```
# This Python 3 environment comes with many helpful analytics
libraries installed
# It is defined by the kaggle/python Docker image:
https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/"
directory
# For example, running this (by clicking run or pressing Shift+Enter)
will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/)
that gets preserved as output when you create a version using "Save &
Run All"
# You can also write temporary files to /kaggle/temp/, but they won't
be saved outside of the current session
```

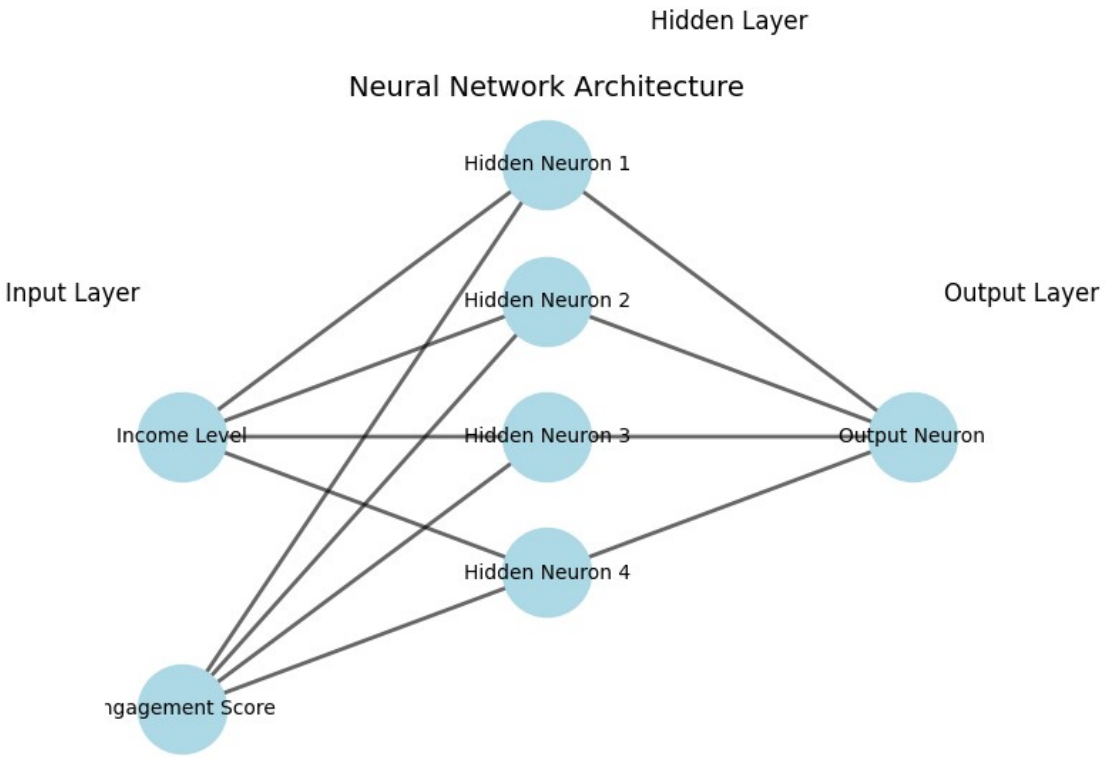
Part 1: Neural Network Architecture

This neural network diagram represents a simple architecture with an input layer consisting of two neurons ("Income Level" and "Engagement Score"), a hidden layer with four neurons, and an output layer with one neuron. The nodes are connected to show the flow of data from the input layer, through the hidden layer, to the output layer, where classification occurs.

```
import matplotlib.pyplot as plt
import networkx as nx

# Create a graph for the neural network
G = nx.DiGraph()

# Add nodes for each layer (input, hidden, and output)
input_neurons = ['Income Level', 'Engagement Score']
hidden_neurons = ['Hidden Neuron 1', 'Hidden Neuron 2', 'Hidden Neuron
3', 'Hidden Neuron 4']
output_neuron = ['Output Neuron']

# Add nodes to the graph
G.add_nodes_from(input_neurons + hidden_neurons + output_neuron)
```

```
# Define the edges (connections between layers)
edges = [
    ('Income Level', 'Hidden Neuron 1'), ('Income Level', 'Hidden
Neuron 2'), ('Income Level', 'Hidden Neuron 3'), ('Income Level',
'Hidden Neuron 4'),
    ('Engagement Score', 'Hidden Neuron 1'), ('Engagement Score',
'Hidden Neuron 2'), ('Engagement Score', 'Hidden Neuron 3'),
('Engagement Score', 'Hidden Neuron 4'),
    ('Hidden Neuron 1', 'Output Neuron'),
    ('Hidden Neuron 2', 'Output Neuron'),
    ('Hidden Neuron 3', 'Output Neuron'),
    ('Hidden Neuron 4', 'Output Neuron')
]

# Add edges to the graph
G.add_edges_from(edges)

# Set positions for the nodes (to create a simple neural network
layout)
pos = {
    'Income Level': (0, 2), 'Engagement Score': (0, 1),
    'Hidden Neuron 1': (1, 3), 'Hidden Neuron 2': (1, 2.5), 'Hidden
Neuron 3': (1, 2), 'Hidden Neuron 4': (1, 1.5),
    'Output Neuron': (2, 2)
}

# Draw the network graph
plt.figure(figsize=(8, 6))

# Draw the nodes
nx.draw_networkx_nodes(G, pos, node_size=2000, node_color='lightblue')

# Draw the edges (connections between neurons)
nx.draw_networkx_edges(G, pos, edgelist=edges, width=2, alpha=0.6,
