Awarding Body: Arden University
Programme Name: Master of Data Science
Module Name (and Part if applicable): Artificial Intelligence and Neural Networks
Assessment Title: MNIST Digit Recognition: A Neural Network Approach with ReLU and Tanh Comparison
Student Number: STU221742
Tutor Name: Mr. Ali Vaisifard
Word Count: 2900

Introduction:

Nowadays, Artificial Neural Networks have led to considerable progression in the field of image processing and classification, with wide-ranging applications in banking systems, handwritten document recognition, and Facial recognition systems within organizations. One of the most prominent applications of the MNIST dataset (**Modified National Institute of Standards and Technology**)is the identification and classification of handwritten digits. In this project, the standard MNIST dataset is utilized to design and implement a multi-layer neural network model. The primary purpose is to evaluate and compare the model's performance using different activation functions ([1]**ReLU, [2]Tanh**) in the hidden layers, in order to examine their impact on the accuracy of digit recognition. We also continue to examine the legal and ethical aspects of the models, which are no less important than the technical performance of the models. The implementation and experiments are carried out using **Google Colab** notebooks, which provide a appropriate and efficient platform for developing and testing machine-learning models.

MNIST: A Standard Dataset for Handwritten Digit Classification:

The **MNIST** dataset is one of the most well-known open-source datasets in the fields of machine learning and computer vision, introduced by Cortes and Burges (1998). With a manageable size, it serves as a standard benchmark for evaluating classification models. This dataset contains **70,000** grayscale images of handwritten digits from **0** to **9**, each image having a size of **28** * **28** pixels. Out of these, **60,000** images are allocated for training the model, and **10,000** images are reserved for testing it.

TASK 1

Environment Setup:

Using the **pip,** the **TensorFlow** library, which is one of the most powerful libraries for designing, training, building, and evaluating neural network models, is installed.

```
pip install tensorflow
```

Figure 1- python codes for install tenssorflow

[1] ReLU: If the input value x is greater than 0, the output is x, and If x is less than or equal to 0, the output will be 0.
[2] Tanh: For any input value, the output of Tanh will be a number between -1 and 1, and the output of Tanh at the zero point is zero.

Importing Libraries:

```
import tensorflow as tf                          # To build and train machine learning models
import numpy as np
from tensorflow.keras.datasets import mnist          # Importing MINST data set
from tensorflow.keras.models import Sequential       # Importing Sequential Model
from tensorflow.keras.layers import Dense, Flatten   # Importing Layers
from tensorflow.keras.utils import to_categorical    # Convert numeric labels to vectors one-hot
import matplotlib.pyplot as plt
from keras import Input
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

Figure2- python codes for Importing Liebraries

Keras: A high level, user-friendly library for making and training artificial neural networks. It has advantages of simplicity in coding, flexibility, and extensive support. This library runs on TensorFlow and has been integrated with TensorFlow by default in the latest update.

Sequential: A simple linear model for inputting the layers in order (input, hidden, output). Each line is a definition of overlapping layers placed one after another.

Dense: Each neuron in this layer connects to every neuron in the layer before it, so it can take in all the available information.

Input: It is used to define the shape of the input data.

Flatten: It is the layer that reshapes the input from multiple dimensions into a single line of values, making it ready for the dense layers to process.

To_categorical: This function converts numerical labels to one-hot vectors that is essential for training a multi-class classification model.

Loading and Exploring Dataset:

```
[3] # Loading Dataset
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    # Display data shapes
    print("Training data shape:", x_train.shape)
    print("Test data shape:", x_test.shape)
```

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

Figure 3- python codes for Loading Data set

Note: Dataset is downloaded from the internet for the first time, subsequent times it uses the cache.

During the data loading process, two sets of **data** are loaded. In **x_train**, the training data, **(60,000** images of handwritten digits, each **28 * 28 pixels**), and in **y_train**, the numerical training labels (**0 to 9**) are placed. Following that, x_test and y_test contain **10,000** test data points. The dimensions of these two arrays are displayed using the shape function.

`11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step`: The total amount of bytes that need to be loaded in **n** seconds.

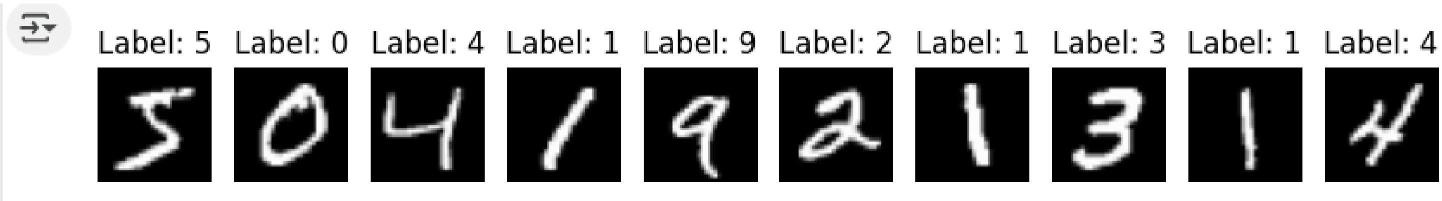
```
# Display sample images
def show_sample_images(X, y, n=10):
    plt.figure(figsize=(10,2))
    for i in range(n):
        plt.subplot(1,n,i+1)
        plt.imshow(X[i], cmap='gray')
        plt.title(f"Label: {y[i]}")
        plt.axis('off')
    plt.show()

show_sample_images(x_train, y_train)
```

Figure 4- python codes for Exploration of Dataset

.imshow from the matplotlib library graphically displays 10 elements of the array in gray.

pLt.imshow(): image from the dataset is displayed in gray scale.



Data pre-processing:

Step1: Normalization

“ Image normalization helps improve the convergence of deep learning models, Data normalization is a key step in improving model performance” (Han et al., 2011)

```
# If you intend to use ReLU, use the range [0, 1]
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Figure 5- python codes for Normalization

In the **MNIST** dataset, the value of each pixel ranges from **0** to **255**. To make the model training faster and better, especially when using an activation function like **ReLU**, which doesn't handle large numbers well, we simplify these values and bring them into the range of **0** to **1**. To do this, we divide each pixel value by **255**.

