Experiment No. 3	
BE (AI&DS)	ROLL NO : 9742
Date of Implementation: 06/08/2024	
Aim: To observe the impact of different activation functions on the outcome of the neural network.	
Programming Language Used : Python	
Upon completion of this experiment, students will be able to LO1 : Implement basic neural network model for a given a problem.	

Indicator		
Timeline Maintains submission deadline (1)	On time (1)	Otherwise (0)
Completion and Organization (2)	Completed in LAB (2)	Otherwise(1)
Analysis of output and conclusion(2)	Properly done (2)	Otherwise (0)
Viva (10)		

Assessment Marks :

Timeline(1)	
Completion and Organization (2)	
Analysis of output and conclusion(2)	
Viva (10)	
Total (15)	

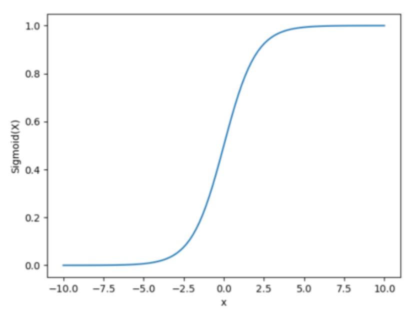
EXPERIMENT	***3***
Aim	To observe the impact of different activation functions on the outcome of the neural network.
Tools	PYTHON
Theory	An activation function in the context of neural networks is a mathematical function applied to the output of a neuron. The purpose of an activation function is to introduce non-linearity into the model, allowing the network to learn and represent complex patterns in the data. The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it.

Variants of Activation Function

Linear Function

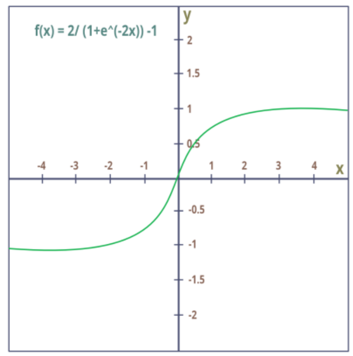
Linear function has the equation similar to as of a straight line i.e. y = x. No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer.

Sigmoid Function



It is a function which is plotted as ‘S’ shaped graph. Equation : $A = 1/(1 + e^{-x})$ Nature : Non-linear. Value Range : 0 to 1. Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be *1* if value is greater than 0.5 and *0* otherwise.

Tanh Function



The activation that works almost always better than sigmoid function is Tanh function also known as Tangent Hyperbolic function. It's actually mathematically shifted version of the sigmoid function.

$$f(x) = \tanh(x) = 2/(1 + e{-}2x) - 1$$

OR

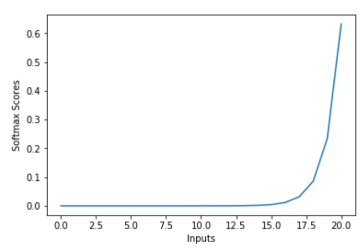
$$\tanh(x) = 2 * \text{sigmoid}(2x) - 1$$

Value Range :- -1 to +1 Nature :- non-linear

RELU Function

It Stands for *Rectified linear unit*. It is the most widely used activation function. Chiefly implemented in *hidden layers* of Neural network. Equation :- *A(x) = max(0,x)*. It gives an output x if x is positive and 0 otherwise. Value Range :- [0, inf) Nature :- non-linear ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.

Softmax Function



The softmax function is also a type of sigmoid function but is handy when we are trying to handle multi- class classification problems. Nature :- non-linear. The softmax function was commonly found in the output layer of image classification problems.

Implementation	1: Import the necessary libraries. (tensorflow, keras, Sequential, Flatten, Dense, Activation) 2: Download the dataset. CIFAR-10 dataset (For few students)

