1257/1257 [==============================] - 5s 4ms/step - loss: 0.4176 - accuracy
Epoch 4/10
1257/1257 [==============================] - 4s 3ms/step - loss: 0.4083 - accuracy
Epoch 5/10
1257/1257 [==============================] - 4s 3ms/step - loss: 0.3940 - accuracy
Epoch 6/10
1257/1257 [==============================] - 8s 6ms/step - loss: 0.3869 - accuracy
Epoch 7/10
1257/1257 [==============================] - 4s 3ms/step - loss: 0.3782 - accuracy
Epoch 8/10
1257/1257 [==============================] - 4s 3ms/step - loss: 0.3754 - accuracy
Epoch 9/10
1257/1257 [==============================] - 5s 4ms/step - loss: 0.3720 - accuracy
Epoch 10/10
1257/1257 [==============================] - 4s 3ms/step - loss: 0.3697 - accuracy
```

```
print(history.history.keys())
```

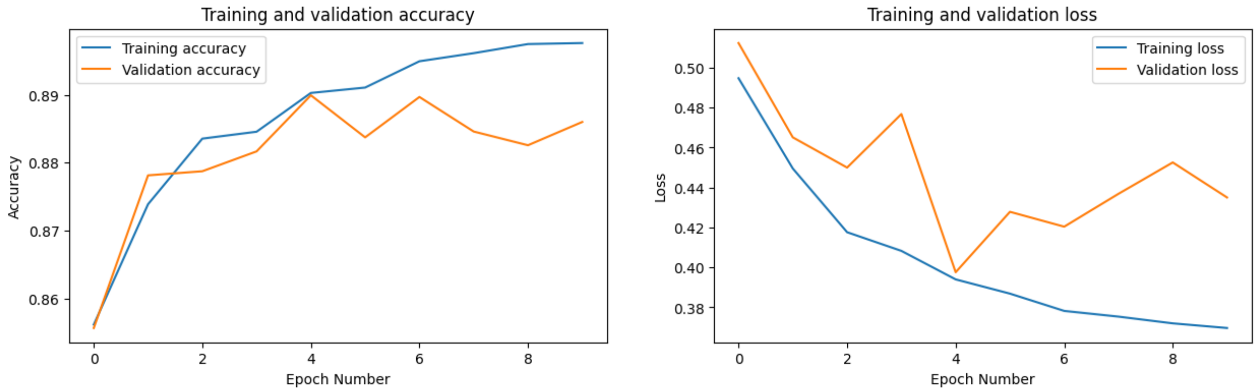
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

```
fig = plt.figure(figsize=(15,4))

fig.add_subplot(121)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['Training accuracy', 'Validation accuracy'])
plt.title('Training and validation accuracy')
plt.xlabel('Epoch Number')
plt.ylabel('Accuracy')

fig.add_subplot(122)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['Training loss', 'Validation loss'])
```

```
plt.title('Training and validation loss')
plt.xlabel('Epoch Number')
plt.ylabel('Loss')
plt.show()
```




```
predict = model.predict(test_X[:1,:,:])
print('Predict shape: ', predict.shape)
print('Prediction for first test image: \n', predict[0])
print('Classification of the first test image: digit ', np.argmax(predict[0]))
```

```
1/1 [==============================] - 0s 61ms/step
Predict shape:  (1, 10)
Prediction for first test image: 
 [3.3607513e-08 1.1154112e-04 2.5045469e-02 6.1610201e-03 7.2616564e-07
 6.9169505e-06 4.3071849e-14 9.6867406e-01 1.9640513e-08 3.9236642e-08]
Classification of the first test image: digit  7
```

```
train_loss, train_acc = model.evaluate(train_X,  train_Y_categorical)
test_loss, test_acc = model.evaluate(test_X,  test_Y_categorical)
print('Classification accuracy on training set: ', train_acc)
print('Classification accuracy on test set: ', test_acc)
```

```
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3831 - accuracy
313/313 [==============================] - 1s 2ms/step - loss: 0.4246 - accuracy:
Classification accuracy on training set:  0.895716667175293
Classification accuracy on test set:  0.8912000060081482
```

```
test_predict = model.predict(test_X)
# Get the classification labels
test_predict_labels = np.argmax(test_predict, axis=1)
confusion_matrix = tf.math.confusion_matrix(labels=test_Y, predictions=test_predict
print('Confusion matrix of the test set:\n', confusion_matrix)
```