Step2: Converting labels to One-Hot Encoding

```
# Converting labels to one-hot
y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)
```

Figure 6- python codes for Normalization

We convert the labels of each image, which represent the written number, into integer numbers from **0** to **9** so that the model can categorize more easily.

To_categorical (): Convert the data into a binary array where all values are represented except for the position related to the real number. (**00010 = 1**)

Creating Models:

Python libraries such as scikit-learn are effective tools for building models (Müller and Guido, 2016)

Building a Sequential model with ReLU:

```
def build_model(activation_fn):                                # Definition of a function
    model = Sequential()                                       # A simple linear model in Keras
    model.add(Flatten(input_shape=(28, 28)))                   # reshape
    model.add(Dense(128, activation=activation_fn))            # creating the first layer
    model.add(Dense(64, activation=activation_fn))             # creating the second layer
    model.add(Dense(32, activation=activation_fn))             # creating the third layer
    model.add(Dense(10, activation='softmax'))                 # Probability output
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])   # compiling
    return model
model_relu = build_model('relu')     # calling
model_relu.summary()                 # summurizing
```

Figure7- Python Codes for Defining Model

We define a function with an input variable for the activation function to create a Sequential model. Initially, we convert all inputs, which are in the form of two-dimensional arrays, into a vector with [3]**784** features. Then, we create three hidden layers, each with **128**, **64**, and **32** neurons respectively, and the layers learn the features. The output layer is created with **10** neurons applying Softmax activation function, which is used for tasks like classifying a **10-class** dataset **MNIST** (from 0 to 9). In the compile phase, we use the **adam** algorithm, which is a combination of two models **Momentum** and **RMSprop** to adjust the weights and help improve the model during training. Then, using the parameters **loss** and **metrics**, we measure the **model's error** and determine how **accurate** the model is. After calling and referring activation function **ReLU** as input to the **build_model** function, we provide a report on the created layers and the performance of the layer creation function using **summary ()**.

[3] 784= 28*28

Softmax: This function helps the model express its prediction as a probability, which is very useful and supportive. It transforms the output of each neuron into a number between 0 and 1 and ensures that the sum of all outputs equals 1. The outputs are in the form of probabilities. In fact, the purpose of this function is to determine, for example, how likely it is that the desired number is 5 and how likely it is that it is 10?

```
history_relu = model_relu.fit(x_train, y_train_cat, epochs=10, validation_data=(x_test, y_test_cat), verbose=2)   # Training

# Prediction and evaluation of the ReLU model
y_pred_prob_relu = model_relu.predict(x_test)
y_pred_relu = np.argmax(y_pred_prob_relu, axis=1)
accuracy_relu = np.sum(y_pred_relu == y_test) / len(y_test)
print(f"Accuracy with ReLU: {accuracy_relu:.4f}")
```

Figure8- python codes for Training & Testing ReLU

The model is trained **10** times with the training data, and the models are validated. Through the parameter **verbos** the performance of each epoch is displayed to the extent of a line. Ultimately, the output of **fit ()** is an object called **history_relu**, which contains information about **accuracy** and **error at each epoch.** By **predict** function, predict and evaluate the accuracy of the model through compare predicted labels with the actual labels.

Model: "sequential_7"

Layer (type)	Output Shape	Param #
flatten_7 (Flatten)	(None, 784)	0
dense_26 (Dense)	(None, 128)	100,480
dense_27 (Dense)	(None, 64)	8,256
dense_28 (Dense)	(None, 32)	2,080
dense_29 (Dense)	(None, 10)	330

