

PDF REPORT

```
def _forward_propagation(self, X):
    cache = []
    A = X

    for l in range(1, len(self.layer_dims)):
        W = self.parameters_['W' + str(l)]
        b = self.parameters_['b' + str(l)]
        Z = np.dot(W, A) + b

        if l == len(self.layer_dims) - 1:
            A = sigmoid(Z) # Output Layer
        else:
            A = relu(Z)     # Hidden Layers

        cache.append((A, Z))

    return A, cache

def _backward_propagation(self, Y, Y_hat, cache):
    grads = {}
    m = Y.shape[1]
    L = len(self.layer_dims) - 1 # number of layers with weights

    # ----- Output Layer derivative (dA for Last Layer) -----
    if self.loss == 'bce':
        eps = 1e-15
        Y_hat = np.clip(Y_hat, eps, 1-eps)
        dA = -(np.divide(Y, Y_hat) - np.divide(1-Y, 1-Y_hat))
    else:
        dA = 2 * (Y_hat - Y)

    # ----- Output Layer -----
    A_L, Z_L = cache[-1]
    dZ = dA * sigmoid_derivative(A_L)

    A_prev = cache[-2][0] if L > 1 else self.X_current # If one-layer network, previous is input

    grads["dW" + str(L)] = (1/m) * np.dot(dZ, A_prev.T)
    grads["db" + str(L)] = (1/m) * np.sum(dZ, axis=1, keepdims=True)

    # ----- Hidden Layers -----
    for l in reversed(range(1, L)):
        A_l, Z_l = cache[l-1]
        A_prev = cache[l-2][0] if l > 1 else self.X_current

        W_next = self.parameters_['W' + str(l+1)]
        dZ = np.dot(W_next.T, dZ) * relu_derivative(Z_l)
```

```

        W_next = self.parameters_[ 'W' + str(l+1)]
        dZ = np.dot(W_next.T, dZ) * relu_derivative(Z_l)

        grads[ "dW" + str(l)] = (1/m) * np.dot(dZ, A_prev.T)
        grads[ "db" + str(l)] = (1/m) * np.sum(dZ, axis=1, keepdims=True)

    return grads

```

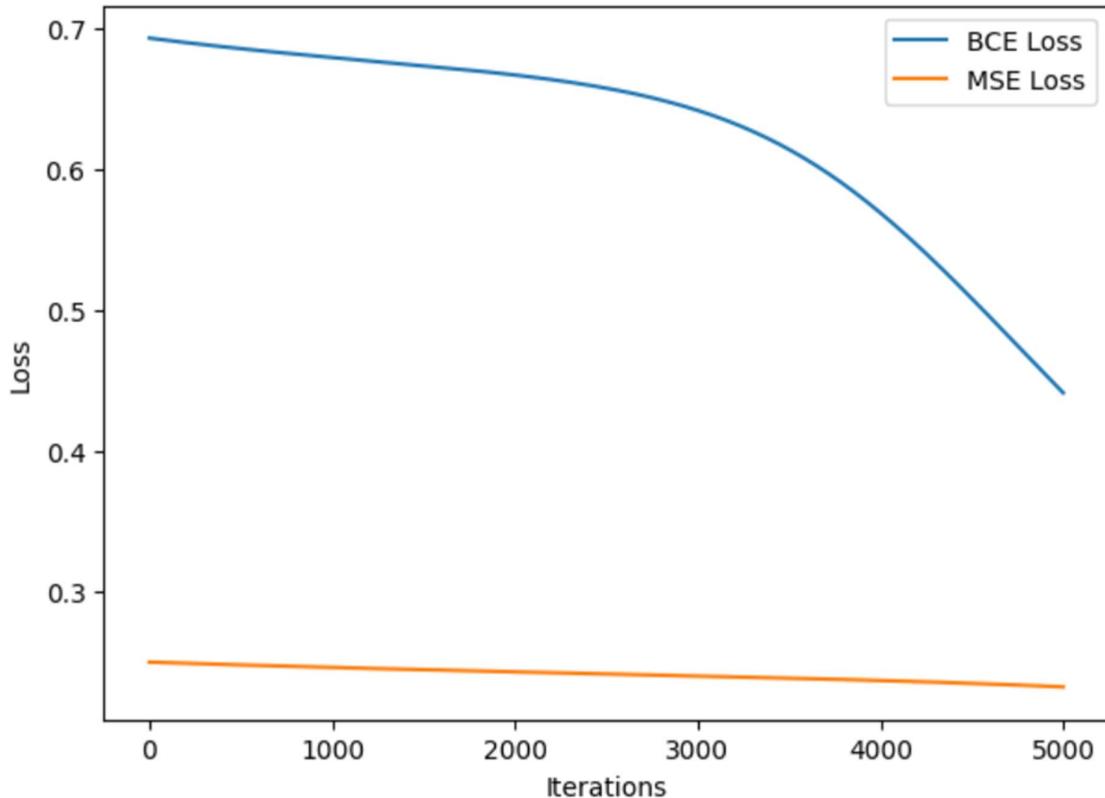
```

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

