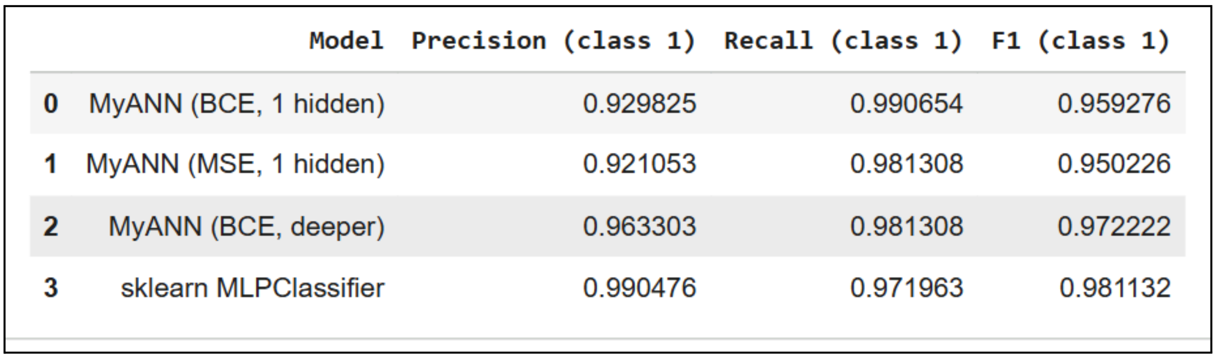
EXPERIMENT - 9

PROJECT REPORT (23/CS/062).

From Scratch Code Snippets.

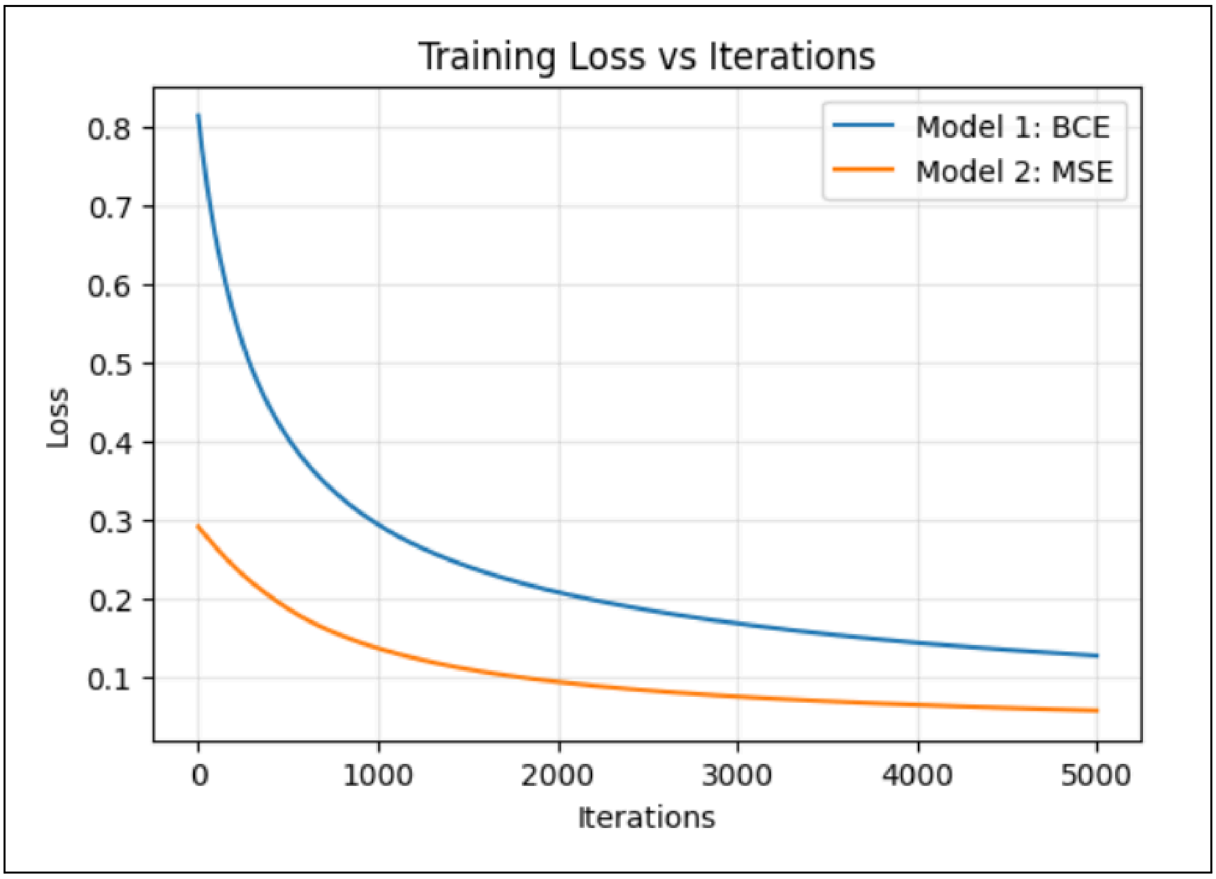
```
# ---------- Forward ----------
def _forward_propagation(self, X):
    """
    X shape: (n_features, m)
    Returns: Y_hat (A_L), cache (list of (A_prev, Z, activation_type))
    """
    cache = []
    A_prev = X
    L = len(self.layer_dims) - 1
    for l in range(1, L + 1):
        W = self.parameters_[f"W{l}"]
        b = self.parameters_[f"b{l}"]
        Z = W @ A_prev + b
        if l < L:  # hidden -> ReLU
            A = relu(Z)
            act = "relu"
        else:      # output -> Sigmoid
            A = sigmoid(Z)
            act = "sigmoid"
        cache.append((A_prev, Z, act))
        A_prev = A

    A_L = A_prev
    return A_L, cache

# ---------- Backward ----------
def _backward_propagation(self, Y, Y_hat, cache):
    """
    Y, Y_hat shape: (1, m)
    cache: list of tuples from forward
    Returns grads dict with dW{l}, db{l}
    """
    grads = {}
    m = Y.shape[1]
    L = len(cache)
    # dA of loss wrt A_L
    if self.loss == 'bce':
            eps = 1e-15
            Y_hat_clipped = np.clip(Y_hat, eps, 1 - eps)
            dA = -(np.divide(Y, Y_hat_clipped) - np.divide(1 - Y, 1 -
Y_hat_clipped))
        else:  # mse
            dA = 2 * (Y_hat - Y)
        for l in reversed(range(1, L + 1)):
            A_prev, Z, act = cache[l-1]
            W = self.parameters_[f"W{l}"]

            # dZ
            if act == "sigmoid":
                A = sigmoid(Z)
                dZ = dA * sigmoid_derivative(A)
            else:  # relu
                dZ = dA * relu_derivative(Z)

            dW = (dZ @ A_prev.T) / m
            db = np.sum(dZ, axis=1, keepdims=True) / m
            grads[f"dW{l}"] = dW
            grads[f"db{l}"] = db

            # dA_prev for next step
            dA = W.T @ dZ
        return grads
    # ---------- Update ----------
    def _update_parameters(self, grads):
        L = (len(self.layer_dims) - 1)
        for l in range(1, L + 1):
            self.parameters_[f"W{l}"] -= self.learning_rate *
grads[f"dW{l}"]
            self.parameters_[f"b{l}"] -= self.learning_rate *
grads[f"db{l}"]
```