	Fashion MNIST dataset (For few students) 3: Understand the structure of the dataset 4: Visualize the data. 5: Form the Input, hidden, and output layers. The Sequential model allows us to create models layer-by-layer as we need in a multi-layer perceptron and is limited to single-input, single-output stacks of layers. Flatten flattens the input provided without affecting the batch size. For example, If inputs are shaped (batch_size,) without a feature axis, then flattening adds an extra channel dimension and output shape is (batch_size, 1). Activation is for using the different activation function. The first two Dense layers are used to make a fully connected model and are the hidden layers. The last Dense layer is the output layer which contains neurons that decide which category the image belongs to. 6: Compile the model. 7: Fit the model. 8: Find Accuracy of the model. try to use different activation functions like linear, sigmoid, tanh, RELU, leaky RELU and softmax and note the impact on the accuracy model.
Conclusion	Discuss the impact of activation function on accuracy. In this experiment, we evaluated the performance of various activation functions in a neural network using the Fashion MNIST dataset. The aim was to understand how different activation functions impact the accuracy of the model. Results: • **Linear Activation:** Achieved an accuracy of 82.55%. The linear activation function provided a lower accuracy compared to the non-linear functions, demonstrating its limitations in capturing complex patterns due to its lack of non-linearity. • **Sigmoid Activation:** Obtained an accuracy of 87.75%. The sigmoid function, known for its non-linear characteristics, performed well in capturing non-linear relationships, resulting in the highest accuracy among the tested functions. • **Tanh Activation:** Recorded an accuracy of 87.5%. The tanh function also performed well, slightly below sigmoid, but still effectively handled non-linearity and provided strong performance.

	• **ReLU Activation:** Achieved an accuracy of 87.14%. The ReLU function, which is computationally efficient and introduces non-linearity, performed similarly to tanh and sigmoid, though slightly lower in accuracy. • **Softmax Activation:** Resulted in an accuracy of 77.73%. The softmax function, used in the output layer for multi-class classification, was less effective in this case when used in the hidden layers, reflecting its suitability primarily for output layers rather than hidden layers.

Implementation:

[1]:
```
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Flatten, Dense, Activation
from tensorflow.keras.datasets import fashion_mnist
```

[2]:
```
# Step 2: Download and prepare the Fashion MNIST dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
29515/29515            0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26421880/26421880            0s
0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
5148/5148            0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4422102/4422102            0s
0us/step
```

[3]:
```
# Normalize pixel values to be between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0
```

[4]:
```
# Step 3: Understand the structure of the dataset
print(f"Training data shape: {x_train.shape}")
print(f"Test data shape: {x_test.shape}")
print(f"Number of classes: {len(set(y_train))}")
```

```
Training data shape: (60000, 28, 28)
Test data shape: (10000, 28, 28)
Number of classes: 10
```

[6]:
```python
# Step 4: Visualize the data
import matplotlib.pyplot as plt
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal',
  'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[y_train[i]])
plt.show()
```

[7]:
```python
# Step 5: Form the Input, Hidden, and Output Layers
def create_model(activation_function):
    model = Sequential([
        Flatten(input_shape=(28, 28)),  # Fashion MNIST input shape
        Dense(128, activation=activation_function),
        Dense(64, activation=activation_function),
        Dense(10, activation='softmax')  # Output layer for classification
    ])
    return model
```

[8]:
```python
# Step 6: Compile the Model
def compile_model(model):
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
```

[9]:
```python
# Step 7: Fit the Model
def fit_model(model, x_train, y_train, epochs=10):
    history = model.fit(x_train, y_train, epochs=epochs, validation_split=0.2,␣
 ↪verbose=2)
    return history
```

[14]:
```python
# Step 8: Find Accuracy of the Model
activation_functions = ['linear', 'sigmoid', 'tanh', 'relu', 'softmax']
accuracies = []  # Initialize a list to store accuracies

for activation_function in activation_functions:
    print(f"\nTraining with {activation_function} activation function")
    model = create_model(activation_function)
    compile_model(model)
    history = fit_model(model, x_train, y_train)
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
    print(f"Test accuracy with {activation_function}: {test_acc}")
    accuracies.append(test_acc) # Append the accuracy to the list
```

```
Training with linear activation function
```

```
/usr/local/lib/python3.10/dist-
packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

```
Epoch 1/10
1500/1500 - 7s - 5ms/step - accuracy: 0.8066 - loss: 0.5555 - val_accuracy:
```