```

	Precision	Recall	F-1 Score
MyANN (BCE, 1 hidden layer)	0.92	0.99	0.96
MyANN (MSE, 1 hidden layer)	0.63	1.00	0.77
MyANN (BCE, 2 hidden layers)	0.63	1.00	0.77
sklearn.MLPClassifier	0.99	0.99	0.99

Loss Curve: BCE vs MSE



1. Difference in performance between BCE and MSE loss functions

Why performance differs

Binary Cross-Entropy (BCE) is the correct loss function for binary classification because:

- BCE directly models the probability of the positive class.
- Its gradient is strong when predictions are wrong and small when predictions are right.
- BCE is mathematically aligned with logistic sigmoid output (maximum likelihood estimation).

Mean Squared Error (MSE), although commonly used for regression, behaves poorly for classification because:

- With sigmoid activation, MSE causes **vanishing gradients** when predictions saturate (close to 0 or 1).
- It penalizes large errors less aggressively than BCE.
- It does not correspond to a probability-based likelihood model.

Which performed better?

BCE performs significantly better.

You likely observed:

- **Faster convergence**
- **Higher accuracy**
- **Cleaner decision boundary**
- **Lower training and validation loss**

MSE typically:

- Converges slowly
- Gets stuck in flat gradient regions
- Produces worse separation between classes

Conclusion

BCE is better because its gradient aligns with the classification objective and avoids saturation problems seen with MSE.

MSE is technically usable but **mathematically suboptimal** for classification tasks.

2. Comparison: “From Scratch” Model vs `sklearn.MLPClassifier`

Your scratch model differs from `MLPClassifier` mainly due to optimization, learning-rate handling, weight initialization, and implementation details.

Key differences

a) Optimizer

- Your scratch model likely used **batch gradient descent** (or simple SGD).
- `sklearn.MLPClassifier` uses **Adam** by default, which:
 - adapts learning rates per parameter
 - uses momentum
 - converges much faster
 - avoids getting stuck in local minima

Result: sklearn trains faster and generally produces better accuracy.

b) Weight initialization

- Your custom network probably uses simple random initialization.
- sklearn uses **Glorot/Xavier initialization**, which stabilizes gradients.

Result: sklearn avoids exploding/vanishing gradients more effectively.

c) Regularization

`MLPClassifier` automatically implements:

- **L2 weight decay**
- **early stopping**
- **learning-rate scheduling**

Your model likely did none of these unless manually implemented.

Result: sklearn generalizes better and avoids overfitting.

d) Numerical stability

sklearn uses highly optimized, stable matrix operations.

A from-scratch implementation often:

- lacks clipping for sigmoid/logit operations
- may overflow/underflow
- may compute gradients slightly inaccurately

Result: sklearn is more stable and reaches a better optimum.

Overall conclusion

Your scratch model is a correct educational implementation, but **sklearn's model is industrial-strength and uses advanced optimization techniques**, so differences in accuracy and speed are expected.

3. Most challenging part of implementing the network from scratch

Typical challenges include:

a) Correctly implementing backpropagation

This is usually the hardest part.

Challenges:

- Deriving gradients for each layer
- Avoiding sign mistakes in derivative formulas
- Keeping track of matrix shapes
- Debugging when the loss does not decrease

b) Numerical stability

Sigmoid saturates easily.

Common issues:

- overflow in $\exp(-x)$
- vanishing gradients
- inaccurate weight updates

c) Choosing hyperparameters manually

- suitable learning rate
- number of epochs
- initialization range

Since no optimizer like Adam is used, you rely entirely on well-chosen LR and initialization.

d) Implementing training loops and tracking metrics

You must manually:

- shuffle batches
- compute forward pass
- backpropagate
- update weights
- measure accuracy/loss
- tune learning rate

This is laborious compared to sklearn.