PERCEPTRON

BSDSF21M020

Q1. Search the internet for different activation functions(at least 4) that are used in Single/Multi Layer Perceptrons. Enlist them and also mention what they are mostly used for in a neural network/multilayer perceptron.

It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function).

The Activation Functions can be basically divided into 2 types-

- ✓ Linear Activation Function
- ✓ Non-linear Activation Functions

Activation functions are :

1. Sigmoid or Logistic Activation Function
2. Tanh or hyperbolic tangent Activation Function
3. ReLU (Rectified Linear Unit) Activation Function
4. Leaky ReLU

- **Sigmoid or Logistic Activation Function**
  - Formula:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

  - Range: (0, 1)
  - Common Usage: Historically used in the output layer for binary classification problems, but less common in hidden layers due to the vanishing gradient problem.

- **<u>Hyperbolic Tangent (tanh) Activation Function:</u>**
    - Formula:

$$Tanh$$

$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

    - Range: (-1, 1)
    - Common Usage: Similar to the sigmoid, but with a wider output range. Often used in hidden layers for classification problems.

▪ **<u>ReLU (Rectified Linear Unit) Activation Function</u>**
- Formula:

$$f(x) = \max(0, x)$$

- Range: [0, +∞)
- Common Usage: One of the most popular choices for hidden layers due to its simplicity and effectiveness. It helps with the vanishing gradient problem and accelerates convergence

- **Leaky Rectified Linear Unit Activation Function:**
- Formula:

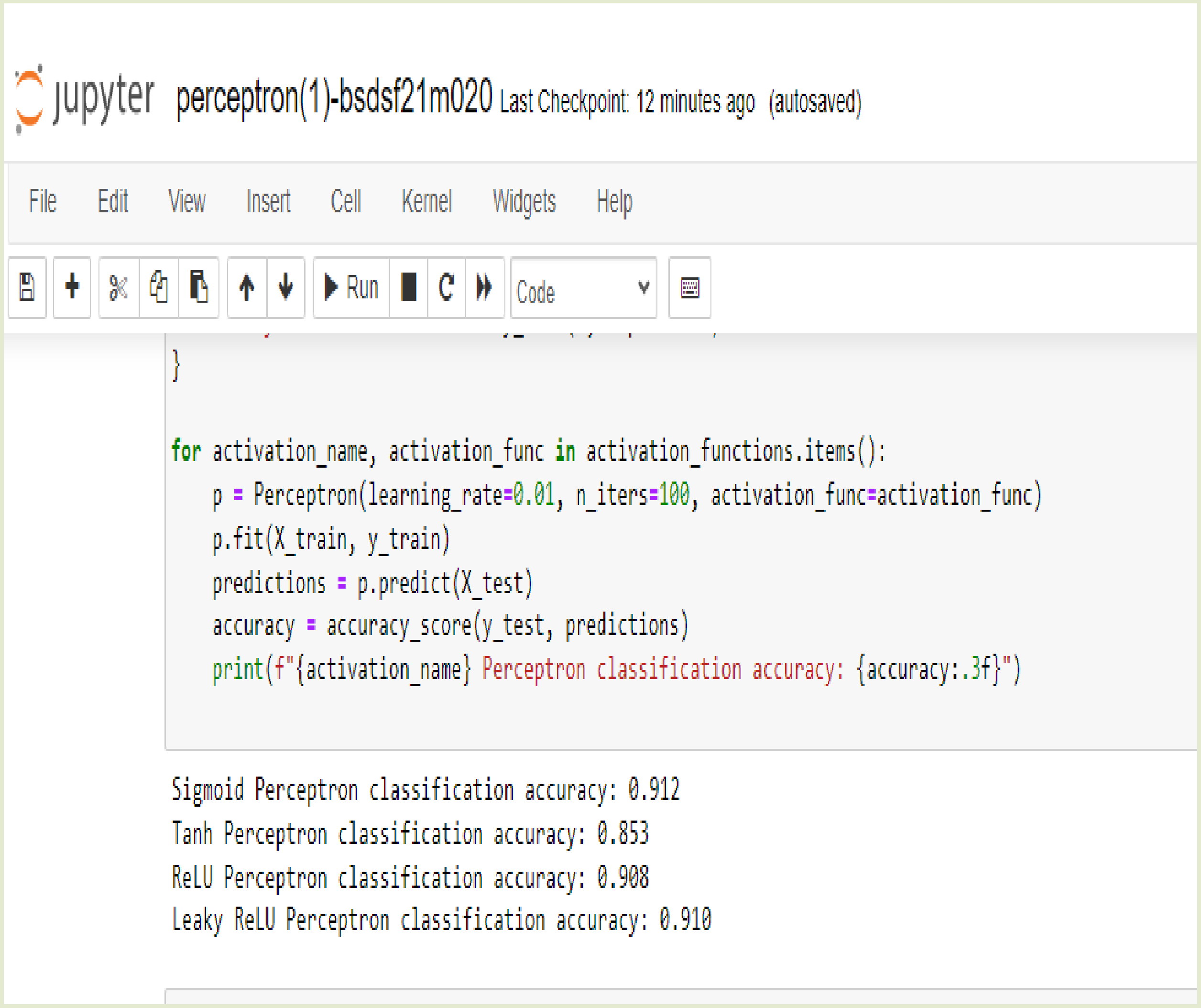
$$leakyrelu(z) = \begin{cases} 0.01z & for\, z < 0 \\ z & for\, z \geq 0 \end{cases}$$

- Range: (-∞, +∞)
- Common Usage: Addresses the "dying ReLU" problem by allowing a small, non-zero gradient for negative inputs. It is a variation of the ReLU and aims to improve learning performance.