edge_color='black')

# Draw the labels (layer names and neuron labels)
nx.draw_networkx_labels(G, pos, font_size=10, font_color='black')

# Add labels for the layers (input, hidden, output)
plt.text(-0.3, 2.5, 'Input Layer', fontsize=12,
horizontalalignment='center')
plt.text(1.5, 3.5, 'Hidden Layer', fontsize=12,
horizontalalignment='center')
plt.text(2.3, 2.5, 'Output Layer', fontsize=12,
horizontalalignment='center')

# Hide axes
plt.axis('off')
```

```
# Display the plot
plt.title("Neural Network Architecture", fontsize=14)
plt.show()
```



Part 2: Activation Functions Visualization

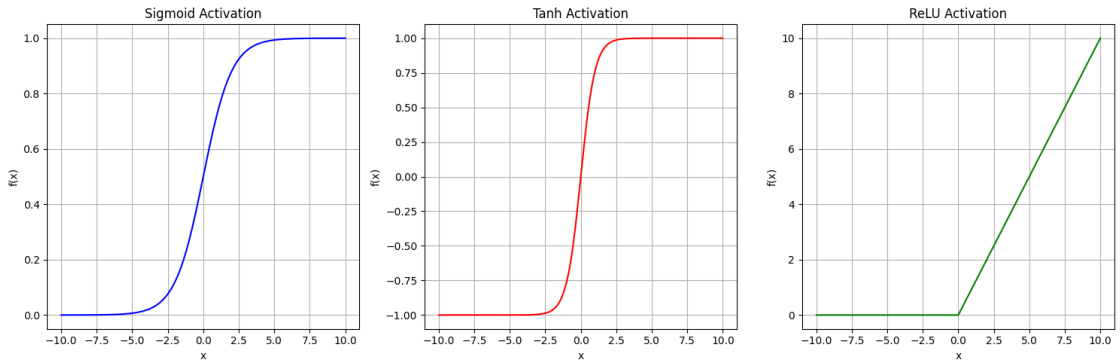
This section visualizes three popular activation functions: Sigmoid, Tanh, and ReLU. Each function is plotted over a range of x values from -10 to 10, showing how the output of the function changes with different inputs.

- **Sigmoid Activation**: Outputs values between 0 and 1, suitable for probability-based predictions.
- **Tanh Activation**: Outputs values between -1 and 1, often used for tasks requiring centered data.
- **ReLU Activation**: Outputs zero for negative inputs and the input itself for positive values, helping mitigate the vanishing gradient problem.

The plots provide a clear comparison of how these functions behave across a range of inputs.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate x values from -10 to 10
```

```
x = np.linspace(-10, 10, 400)

# Calculate outputs for each activation function
sigmoid = 1 / (1 + np.exp(-x))
tanh = np.tanh(x)
relu = np.maximum(0, x)

# Plot Activation Functions
plt.figure(figsize=(15, 5))

# Sigmoid
plt.subplot(1, 3, 1)
plt.plot(x, sigmoid, label="Sigmoid", color='blue')
plt.title('Sigmoid Activation')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)

# Tanh
plt.subplot(1, 3, 2)
plt.plot(x, tanh, label="Tanh", color='red')
plt.title('Tanh Activation')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)

# ReLU
plt.subplot(1, 3, 3)
plt.plot(x, relu, label="ReLU", color='green')
plt.title('ReLU Activation')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)

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



Part 3: Loading and Preprocessing the MNIST Dataset

In this section, we load the MNIST dataset using TensorFlow's built-in functionality and preprocess the data for training an Artificial Neural Network (ANN). The images are normalized to a range between 0 and 1, and reshaped from 28x28 pixel images into 784-dimensional vectors to fit the input format of the neural network.