Total params: 111,146 (434.16 KB)
Trainable params: 111,146 (434.16 KB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
1875/1875 - 9s - 5ms/step - accuracy: 0.9214 - loss: 0.2670 - val_accuracy: 0.9610 - val_loss: 0.1271
Epoch 2/10
1875/1875 - 7s - 3ms/step - accuracy: 0.9670 - loss: 0.1081 - val_accuracy: 0.9561 - val_loss: 0.1346
Epoch 3/10
1875/1875 - 8s - 4ms/step - accuracy: 0.9755 - loss: 0.0780 - val_accuracy: 0.9711 - val_loss: 0.0924
Epoch 4/10
1875/1875 - 7s - 4ms/step - accuracy: 0.9814 - loss: 0.0587 - val_accuracy: 0.9710 - val_loss: 0.0888
Epoch 5/10
1875/1875 - 8s - 4ms/step - accuracy: 0.9852 - loss: 0.0467 - val_accuracy: 0.9777 - val_loss: 0.0739
Epoch 6/10
1875/1875 - 11s - 6ms/step - accuracy: 0.9872 - loss: 0.0390 - val_accuracy: 0.9776 - val_loss: 0.0768
Epoch 7/10
1875/1875 - 10s - 5ms/step - accuracy: 0.9891 - loss: 0.0327 - val_accuracy: 0.9768 - val_loss: 0.0879
Epoch 8/10
1875/1875 - 9s - 5ms/step - accuracy: 0.9899 - loss: 0.0300 - val_accuracy: 0.9752 - val_loss: 0.0934
Epoch 9/10
1875/1875 - 12s - 6ms/step - accuracy: 0.9919 - loss: 0.0243 - val_accuracy: 0.9768 - val_loss: 0.0918
Epoch 10/10
1875/1875 - 8s - 5ms/step - accuracy: 0.9923 - loss: 0.0230 - val_accuracy: 0.9762 - val_loss: 0.0989
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Accuracy with ReLU: 0.9762
```

Figure9- Result of ReLU Accuracy

This is a report table of the **one-dimensionalization** process and the number of input pixels to the layers. Each neuron is connected to the neurons of the next layer and has about **111,000** trainable parameters. Ultimately, the model will learn to convert each handwritten image into a correct number.

Train Accuracy has a rising and stable trend at **0.9762**. Considering the downward trend of **Training loss**, it can be concluded that the model has learned better over periods. **Validation Accuracy** have been slight differences during the periods, but the overall trend is acceptable. Finally, based on the declining trend of **Validation Loss** during the periods, it can be concluded that the error rate is very low.

Building a Sequential model with Tanh:

```python
model_tanh = build_model('tanh')
model_tanh.summary()
history_tanh = model_tanh.fit(x_train, y_train_cat, epochs=10, validation_data=(x_test, y_test_cat), verbose=2)

# Prediction and evaluation of the Tanh model
y_pred_prob_tanh = model_tanh.predict(x_test)
y_pred_tanh = np.argmax(y_pred_prob_tanh, axis=1)
accuracy_tanh = np.sum(y_pred_tanh == y_test) / len(y_test)
print(f"Accuracy with Tanh: {accuracy_tanh:.4f}")
```

Figure10- python codes for Training & Testing Tanh

Activation function operations **Tanh.**

```
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 784)	0
dense_12 (Dense)	(None, 128)	100,480
dense_13 (Dense)	(None, 64)	8,256
dense_14 (Dense)	(None, 32)	2,080
dense_15 (Dense)	(None, 10)	330

```
Total params: 111,146 (434.16 KB)
Trainable params: 111,146 (434.16 KB)
Non-trainable params: 0 (0.00 B)
```

```
Epoch 1/10
1875/1875 - 10s - 5ms/step - accuracy: 0.9229 - loss: 0.2727 - val_accuracy: 0.9598 - val_loss: 0.1353
Epoch 2/10
1875/1875 - 7s - 4ms/step - accuracy: 0.9638 - loss: 0.1212 - val_accuracy: 0.9626 - val_loss: 0.1154
Epoch 3/10
1875/1875 - 10s - 5ms/step - accuracy: 0.9749 - loss: 0.0845 - val_accuracy: 0.9712 - val_loss: 0.0900
Epoch 4/10
1875/1875 - 8s - 4ms/step - accuracy: 0.9799 - loss: 0.0645 - val_accuracy: 0.9680 - val_loss: 0.0996
Epoch 5/10
1875/1875 - 10s - 5ms/step - accuracy: 0.9842 - loss: 0.0500 - val_accuracy: 0.9748 - val_loss: 0.0799
Epoch 6/10
1875/1875 - 8s - 4ms/step - accuracy: 0.9869 - loss: 0.0408 - val_accuracy: 0.9761 - val_loss: 0.0767
Epoch 7/10
1875/1875 - 8s - 4ms/step - accuracy: 0.9904 - loss: 0.0319 - val_accuracy: 0.9731 - val_loss: 0.0899
Epoch 8/10
1875/1875 - 10s - 5ms/step - accuracy: 0.9905 - loss: 0.0292 - val_accuracy: 0.9732 - val_loss: 0.0997
Epoch 9/10
1875/1875 - 9s - 5ms/step - accuracy: 0.9908 - loss: 0.0285 - val_accuracy: 0.9761 - val_loss: 0.0890
Epoch 10/10
1875/1875 - 10s - 5ms/step - accuracy: 0.9936 - loss: 0.0201 - val_accuracy: 0.9729 - val_loss: 0.1013
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Accuracy with Tanh: 0.9729
```

Figure11- Result of Tanh Accuracy

Tanh model has a similar performance to **ReLU** model in **Train Accuracy** at **0.9729**. It has good accuracy in training and high validation accuracy. Meanwhile, it has a low error rate in both training and validation.

Comparison:

Both models have shown good and growing performance in learning throughout the periods, but the **ReLU** model has more stable performance with higher accuracy in validation. In addition, due to lower error rates, it learns patterns better and models them more effectively, being more resilient than **Tanh** and having a higher capability for generalization when faced with new and unseen data. Although the **Tanh** model performs well in the early and middle stages, it experiences a slight decline in accuracy and an increase in errors as time progresses and approaches the end of the periods. Despite this issue being minor, it indicates overfitting. Therefore, the **ReLU** model is more reliable and generalizable for use in real-world applications.