Q2. Implement different perceptrons in Python using the activation functions that you have found in the above question. Use the same data to train each of these and then check which perceptron gives you the highest accuracy.

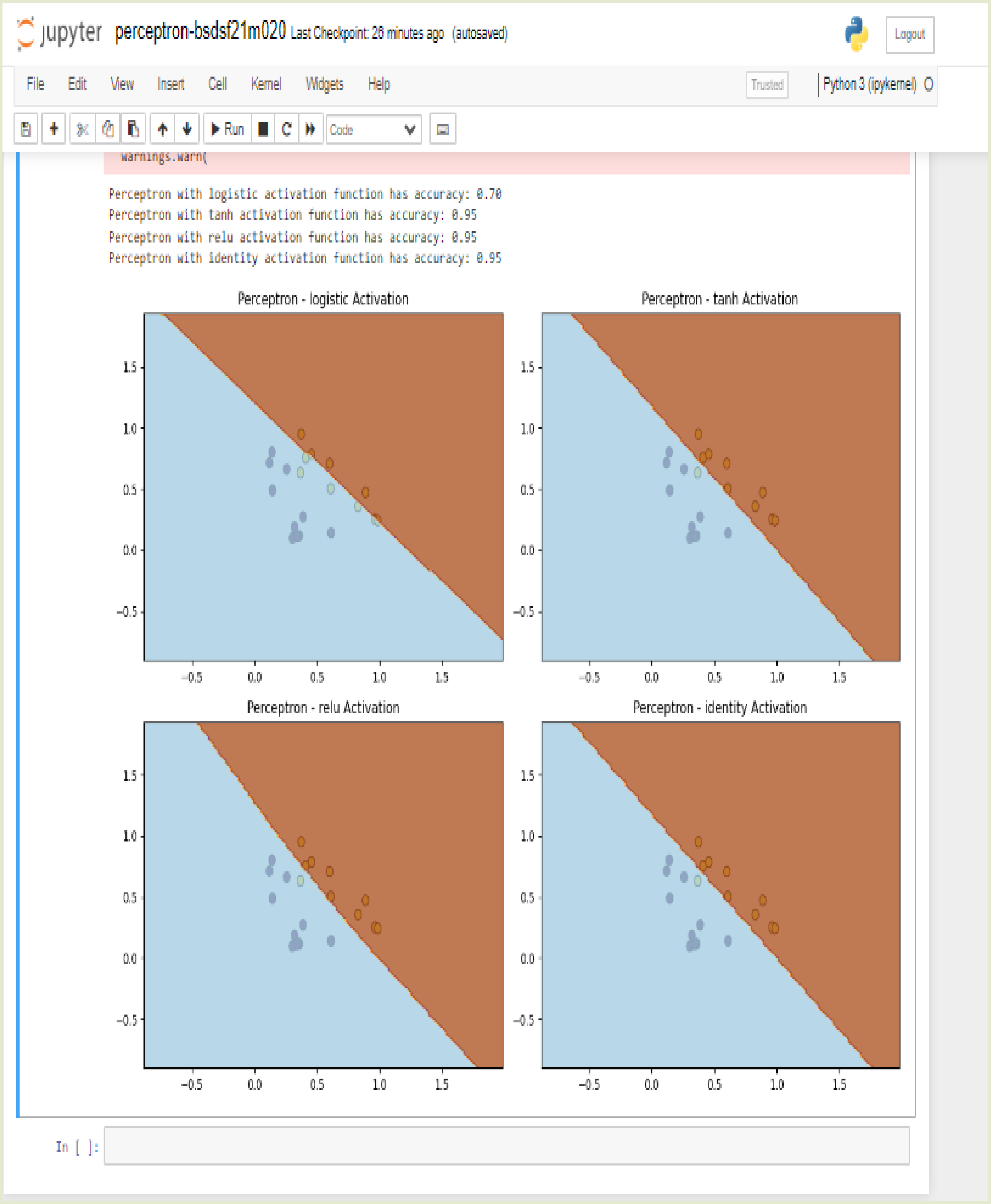
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score

# Assuming you have an activation function like sigmoid
        and step function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def step_function(x):
    return np.where(x > 0, 1, 0)

# Assuming you have a Perceptron class
class Perceptron:
    def __init__(self, learning_rate=0.01, n_iters=100,
              activation_func=sigmoid):
        self.learning_rate = learning_rate
        self.n_iters = n_iters
        self.activation_func = activation_func
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iters):
            linear_model = np.dot(X, self.weights) + self.bias
            predictions = self.activation_func(linear_model)