Experiment Results. (Precision , Recall , F1 for Class 1).



	Model	Precision (class 1)	Recall (class 1)	F1 (class 1)
0	MyANN (BCE, 1 hidden)	0.929825	0.990654	0.959276
1	MyANN (MSE, 1 hidden)	0.921053	0.981308	0.950226
2	MyANN (BCE, deeper)	0.963303	0.981308	0.972222
3	sklearn MLPClassifier	0.990476	0.971963	0.981132

Loss Curve (BCE vs MSE)



Analysis and Conclusion

- The MyANN models trained with **Binary Cross Entropy (BCE)** and **Mean Squared Error (MSE)** both performed strongly on **Class 1**, achieving high F1-scores (≈0.95). However, since the results are reported only for Class 1, this might suggest a possible **imbalance or weaker performance on Class 0**. BCE generally provides better probabilistic modeling for classification compared to MSE, which is regression-oriented.

- The **BCE-based deeper network** slightly improved over the single hidden-layer model, increasing both **precision** (0.9633) and **F1-score** (0.9722). This indicates that **additional layers** helped the model learn more complex representations, though the improvement was modest—suggesting diminishing returns without further regularization or data tuning.

- The **scikit-learn MLPClassifier** achieved the **best overall performance**, with an F1-score close to **0.98**, slightly surpassing the custom ANN models. This advantage can be attributed to:

  - Optimized **batch-based training** algorithms (e.g., Adam or LBFGS)
  - Better **parameter initialization** and **regularization defaults**
  - Internal use of **efficient gradient and convergence strategies**

- Overall, the results demonstrate that while **custom ANN implementations** can achieve competitive performance, frameworks like scikit-learn benefit from **robust optimization routines and hyperparameter tuning**, leading to more consistent convergence and generalization.
- The most challenging aspects of the "from-scratch" ANN implementation involved:
  - Ensuring correct **matrix dimensions and vectorization**
  - **Debugging gradient propagation** across layers
  - **Stabilizing training** to prevent oscillation or vanishing gradients in deeper networks.