Confusion Matrix:

In this matrix, we evaluate the performance of the models to see how well these models has performed in making correct or incorrect predictions and to help us understand the extent to which the model needs data or adjustments in its architecture.

```
[43] # Confusion Matrix
     def plot_confusion(y_true, y_pred, title):
         cm = confusion_matrix(y_true, y_pred)
         plt.figure(figsize=(5,5))
         sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
         plt.title(f'Confusion Matrix - {title}')
         plt.xlabel('Predicted')
         plt.ylabel('True')
         plt.show()

     plot_confusion(y_test, y_pred_relu, "ReLU")
     plot_confusion(y_test, y_pred_tanh, "Tanh")
```

Figure12- python codes for Confusion Matrix

The main diameters indicate the number of correct predictions, while outside the diameter are the cases of incorrect predictions.

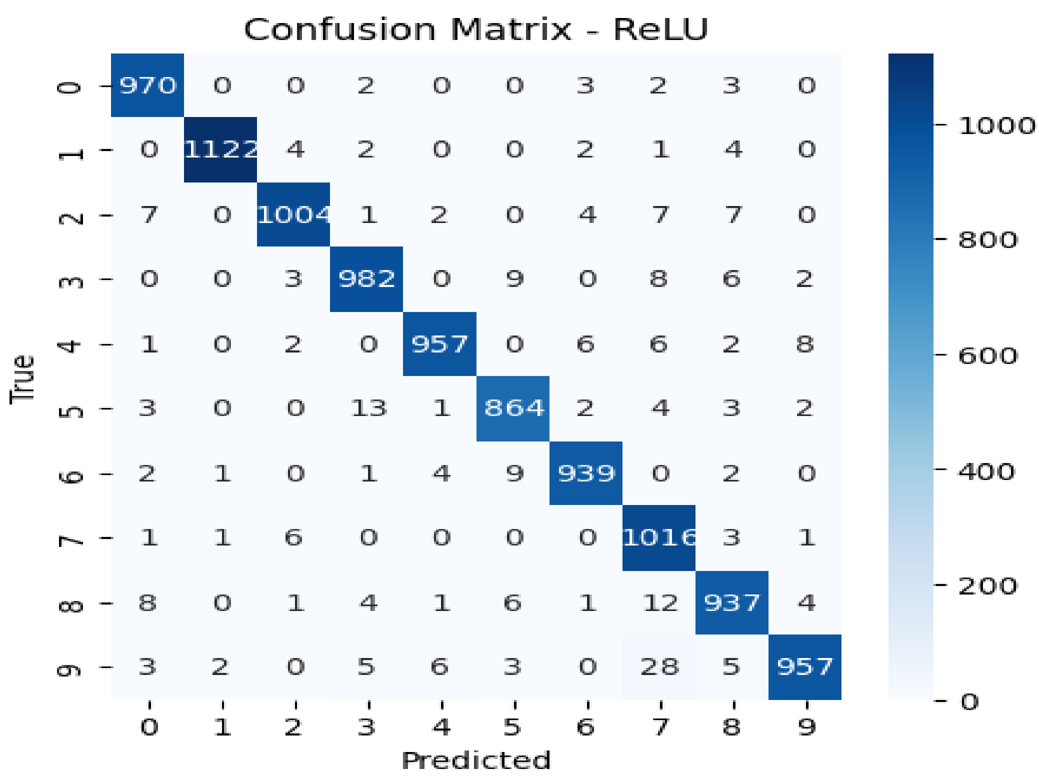


Figure13- Confusion Matrix of ReLU

The highest number of correctly predicted cases is related to **1**, which is **1122**, and the lowest number belongs to the number **8** and **6** at **937, 939,** respectively. On the other hand, the most common forecast errors involve confusing **9** with **7, 5** with **3,** and **8** with **7** at **28**, **13** and **12**, respectively.