            # Update weights and bias
            self.weights -= self.learning_rate * (1/n_samples) *
              np.dot(X.T, (predictions - y))
            self.bias -= self.learning_rate * (1/n_samples) *
              np.sum(predictions - y)

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        predictions = self.activation_func(linear_model)
        return np.where(predictions > 0.5, 1, 0)

# Load data using make_classification
X, y = make_classification(
    n_samples=5000, n_features=2, n_informative=2,
              n_redundant=0,
    n_clusters_per_class=1, flip_y=0, random_state=20
)

# Convert labels to binary (ensure they are already binary)
y_binary = (y > 0).astype(int)

X_train, X_test, y_train, y_test = train_test_split(
    X, y_binary, test_size=0.2, random_state=20
)

# Activation functions
activation_functions = {
    "Sigmoid": sigmoid,
    "Step": step_function,
}
```

```
# Train and evaluate perceptrons with different activation
            functions
for activation_name, activation_func in
            activation_functions.items():
   p = Perceptron(learning_rate=0.01, n_iters=100,
            activation_func=activation_func)
   p.fit(X_train, y_train)
   predictions = p.predict(X_test)
   accuracy = accuracy_score(y_test, predictions)
   print(f"{activation_name} Perceptron classification
            accuracy: {accuracy:.3f}")
```


jupyter
perceptron(1)-bsdsf21m020 Last Checkpoint: 12 minutes ago (autosaved)
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Run
Code

```
}

for activation_name, activation_func in activation_functions.items():
    p = Perceptron(learning_rate=0.01, n_iters=100, activation_func=activation_func)
    p.fit(X_train, y_train)
    predictions = p.predict(X_test)
    accuracy = accuracy_score(y_test, predictions)
    print(f"{activation_name} Perceptron classification accuracy: {accuracy:.3f}")
```

```
Sigmoid Perceptron classification accuracy: 0.912
Tanh Perceptron classification accuracy: 0.853
ReLU Perceptron classification accuracy: 0.908
Leaky ReLU Perceptron classification accuracy: 0.910
```


jupyter
perceptron-bsdsf21m020 Last Checkpoint: 28 minutes ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3 (ipykernel)
Run
Code

```
  warnings.warn(
```

```
Perceptron with logistic activation function has accuracy: 0.70
Perceptron with tanh activation function has accuracy: 0.95
Perceptron with relu activation function has accuracy: 0.95
Perceptron with identity activation function has accuracy: 0.95
```

Perceptron - logistic Activation
Perceptron - tanh Activation
Perceptron - relu Activation
Perceptron - identity Activation
1.5
1.0
0.5
0.0
−0.5
−0.5
0.0
0.5
1.0
1.5
In [ ]:

Q3. Research and describe in your own words the following topics
1. Epochs
2. Forward Propagation
3. Backward Propagation
4. Bias in a perceptron
5. XOR implementation using Single Layer Perceptron

Epochs: An epoch is a complete pass over the entire training dataset during the training of a model in the context of machine learning. The model is exposed to each training sample once during each epoch, allowing it to alter its internal parameters (weights and biases) based on the difference between its predictions and the actual target values. Multiple epochs are frequently required to adequately train a model, and the number of epochs is a hyperparameter that can be tweaked during the training process.

Forward propagation:
Forward propagation is the process of processing input data through a neural network layer by layer in order to generate predictions or outputs. Each layer of the network in this procedure conducts a linear transformation (weighted sum of inputs) followed by a non-linear activation function. The output of one layer becomes the input for the next

layer, and so on until the network's output is produced by the last layer. Forward propagation is required for neural network prediction during both the training and testing phases.

Backward propagation:

Backward propagation, also known as backpropagation, is a technique used in neural network training to update the model's parameters (weights and biases) depending on the computed error. It entails calculating the gradient of the loss function with respect to the parameters of the model. This gradient information is then utilized to update the parameters in the gradient's opposite direction, with the goal of minimizing error. The gradients are efficiently computed using the backpropagation method by iteratively applying the chain rule of calculus from the output layer to the input layer.

Bias in a Perceptron:

Bias is an additional parameter that is added to the weighted sum of input characteristics in the setting of a perceptron. It enables the model to take into account instances in which all input features are zero. The bias term essentially alters the decision boundary, allowing the model to more correctly fit the data. It determines the offset or y-intercept of the decision boundary in a geometric sense. Including a bias component in a perceptron increases its ability to simulate a broader set of data interactions.

Implementation of XOR Using a Single Layer Perceptron:

Because of its linear decision boundary, a single-layer perceptron (SLP) cannot properly learn the XOR (exclusive OR) function. The binary operation XOR produces true only when the number of true inputs is odd. Because XOR is not linearly separable, it cannot be correctly modeled by a single-layer perceptron, which generates only linear decision boundaries. A multi-layer perceptron (MLP) or neural network with at least one hidden layer is required to achieve this. The extra layer enables the network to learn complicated, non-linear relationships, allowing for successful modeling of XOR and other non-linear functions.