```
313/313 [==============================] - 1s 2ms/step
```

```
Confusion matrix of the test set:
 tf.Tensor(
[[ 953    0    2    0    2    1    9    5    8    0]
 [   0 1098    2    6    1    2    1    5   20    0]
 [  10    0  938    4    7    2   16    7   45    3]
 [   1    0   27  899    1   42    4    7   25    4]
 [   1    1    3    0  929    2   12    2    5   27]
 [  45    2    5  151   11  599   16    6   53    4]
 [  28    0    4    0   13    9  885    0   19    0]
 [   2    5   26   20    9    0    0  906    7   53]
 [  24    2   11   10   14   23   23   10  845   12]
 [   4    8    1   22   61    9    0   38    6  860]], shape=(10, 10), dtype=int32
```

Visualizing Hidden Layer Outputs

```
flat_layer = model.layers[0]
hidden_layer = model.layers[1]
output_layer = model.layers[2]
```

```
def get_hidden_layer_output(model, X):
    # Convert X to a tensor
    x = tf.convert_to_tensor(np.reshape(X, (1, 28, 28)),
                             dtype=tf.dtypes.float32)# Model layers
    flat_layer = model.layers[0]
    hidden_layer = model.layers[1]
    output_layer = model.layers[2]# Pass x through different layers
    flat_tensor = flat_layer(x)
    hidden_tensor = hidden_layer(flat_tensor)
    output_tensor = output_layer(hidden_tensor)
    predicted_digit = np.argmax(output_tensor)
    return hidden_tensor, predicted_digit
```

```
total_cols = 4
fig, ax = plt.subplots(nrows=2, ncols=total_cols,
                      figsize=(18,4),
                      subplot_kw=dict(xticks=[], yticks=[]))

for j in range(total_cols):
    image = test_X[j, :, :]
    h, prediction = get_hidden_layer_output(model, image)
    ax[0, j].imshow(image)
    ax[1, j].imshow(np.reshape(h.numpy(), (4,4)))
    ax[0, j].set_title('Digit ' + str(test_Y[j]))
    ax[1, j].set_title('Classification ' + str(prediction))
plt.title('MNIST Digits and Their Hidden Layer Representation', y=-0.4, x=-4)
plt.show()
```

MNIST Digits and Their Hidden Layer Representation

Task 2: Implement dropout regularization within your previously created feed-forward NN. Experiment with different dropout rates and observe the effect on training and validation accuracy/loss.

```
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
```

```
#Loading the MNIST dataset
(train_X, train_Y), (test_X, test_Y) = mnist.load_data()
train_X, test_X = train_X / 255.0, test_X / 255.0
```

```
# Define different dropout rates to experiment with
dropout_rates = [0.0, 0.25, 0.5]

# Loop through different dropout rates and train the models
for rate in dropout_rates:
    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    print(f"Training model with dropout rate: {rate}")
    history = model.fit(train_X, train_Y, epochs=10, validation_data=(test_X, test_

    # Evaluate the model
    test_loss, test_acc = model.evaluate(test_X, test_Y, verbose=2)
    print(f"Test accuracy with dropout rate {rate}: {test_acc}")

```

```
Training model with dropout rate: 0.0
Epoch 1/10
1875/1875 - 6s - loss: 0.6348 - accuracy: 0.8100 - val_loss: 55.4354 - val_accurac
Epoch 2/10
1875/1875 - 4s - loss: 0.3104 - accuracy: 0.9110 - val_loss: 54.4254 - val_accurac
Epoch 3/10
1875/1875 - 4s - loss: 0.2802 - accuracy: 0.9195 - val_loss: 52.7971 - val_accurac
Epoch 4/10
1875/1875 - 5s - loss: 0.2645 - accuracy: 0.9235 - val_loss: 55.5008 - val_accurac
```