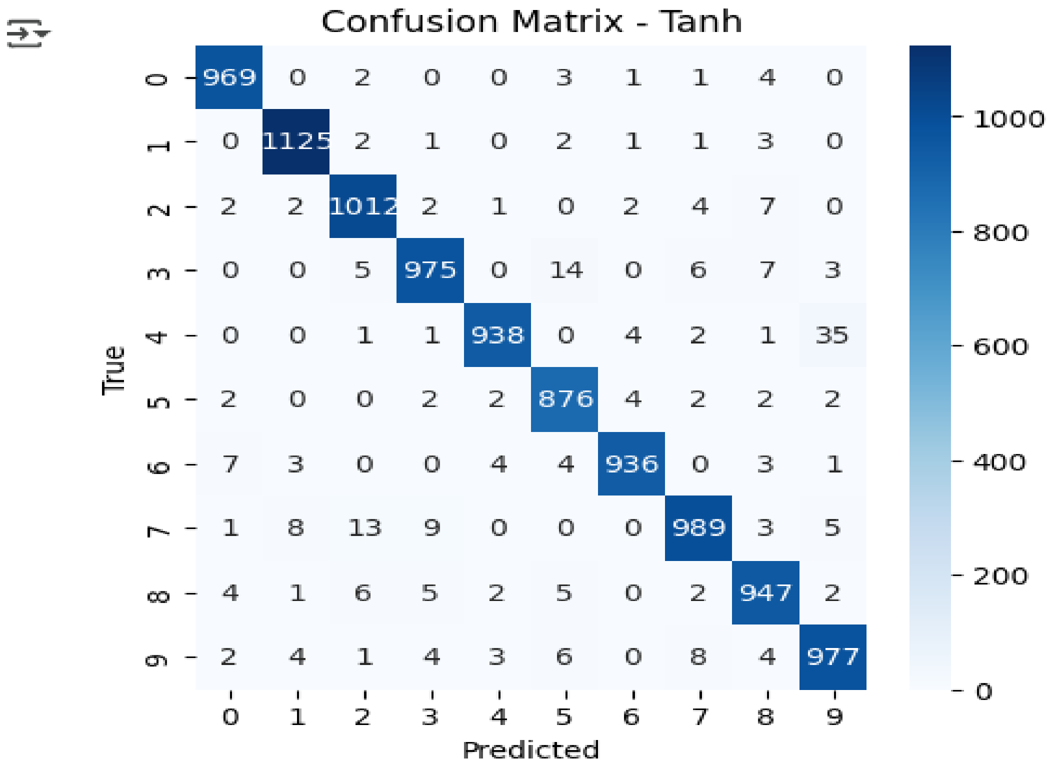


Figure14- Confusion Matrix of Tanh

In this model, just like in the **ReLU**, the highest number of correct predictions is for the number **1**, which is **1125**, and the lowest belong to **6** at **936**. The number **4** has been mistaken with **9** for **35** times. **3** and **7** are the most frequently confused with **5** and **2**, respectively.

Visualization:

```python
def plot_history(history, activation_name):
    # Training Accuracy and Validation
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title(f'Accuracy - Activation: {activation_name}')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()

    # Loss during training and validation
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title(f'Loss - Activation: {activation_name}')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

plot_history(history_relu, 'ReLU')
plot_history(history_tanh, 'Tanh')
```

Figure15- python codes for Drawing Diagrams

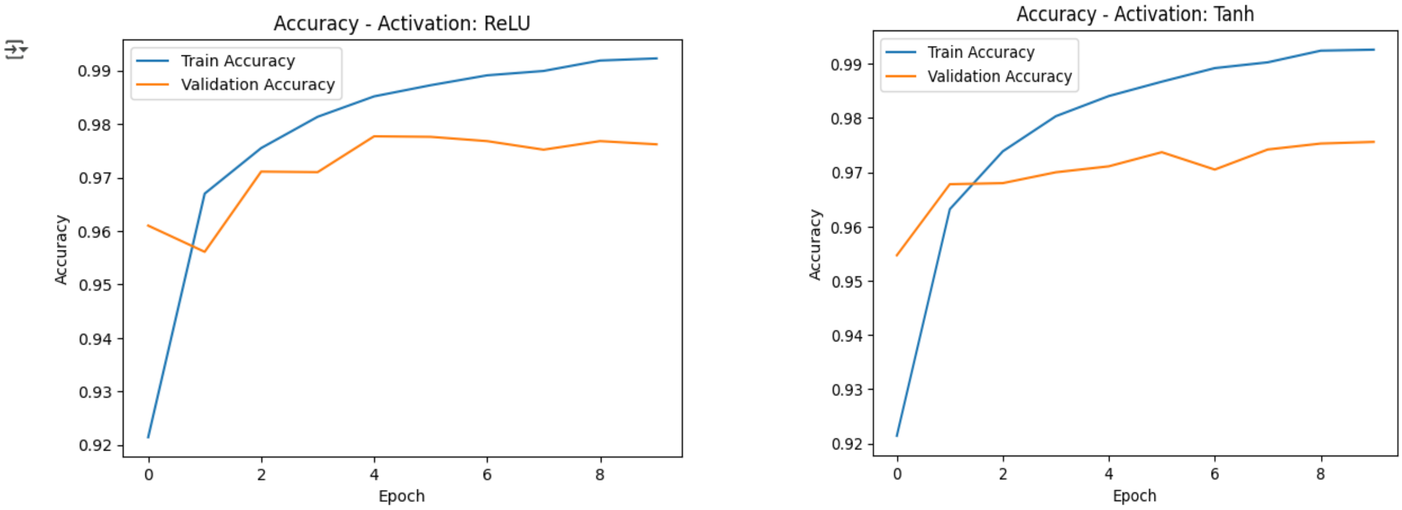


Figure16- Accuracy Diagram of ReLU & Tanh

Both models showed significant **growth** in **Training Accuracy**, And **ReLU** has exhibited slightly faster growth compared to **Tanh**. In terms of **Validation Aaccuracy**, both models have performed similarly, ranging between **96%** and **98%.** The **Tanh** model has shown greater stability and less fluctuation. Meanwhile, Signs of **Overfitting** can be observed in the **Relu** model, as there is a slight decrease after epoch **4** or **5**, followed by stability.