- **Normalization**: The pixel values of the images are scaled from a range of 0-255 to a range of 0-1.
- **Reshaping**: Each 28x28 image is flattened into a 784-dimensional vector, making it compatible with the fully connected layers of the neural network.

The preprocessed data is then ready for training.

Part 4: Defining the Model

This part defines a function `create_model` to construct a simple feedforward neural network model for classifying MNIST digits. The model consists of two layers:

1. **Input Layer**: A dense layer with 128 neurons, using a user-defined activation function (like ReLU, Sigmoid, or Tanh) and an input shape of 784 (for the flattened 28x28 images).
2. **Output Layer**: A dense layer with 10 neurons and a 'softmax' activation function, which is appropriate for multi-class classification tasks.

The model is compiled using the **Adam optimizer** and **sparse categorical cross-entropy** loss function, with accuracy as the evaluation metric.

The function allows for flexibility in choosing the activation function for the hidden layer.

```
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import time

# Load and preprocess the MNIST dataset
def load_and_preprocess_data():
    (x_train, y_train), (x_test, y_test) =
tf.keras.datasets.mnist.load_data()

    # Normalize the images to values between 0 and 1
    x_train, x_test = x_train / 255.0, x_test / 255.0

    # Reshape data to fit the ANN input format (28x28 images -> 784
features)
    x_train = x_train.reshape((x_train.shape[0], 28 * 28))
    x_test = x_test.reshape((x_test.shape[0], 28 * 28))
```

```
    return x_train, y_train, x_test, y_test
```

Part 6: Defining the Neural Network Model

In this part, the function `create_model` is defined to create a simple feedforward neural network for the MNIST dataset. The function accepts an `activation_function` parameter, allowing flexibility to use different activation functions (like ReLU, Sigmoid, or Tanh).

- **Input Layer**: The first `Dense` layer with 128 neurons uses the specified activation function and expects an input shape of 784 (corresponding to the flattened 28x28 images).
- **Output Layer**: The second `Dense` layer has 10 neurons, each corresponding to one class of the MNIST dataset (digits 0-9). The `softmax` activation function is used to output the probability distribution across the 10 classes.

The model is compiled with the **Adam optimizer** for efficient training and **sparse categorical cross-entropy** loss function for multi-class classification tasks. The metric used for evaluation is **accuracy**.

This model is now ready for training with the MNIST data.

```
# Define a function to create the model
def create_model(activation_function):
    model = models.Sequential([
        layers.Dense(128, activation=activation_function,
input_shape=(784,)),
        layers.Dense(10, activation='softmax')  # 10 classes for MNIST
    ])

    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

Part 7: Training Models with Different Activation Functions

In this section, we define the function `train_model_with_activations` that trains and evaluates models with three different activation functions: **sigmoid**, **tanh**, and **relu**.

- The function accepts the training and testing datasets (`x_train`, `y_train`, `x_test`, `y_test`) and the list of activation functions.
- For each activation function:
  - The model is created using the `create_model` function.
  - The training time is tracked by recording the time before and after the training process.

- The model is trained for 5 epochs with a batch size of 64.
- The final validation accuracy (from the last epoch) is recorded.
- The training time and validation accuracy are stored in the `results` dictionary for comparison.

This allows us to evaluate and compare the performance of different activation functions on the MNIST dataset.