```
Epoch 5/10
1875/1875 - 3s - loss: 0.2545 - accuracy: 0.9261 - val_loss: 55.1760 - val_accurac
Epoch 6/10
1875/1875 - 5s - loss: 0.2466 - accuracy: 0.9282 - val_loss: 59.4062 - val_accurac
Epoch 7/10
1875/1875 - 5s - loss: 0.2399 - accuracy: 0.9311 - val_loss: 53.8096 - val_accurac
Epoch 8/10
1875/1875 - 4s - loss: 0.2359 - accuracy: 0.9322 - val_loss: 60.9512 - val_accurac
Epoch 9/10
1875/1875 - 4s - loss: 0.2315 - accuracy: 0.9324 - val_loss: 62.1712 - val_accurac
Epoch 10/10
1875/1875 - 5s - loss: 0.2272 - accuracy: 0.9341 - val_loss: 69.0469 - val_accurac
313/313 - 1s - loss: 0.2583 - accuracy: 0.9247 - 573ms/epoch - 2ms/step
Test accuracy with dropout rate 0.0: 0.9247000217437744
Training model with dropout rate: 0.25
Epoch 1/10
1875/1875 - 4s - loss: 0.2246 - accuracy: 0.9351 - val_loss: 0.2453 - val_accuracy
Epoch 2/10
1875/1875 - 4s - loss: 0.2213 - accuracy: 0.9359 - val_loss: 0.2370 - val_accuracy
Epoch 3/10
1875/1875 - 5s - loss: 0.2187 - accuracy: 0.9369 - val_loss: 0.2458 - val_accuracy
Epoch 4/10
1875/1875 - 4s - loss: 0.2163 - accuracy: 0.9380 - val_loss: 0.2380 - val_accuracy
Epoch 5/10
1875/1875 - 4s - loss: 0.2135 - accuracy: 0.9384 - val_loss: 0.2420 - val_accuracy
Epoch 6/10
1875/1875 - 5s - loss: 0.2120 - accuracy: 0.9390 - val_loss: 0.2373 - val_accuracy
Epoch 7/10
1875/1875 - 4s - loss: 0.2098 - accuracy: 0.9397 - val_loss: 0.2344 - val_accuracy
Epoch 8/10
1875/1875 - 4s - loss: 0.2082 - accuracy: 0.9399 - val_loss: 0.2414 - val_accuracy
Epoch 9/10
1875/1875 - 5s - loss: 0.2069 - accuracy: 0.9402 - val_loss: 0.2372 - val_accuracy
Epoch 10/10
1875/1875 - 4s - loss: 0.2052 - accuracy: 0.9409 - val_loss: 0.2402 - val_accuracy
313/313 - 0s - loss: 0.2402 - accuracy: 0.9329 - 444ms/epoch - 1ms/step
Test accuracy with dropout rate 0.25: 0.9329000115394592
Training model with dropout rate: 0.5
Epoch 1/10
1875/1875 - 5s - loss: 0.2036 - accuracy: 0.9419 - val_loss: 0.2343 - val_accuracy
Epoch 2/10
1875/1875 - 4s - loss: 0.2023 - accuracy: 0.9421 - val_loss: 0.2430 - val_accuracy
Epoch 3/10
1875/1875 - 4s - loss: 0.2015 - accuracy: 0.9422 - val_loss: 0.2323 - val_accuracy
Epoch 4/10
1875/1875 - 4s - loss: 0.1994 - accuracy: 0.9424 - val_loss: 0.2404 - val_accuracy
Epoch 5/10
1875/1875 - 4s - loss: 0.1988 - accuracy: 0.9431 - val_loss: 0.2375 - val_accuracy
Epoch 6/10
```

```
# Define different dropout rates to experiment with
dropout_rates = [0.4, 0.45, 0.55]

# Loop through different dropout rates and train the models
for rate in dropout_rates:
    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
```

```python

    print(f"Training model with dropout rate: {rate}")
    history = model.fit(train_X, train_Y, epochs=10, validation_data=(test_X, test_Y)

    # Evaluate the model
    test_loss, test_acc = model.evaluate(test_X, test_Y, verbose=2)
    print(f"Test accuracy with dropout rate {rate}: {test_acc}")

```