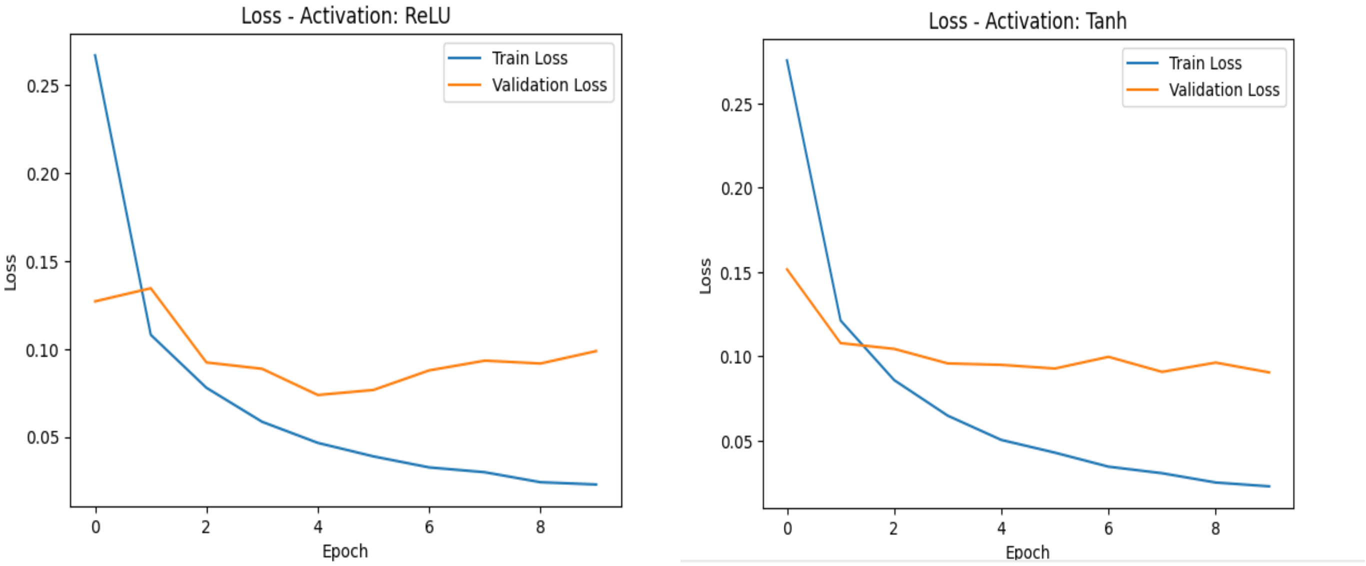


Figure17- Loss Diagram of ReLU & Tanh

Both models have shown a decreasing and rapid trend in **Training Error**. The **ReLU** model exprienced a faster reduction. In the **ReLU** model, the **Validation Error** initially experiences a decrease at the beginning of training, but after several epochs, it slightly **increases** again and keeps until the end of training, indicating **overfitting**. In the **Tanh** model, the validation error after sharp decreasing remains relatively stable, that indicat better stability on unseen data.

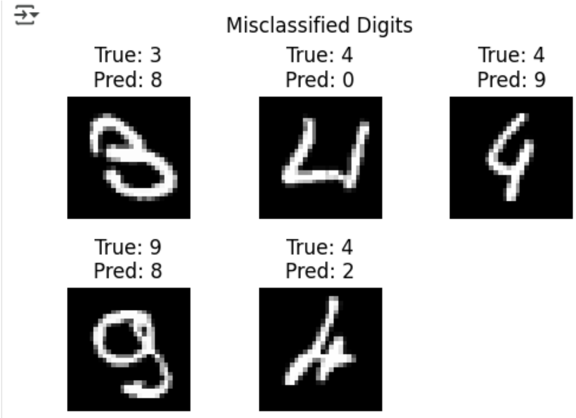
Display Examples of Mistakes:

```python
# Displaying Examples of Mistake
def show_misclassified(X, y_true, y_pred, n=5):
    wrong = np.where(y_true != y_pred)[0]              #Comparison of true labels with predicted labels
    plt.figure(figsize=(5,5))
    for i, idx in enumerate(wrong[:n]):
        plt.subplot(3,3,i+1)
        plt.imshow(X[idx], cmap='gray')                  # print lables
        plt.title(f"True: {y_true[idx]}\nPred: {y_pred[idx]}")
        plt.axis('off')
    plt.tight_layout()
    plt.suptitle("Misclassified Digits", y=1.02)
    plt.show()

show_misclassified(x_test, y_test, y_pred_relu)          #Mistakes of each model should be displayed separately.
show_misclassified(x_test, y_test, y_pred_tanh)
```

Figure18- python codes for Displaying example of Mistakes

In this piece of code, it prints the labels that have been incorrectly predicted by examining epoch 5.



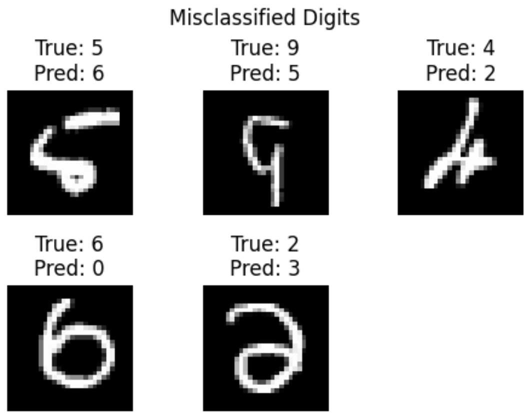


Figure19- Examples of Mistakes

Conclusion:

In summary, after creating a neural network model based on the MNIST dataset and applying two activation functions, Tanh and ReLU, for handwritten digit recognition, it was concluded that both models perform satisfactorily in terms of training and validation accuracy as well as reducing the error rate in predictions and recognizing image patterns. They showed that simple neural networks can perform well in recognizing handwritten digits. Some differences were observed, including: the speed of learning, stability, validation, and resistance to noise in the **ReLU** based model is much higher. In contrast, the **Tanh** based model has high sensitivity to input values and gradients due to its declining performance until the end of the period.