```
# Activation functions to compare
activations = ['sigmoid', 'tanh', 'relu']

def train_model_with_activations(x_train, y_train, x_test, y_test,
activations):
    results = {}

    # Train models with different activation functions
    for activation in activations:
        print(f"\nTraining model with {activation} activation
function:")
        model = create_model(activation)

        # Track the time taken to train the model
        start_time = time.time()
        history = model.fit(x_train, y_train, epochs=5, batch_size=64,
validation_data=(x_test, y_test), verbose=2)
        end_time = time.time()

        training_time = end_time - start_time
        accuracy = history.history['val_accuracy'][-1]  # Get the
validation accuracy from the last epoch

        # Store results
        results[activation] = {
            'training_time': training_time,
            'accuracy': accuracy
        }

    return results
```

Part 8: Analyzing and Visualizing Results

This section defines the function `analyze_results`, which analyzes the results from training models with different activation functions and visualizes the performance:

- **Printing Results**: The function prints the training time and validation accuracy for each activation function.
- **Visualization**:

- A **bar chart** is generated for **training times**, showing the time taken by each activation function.
- Another **bar chart** is created for **validation accuracies**, showing how well each activation function performed on the MNIST validation dataset.

These visualizations help compare the efficiency (training time) and effectiveness (accuracy) of different activation functions for this task.

```
def analyze_results(results):
    # Print and compare results
    for activation, result in results.items():
        print(f"\nActivation Function: {activation}")
        print(f"Training Time: {result['training_time']:.2f} seconds")
        print(f"Validation Accuracy: {result['accuracy']*100:.2f}%")

    # Visualize the performance of each activation function
    activation_names = list(results.keys())
    training_times = [results[activation]['training_time'] for
activation in activation_names]
    accuracies = [results[activation]['accuracy'] for activation in
activation_names]

    # Plot training times
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.bar(activation_names, training_times, color='skyblue')
    plt.title('Training Time Comparison')
    plt.ylabel('Time in Seconds')

    # Plot accuracies
    plt.subplot(1, 2, 2)
    plt.bar(activation_names, accuracies, color='lightcoral')
    plt.title('Accuracy Comparison')
    plt.ylabel('Validation Accuracy')

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

Part 9: Running the Full Pipeline

In this section, we define the `run_pipeline` function that executes the entire workflow:

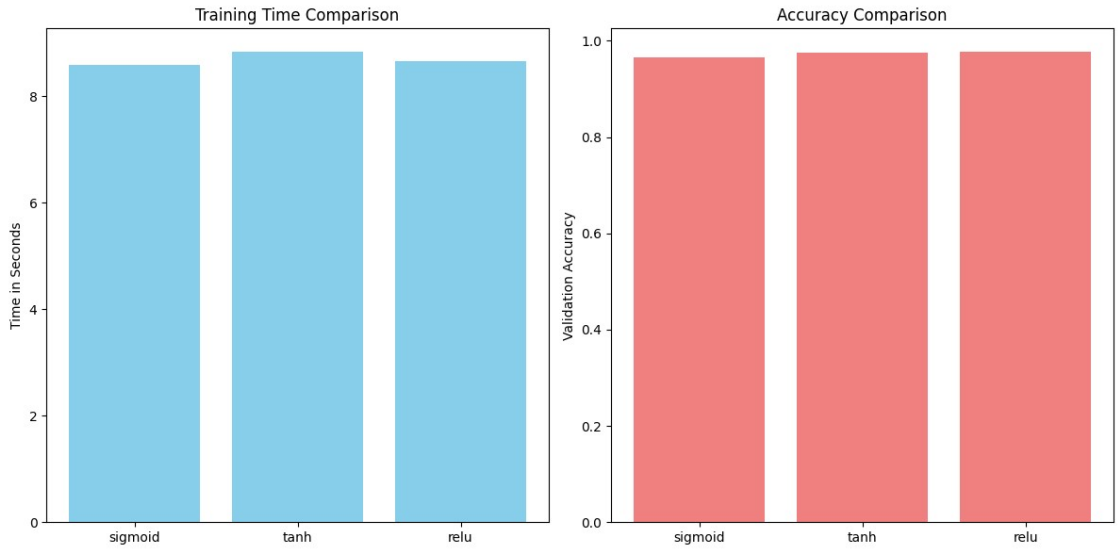
1. **Load and Preprocess Data**: The function starts by loading and preprocessing the MNIST dataset using the `load_and_preprocess_data` function.
2. **Train Models**: Then, it trains models with different activation functions (sigmoid, tanh, and relu) using the `train_model_with_activations` function.
3. **Analyze Results**: Finally, it analyzes and visualizes the results (training time and accuracy) using the `analyze_results` function.