```
Training model with dropout rate: 0.4
Epoch 1/10
1875/1875 - 6s - loss: 0.1917 - accuracy: 0.9442 - val_loss: 0.2357 - val_accuracy
Epoch 2/10
1875/1875 - 4s - loss: 0.1911 - accuracy: 0.9443 - val_loss: 0.2379 - val_accuracy
Epoch 3/10
1875/1875 - 4s - loss: 0.1905 - accuracy: 0.9450 - val_loss: 0.2380 - val_accuracy
Epoch 4/10
1875/1875 - 5s - loss: 0.1890 - accuracy: 0.9449 - val_loss: 0.2379 - val_accuracy
Epoch 5/10
1875/1875 - 4s - loss: 0.1886 - accuracy: 0.9455 - val_loss: 0.2394 - val_accuracy
Epoch 6/10
1875/1875 - 4s - loss: 0.1873 - accuracy: 0.9453 - val_loss: 0.2346 - val_accuracy
Epoch 7/10
1875/1875 - 5s - loss: 0.1860 - accuracy: 0.9454 - val_loss: 0.2411 - val_accuracy
Epoch 8/10
1875/1875 - 4s - loss: 0.1855 - accuracy: 0.9460 - val_loss: 0.2422 - val_accuracy
Epoch 9/10
1875/1875 - 3s - loss: 0.1853 - accuracy: 0.9460 - val_loss: 0.2354 - val_accuracy
Epoch 10/10
1875/1875 - 5s - loss: 0.1847 - accuracy: 0.9463 - val_loss: 0.2375 - val_accuracy
313/313 - 0s - loss: 0.2375 - accuracy: 0.9365 - 429ms/epoch - 1ms/step
Test accuracy with dropout rate 0.4: 0.9365000128746033
Training model with dropout rate: 0.45
Epoch 1/10
1875/1875 - 5s - loss: 0.1842 - accuracy: 0.9462 - val_loss: 0.2410 - val_accuracy
Epoch 2/10
1875/1875 - 5s - loss: 0.1827 - accuracy: 0.9467 - val_loss: 0.2388 - val_accuracy
Epoch 3/10
1875/1875 - 4s - loss: 0.1826 - accuracy: 0.9466 - val_loss: 0.2450 - val_accuracy
Epoch 4/10
1875/1875 - 4s - loss: 0.1814 - accuracy: 0.9469 - val_loss: 0.2446 - val_accuracy
Epoch 5/10
1875/1875 - 5s - loss: 0.1810 - accuracy: 0.9460 - val_loss: 0.2428 - val_accuracy
Epoch 6/10
1875/1875 - 4s - loss: 0.1815 - accuracy: 0.9472 - val_loss: 0.2408 - val_accuracy
Epoch 7/10
1875/1875 - 4s - loss: 0.1808 - accuracy: 0.9475 - val_loss: 0.2364 - val_accuracy
Epoch 8/10
1875/1875 - 5s - loss: 0.1799 - accuracy: 0.9478 - val_loss: 0.2370 - val_accuracy
Epoch 9/10
1875/1875 - 4s - loss: 0.1796 - accuracy: 0.9470 - val_loss: 0.2396 - val_accuracy
Epoch 10/10
1875/1875 - 4s - loss: 0.1790 - accuracy: 0.9473 - val_loss: 0.2479 - val_accuracy
313/313 - 1s - loss: 0.2479 - accuracy: 0.9324 - 696ms/epoch - 2ms/step
Test accuracy with dropout rate 0.45: 0.9323999881744385
Training model with dropout rate: 0.55
Epoch 1/10
1875/1875 - 5s - loss: 0.1788 - accuracy: 0.9475 - val_loss: 0.2387 - val_accuracy
```

```
Epoch 2/10
1875/1875 - 3s - loss: 0.1785 - accuracy: 0.9476 - val_loss: 0.2377 - val_accuracy
Epoch 3/10
1875/1875 - 5s - loss: 0.1783 - accuracy: 0.9475 - val_loss: 0.2412 - val_accuracy
Epoch 4/10
1875/1875 - 4s - loss: 0.1775 - accuracy: 0.9480 - val_loss: 0.2398 - val_accuracy
Epoch 5/10
1875/1875 - 3s - loss: 0.1764 - accuracy: 0.9477 - val_loss: 0.2467 - val_accuracy
Epoch 6/10
```

Task 3: Modify your feed-forward neural network to include batch normalization layers after each hidden layer.

1. Train the network using the same dataset and track its performance.
2. Compare the convergence speed and final accuracy/loss with and without batch normalization.

```
from tensorflow.keras.layers import Dense, BatchNormalization

# Define the neural network architecture with batch normalization
def build_model_with_batch_norm():
    model = Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(128, activation='relu'),
        BatchNormalization(),  # Batch normalization layer
        tf.keras.layers.Dense(64, activation='relu'),
        BatchNormalization(),  # Batch normalization layer
        tf.keras.layers.Dense(10, activation='softmax')
    ])
    return model

# Define the neural network architecture without batch normalization
def build_model_without_batch_norm():
    model = Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(16, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
    return model

# Compile and train the models
def train_and_evaluate(model, train_X, train_Y, test_X, test_Y):
    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    history = model.fit(train_X, train_Y, epochs=10, validation_data=(test_X, test_

