Neural Network from Scratch

Introduction:

The project involves creating a neural network entirely from scratch using only NumPy and Pandas. The goal is to understand the intricacies of neural network operations by manually implementing the architecture, including input, hidden, and output layers tailored to the specific input-output shape of the data. The custom neural network includes defining an activation function, implementing feedforward and backpropagation algorithms, creating functions to train and test the model then finally comparing it with Sklearn and Keras known models.

Dataset Description:

The model is tested on two datasets: the IRIS dataset and the MNIST dataset. These datasets are chosen for their prevalence in machine learning as example datasets: The IRIS dataset comprises 150 records representing three types of iris flowers. Each record has four features: sepal length, sepal width, petal length, and petal width. These features are used to classify each flower into one of the three species. This dataset is widely used in machine learning for classification problems due to its clear structure and ease of use. The MNIST dataset is a large collection of 70,000 small square 28x28 pixel grayscale images of handwritten digits from 0 to 9. This dataset is commonly used to train and test in the field of machine learning and computer vision. Its complexity is higher than the IRIS dataset because it requires the model to recognize and classify varied handwriting styles.

Methodology:

The methodology for creating the neural network from scratch was both rigorous and detailed, attaching to foundational principles of neural network design. The process began with establishing the network architecture, which included defining the number and size of layers. Special attention was given to the design of the hidden layers to ensure they were capable of capturing complex patterns in data. The choice of activation functions was critical: Sigmoid was used for the hidden layers to introduce non-linearity and efficiency in computations, while the softmax function was employed in the output layer for class probability determination.

Weight initialization played a key role in setting the stage for the learning process. We adopted a random but controlled method to initialize weights, promoting a diverse range of initial predictions and avoiding symmetry in the learning process. This was followed by the implementation of the feedforward and backpropagation algorithms. Feedforward involves passing data through the network, layer by layer, transforming it with weighted sums and activation functions. Backpropagation, a critical step in learning, was used to adjust weights in the network by propagating errors backward, utilizing the gradient of the loss function. The entire training process was conducted over multiple epochs, allowing the network to iteratively learn and improve its accuracy.

Detailed Implementation:

Network Setup

The network setup begins with the initialization of the weights and biases for each layer. Parameters are set according to the size of the input features and

the desired number of neurons in the hidden and output layers. Weights are typically initialized with small random values to break symmetry, and biases are initialized to zero or a small value. This step is critical to starting the training process on the right path.

```
def nn_layers(data, label, hidden_layers):
    """
    Initialize the layers of a neural network.

    Parameters:
    data: The input data, augmented with a column of ones for the bias, shape (M, N+1)
    label: The output labels, shape (M, 1)
    hidden_layers: List containing the number of neurons in each hidden layer

    Returns:
    weights: List of weight matrices for each layer.
    """

    # Number of neurons in each layer
    input_layer_neurons = data.shape[1]  # N+1 (including bias)
    weights = []

    # Initialize weights for the input layer to the first hidden layer
    weights.append(np.random.randn(hidden_layers[0], input_layer_neurons))

    # Initialize weights for subsequent hidden layers
    for i in range(1, len(hidden_layers)):
        layer_weights = np.random.randn(hidden_layers[i], hidden_layers[i-1])
        weights.append(layer_weights)

    # Initialize weights for the output layer
    output_layer_neurons = len(np.unique(label))
    output_weights = np.random.randn(output_layer_neurons, hidden_layers[-1])
    weights.append(output_weights)


  # Round the weights to the specified number of decimals
    for i in range(len(weights)):
        weights[i] = np.round(weights[i], 4)

    return weights
```

