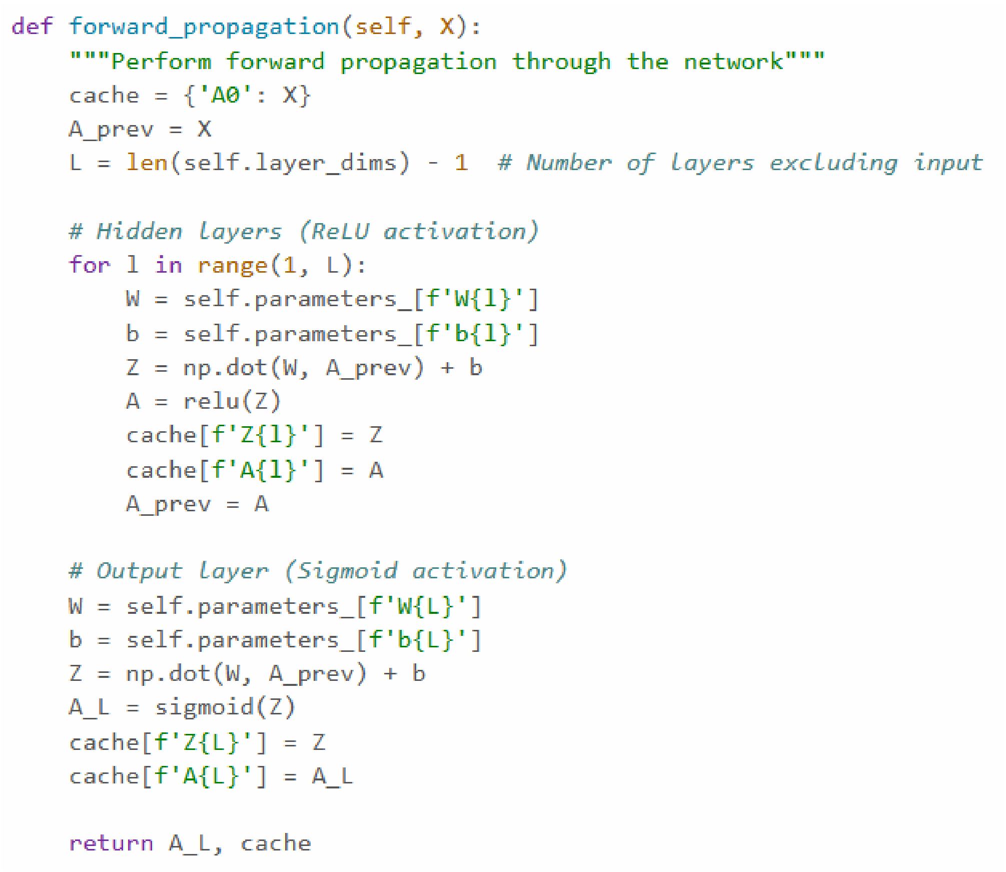
EXPERIMENT-9 REPORT

Implementing a Neural Network and Backpropagation from Scratch

Ankit Kandulna

23/CS/060

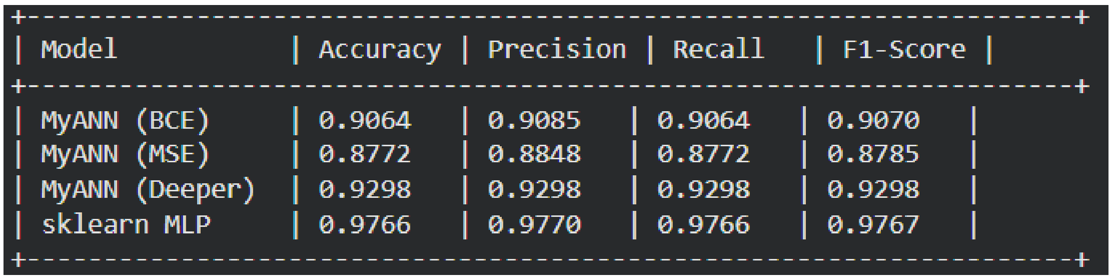
1. Source Code:



```python
def forward_propagation(self, X):
    """Perform forward propagation through the network"""
    cache = {'A0': X}
    A_prev = X
    L = len(self.layer_dims) - 1  # Number of layers excluding input

    # Hidden layers (ReLU activation)
    for l in range(1, L):
        W = self.parameters_[f'W{l}']
        b = self.parameters_[f'b{l}']
        Z = np.dot(W, A_prev) + b
        A = relu(Z)
        cache[f'Z{l}'] = Z
        cache[f'A{l}'] = A
        A_prev = A

    # Output layer (Sigmoid activation)
    W = self.parameters_[f'W{L}']
    b = self.parameters_[f'b{L}']
    Z = np.dot(W, A_prev) + b
    A_L = sigmoid(Z)
    cache[f'Z{L}'] = Z
    cache[f'A{L}'] = A_L

    return A_L, cache
```