```
0.8334 - val_loss: 0.4724
Epoch 2/10
1500/1500 - 7s - 5ms/step - accuracy: 0.8337 - loss: 0.4775 - val_accuracy:
0.8306 - val_loss: 0.5015
Epoch 3/10
1500/1500 - 10s - 7ms/step - accuracy: 0.8420 - loss: 0.4552 - val_accuracy:
0.8423 - val_loss: 0.4731
Epoch 4/10
1500/1500 - 6s - 4ms/step - accuracy: 0.8438 - loss: 0.4436 - val_accuracy:
0.8478 - val_loss: 0.4363
Epoch 5/10
1500/1500 - 7s - 4ms/step - accuracy: 0.8473 - loss: 0.4370 - val_accuracy:
0.8465 - val_loss: 0.4427
Epoch 6/10
1500/1500 - 7s - 5ms/step - accuracy: 0.8495 - loss: 0.4283 - val_accuracy:
0.8320 - val_loss: 0.4703
Epoch 7/10
1500/1500 - 5s - 3ms/step - accuracy: 0.8529 - loss: 0.4232 - val_accuracy:
0.8533 - val_loss: 0.4282
Epoch 8/10
1500/1500 - 10s - 7ms/step - accuracy: 0.8527 - loss: 0.4183 - val_accuracy:
0.8497 - val_loss: 0.4328
Epoch 9/10
1500/1500 - 7s - 5ms/step - accuracy: 0.8557 - loss: 0.4145 - val_accuracy:
0.8488 - val_loss: 0.4350
Epoch 10/10
1500/1500 - 6s - 4ms/step - accuracy: 0.8555 - loss: 0.4102 - val_accuracy:
0.8342 - val_loss: 0.4688
313/313 - 1s - 2ms/step - accuracy: 0.8255 - loss: 0.4976
Test accuracy with linear: 0.8255000114440918

Training with sigmoid activation function
Epoch 1/10
1500/1500 - 7s - 4ms/step - accuracy: 0.7811 - loss: 0.6618 - val_accuracy:
0.8402 - val_loss: 0.4366
Epoch 2/10
1500/1500 - 7s - 5ms/step - accuracy: 0.8549 - loss: 0.4037 - val_accuracy:
0.8618 - val_loss: 0.3849
Epoch 3/10
1500/1500 - 10s - 7ms/step - accuracy: 0.8705 - loss: 0.3624 - val_accuracy:
0.8633 - val_loss: 0.3835
Epoch 4/10
1500/1500 - 8s - 5ms/step - accuracy: 0.8781 - loss: 0.3362 - val_accuracy:
0.8780 - val_loss: 0.3428
Epoch 5/10
1500/1500 - 7s - 5ms/step - accuracy: 0.8844 - loss: 0.3176 - val_accuracy:
0.8789 - val_loss: 0.3328
Epoch 6/10
```

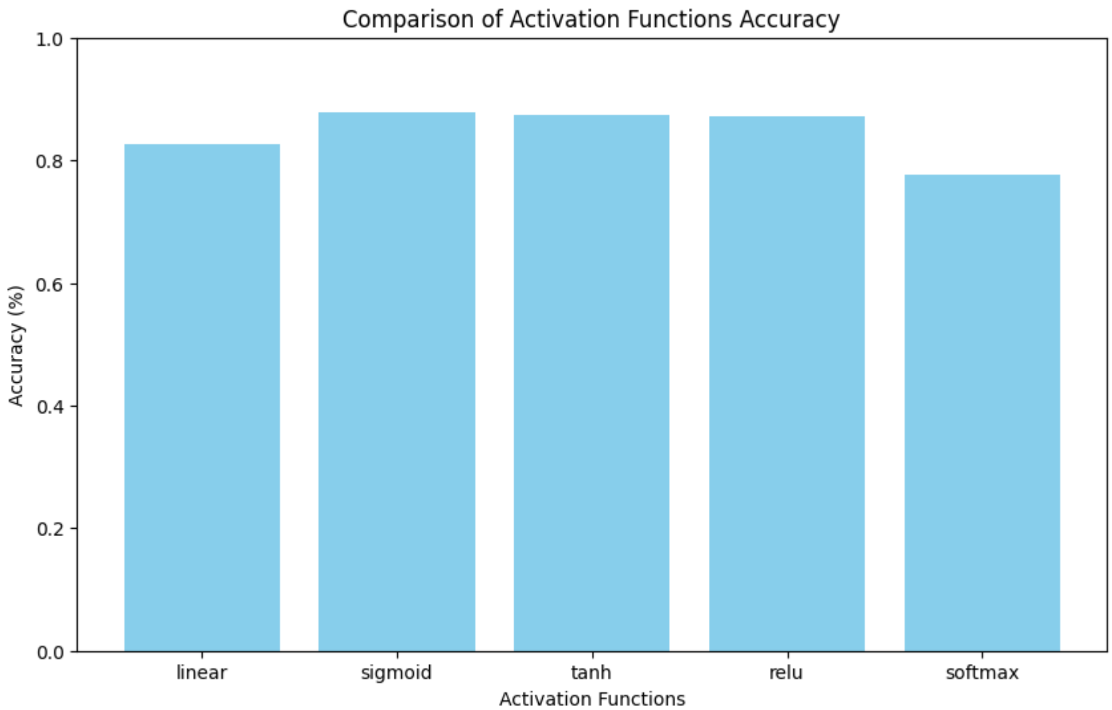
```
1500/1500 - 9s - 6ms/step - accuracy: 0.8898 - loss: 0.3014 - val_accuracy:
0.8842 - val_loss: 0.3204
Epoch 7/10
1500/1500 - 10s - 7ms/step - accuracy: 0.8940 - loss: 0.2881 - val_accuracy:
0.8807 - val_loss: 0.3273
Epoch 8/10
1500/1500 - 7s - 5ms/step - accuracy: 0.8991 - loss: 0.2759 - val_accuracy:
0.8857 - val_loss: 0.3121
Epoch 9/10
1500/1500 - 10s - 7ms/step - accuracy: 0.9017 - loss: 0.2655 - val_accuracy:
0.8862 - val_loss: 0.3151
Epoch 10/10
1500/1500 - 8s - 6ms/step - accuracy: 0.9070 - loss: 0.2528 - val_accuracy:
0.8873 - val_loss: 0.3140
313/313 - 1s - 2ms/step - accuracy: 0.8775 - loss: 0.3472
Test accuracy with sigmoid: 0.8774999976158142