    # Evaluate the model
    test_loss, test_acc = model.evaluate(test_X, test_Y, verbose=2)
    return history, test_loss, test_acc

# Train and evaluate models with and without batch normalization
```

```
# Train and evaluate models with and without batch normalization
model_with_bn = build_model_with_batch_norm()
history_with_bn, test_loss_with_bn, test_acc_with_bn = train_and_evaluate(model_wit

model_without_bn = build_model_without_batch_norm()
history_without_bn, test_loss_without_bn, test_acc_without_bn = train_and_evaluate(

# Compare convergence speed and final accuracy/loss
print("With Batch Normalization:")
print(f"Test accuracy: {test_acc_with_bn}")
print("Without Batch Normalization:")
print(f"Test accuracy: {test_acc_without_bn}")
```

```
Epoch 1/10
1875/1875 - 12s - loss: 0.2486 - accuracy: 0.9251 - val_loss: 0.1287 - val_accurac
Epoch 2/10
1875/1875 - 9s - loss: 0.1238 - accuracy: 0.9621 - val_loss: 0.0978 - val_accuracy
Epoch 3/10
1875/1875 - 9s - loss: 0.0961 - accuracy: 0.9701 - val_loss: 0.0892 - val_accuracy
Epoch 4/10
1875/1875 - 8s - loss: 0.0798 - accuracy: 0.9754 - val_loss: 0.0884 - val_accuracy
Epoch 5/10
1875/1875 - 9s - loss: 0.0679 - accuracy: 0.9780 - val_loss: 0.0790 - val_accuracy
Epoch 6/10
1875/1875 - 9s - loss: 0.0591 - accuracy: 0.9811 - val_loss: 0.0702 - val_accuracy
Epoch 7/10
1875/1875 - 9s - loss: 0.0509 - accuracy: 0.9833 - val_loss: 0.0655 - val_accuracy
Epoch 8/10
1875/1875 - 8s - loss: 0.0459 - accuracy: 0.9850 - val_loss: 0.0760 - val_accuracy
Epoch 9/10
1875/1875 - 10s - loss: 0.0426 - accuracy: 0.9856 - val_loss: 0.0703 - val_accurac
Epoch 10/10
1875/1875 - 9s - loss: 0.0386 - accuracy: 0.9869 - val_loss: 0.0782 - val_accuracy
313/313 - 1s - loss: 0.0782 - accuracy: 0.9774 - 530ms/epoch - 2ms/step
Epoch 1/10
1875/1875 - 4s - loss: 0.4342 - accuracy: 0.8787 - val_loss: 0.2602 - val_accuracy
Epoch 2/10
1875/1875 - 4s - loss: 0.2494 - accuracy: 0.9291 - val_loss: 0.2292 - val_accuracy
Epoch 3/10
1875/1875 - 4s - loss: 0.2147 - accuracy: 0.9388 - val_loss: 0.2084 - val_accuracy
Epoch 4/10
1875/1875 - 3s - loss: 0.1943 - accuracy: 0.9434 - val_loss: 0.1882 - val_accuracy
Epoch 5/10
1875/1875 - 4s - loss: 0.1793 - accuracy: 0.9477 - val_loss: 0.1803 - val_accuracy
Epoch 6/10
1875/1875 - 5s - loss: 0.1677 - accuracy: 0.9510 - val_loss: 0.1776 - val_accuracy
Epoch 7/10
1875/1875 - 4s - loss: 0.1590 - accuracy: 0.9532 - val_loss: 0.1726 - val_accuracy
Epoch 8/10
1875/1875 - 5s - loss: 0.1514 - accuracy: 0.9552 - val_loss: 0.1642 - val_accuracy
Epoch 9/10
1875/1875 - 4s - loss: 0.1465 - accuracy: 0.9565 - val_loss: 0.1600 - val_accuracy
Epoch 10/10
1875/1875 - 4s - loss: 0.1409 - accuracy: 0.9582 - val_loss: 0.1752 - val_accuracy
313/313 - 0s - loss: 0.1752 - accuracy: 0.9479 - 426ms/epoch - 1ms/step
With Batch Normalization:
Test accuracy: 0.977400004863739
Without Batch Normalization:
```