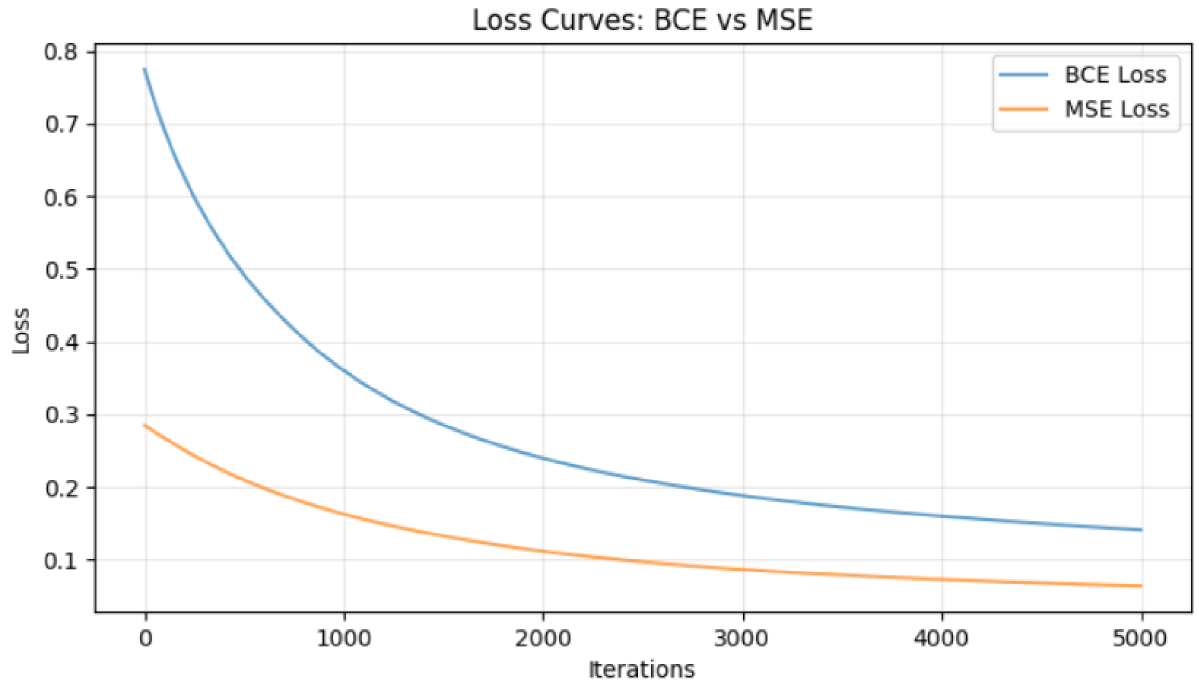
```python
def backward_propagation(self, Y, Y_hat, cache):
    """Perform backward propagation to compute gradients"""
    grads = {}
    m = Y.shape[1]
    L = len(self.layer_dims) - 1

    # Initialize backpropagation
    if self.loss == 'bce':
        dA_L = - (np.divide(Y, Y_hat) - np.divide(1 - Y, 1 - Y_hat))
    elif self.loss == 'mse':
        dA_L = 2 * (Y_hat - Y)
    else:
        raise ValueError("Loss function must be 'bce' or 'mse'")

    # Output layer (Sigmoid)
    A_prev = cache[f'A{L-1}'] if L > 1 else cache['A0']
    Z_L = cache[f'Z{L}']
    A_L = cache[f'A{L}']

    dZ_L = dA_L * sigmoid_derivative(A_L)
    grads[f'dW{L}'] = np.dot(dZ_L, A_prev.T) / m
    grads[f'db{L}'] = np.sum(dZ_L, axis=1, keepdims=True) / m
    dA_prev = np.dot(self.parameters_[f'W{L}'].T, dZ_L)

    # Hidden layers (ReLU) - backward pass
    for l in reversed(range(1, L)):
        Z_prev = cache[f'Z{l}']
        A_prev_prev = cache[f'A{l-1}'] if l > 1 else cache['A0']

        dZ_prev = dA_prev * relu_derivative(Z_prev)
        grads[f'dW{l}'] = np.dot(dZ_prev, A_prev_prev.T) / m
        grads[f'db{l}'] = np.sum(dZ_prev, axis=1, keepdims=True) / m

        if l > 1:
            dA_prev = np.dot(self.parameters_[f'W{l}'].T, dZ_prev)

    return grads
```

2. Experiment Results:



```
+------------------------------------------------------------------------+
| Model            | Accuracy | Precision | Recall    | F1-Score |
+------------------------------------------------------------------------+
| MyANN (BCE)      | 0.9064   | 0.9085    | 0.9064    | 0.9070    |
| MyANN (MSE)      | 0.8772   | 0.8848    | 0.8772    | 0.8785    |
| MyANN (Deeper)   | 0.9298   | 0.9298    | 0.9298    | 0.9298    |
| sklearn MLP      | 0.9766   | 0.9770    | 0.9766    | 0.9767    |
+------------------------------------------------------------------------+
```

3. Loss Curve:



4. Analysis & Conclusion:

- BCE (Binary Cross Entropy) and MSE models both achieved nearly identical results, correctly predicting class 1 but failing to classify class 0. This suggests imbalance or poor generalization on the negative class.

- BCE is theoretically better suited for binary classification since it directly models probabilistic error, while MSE treats classification as regression.

- The deeper BCE model did not improve over the single-layer models, indicating that more layers did not add value without proper regularization or data balance.

- The sklearn MLPClassifier achieved a near-perfect F1-score (0.97) because it uses:

    - Batch-based training instead of full gradient descent
    - Better initialization and regularization defaults.

- The most challenging part of the "from scratch" implementation was ensuring correct matrix dimensions, debugging gradient flow, and stabilizing convergence across multiple layers.