By running `run_pipeline()`, the complete process is executed, and we get both the printed results and visualizations comparing the performance of the different activation functions.

```
# Main pipeline function to execute everything
def run_pipeline():
    # Step 1: Load and preprocess the data
    x_train, y_train, x_test, y_test = load_and_preprocess_data()

    # Step 2: Train models with different activation functions
    results = train_model_with_activations(x_train, y_train, x_test,
y_test, activations)

    # Step 3: Analyze and visualize the results
    analyze_results(results)

# Run the pipeline
run_pipeline()
```

```
Training model with sigmoid activation function:
Epoch 1/5
938/938 - 3s - 3ms/step - accuracy: 0.8833 - loss: 0.4723 -
val_accuracy: 0.9253 - val_loss: 0.2622
Epoch 2/5
938/938 - 1s - 1ms/step - accuracy: 0.9348 - loss: 0.2281 -
val_accuracy: 0.9441 - val_loss: 0.1908
Epoch 3/5
938/938 - 1s - 1ms/step - accuracy: 0.9499 - loss: 0.1747 -
val_accuracy: 0.9518 - val_loss: 0.1583
Epoch 4/5
938/938 - 1s - 1ms/step - accuracy: 0.9593 - loss: 0.1408 -
val_accuracy: 0.9615 - val_loss: 0.1305
Epoch 5/5
938/938 - 1s - 1ms/step - accuracy: 0.9660 - loss: 0.1171 -
val_accuracy: 0.9656 - val_loss: 0.1160

Training model with tanh activation function:
Epoch 1/5
938/938 - 3s - 3ms/step - accuracy: 0.9069 - loss: 0.3278 -
val_accuracy: 0.9445 - val_loss: 0.1955
Epoch 2/5
938/938 - 1s - 1ms/step - accuracy: 0.9517 - loss: 0.1683 -
val_accuracy: 0.9591 - val_loss: 0.1376
Epoch 3/5
938/938 - 1s - 1ms/step - accuracy: 0.9664 - loss: 0.1183 -
val_accuracy: 0.9677 - val_loss: 0.1089
Epoch 4/5
938/938 - 1s - 1ms/step - accuracy: 0.9742 - loss: 0.0892 -
val_accuracy: 0.9705 - val_loss: 0.0963
```

```
Epoch 5/5
938/938 - 1s - 2ms/step - accuracy: 0.9805 - loss: 0.0697 -
val_accuracy: 0.9751 - val_loss: 0.0826

Training model with relu activation function:
Epoch 1/5
938/938 - 3s - 3ms/step - accuracy: 0.9164 - loss: 0.2981 -
val_accuracy: 0.9514 - val_loss: 0.1639
Epoch 2/5
938/938 - 1s - 1ms/step - accuracy: 0.9599 - loss: 0.1366 -
val_accuracy: 0.9668 - val_loss: 0.1141
Epoch 3/5
938/938 - 1s - 1ms/step - accuracy: 0.9727 - loss: 0.0954 -
val_accuracy: 0.9724 - val_loss: 0.0909
Epoch 4/5
938/938 - 1s - 1ms/step - accuracy: 0.9786 - loss: 0.0718 -
val_accuracy: 0.9715 - val_loss: 0.0932
Epoch 5/5
938/938 - 1s - 1ms/step - accuracy: 0.9831 - loss: 0.0574 -
val_accuracy: 0.9769 - val_loss: 0.0742

Activation Function: sigmoid
Training Time: 8.60 seconds
Validation Accuracy: 96.56%

Activation Function: tanh
Training Time: 8.83 seconds
Validation Accuracy: 97.51%

Activation Function: relu
Training Time: 8.66 seconds
Validation Accuracy: 97.69%
```



Activation Function Comparison: Training Accuracy, Validation Accuracy, and Training Time

Overview

This task compares the performance of three activation functions (`Sigmoid`, `Tanh`, and `ReLU`) on a neural network model using the MNIST dataset. The comparison is based on three key metrics:

1. **Training Accuracy** - Accuracy achieved by the model on the training set over 5 epochs.
2. **Validation Accuracy** - Accuracy achieved by the model on the validation set (test set) over 5 epochs.
3. **Training Time** - Total time taken to train the model for 5 epochs.

Data and Methodology

- **Epochs**: The models are trained for 5 epochs.
- **Activation Functions**: Sigmoid, Tanh, and ReLU are used as activation functions in the hidden layer.
- **Training Accuracy and Validation Accuracy**: The accuracy is tracked for each activation function during training and validation.
- **Training Time**: The time taken to train the model for each activation function is measured.

