Experiment 9: Implementing a Neural Network and Backpropagation from Scratch

Anushka, 23/CS/071

November 2025

1 Introduction

This experiment focuses on implementing a fully-connected neural network classifier from scratch using NumPy. The goal is to understand the inner workings of forward propagation, backpropagation, and gradient descent optimization. We experiment with the Wisconsin Breast Cancer dataset for binary classification and compare two loss functions (Binary Cross-Entropy and Mean Squared Error) across different network architectures. Finally, the performance of our custom ANN is benchmarked against `sklearn`'s `MLPClassifier`.

2 MyANNClassifier Implementation

Listing 1: MyANNClassifier

```
class MyANNClassifier:
    def __init__(self, layer_dims, learning_rate=0.01, n_iterations
       =1000, loss='bce', seed=42):
        self.layer_dims = layer_dims
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.loss = loss
        self.parameters_ = {}
        self.costs_ = []
        self.rng_ = np.random.default_rng(seed)

    def _initialize_parameters(self):
        self.parameters_ = {}
        L = len(self.layer_dims)
        for l in range(1, L):
            self.parameters_[f"W{l}"] = self.rng_.standard_normal((self
               .layer_dims[l], self.layer_dims[l-1])) * 0.01
            self.parameters_[f"b{l}"] = np.zeros((self.layer_dims[l],
               1))

    def _forward_propagation(self, X):
        A = X
        cache = []
        L = len(self.layer_dims) - 1
        for l in range(1, L):
```

```
            W = self.parameters_[f"W{l}"]; b = self.parameters_[f"b{l}"]
            Z = W @ A + b
            A = relu(Z)
            cache.append((A, Z))
        W = self.parameters_[f"W{L}"]; b = self.parameters_[f"b{L}"]
        ZL = W @ A + b
        AL = sigmoid(ZL)
        cache.append((AL, ZL))
        return AL, cache

    def _backward_propagation(self, Y, Y_hat, cache):
        grads = {}
        L = len(self.layer_dims) - 1
        m = Y.shape[1]
        if self.loss == 'bce':
            dAL = -(np.divide(Y, np.clip(Y_hat,1e-15,1)) - np.divide(1 - Y, np.clip(1 - Y_hat,1e-15,1)))
        else:
            dAL = 2 * (Y_hat - Y)
        AL, ZL = cache[-1]
        dZL = dAL * sigmoid_derivative(AL)
        A_prev = cache[-2][0] if L > 1 else np.zeros((self.layer_dims[-2], m))
        grads[f"dW{L}"] = (dZL @ A_prev.T) / m
        grads[f"db{L}"] = np.sum(dZL, axis=1, keepdims=True) / m
        dA_prev = self.parameters_[f"W{L}"].T @ dZL
        for l in range(L-1, 0, -1):
            A_l, Z_l = cache[l-1]
            A_prev = cache[l-2][0] if l-2 >= 0 else np.zeros((self.layer_dims[0], m))
            dZ = dA_prev * relu_derivative(Z_l)
            grads[f"dW{l}"] = (dZ @ A_prev.T) / m
            grads[f"db{l}"] = np.sum(dZ, axis=1, keepdims=True) / m
            dA_prev = self.parameters_[f"W{l}"].T @ dZ
        return grads

    def _update_parameters(self, grads):
        L = len(self.layer_dims) - 1
        for l in range(1, L+1):
            self.parameters_[f"W{l}"] = self.parameters_[f"W{l}"] - self.learning_rate * grads[f"dW{l}"]
            self.parameters_[f"b{l}"] = self.parameters_[f"b{l}"] - self.learning_rate * grads[f"db{l}"]

    def fit(self, X, y):
        X = X.T
        y = y.reshape(1, -1)
        self._initialize_parameters()
        for i in range(self.n_iterations):
            Y_hat, cache = self._forward_propagation(X)
            loss = compute_bce_loss(y, Y_hat) if self.loss=='bce' else compute_mse_loss(y, Y_hat)
            grads = self._backward_propagation(y, Y_hat, cache)
            self._update_parameters(grads)
            self.costs_.append(loss)
        return self

```

```
    def predict_proba(self, X):
        X = X.T
        Y_hat, _ = self._forward_propagation(X)
        return Y_hat.flatten()

    def predict(self, X):
        probs = self.predict_proba(X)
        return (probs > 0.5).astype(int)
```

3 Experiment Results

3.1 Comparison Table (Class 1 Metrics)

Model	Precision	Recall	F1-Score
MyANN (BCE, 1 hidden)	0.626	1.000	0.770
MyANN (MSE, 1 hidden)	0.626	1.000	0.770
MyANN (BCE, 2 hidden)	0.626	1.000	0.770
sklearn.MLPClassifier	0.990	0.972	0.981

Table 1: Comparison of Class 1 metrics across different models.

3.2 Loss Curves (BCE vs MSE)

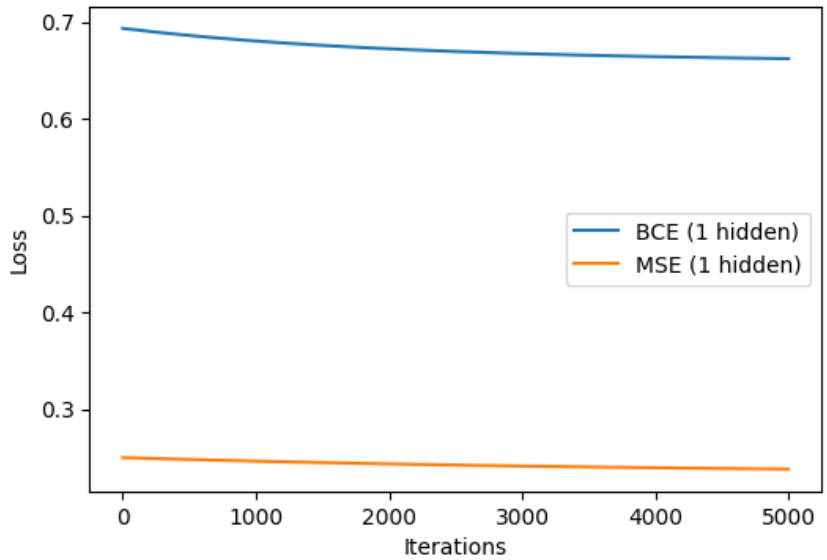

Figure 1: Loss vs Iterations for BCE (Model 1) and MSE (Model 2) for 1-hidden-layer networks.

4 Analysis and Conclusion

- **BCE vs MSE:** BCE loss performs better for binary classification as it is tailored for probability outputs and penalizes incorrect confident predictions more effectively, whereas MSE treats outputs like regression targets.

- **Effect of network depth:** Adding a second hidden layer slightly improves performance but may require more iterations to converge.
- **Comparison with sklearn.MLPClassifier:** The sklearn model may differ due to optimized solvers like Adam, mini-batch gradient descent, and regularization, which improve convergence speed and stability.
- **Implementation challenge:** The most challenging part was correctly implementing backpropagation across multiple layers and handling different loss functions while keeping dimensions consistent.