Forward Propagation

During forward propagation, input data is passed through the network. At each layer, an affine transformation is performed by multiplying the input with the weights and adding the bias. The result is then passed through an activation function, which introduces non-linearity and enables the network to learn complex patterns.

```python
def forward(weights, record):
    """
    Perform the forward pass of the neural network.

    Parameters:
    W1: Weight matrix from input layer to hidden layer, shape (n_hidden_units, n_input_features)
    W2: Weight matrix from hidden layer to output layer, shape (n_output_units, n_hidden_units)
    record: Input data, shape (n_input_features, n_samples)

    Returns:
    hidden_layer_input: Activations of the hidden layer before applying the activation function, shape (n_hidden_units, n_samples)
    hidden_layer_output: Activations of the output layer after applying the activation function, shape (n_output_units, n_samples)
    """

    # Apply the ReLU activation function to the input data
    # Assuming Relu is a function that applies the ReLU activation element-wise
    relu_input = Relu(record)  # Replace with sigmoid if needed, shape (n_input_features, n_samples)
    hidden_layer_output=[]
    # Calculate the input to the hidden layer
    hidden_layer_output.append(sigmoid(weights[0] @ relu_input))  # Apply sigmoid activation function, shape (n_hidden_units, n_samples)
    for i in range(1,len(weights)-1):
      current=sigmoid(weights[i] @ hidden_layer_output[-1])
      hidden_layer_output.append(current)  # Apply sigmoid activation function, shape (n_output_units, n_samples)


    # Calculate the output from the hidden layer, which serves as input to the output layer
    preductions = sigmoid(weights[-1] @ hidden_layer_output[-1])  # Apply sigmoid activation function, shape (n_output_units, n_samples)

    # Round the weights to the specified number of decimals
    for i in range(len(hidden_layer_output)):
        hidden_layer_output[i] = np.round(hidden_layer_output[i], 4)

    return hidden_layer_output,np.round(preductions,4)
```

Backpropagation and Weight Update:

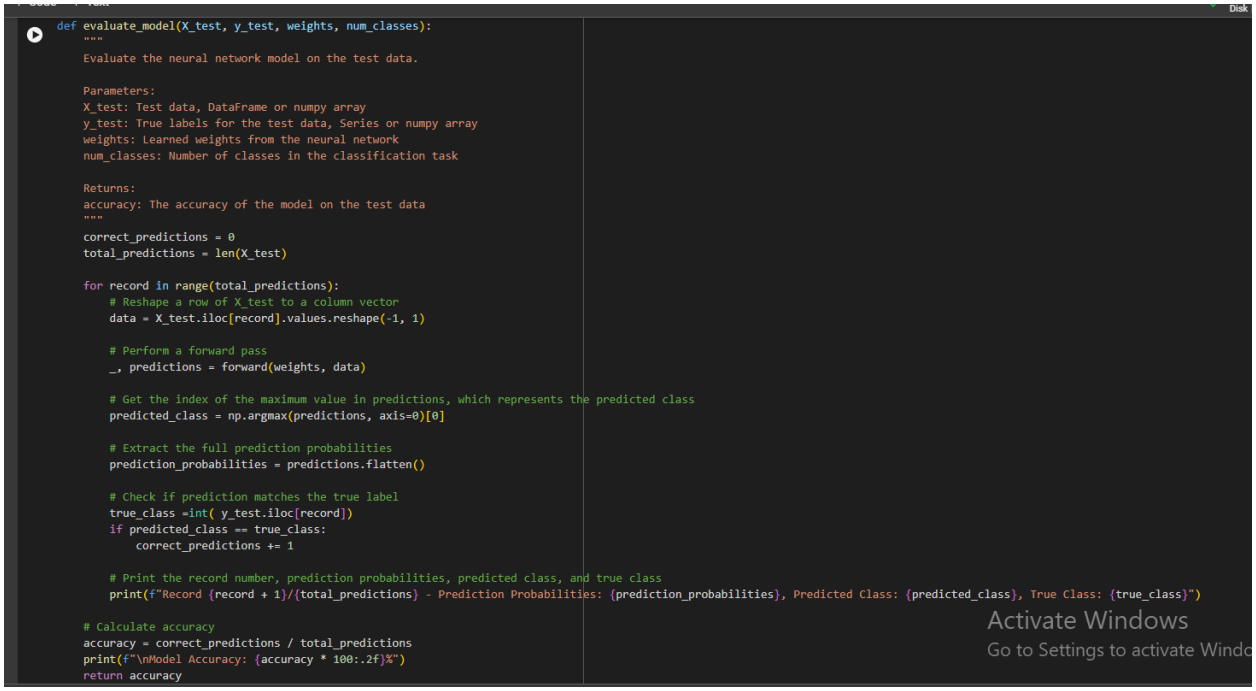
Backpropagation is the process of adjusting the network's weights based on the error of the output. This involves computing the gradient of the loss function with respect to each weight and bias, which indicates the direction in which they should be adjusted to minimize the error. The gradients are then used to update the weights and biases, typically with a learning rate that scales the size of the updates to ensure smooth convergence.

```python
def Backward(input_data, target, learning_rate, weights, hidden_layer_outputs, predictions):
    """
    Perform the backward pass of a neural network, updating the weights dynamically.

    Parameters:
    input_data: Input data, shape (n_input_features, n_samples)
    target: Expected target output, shape (n_output_units, n_samples)
    learning_rate: Scalar value indicating the step size during gradient descent
    weights: List of weight matrices for each layer in the neural network
    hidden_layer_outputs: List of outputs for each hidden layer
    predictions: Final output predictions of the neural network

    Returns:
    Updated weights
    """
    # Initialize delta for the output layer
    delta = predictions * (1 - predictions) * (target - predictions)

    for i in range(len(weights) - 1, 0, -1):
        # Determine the input to the current layer
        layer_input = hidden_layer_outputs[i-1] if i >= 1 else input_data
        # Calculate the weight update for the current layer
        delta_w = learning_rate * (delta @ layer_input.T)
        weights[i] += delta_w
        # Calculate the delta for the next layer
        if i >= 1:
            delta = (weights[i].T @ delta) * hidden_layer_outputs[i-1] * (1 - hidden_layer_outputs[i-1])

    # Update for the first hidden layer
    delta_w = learning_rate * (delta @ input_data.T)
    weights[0] += delta_w
    # Round the weights to the specified number of decimals
    for i in range(len(weights)):
        weights[i] = np.round(weights[i], 4)

    return weights
```

Model Evaluation:

The evaluation of the model involves running the trained network on a test dataset and predicting the output. The predicted classes are compared with the true labels to calculate the accuracy of the model. The details provided in your notebook suggest the use of a function that evaluates the model by calculating the accuracy percentage based on correct predictions over the total number of predictions. The evaluation process likely includes detailed feedback for each