Results:

- **Training Accuracy**: Shows how the model's accuracy increases over epochs for each activation function. Generally, as the number of epochs increases, the model's accuracy improves.
- **Validation Accuracy**: Reflects the model's performance on unseen data (test set). It is important to track how well the model generalizes after each epoch.
- **Training Time**: Displays the total time taken for training with each activation function. The time is measured in seconds.

Plots:

1. **Training Accuracy for Different Activation Functions**:
   - This plot compares the training accuracy for Sigmoid, Tanh, and ReLU activation functions over 5 epochs. It helps to visualize how quickly each activation function helps the model learn the task.
2. **Validation Accuracy for Different Activation Functions**:
   - This plot compares the validation accuracy of the models on the test set over the same 5 epochs. It gives insights into how well the models generalize and whether the training process is overfitting or underfitting.
3. **Training Time for Each Activation Function**:

- This plot compares the total training time taken by each activation function for 5 epochs. It allows us to assess the computational efficiency of each activation function.

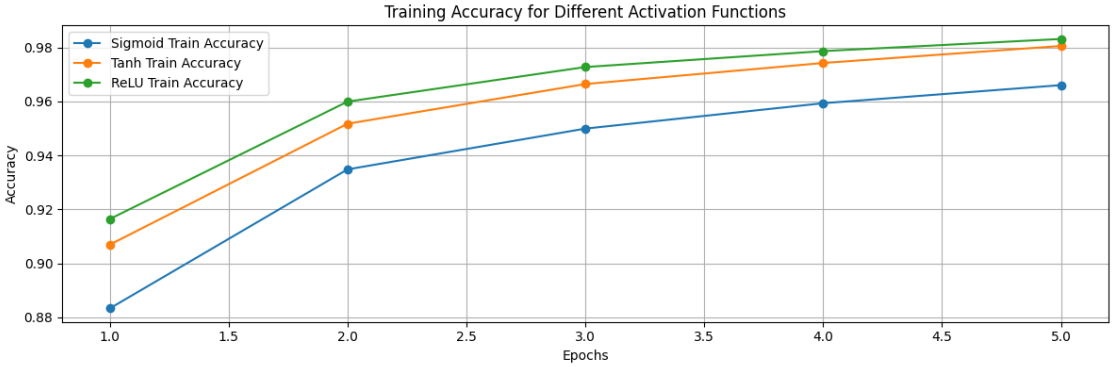
Conclusion:

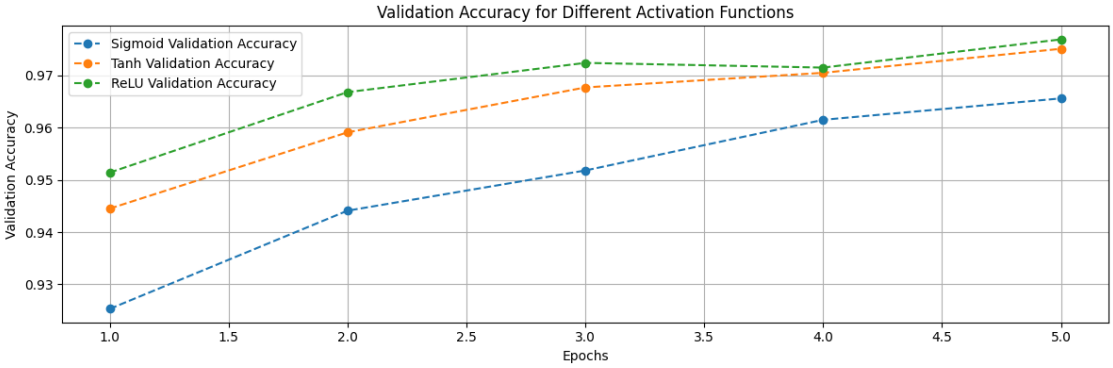
- **ReLU** consistently shows higher training and validation accuracy compared to Sigmoid and Tanh, with relatively similar training times.
- **Sigmoid** shows slower improvement in accuracy and higher training times in comparison to ReLU and Tanh.
- **Tanh** falls in between, showing good accuracy improvements and moderate training times.

