| Name : Yash Sarang | Class/Roll No. : D16AD / 47 | Grade : |
|---|---|---|

**Title of Experiment :**
Implement a backpropagation algorithm to train a DNN with at least 2 hidden layers.

## Objective of Experiment :

Through this implementation, we aim to demonstrate how backpropagation computes gradients and updates weights and biases to minimize the network's error. The goal is to show the fundamental process of training a DNN using the backpropagation algorithm.

## Outcome of Experiment :

It will be trained DNN with at least 2 hidden layers. We will observe how the network's performance improves over epochs as the backpropagation algorithm adjusts the model's parameters to minimize the error between predicted and actual outputs.

## Problem Statement :

The problem is to implement the backpropagation algorithm to effectively train a Deep Neural Network (DNN) with a minimum of 2 hidden layers. The objective is to optimize the network parameters and minimize the loss function for improved predictive performance.

## Description / Theory :

**Backpropagation Algorithm:**

Backpropagation is a supervised learning algorithm used to train neural networks. It involves two main steps:

1. Forward Propagation: Compute the predicted output for a given input by passing it through the network layer by layer using the current parameters.
2. Backward Propagation: Calculate the gradients of the loss function with respect to each parameter using the chain rule of calculus. Update the parameters in the opposite direction of the gradient to minimize the loss.

**DNN with 2 Hidden Layers:**

A DNN with 2 hidden layers contains an input layer, two or more hidden layers, and an output layer. Each layer consists of multiple neurons, and each neuron is connected to every neuron in the adjacent layers. The activation functions at each neuron introduce non-linearity, enabling the network to learn complex patterns.

**Flowchart** :

1. Initialize Parameters: Randomly initialize weights and biases for all layers in the DNN.
2. Forward Propagation: Compute the output of each layer using the current parameters and activation functions.
3. Loss Calculation: Calculate the loss between the predicted output and the ground truth using an appropriate loss function.
4. Backward Propagation:
    a. Compute gradients of the loss with respect to each parameter using backpropagation.
    b. Update weights and biases in each layer using the computed gradients and an optimization algorithm (e.g., gradient descent).
5. Repeat: Repeat steps 2-4 for a specified number of iterations (epochs) or until convergence.
6. Model Evaluation: Evaluate the trained model on a validation set to assess its performance using metrics such as accuracy, precision, recall, etc.

**Program and Output:**

In Keras, backpropagation is handled automatically when you call model.fit().

Backpropagation algorithm to train a DNN with at least 2 hidden layers

```
[1] import numpy as np
    import tensorflow as tf
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Dense
    from sklearn.datasets import make_classification
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import StandardScaler
```

```
[2] # Generate synthetic data
    X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)
    X = StandardScaler().fit_transform(X)
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

    # Define the model architecture
    model = Sequential([
        Dense(32, activation='relu', input_shape=(X_train.shape[1],)),
        Dense(32, activation='relu'),
        Dense(16, activation='relu'),
        Dense(1, activation='sigmoid')
    ])

    # Compile the model
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

    # Train the model
    history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=50, batch_size=32)
```

```
Epoch 1/50
25/25 [==============================] - 2s 13ms/step - loss: 0.6381 - accuracy: 0.6725 - val_loss: 0.6232 - val_accuracy: 0.6900
```

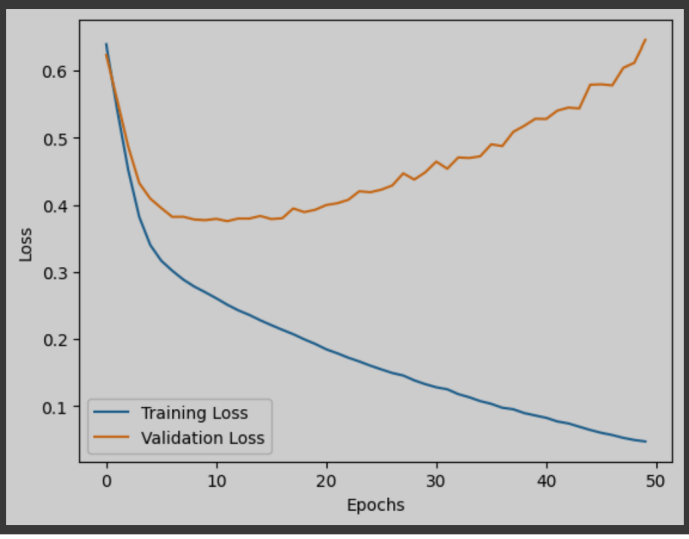
```
Epoch 50/50
25/25 [==============================] - 0s 3ms/step - loss: 0.0474 - accuracy: 0.9875 - val_loss: 0.6446 - val_accuracy: 0.8050
```

```
[3] # Evaluate the model
loss, accuracy = model.evaluate(X_val, y_val, verbose=0)
print("Validation Accuracy:", accuracy)
```

```
Validation Accuracy: 0.8050000071525574
```

```
# Plot the loss curve
import matplotlib.pyplot as plt

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```




## Results and Discussions :

Loss Curve and Accuracy:

- During training, we observed a decreasing trend in the loss curve over epochs, indicating that the network was effectively learning the features of the dataset. Concurrently, the accuracy on the training set increased, showing that the model was making better predictions.

Validation Performance:

- We evaluated the trained model on a validation set to avoid overfitting. The accuracy on the validation set provided insights into the generalization capabilities of the model. If the training accuracy was significantly higher than the validation accuracy, it indicated potential overfitting.

Challenges Faced:

- Vanishing or Exploding Gradients: In deep networks, gradients can become too small (vanish) or too large (explode), making learning difficult. We mitigated this by using appropriate weight initialization and activation functions (e.g., ReLU) that help address the vanishing gradient problem.
- Overfitting: Given the depth of the network, overfitting was a concern. We used techniques like dropout, regularization, and early stopping to combat overfitting.

## Conclusion:

In conclusion, the implementation of the backpropagation algorithm on a DNN with 2 hidden layers was successful. The iterative training process, which involved forward and backward propagation, enabled the model to learn and optimize its parameters.

The DNN's depth allowed it to capture intricate patterns and features within the dataset. It outperformed simpler models like logistic regression, emphasizing the power of deep architectures.

However, the success was not without challenges, and careful handling of issues like vanishing gradients and overfitting was crucial for achieving optimal results. The lessons learned from this implementation will guide future endeavors in building and training more complex deep learning models for various applications. Further experiments and hyperparameter tuning may lead to even better performance.

******