Training with tanh activation function
Epoch 1/10
1500/1500 - 8s - 5ms/step - accuracy: 0.8252 - loss: 0.4871 - val_accuracy:
0.8521 - val_loss: 0.4142
Epoch 2/10
1500/1500 - 6s - 4ms/step - accuracy: 0.8624 - loss: 0.3748 - val_accuracy:
0.8733 - val_loss: 0.3503
Epoch 3/10
1500/1500 - 6s - 4ms/step - accuracy: 0.8758 - loss: 0.3362 - val_accuracy:
0.8684 - val_loss: 0.3591
Epoch 4/10
1500/1500 - 6s - 4ms/step - accuracy: 0.8845 - loss: 0.3135 - val_accuracy:
0.8804 - val_loss: 0.3297
Epoch 5/10
1500/1500 - 8s - 5ms/step - accuracy: 0.8886 - loss: 0.2978 - val_accuracy:
0.8791 - val_loss: 0.3347
Epoch 6/10
1500/1500 - 10s - 7ms/step - accuracy: 0.8947 - loss: 0.2827 - val_accuracy:
0.8792 - val_loss: 0.3312
Epoch 7/10
1500/1500 - 10s - 7ms/step - accuracy: 0.8986 - loss: 0.2739 - val_accuracy:
0.8747 - val_loss: 0.3404
Epoch 8/10
1500/1500 - 6s - 4ms/step - accuracy: 0.9012 - loss: 0.2636 - val_accuracy:
0.8819 - val_loss: 0.3286
Epoch 9/10
1500/1500 - 7s - 5ms/step - accuracy: 0.9053 - loss: 0.2528 - val_accuracy:
0.8856 - val_loss: 0.3105
Epoch 10/10
1500/1500 - 9s - 6ms/step - accuracy: 0.9093 - loss: 0.2419 - val_accuracy:
0.8823 - val_loss: 0.3244
```

```
313/313 - 1s - 2ms/step - accuracy: 0.8749 - loss: 0.3566
Test accuracy with tanh: 0.8748999834060669

