

Experiment Number: 05

Title: Write a program to create a neural network architecture from scratch, focusing on multi-class classification with customizable parameters such as hidden layers, neurons, non-linearity, and optimization algorithm.

Objective

The objective of this assignment is to design and implement a **neural network architecture from scratch** for **multi-class classification**. The implementation should allow customization of the number of hidden layers, neurons, activation functions, and optimization algorithm. The goal is to understand the underlying mathematics and mechanisms of neural networks, without relying on high-level libraries like TensorFlow or PyTorch..

Theory

A **neural network** is a computational model inspired by the human brain. It consists of interconnected layers of nodes (neurons) that process data through weighted connections.

1. **Input Layer** – Accepts the features of the dataset.
2. **Hidden Layers** – Perform transformations through **weighted sums** and **activation functions** (like ReLU, sigmoid, tanh).
3. **Output Layer** – Produces probabilities for each class using **softmax** activation.
4. **Loss Function** – For multi-class classification, **cross-entropy loss** is commonly used.
5. **Optimization Algorithm** – Updates the weights using methods like **Stochastic Gradient Descent (SGD)** or **Adam Optimizer**.

Algorithm

□ Initialize Parameters

- Randomly initialize weights and biases for each layer.

□ Forward Propagation

- Compute linear combination:

$$Z = W \cdot X + b$$

- Apply activation function:

$$A = f(Z)$$

□ Compute Loss

- Use cross-entropy loss:

$$L = -\frac{1}{m} \sum [Y \log(\hat{Y})]$$

- ❑ **Backward Propagation**
 - Calculate gradients of loss with respect to weights and biases.
 - ❑ **Update Parameters**
 - Update weights using the chosen optimizer (e.g., SGD, Adam).
 - ❑ **Repeat** for several epochs until convergence.
-

Program (Python Implementation)

```
# Neural Network from Scratch for Multi-Class Classification

import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import matplotlib.pyplot as plt

# -----
# Utility Functions
# -----
def relu(Z):
    return np.maximum(0, Z)

def relu_derivative(Z):
    return Z > 0

def softmax(Z):
    expZ = np.exp(Z - np.max(Z, axis=1, keepdims=True))
    return expZ / np.sum(expZ, axis=1, keepdims=True)

def cross_entropy_loss(y_true, y_pred):
    m = y_true.shape[0]
    loss = -np.sum(y_true * np.log(y_pred + 1e-9)) / m
    return loss

# -----
# Neural Network Class
# -----
class NeuralNetwork:
    def __init__(self, layers, learning_rate=0.01, activation='relu',
optimizer='sgd'):
        self.layers = layers
        self.learning_rate = learning_rate
        self.activation = activation
        self.optimizer = optimizer
        self.params = {}
        self.initialize_parameters()
```

```

def initialize_parameters(self):
    np.random.seed(42)
    for i in range(1, len(self.layers)):
        self.params['W' + str(i)] = np.random.randn(self.layers[i-1],
self.layers[i]) * 0.01
        self.params['b' + str(i)] = np.zeros((1, self.layers[i]))

def activation_forward(self, Z):
    if self.activation == 'relu':
        return relu(Z)
    elif self.activation == 'tanh':
        return np.tanh(Z)
    elif self.activation == 'sigmoid':
        return 1 / (1 + np.exp(-Z))

def activation_backward(self, Z):
    if self.activation == 'relu':
        return relu_derivative(Z)
    elif self.activation == 'tanh':
        return 1 - np.tanh(Z)**2
    elif self.activation == 'sigmoid':
        s = 1 / (1 + np.exp(-Z))
        return s * (1 - s)

def forward_propagation(self, X):
    cache = {'A0': X}
    L = len(self.layers) - 1
    for i in range(1, L):
        Z = np.dot(cache['A' + str(i-1)], self.params['W' + str(i)]) +
self.params['b' + str(i)]
        A = self.activation_forward(Z)
        cache['Z' + str(i)] = Z
        cache['A' + str(i)] = A
    ZL = np.dot(cache['A' + str(L-1)], self.params['W' + str(L)]) +
self.params['b' + str(L)]
    AL = softmax(ZL)
    cache['Z' + str(L)] = ZL
    cache['A' + str(L)] = AL
    return AL, cache

def backward_propagation(self, X, Y, cache):
    grads = {}
    L = len(self.layers) - 1
    m = X.shape[0]

    dZL = cache['A' + str(L)] - Y
    grads['dW' + str(L)] = np.dot(cache['A' + str(L-1)].T, dZL) / m
    grads['db' + str(L)] = np.sum(dZL, axis=0, keepdims=True) / m

```

```

        for i in reversed(range(1, L)):
            dA = np.dot(dZL, self.params['W' + str(i+1)].T)
            dZ = dA * self.activation_backward(cache['Z' + str(i)])
            grads['dW' + str(i)] = np.dot(cache['A' + str(i-1)].T, dZ) / m
            grads['db' + str(i)] = np.sum(dZ, axis=0, keepdims=True) / m
            dZL = dZ

        return grads

    def update_parameters(self, grads):
        for i in range(1, len(self.layers)):
            self.params['W' + str(i)] -= self.learning_rate * grads['dW' +
str(i)]
            self.params['b' + str(i)] -= self.learning_rate * grads['db' +
str(i)]

    def fit(self, X, Y, epochs=500):
        losses = []
        for epoch in range(epochs):
            AL, cache = self.forward_propagation(X)
            loss = cross_entropy_loss(Y, AL)
            grads = self.backward_propagation(X, Y, cache)
            self.update_parameters(grads)
            losses.append(loss)

            if epoch % 50 == 0:
                print(f"Epoch {epoch}, Loss: {loss:.4f}")
        return losses

    def predict(self, X):
        AL, _ = self.forward_propagation(X)
        return np.argmax(AL, axis=1)

# -----
# Dataset Creation
# -----
# Create an easier separable dataset and scale features for faster
convergence.
# Increase samples and class separation so the network can achieve high
accuracy on this synthetic task.
X, y = make_classification(n_samples=1000, n_features=4, n_classes=3,
n_informative=3, n_redundant=0, n_repeated=0, n_clusters_per_class=1,
class_sep=2.0, random_state=42)
y = y.reshape(-1, 1)
# One-hot encode labels
encoder = OneHotEncoder(sparse_output=False)
Y = encoder.fit_transform(y)