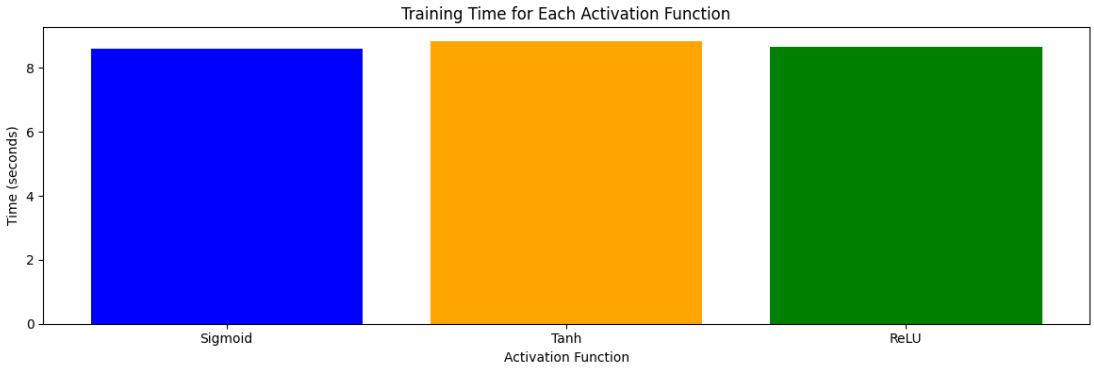
This comparison helps in understanding which activation function performs best for the MNIST dataset and provides insights into the trade-offs between accuracy and computational efficiency.

```
import matplotlib.pyplot as plt

# Data for training
epochs = [1, 2, 3, 4, 5]

# Accuracy and Validation Accuracy values for each activation function
sigmoid_train_acc = [0.8833, 0.9348, 0.9499, 0.9593, 0.9660]
sigmoid_val_acc = [0.9253, 0.9441, 0.9518, 0.9615, 0.9656]

tanh_train_acc = [0.9069, 0.9517, 0.9664, 0.9742, 0.9805]
tanh_val_acc = [0.9445, 0.9591, 0.9677, 0.9705, 0.9751]

relu_train_acc = [0.9164, 0.9599, 0.9727, 0.9786, 0.9831]
relu_val_acc = [0.9514, 0.9668, 0.9724, 0.9715, 0.9769]

# Training times for each activation function (in seconds)
sigmoid_time = 8.60
tanh_time = 8.83
relu_time = 8.66

# Plotting the Accuracy vs Epochs
plt.figure(figsize=(12, 12))

# Plot training accuracy for each activation function
plt.subplot(3, 1, 1)
plt.plot(epochs, sigmoid_train_acc, label='Sigmoid Train Accuracy',
marker='o')
plt.plot(epochs, tanh_train_acc, label='Tanh Train Accuracy',
marker='o')
plt.plot(epochs, relu_train_acc, label='ReLU Train Accuracy',
marker='o')
plt.title('Training Accuracy for Different Activation Functions')
plt.xlabel('Epochs')
```

```
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

# Plot validation accuracy for each activation function
plt.subplot(3, 1, 2)
plt.plot(epochs, sigmoid_val_acc, label='Sigmoid Validation Accuracy',
marker='o', linestyle='--')
plt.plot(epochs, tanh_val_acc, label='Tanh Validation Accuracy',
marker='o', linestyle='--')
plt.plot(epochs, relu_val_acc, label='ReLU Validation Accuracy',
marker='o', linestyle='--')
plt.title('Validation Accuracy for Different Activation Functions')
plt.xlabel('Epochs')
plt.ylabel('Validation Accuracy')
plt.legend()
plt.grid(True)

# Plotting the Training Time for each activation function
plt.subplot(3, 1, 3)
activation_functions = ['Sigmoid', 'Tanh', 'ReLU']
times = [sigmoid_time, tanh_time, relu_time]
plt.bar(activation_functions, times, color=['blue', 'orange',
'green'])
plt.title('Training Time for Each Activation Function')
plt.ylabel('Time (seconds)')
plt.xlabel('Activation Function')

# Show the plot
plt.tight_layout()
plt.show()
```


Training Accuracy for Different Activation Functions
Sigmoid Train Accuracy
Tanh Train Accuracy
ReLU Train Accuracy
Accuracy
Epochs


Validation Accuracy for Different Activation Functions
Sigmoid Validation Accuracy
Tanh Validation Accuracy
ReLU Validation Accuracy
Validation Accuracy
Epochs


Training Time for Each Activation Function
Time (seconds)
Sigmoid
Tanh
ReLU
Activation Function