```
Test accuracy: 0.9478999972343445
```

Task 4: Extend your neural network to include multiple activation functions. Implement the following activation functions in your model:

1. Tanh
2. Sigmoid
3. ReLU (Rectified Linear Unit)
4. Softmax

Experiment with different activation functions for different layers and document the impact on training

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.activations import tanh, sigmoid, relu, softmax
from tensorflow.keras.utils import to_categorical

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

# Define a function to create a model with different activation functions
def create_model(activation_1, activation_2):
    model = Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        Dense(128, activation=activation_1),
        Dense(64, activation=activation_2),
        Dense(10, activation=softmax)
    ])
    return model

# Define different activation functions to experiment with
activation_functions = [tanh, sigmoid, relu]

# Experiment with different combinations of activation functions
for activation_1 in activation_functions:
    for activation_2 in activation_functions:
        model = create_model(activation_1, activation_2)
        model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['

        print(f"Training model with activation functions: {activation_1.__name__},
        history = model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_
        print("\n")
```

```
Training model with activation functions: tanh, tanh
Epoch 1/5
1875/1875 - 8s - loss: 0.2638 - accuracy: 0.9229 - val_loss: 0.1689 - val_accuracy
```

```
1875/1875 - 8s - loss: 0.2658 - accuracy: 0.9229 - val_loss: 0.1689 - val_accuracy
Epoch 2/5
1875/1875 - 6s - loss: 0.1205 - accuracy: 0.9646 - val_loss: 0.1030 - val_accuracy
Epoch 3/5
1875/1875 - 7s - loss: 0.0816 - accuracy: 0.9751 - val_loss: 0.0973 - val_accuracy
Epoch 4/5
1875/1875 - 6s - loss: 0.0595 - accuracy: 0.9814 - val_loss: 0.0973 - val_accuracy
Epoch 5/5
1875/1875 - 7s - loss: 0.0464 - accuracy: 0.9856 - val_loss: 0.0907 - val_accuracy


Training model with activation functions: tanh, sigmoid
Epoch 1/5
1875/1875 - 8s - loss: 0.3329 - accuracy: 0.9119 - val_loss: 0.1553 - val_accuracy
Epoch 2/5
1875/1875 - 6s - loss: 0.1292 - accuracy: 0.9624 - val_loss: 0.1096 - val_accuracy
Epoch 3/5
1875/1875 - 7s - loss: 0.0845 - accuracy: 0.9750 - val_loss: 0.0853 - val_accuracy
Epoch 4/5
1875/1875 - 7s - loss: 0.0618 - accuracy: 0.9818 - val_loss: 0.0844 - val_accuracy
Epoch 5/5
1875/1875 - 7s - loss: 0.0455 - accuracy: 0.9868 - val_loss: 0.0798 - val_accuracy


Training model with activation functions: tanh, relu
Epoch 1/5
1875/1875 - 8s - loss: 0.2498 - accuracy: 0.9264 - val_loss: 0.1377 - val_accuracy
Epoch 2/5
1875/1875 - 6s - loss: 0.1107 - accuracy: 0.9657 - val_loss: 0.0988 - val_accuracy
Epoch 3/5
1875/1875 - 7s - loss: 0.0746 - accuracy: 0.9765 - val_loss: 0.0965 - val_accuracy
Epoch 4/5
1875/1875 - 6s - loss: 0.0568 - accuracy: 0.9825 - val_loss: 0.0801 - val_accuracy
Epoch 5/5
1875/1875 - 7s - loss: 0.0417 - accuracy: 0.9867 - val_loss: 0.0779 - val_accuracy


Training model with activation functions: sigmoid, tanh
Epoch 1/5
1875/1875 - 8s - loss: 0.3419 - accuracy: 0.9029 - val_loss: 0.1957 - val_accuracy
Epoch 2/5
1875/1875 - 8s - loss: 0.1561 - accuracy: 0.9541 - val_loss: 0.1302 - val_accuracy
Epoch 3/5
1875/1875 - 6s - loss: 0.1056 - accuracy: 0.9682 - val_loss: 0.1026 - val_accuracy
Epoch 4/5
1875/1875 - 7s - loss: 0.0768 - accuracy: 0.9763 - val_loss: 0.0941 - val_accuracy
Epoch 5/5
1875/1875 - 6s - loss: 0.0582 - accuracy: 0.9823 - val_loss: 0.0759 - val_accuracy


Training model with activation functions: sigmoid, sigmoid
Epoch 1/5
1875/1875 - 8s - loss: 0.4662 - accuracy: 0.8832 - val_loss: 0.2118 - val_accuracy
Epoch 2/5
1875/1875 - 6s - loss: 0.1786 - accuracy: 0.9470 - val_loss: 0.1465 - val_accuracy
Epoch 3/5
```