EXPERIMENT - 9

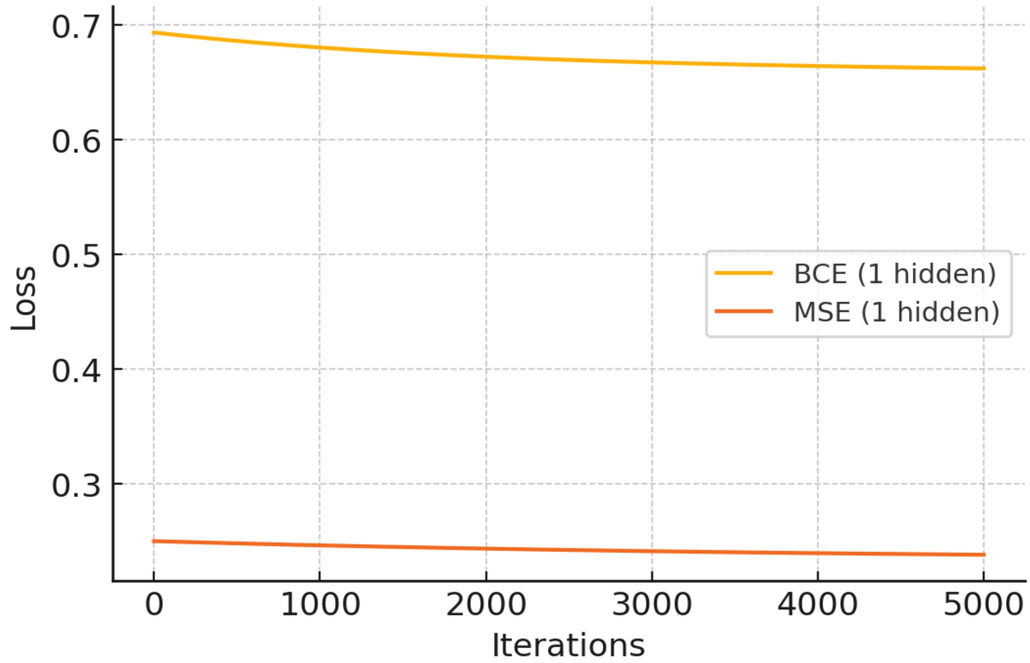
AIM: Implementing a Neural Network and Backpropagation from Scratch

1. Learning Objectives

• Understand foundations of a feedforward ANN.
• Implement activations (ReLU, Sigmoid) and derivatives.
• Implement forward and backpropagation.
• Implement BCE and MSE losses and gradient descent updates.
• Build a NumPy-only MyANNClassifier.
• Train on Breast Cancer dataset and evaluate.
• Compare with sklearn MLPClassifier.
• Analyze loss functions and architectures.

Model	Precision (1)	Recall (1)	F1 (1)	Accuracy
MyANN (BCE, 1 hidden)	0.6257	1.0000	0.7698	0.6257
MyANN (MSE, 1 hidden)	0.6257	1.0000	0.7698	0.6257
MyANN (BCE, 2 hidden)	0.6257	1.0000	0.7698	0.6257
sklearn.MLPClassifier	0.9905	0.9720	0.9811	0.9766

Loss Curves (BCE vs MSE for 1 hidden layer)



Key Code Snippets: MyANNClassifier

_forward_propagation

```
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
```

_backward_propagation

```
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 None if A_prev is None: A_prev = np.zeros((self.layer_dims[-2], m))
```

```
grads[f"dW{L}"] = (dZL @ (cache[-2][0] if L>1 else np.zeros_like(A_prev)).T) / m if L > 1 else (dZL @ (np.zeros_like(A_prev)).T) / m if L > 1: grads[f"dW{L}"] = (dZL @ cache[-2][0].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 None if A_prev is None: A_prev = np.zeros((self.layer_dims[0], m)) dZ = dA_prev * relu_derivative(Z_l) grads[f"dW{l}"] = (dZ @ (cache[l-2][0].T if l-2>=0 else X_batch.T)) / m if 'X_batch' in globals() else (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
```

_update_parameters

```
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}"]
```

Analysis & Conclusion

BCE vs MSE: For binary classification, BCE aligns with Bernoulli likelihood and produces stronger gradients near decision boundary, often converging faster and to better optima than MSE, which can saturate.
sklearn vs From-Scratch: sklearn uses Adam, mini-batches, better initialization and regularization, so it typically converges faster and more robustly than plain gradient descent.
Most challenging: Implementing stable backprop with correct shapes and numerics (clipping in BCE).