TASK 2

Critical Evaluation Report

❖ Summary and Technical evaluation of the neural network model

In this study, a simple neural network model based on Sequential was created for handwritten digit classification. The inputs were taken from the **MNIST** dataset, reshaped into a **one-dimensional** vector with **784** features. This model includes **3** connected input layers and finally an output layer based on the softmax function with 10 neurons. Our goal is to compare the performance of two activation functions **ReLU** and **Tanh** in the hidden layer. Therefore, two activation functions, ReLU and Tanh, were applied to the model, and accuracy and error size were evaluated.

In the result of assessment, it concludes:

- **Technical and Performance Evaluation:**

The ReLU-based model has a higher training accuracy compared to the **Tanh** model. In terms of validation accuracy**, ReLU** remained at **97.8%,** while **Tanh** reached around **97.7%** and remained stable at this level with less fluctuation. Based on the error charts, there are signs of Overfitting in the **ReLU** model, as the validation error in the ReLU model showed more fluctuations. Overall, both models performed well, but ReLU had faster convergence and generalized slightly better, and Tanh has greater stability.

- **Strengths of the Project**

• Implementation of various activation functions to gain a deeper understanding of performance

• Systematic and coherent design

• Proper preparation of the dataset (**normalize, one-hot encoding**)

• Accurate assessment with test and validation data

- **Limitations and Considerations**

As mentioned, the model has been tested on a standard dataset (**MNIST**). If the model is applied to real data with noise, it may make different results. Then it may require further tuning of **hyperparameters** (**number of neurons, batch size, number of epochs**) to achieve acceptable performance on other datasets. Finally emphasis and focus on examining **overfitting/ underfitting** can be done more accurately (for example, using dropout).

❖ Moral and social considerations

Alongside the technical assessment, the ethical and social implications of artificial intelligence models must be considered, as these models encompass all aspects related to the privacy of individuals in society .Any use of technology must be accompanied by consideration of its social and ethical implications (Floridi, 2013).

1) Input bias and Equity:

Real data in the real world is much more diverse and sensitive. Such as the age of individuals, level of education, hereditary traits, physical and mental disabilities, etc. If the model is trained only with a set of simple, predetermined, and non-diverse data, it will become biased when faced with diverse and real data and will not be able to generalize well to data outside the scope of what it has been trained on. In the meantime, justice may be called into question because the results may be one-sided. Therefore, when creating and training models, especially in real-world applications, attention to data diversity, identifying biases, and applying techniques is essential.

2) Privacy and data protection

In the real world, all forms of bank documents, administrative papers, medical records, include identity information, financial data, and sensitive data such as signatures, identification numbers, and financial information related to individuals. Transparency and explain ability. In these cases, considering users' trust in the model to provide their important information, the model must make every effort to safeguard privacy and protect individuals' data from access and misuse by opportunistic individuals. This issue is of particular importance in countries with strict laws, the

European countries, under the GDPR law. As a result, when designing systems that use handwritten data, attention to these matters is very important:

Use of anonymization and coding methods to prevent the identification of individuals' data.

Prior consent of users for accessing their data.

Adoption and implementation of security measures to protect users' data. Necessary security measures should be applied to protect the data.

3) Transparency and explain ability

Given the existence of trust in decision-making and presenting the results of the model's solution, the model must clearly and understandably explain and clarify the reasons for its decisions. In many areas with sensitive applications such as finance, medicine, or legal matters, lack of transparency and ambiguity can create new problems. The model must always provide a logical explanation to convince users, for example, the reason for denying a loan application. Cases that the model should not allow:

- Erosion of user trust
- The organization may face legal or ethical complaints.
- Violation of regulations including **GDPR**.

4) Responsibility

In situations where the model makes a vital and sensitive decision, accountability for the decision made is very important. For example, a medical, legal, financial decision, etc. The lack of accountability leads to public distrust, legal issues, and even financial and psychological harm to users.

❖ Recommendations for Deployment

- **Training the model for greater flexibility when facing various data types in the real world**

  Training the model in real environments with real and noisy data, and enhancing its ability to better adapt to the data in terms of structure and quality, causes the model to be less confused when operating in the real world and to have a more stable performance.

- **Evaluating the model's performance by implementing supervisory mechanisms**

  Implementing supervisory mechanisms such as frequent testing, cross-validation, and monitoring helps increase the speed of problem detection and prevent performance degradation of the model.

- **Having the capability of Explainable AI**

  In applications related to health, finance, or law, using techniques to make the model's output more explainable and humanized increases the trust of users and managers in the model.

- **Emphasizing compliance with privacy regulations (such as GDPR)**

  Paying attention to user data privacy and complying with international laws such as GDPR, (General Data Protection Regulation of the European Union), which set requirements for the collection, storage, and processing of data, is very important in designing a professional model to prevent legal issues in the future.

Conclusion:

Although the model created based on the **MNIST** dataset has a good technical performance, and in the evaluation stages, it has obtained an acceptable score in terms of accuracy, validity and error rate. However, when designing a model, especially models based on neural networks, ethical, legal and social issues must be considered. Because when a model observes all aspects of security and legality, both technically and ethically, it gains stability and trust, and at the same time, greater durability in the real world.