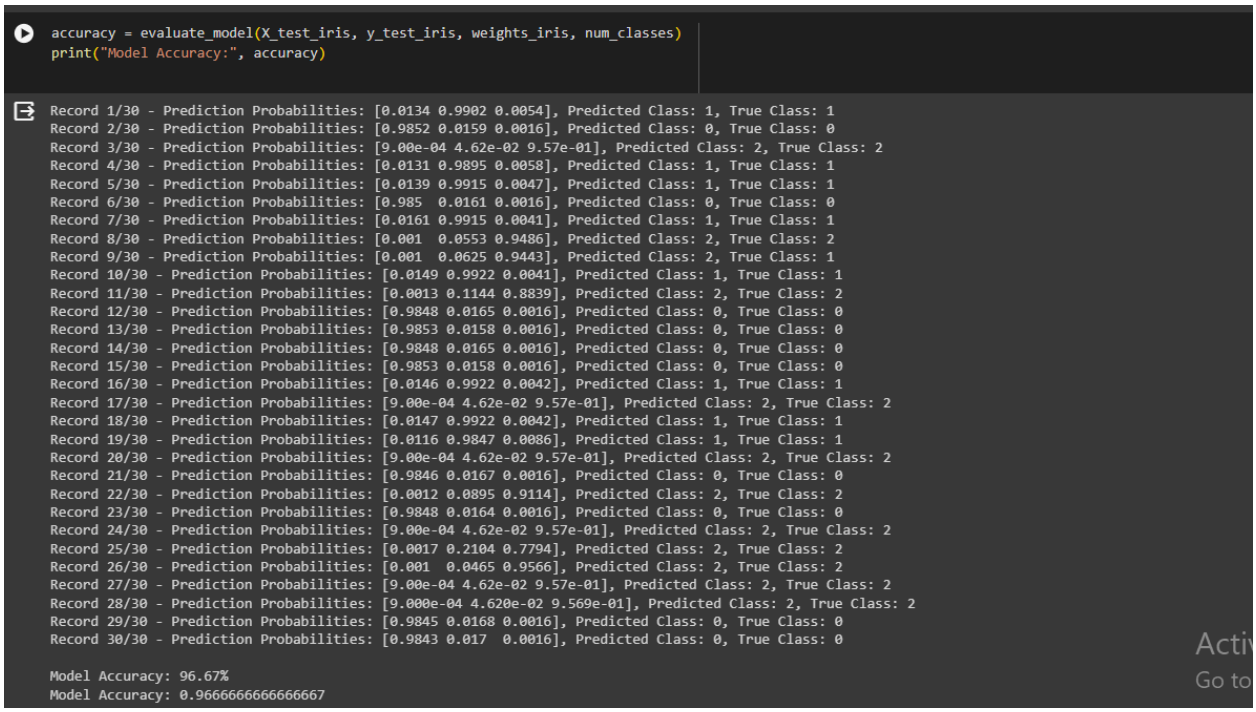
record, presenting the predicted probabilities, the predicted class, and the true class.



```python
def evaluate_model(X_test, y_test, weights, num_classes):
    """
    Evaluate the neural network model on the test data.

    Parameters:
    X_test: Test data, DataFrame or numpy array
    y_test: True labels for the test data, Series or numpy array
    weights: Learned weights from the neural network
    num_classes: Number of classes in the classification task

    Returns:
    accuracy: The accuracy of the model on the test data
    """
    correct_predictions = 0
    total_predictions = len(X_test)

    for record in range(total_predictions):
        # Reshape a row of X_test to a column vector
        data = X_test.iloc[record].values.reshape(-1, 1)

        # Perform a forward pass
        _, predictions = forward(weights, data)

        # Get the index of the maximum value in predictions, which represents the predicted class
        predicted_class = np.argmax(predictions, axis=0)[0]

        # Extract the full prediction probabilities
        prediction_probabilities = predictions.flatten()

        # Check if prediction matches the true label
        true_class =int( y_test.iloc[record])
        if predicted_class == true_class:
            correct_predictions += 1

        # Print the record number, prediction probabilities, predicted class, and true class
        print(f"Record {record + 1}/{total_predictions} - Prediction Probabilities: {prediction_probabilities}, Predicted Class: {predicted_class}, True Class: {true_class}")

    # Calculate accuracy
    accuracy = correct_predictions / total_predictions
    print(f"\nModel Accuracy: {accuracy * 100:.2f}%")
    return accuracy
```

Results:

