Department of Information Engineering Technology

The Superior University, Lahore

Artificial Intelligence Lab

Experiment No.10

Implementation of Neural Network in Python from Scratch

(Part -2)

Prepared for

_____

By:

Name: _____

ID: _____

Section: _____

Semester: _____

Total Marks: _____

Obtained Marks: _____

Signature: _____

Date: _____

Experiment No. 10

Implementation of Neural Network in Python from Scratch (Part -2)

(Rubrics)

Name: _____ **Roll No. : _____**

A. PSYCHOMOTOR

Sr. No.	Criteria	Allocated Marks	Unacceptable 0%	Poor 25%	Fair 50%	Good 75%	Excellent 100%	Total Obtained
1	Follow Procedures	1	0	0.25	0.5	0.75	1	
2	Software and Simulations	2	0	0.5	1	1.5	2	
3	Accuracy in Output Results	3	0	0.75	1.5	2.25	3	
Sub Total		**6**	**Sub Total marks Obtained in Psychomotor(P)**					

B. AFFECTIVE

Sr. No.	Criteria	Allocated Marks	Unacceptable 0%	Poor 25%	Fair 50%	Good 75%	Excellent 100%	Total Obtained
1	Respond to Questions	1	0	0.25	0.5	0.75	1	
2	Lab Report	1	0	0.25	0.5	0.75	1	
3	Assigned Task	2	0	0.5	1	1.5	2	
Sub Total		**4**	**Sub Total marks Obtained in Affective (A)**					

Instructor Name:_____ **Total Marks (P+A): _____**

Instructor Signature:_____ **Obtained Marks (P+A): _____**

Introduction to Gradient Descent

Gradient Descent is a widely used optimization algorithm in the field of deep learning. Its primary purpose is to minimize a cost function by adjusting the model parameters iteratively. In the context of deep learning, the model parameters are the weights and biases of the neural network.

Here's a high-level overview of how Gradient Descent works in the context of deep learning:

Cost Function (Loss Function): The first step is to define a cost function that measures how well the model is performing. The goal of the optimization process is to minimize this cost function.

Initialize Parameters: Initialize the weights and biases of the neural network with small random values.

Forward Propagation: Perform a forward pass through the neural network to make predictions on the training data.

Compute Loss: Calculate the loss, which is the difference between the predicted output and the actual output. The loss is a measure of how far off the model's predictions are from the true values.

Backward Propagation (Backpropagation): Compute the gradient of the cost function with respect to each model parameter. This is done by applying the chain rule of calculus to calculate the partial derivatives of the cost function with respect to each parameter.

Update Parameters: Adjust the weights and biases in the opposite direction of the gradient to minimize the cost function. The learning rate is a hyperparameter that determines the size of the steps taken during the optimization process.

Repeat: Repeat steps 3-6 for a specified number of iterations or until the cost function converges to a satisfactory minimum.

BUILD A MODEL IN KERAS/TENSORFLOW

Here's how it will work:

Import the libraries

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import pandas as pd
from matplotlib import pyplot as plt
```

Load Dataset

```
df = pd.read_csv("insurance_data.csv")
df.head()
```

we're splitting our data into training and testing sets using train_test_split and then scaling the 'age' feature to bring it into the same range

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(df[['age','affordibility']],df.bought_insurance,test_si
ze=0.2, random_state=25)
X_train_scaled = X_train.copy()
X_train_scaled['age'] = X_train_scaled['age'] / 100

X_test_scaled = X_test.copy()
X_test_scaled['age'] = X_test_scaled['age'] / 100
```

This code defines a simple neural network with one neuron, sigmoid activation, and initializes weights to 'ones' and biases to 'zeros'. The model is then compiled and trained on the scaled training data for 5000 epochs.

```
model = keras.Sequential([
    keras.layers.Dense(1, input_shape=(2,), activation='sigmoid',
kernel_initializer='ones', bias_initializer='zeros')
])

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

model.fit(X_train_scaled, y_train, epochs=5000)
```

This line evaluates the model's performance on the test set using the specified metrics.

```
model.evaluate(X_test_scaled,y_test)
model.predict(X_test_scaled)
```

We extract the weights and bias from the trained model using get_weights().

```
coef, intercept = model.get_weights()
coef, intercept
```

BASIC GRADIENT DESCENT ALGORITHM TO OPTIMIZE THE WEIGHTS AND BIAS

.

Here's how it works:

This is a simple sigmoid function implementation, which we use later in our custom prediction function.

```
def sigmoid(x):
        import math
        return 1 / (1 + math.exp(-x))
sigmoid(18)
```

We've defined a custom prediction function that uses the weights (coef[0], coef[1]) and bias (intercept) obtained from our Keras model to make predictions.

```
def prediction_function(age, affordibility):
    weighted_sum = coef[0]*age + coef[1]*affordibility + intercept
    return sigmoid(weighted_sum)

prediction_function(.47, 1)
```

These are helper functions for the sigmoid activation function and log loss calculation.

```
def sigmoid_numpy(X):
   return 1/(1+np.exp(-X))

sigmoid_numpy(np.array([12,0,1]))

def log_loss(y_true, y_predicted):
    epsilon = 1e-15
    y_predicted_new = [max(i,epsilon) for i in y_predicted]
    y_predicted_new = [min(i,1-epsilon) for i in y_predicted_new]
    y_predicted_new = np.array(y_predicted_new)
    return -np.mean(y_true*np.log(y_predicted_new)+(1-y_true)*np.log(1-
y_predicted_new))
```