Things to consider:

- Capable of using a variety of data types
- Being able to explain when using AI techniques
- Ensuring compliance with privacy regulations
- Considering and examining all social and economic impacts of models

Critical evaluations are to ensure that the models are responsible and reliable, as well as accurate and error-free**.**

References:

- LeCun, Y., Cortes, C. and Burges, C.J.C., 1998. *The MNIST database of handwritten digits*. New York: NYU.
- Goodfellow, I., Bengio, Y. and Courville, A., 2016. Deep Learning. Cambridge, MA: MIT Press.
- Müller, A. and Guido, S., (2016). Introduction to Machine Learning with Python: A Guide for Data Scientists. Sebastopol, CA: O'Reilly Media
- Floridi, L., 2013. The ethics of information. Oxford: Oxford University Press.

Code Appendix

```
pip install tensorflow

import tensorflow as tf                      # To build and train machine learning models

import numpy as np

from tensorflow.keras.datasets import mnist              # Importing MINST data set

from tensorflow.keras.models import Sequential           # Importing Sequential Model

from tensorflow.keras.layers import Dense, Flatten       # Importing Layers

from tensorflow.keras.utils import to_categorical        # Convert numeric labels to vectors
one-hot

import matplotlib.pyplot as plt

from keras import Input

from sklearn.metrics import confusion_matrix

import seaborn as sns


# Loading Dataset

(x_train, y_train), (x_test, y_test) = mnist.load_data()


# Display data shapes

print("Training data shape:", x_train.shape)

print("Test data shape:", x_test.shape)

# Display sample images

def show_sample_images(X, y, n=10):

    plt.figure(figsize=(10,2))

    for i in range(n):

        plt.subplot(1,n,i+1)

        plt.imshow(X[i], cmap='gray')

        plt.title(f"Label: {y[i]}")

        plt.axis('off')

    plt.show()
```

```
show_sample_images(x_train, y_train)

# If you intend to use ReLU, use the range [0, 1]

x_train, x_test = x_train / 255.0, x_test / 255.0


# Converting labels to one-hot

y_train_cat = to_categorical(y_train, 10)

y_test_cat = to_categorical(y_test, 10)

def build_model(activation_fn):                              # Definition of a function

    model = Sequential()                                     # A simple linear model in
Keras

    model.add(Flatten(input_shape=(28, 28)))                 # reshape

    model.add(Dense(128, activation=activation_fn))          # creating the first layer

    model.add(Dense(64, activation=activation_fn))           # creating the second layer

    model.add(Dense(32, activation=activation_fn))           # creating the third layer

    model.add(Dense(10, activation='softmax'))               # Probability output

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

    return model


model_relu = build_model('relu')      # calling

model_relu.summary()                  # summarizing
```

```
history_relu = model_relu.fit(x_train, y_train_cat, epochs=10, validation_data=(x_test, y_test_cat), verbose=2)   # Training


# Prediction and evaluation of the ReLU model

y_pred_prob_relu = model_relu.predict(x_test)

y_pred_relu = np.argmax(y_pred_prob_relu, axis=1)

accuracy_relu = np.sum(y_pred_relu == y_test) / len(y_test)

print(f"Accuracy with ReLU: {accuracy_relu:.4f}")

model_tanh = build_model('tanh')

model_tanh.summary()

history_tanh = model_tanh.fit(x_train, y_train_cat, epochs=10, validation_data=(x_test, y_test_cat), verbose=2)


# Prediction and evaluation of the Tanh model

y_pred_prob_tanh = model_tanh.predict(x_test)

y_pred_tanh = np.argmax(y_pred_prob_tanh, axis=1)

accuracy_tanh = np.sum(y_pred_tanh == y_test) / len(y_test)

print(f"Accuracy with Tanh: {accuracy_tanh:.4f}")


# Confusion Matrix

def plot_confusion(y_true, y_pred, title):

    cm = confusion_matrix(y_true, y_pred)

    plt.figure(figsize=(5,5))

    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")

    plt.title(f'Confusion Matrix - {title}')

    plt.xlabel('Predicted')

    plt.ylabel('True')

    plt.show()
```

```
plot_confusion(y_test, y_pred_relu, "ReLU")

plot_confusion(y_test, y_pred_tanh, "Tanh")

def plot_history(history, activation_name):

    # Training Accuracy and Validation

    plt.plot(history.history['accuracy'], label='Train Accuracy')

    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')

    plt.title(f'Accuracy - Activation: {activation_name}')

    plt.xlabel('Epoch')

    plt.ylabel('Accuracy')

    plt.legend()

    plt.show()


    # Loss during training and validation

    plt.plot(history.history['loss'], label='Train Loss')

    plt.plot(history.history['val_loss'], label='Validation Loss')

    plt.title(f'Loss - Activation: {activation_name}')

    plt.xlabel('Epoch')

    plt.ylabel('Loss')

    plt.legend()

    plt.show()


plot_history(history_relu, 'ReLU')

plot_history(history_tanh, 'Tanh')

# Displaying Examples of Mistake
```

```
def show_misclassified(X, y_true, y_pred, n=5):

    wrong = np.where(y_true != y_pred)[0]                 #Comparison of true labels with predicted labels

    plt.figure(figsize=(5,5))

    for i, idx in enumerate(wrong[:n]):

        plt.subplot(3,3,i+1)

        plt.imshow(X[idx], cmap='gray')                    # print lables

        plt.title(f"True: {y_true[idx]}\nPred: {y_pred[idx]}")

        plt.axis('off')

    plt.tight_layout()

    plt.suptitle("Misclassified Digits", y=1.02)

    plt.show()


show_misclassified(x_test, y_test, y_pred_relu)          #Mistakes of each model should be displayed separately.

show_misclassified(x_test, y_test, y_pred_tanh)
```