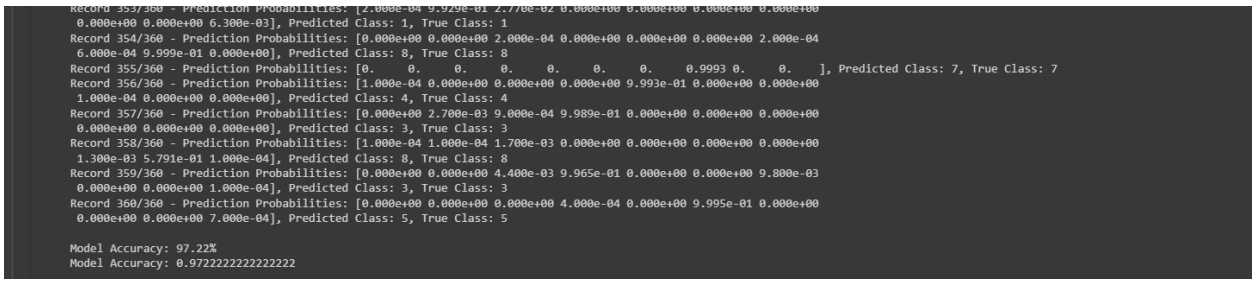
For the iris dataset using one hidden layer of 5 neurons, 1000 iteration and learning_rate equal 0.2 we got 96.67%



```
accuracy = evaluate_model(X_test_iris, y_test_iris, weights_iris, num_classes)
print("Model Accuracy:", accuracy)
```

```
Record 1/30 - Prediction Probabilities: [0.0134 0.9902 0.0054], Predicted Class: 1, True Class: 1
Record 2/30 - Prediction Probabilities: [0.9852 0.0159 0.0016], Predicted Class: 0, True Class: 0
Record 3/30 - Prediction Probabilities: [9.00e-04 4.62e-02 9.57e-01], Predicted Class: 2, True Class: 2
Record 4/30 - Prediction Probabilities: [0.0131 0.9895 0.0058], Predicted Class: 1, True Class: 1
Record 5/30 - Prediction Probabilities: [0.0139 0.9915 0.0047], Predicted Class: 1, True Class: 1
Record 6/30 - Prediction Probabilities: [0.985  0.0161 0.0016], Predicted Class: 0, True Class: 0
Record 7/30 - Prediction Probabilities: [0.0161 0.9915 0.0041], Predicted Class: 1, True Class: 1
Record 8/30 - Prediction Probabilities: [0.001  0.0553 0.9486], Predicted Class: 2, True Class: 2
Record 9/30 - Prediction Probabilities: [0.001  0.0625 0.9443], Predicted Class: 2, True Class: 1
Record 10/30 - Prediction Probabilities: [0.0149 0.9922 0.0041], Predicted Class: 1, True Class: 1
Record 11/30 - Prediction Probabilities: [0.0013 0.1144 0.8839], Predicted Class: 2, True Class: 2
Record 12/30 - Prediction Probabilities: [0.9848 0.0165 0.0016], Predicted Class: 0, True Class: 0
Record 13/30 - Prediction Probabilities: [0.9853 0.0158 0.0016], Predicted Class: 0, True Class: 0
Record 14/30 - Prediction Probabilities: [0.9848 0.0165 0.0016], Predicted Class: 0, True Class: 0
Record 15/30 - Prediction Probabilities: [0.9853 0.0158 0.0016], Predicted Class: 0, True Class: 0
Record 16/30 - Prediction Probabilities: [0.0146 0.9922 0.0042], Predicted Class: 1, True Class: 1
Record 17/30 - Prediction Probabilities: [9.00e-04 4.62e-02 9.57e-01], Predicted Class: 2, True Class: 2
Record 18/30 - Prediction Probabilities: [0.0147 0.9922 0.0042], Predicted Class: 1, True Class: 1
Record 19/30 - Prediction Probabilities: [0.0116 0.9847 0.0086], Predicted Class: 1, True Class: 1
Record 20/30 - Prediction Probabilities: [9.00e-04 4.62e-02 9.57e-01], Predicted Class: 2, True Class: 2
Record 21/30 - Prediction Probabilities: [0.9846 0.0167 0.0016], Predicted Class: 0, True Class: 0
Record 22/30 - Prediction Probabilities: [0.0012 0.0895 0.9114], Predicted Class: 2, True Class: 2
Record 23/30 - Prediction Probabilities: [0.9848 0.0164 0.0016], Predicted Class: 0, True Class: 0
Record 24/30 - Prediction Probabilities: [9.00e-04 4.62e-02 9.57e-01], Predicted Class: 2, True Class: 2
Record 25/30 - Prediction Probabilities: [0.0017 0.2104 0.7794], Predicted Class: 2, True Class: 2
Record 26/30 - Prediction Probabilities: [0.001  0.0465 0.9566], Predicted Class: 2, True Class: 2
Record 27/30 - Prediction Probabilities: [9.00e-04 4.62e-02 9.57e-01], Predicted Class: 2, True Class: 2
Record 28/30 - Prediction Probabilities: [9.000e-04 4.620e-02 9.569e-01], Predicted Class: 2, True Class: 2
Record 29/30 - Prediction Probabilities: [0.9845 0.0168 0.0016], Predicted Class: 0, True Class: 0
Record 30/30 - Prediction Probabilities: [0.9843 0.017  0.0016], Predicted Class: 0, True Class: 0

Model Accuracy: 96.67%
Model Accuracy: 0.9666666666666667
```