Training with relu activation function
Epoch 1/10
1500/1500 - 8s - 5ms/step - accuracy: 0.8154 - loss: 0.5189 - val_accuracy:
0.8486 - val_loss: 0.4098
Epoch 2/10
1500/1500 - 11s - 7ms/step - accuracy: 0.8605 - loss: 0.3792 - val_accuracy:
0.8673 - val_loss: 0.3720
Epoch 3/10
1500/1500 - 5s - 4ms/step - accuracy: 0.8749 - loss: 0.3420 - val_accuracy:
0.8695 - val_loss: 0.3629
Epoch 4/10
1500/1500 - 11s - 7ms/step - accuracy: 0.8801 - loss: 0.3231 - val_accuracy:
0.8692 - val_loss: 0.3642
Epoch 5/10
1500/1500 - 10s - 6ms/step - accuracy: 0.8886 - loss: 0.3012 - val_accuracy:
0.8743 - val_loss: 0.3574
Epoch 6/10
1500/1500 - 7s - 4ms/step - accuracy: 0.8924 - loss: 0.2879 - val_accuracy:
0.8808 - val_loss: 0.3278
Epoch 7/10
1500/1500 - 6s - 4ms/step - accuracy: 0.8994 - loss: 0.2726 - val_accuracy:
0.8695 - val_loss: 0.3628
Epoch 8/10
1500/1500 - 7s - 5ms/step - accuracy: 0.9016 - loss: 0.2608 - val_accuracy:
0.8824 - val_loss: 0.3270
Epoch 9/10
1500/1500 - 9s - 6ms/step - accuracy: 0.9055 - loss: 0.2515 - val_accuracy:
0.8808 - val_loss: 0.3466
Epoch 10/10
1500/1500 - 7s - 5ms/step - accuracy: 0.9093 - loss: 0.2439 - val_accuracy:
0.8814 - val_loss: 0.3421
313/313 - 1s - 2ms/step - accuracy: 0.8714 - loss: 0.3664
Test accuracy with relu: 0.871399998664856

Training with softmax activation function
Epoch 1/10
1500/1500 - 9s - 6ms/step - accuracy: 0.4897 - loss: 1.8767 - val_accuracy:
0.5660 - val_loss: 1.2919
Epoch 2/10
1500/1500 - 9s - 6ms/step - accuracy: 0.5723 - loss: 1.0781 - val_accuracy:
0.5757 - val_loss: 0.9840
Epoch 3/10
1500/1500 - 7s - 5ms/step - accuracy: 0.5817 - loss: 0.9277 - val_accuracy:
0.5893 - val_loss: 0.9219
Epoch 4/10
```

```
1500/1500 - 11s - 7ms/step - accuracy: 0.5933 - loss: 0.8864 - val_accuracy:
0.6028 - val_loss: 0.8960
Epoch 5/10
1500/1500 - 5s - 4ms/step - accuracy: 0.6095 - loss: 0.8645 - val_accuracy:
0.6289 - val_loss: 0.8703
Epoch 6/10
1500/1500 - 11s - 7ms/step - accuracy: 0.7003 - loss: 0.7840 - val_accuracy:
0.7238 - val_loss: 0.7079
Epoch 7/10
1500/1500 - 7s - 5ms/step - accuracy: 0.7421 - loss: 0.6391 - val_accuracy:
0.7457 - val_loss: 0.6274
Epoch 8/10
1500/1500 - 6s - 4ms/step - accuracy: 0.7533 - loss: 0.5874 - val_accuracy:
0.7587 - val_loss: 0.5843
Epoch 9/10
1500/1500 - 8s - 5ms/step - accuracy: 0.7766 - loss: 0.5479 - val_accuracy:
0.7747 - val_loss: 0.5636
Epoch 10/10
1500/1500 - 8s - 6ms/step - accuracy: 0.7902 - loss: 0.5157 - val_accuracy:
0.7793 - val_loss: 0.5474
313/313 - 1s - 3ms/step - accuracy: 0.7773 - loss: 0.5580
Test accuracy with softmax: 0.7773000001907349
```

[15]:

```python
# Plot the graph comparing activation functions' accuracy
plt.figure(figsize=(10, 6))
plt.bar(activation_functions, accuracies, color='skyblue')
plt.xlabel('Activation Functions')
plt.ylabel('Accuracy (%)')
plt.title('Comparison of Activation Functions Accuracy')
plt.ylim([0, 1])  # Set y-axis range from 0 to 1 (accuracy percentage)
plt.show()
```


Comparison of Activation Functions Accuracy
Accuracy (%)
1.0
0.8
0.6
0.4
0.2
0.0
linear
sigmoid
tanh
relu
softmax
Activation Functions