This is our main gradient descent implementation. It iteratively updates the weights and bias using the calculated gradients until the loss falls below a specified threshold.

```
def gradient_descent(age, affordability, y_true, epochs, loss_thresold):
    w1 = w2 = 1
    bias = 0
    rate = 0.5
    n = len(age)
    for i in range(epochs):
        weighted_sum = w1 * age + w2 * affordability + bias
        y_predicted = sigmoid_numpy(weighted_sum)
        loss = log_loss(y_true, y_predicted)

        w1d = (1/n)*np.dot(np.transpose(age),(y_predicted-y_true))
        w2d = (1/n)*np.dot(np.transpose(affordability),(y_predicted-
y_true))

        bias_d = np.mean(y_predicted-y_true)
        w1 = w1 - rate * w1d
        w2 = w2 - rate * w2d
        bias = bias - rate * bias_d

        print (f'Epoch:{i}, w1:{w1}, w2:{w2}, bias:{bias}, loss:{loss}')

        if loss<=loss_thresold:
            break

    return w1, w2, bias
gradient_descent(X_train_scaled['age'],X_train_scaled['affordibility'],y
_train,1000, 0.4631)
coef, intercept
```

Introduction to Scratch

In the context of deep learning, "from scratch" often refers to building a neural network or a machine learning model without using pre-trained models or existing libraries for certain components. It means starting with the fundamental building blocks and developing the model architecture and parameters independently.

Here are a few situations where "from scratch" might be used in deep learning:

Training a Neural Network from Scratch: Instead of using pre-trained models, our might decide to train a neural network from scratch on a specific dataset. This involves initializing the network's weights

randomly and updating them through the training process using techniques like backpropagation and gradient descent.

Implementing Layers or Functions from Scratch: Sometimes, researchers or practitioners may implement specific layers or functions of a neural network from scratch, without relying on deep learning libraries like TensorFlow or PyTorch. This could include writing custom activation functions, loss functions, or even implementing backpropagation manually.

Building a Framework or Library from Scratch: In a more ambitious sense, "from scratch" could also mean developing an entire deep learning framework or library without using existing ones. This involves implementing algorithms, optimizations, and utilities that are typically provided by established deep learning libraries.

The decision to build from scratch in deep learning depends on factors like the level of customization needed, the understanding of the underlying algorithms, and the specific requirements of the task at hand. While building from scratch can provide a deep understanding of the inner workings of neural networks, it's often more common to use high-level frameworks that offer pre-implemented components for efficiency and ease of use.

Neural Network from Scratch

Here's how it will work:

In the constructor **(__init__ method),** we initialize the weights (w1 and w2) and the bias to default values.

The **fit** method trains the neural network using the gradient descent optimization algorithm. It takes input features (X), target values (y), the number of epochs, and a loss threshold as parameters. It calls the gradient_descent method to update the weights and bias based on the training data. After training, it prints the final values of the weights and bias.

The **predict** method takes a test dataset (X_test) and calculates the weighted sum of the features using the final weights and bias obtained during training. It then applies the sigmoid activation function using our sigmoid_numpy function to obtain predictions.

The **gradient_descent** method is where the actual optimization happens. It iteratively updates the weights and bias based on the gradients of the loss function with respect to these parameters. The training process continues for a specified number of epochs or until the loss falls below a certain threshold.

```
class myNN:
    def __init__(self):
        self.w1 = 1
        self.w2 = 1
        self.bias = 0

    def fit(self, X, y, epochs, loss_thresold):
        self.w1, self.w2, self.bias =
self.gradient_descent(X['age'],X['affordibility'],y, epochs,
loss_thresold)
```

```
        print(f"Final weights and bias: w1: {self.w1}, w2: {self.w2},
bias: {self.bias}")

    def predict(self, X_test):
        weighted_sum = self.w1*X_test['age'] +
self.w2*X_test['affordibility'] + self.bias
        return sigmoid_numpy(weighted_sum)

    def gradient_descent(self, age,affordability, y_true, epochs,
loss_thresold):
        w1 = w2 = 1
        bias = 0
        rate = 0.5
        n = len(age)
        for i in range(epochs):
            weighted_sum = w1 * age + w2 * affordability + bias
            y_predicted = sigmoid_numpy(weighted_sum)
            loss = log_loss(y_true, y_predicted)

            w1d = (1/n)*np.dot(np.transpose(age),(y_predicted-y_true))
            w2d = (1/n)*np.dot(np.transpose(affordability),(y_predicted-
y_true))

            bias_d = np.mean(y_predicted-y_true)
            w1 = w1 - rate * w1d
            w2 = w2 - rate * w2d
            bias = bias - rate * bias_d

            if i%50==0:
                print (f'Epoch:{i}, w1:{w1}, w2:{w2}, bias:{bias},
loss:{loss}')

            if loss<=loss_thresold:
                print (f'Epoch:{i}, w1:{w1}, w2:{w2}, bias:{bias},
loss:{loss}')
                break

        return w1, w2, bias
```

We create an instance of our custom neural network (myNN) and train it on the scaled training data (X_train_scaled and y_train). The specified number of epochs is 8000, and the training stops if the loss falls below the threshold of 0.4631.

```
customModel = myNN()
customModel.fit(X_train_scaled, y_train, epochs=8000,
loss_thresold=0.4631)
```

Task:

Design a Neural Network of MNIST dataset on plain python.