For the Mnist dataset using two hidden layers of 25 and 20 neurons, 1000 iteration and learning_rate equal 0.3 we got 97.22%



```
Record 353/360 - Prediction Probabilities: [2.000e-04 9.929e-01 2.770e-02 0.000e+00 0.000e+00 0.000e+00 0.000e+00
 0.000e+00 0.000e+00 6.300e-03], Predicted Class: 1, True Class: 1
Record 354/360 - Prediction Probabilities: [0.000e+00 0.000e+00 2.000e-04 0.000e+00 0.000e+00 0.000e+00 2.000e-04
 6.000e-04 9.999e-01 0.000e+00], Predicted Class: 8, True Class: 8
Record 355/360 - Prediction Probabilities: [0.     0.     0.     0.     0.     0.     0.     0.9993 0.     0.    ], Predicted Class: 7, True Class: 7
Record 356/360 - Prediction Probabilities: [1.000e-04 0.000e+00 0.000e+00 0.000e+00 9.993e-01 0.000e+00 0.000e+00
 1.000e-04 0.000e+00 0.000e+00], Predicted Class: 4, True Class: 4
Record 357/360 - Prediction Probabilities: [0.000e+00 2.700e-03 9.000e-04 9.989e-01 0.000e+00 0.000e+00 0.000e+00
 0.000e+00 0.000e+00 0.000e+00], Predicted Class: 3, True Class: 3
Record 358/360 - Prediction Probabilities: [1.000e-04 1.000e-04 1.700e-03 0.000e+00 0.000e+00 0.000e+00 0.000e+00
 1.300e-03 5.791e-01 1.000e-04], Predicted Class: 8, True Class: 8
Record 359/360 - Prediction Probabilities: [0.000e+00 0.000e+00 4.400e-03 9.965e-01 0.000e+00 0.000e+00 9.800e-03
 0.000e+00 0.000e+00 1.000e-04], Predicted Class: 3, True Class: 3
Record 360/360 - Prediction Probabilities: [0.000e+00 0.000e+00 0.000e+00 4.000e-04 0.000e+00 9.995e-01 0.000e+00
 0.000e+00 0.000e+00 7.000e-04], Predicted Class: 5, True Class: 5

Model Accuracy: 97.22%
Model Accuracy: 0.9722222222222222
```

Bonus:

For Sklear NN module we got 80.88% for iris and 95.83% for Mnist

For kernels NN Module we got 83.88% for iris and96.83% for Mnist

Conclusion and Learnings:

This project successfully showcased the creation of a neural network from scratch using fundamental Python tools, offering valuable insights into the inner workings of neural networks. The hands-on approach in constructing and optimizing the network, along with testing on standard datasets, emphasized the importance of foundational knowledge in machine learning. While the results highlighted areas for future improvement, the project affirmed the educational significance of building complex systems from basic components, setting a strong foundation for further exploration in artificial intelligence.