```

```
# Split and scale features
from sklearn.preprocessing import StandardScaler
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
random_state=42, shuffle=True)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# -----
# Model Training
# -----
# Increase capacity and training duration for better performance
nn = NeuralNetwork(layers=[4, 32, 16, 3], learning_rate=0.1,
activation='relu', optimizer='sgd')
losses = nn.fit(X_train, Y_train, epochs=1000)

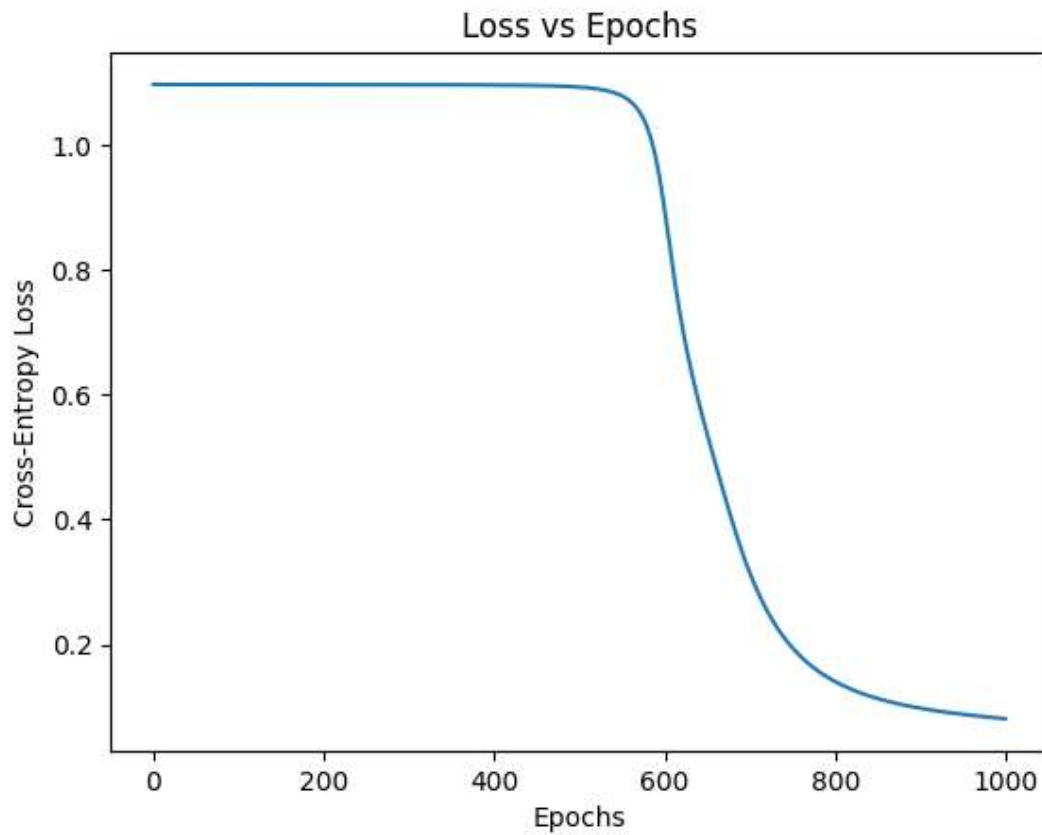
# -----
# Evaluation
# -----
y_pred = nn.predict(X_test)
y_true = np.argmax(Y_test, axis=1)
accuracy = np.mean(y_pred == y_true) * 100
print(f"\nTest Accuracy: {accuracy:.2f}%")

# Plot loss curve
plt.plot(losses)
plt.title("Loss vs Epochs")
plt.xlabel("Epochs")
plt.ylabel("Cross-Entropy Loss")
plt.show()
```

Output

```
... Epoch 0, Loss: 1.0986
Epoch 50, Loss: 1.0983
Epoch 100, Loss: 1.0982
Epoch 150, Loss: 1.0982
Epoch 200, Loss: 1.0982
Epoch 250, Loss: 1.0981
Epoch 300, Loss: 1.0981
Epoch 350, Loss: 1.0979
Epoch 400, Loss: 1.0976
Epoch 200, Loss: 1.0982
Epoch 250, Loss: 1.0981
Epoch 300, Loss: 1.0981
Epoch 350, Loss: 1.0979
Epoch 400, Loss: 1.0976
Epoch 450, Loss: 1.0968
Epoch 500, Loss: 1.0942
Epoch 550, Loss: 1.0799
Epoch 600, Loss: 0.8947
Epoch 650, Loss: 0.5355
Epoch 450, Loss: 1.0968
Epoch 500, Loss: 1.0942
Epoch 550, Loss: 1.0799
Epoch 600, Loss: 0.8947
Epoch 650, Loss: 0.5355
Epoch 700, Loss: 0.3131
...
Test Accuracy: 98.50%
Epoch 950, Loss: 0.0875
```

Conclusion



This experiment demonstrates the internal working of a neural network. By implementing each step manually, we gain deeper insights into how data flows through the network, how errors are propagated back, and how optimization helps the model learn to classify multiple classes effectively.