Homework 1 : Report

110024516 統研所 邱繼賢

1 Introduction

1.1 Abstract

本次作業使用 Fashion-MNIST dataset，希望透過 shallow neural network 分析二維圖片資料來分類其屬於十種服飾類別中的哪一種類，並希望模型在 test data 上的準確率可以達到 87%。

1.2 Data

在 Fashion-MNIST dataset 中，總共有 60000 筆 train data 和 10000 筆 test data，每筆資料的 input variable 都是一個 28×28 的二維陣列，response variable 爲該圖片對應到的 label ($0 \sim 9$)，爲了方便我們之後的模型處理，將每一筆 28×28 matrix form input variable 都轉換成 784×1 vector form，而 label response 都轉換成 one-hot 的形式：(10×1) vector with true label = 1, others = 0，然後再將 train data 以 55000 : 5000 的比例隨機分成 train set 和 validation set，各資料集的維度如下所示：

$$X_{train} : (55000\ ,\ 784)\ \ ,\ \ Y_{train} : (55000\ ,\ 10)$$
$$X_{val} : (5000\ ,\ 784)\ \ ,\ \ Y_{val} : (5000\ ,\ 10)$$
$$X_{test} : (10000\ ,\ 784)\ \ ,\ \ Y_{test} : (10000\ ,\ 10)$$

所有的 input variable 都 element-wise 的除以 255，也就是對 data 做 normalize 用以避免一些太大的數值影響後續參數的估計。

2 Shallow Neural Network

2.1 Architecture

我們建構一個兩層 hidden layers 的 shallow neural network model，兩層 hidden layers 的 neuron 個數分別爲 $(600, 400)$，每層的 hidden layer 使用 ReLU activation function，output layer 使用 softmax activation function，模型的架構如 Figure 1 所示，最後再利用模型預測出來的類別以及眞實資料的類別計算 Accuracy 和 Cross-entropy loss : $L(y, \hat{y}) = -\sum y \log(\hat{y})$

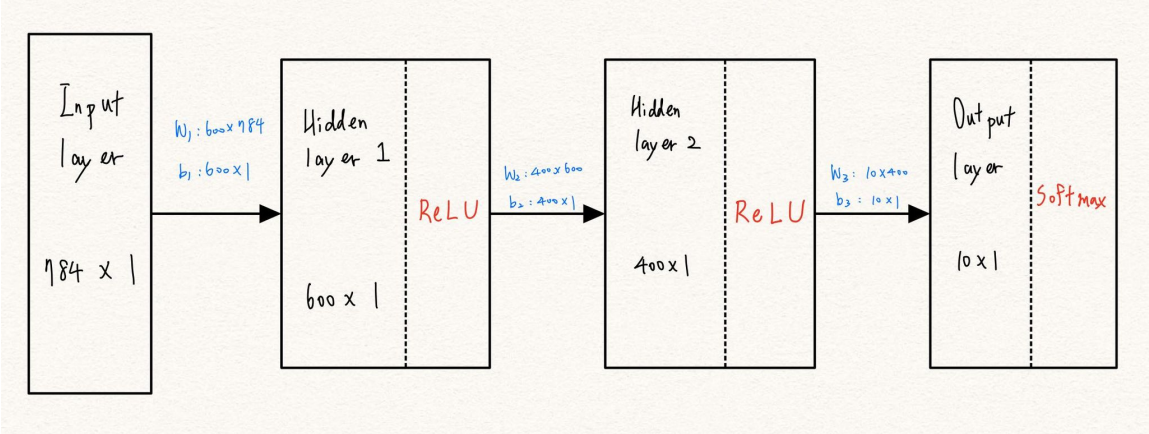

Figure 1: Architecture of model

2.2 Implementation Detail for gradient of loss

Cross-entropy loss : $L = \sum_{n=1}^{M} L^{(n)}$, and our goal is to evaluate $\frac{\partial L^{(n)}}{\partial W_{i,j}^{(k)}}$ in each k-th layer. By chain rule :

$$\frac{\partial L^{(n)}}{\partial W_{i,j}^{(k)}} = \frac{\partial L^{(n)}}{\partial z_j^{(k)}} \cdot \frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$$

The second term $\frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$ can be computed by forward pass

$$\frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}} = \begin{cases} x_i^{(n)} & \text{,for } k = 1 \\ a_i^{(k-1)} = act(z_i^{k-1}) & \text{,for } k > 1 \end{cases}$$

details can be seen in Listing 1

```
def feed_forward(self, x):
        self.cache["X"] = x
        self.cache["Z1"] = np.matmul(self.params["W1"],
                                     self.cache["X"].T) + self.params["b1"]
        self.cache["A1"] = self.activation(self.cache["Z1"])
        self.cache["Z2"] = np.matmul(self.params["W2"],
                                     self.cache["A1"]) + self.params["b2"]
        self.cache["A2"] = self.activation(self.cache["Z2"])
        self.cache["Z3"] = np.matmul(self.params["W3"],
                                     self.cache["A2"]) + self.params["b3"]
        self.cache["A3"] = self.softmax(self.cache["Z3"])
        return self.cache["A3"]
```

Listing 1: Forward Pass

and the first term $\delta_j^{(k)} = \frac{\partial L^{(n)}}{\partial z_j^{(k)}}$ can be computed by backward pass

$$\delta_j^{(k)} = \frac{\partial L^{(n)}}{\partial z_j^{(k)}} = \frac{\partial L^{(n)}}{\partial a_j^{(k)}} \cdot \frac{\partial a_j^{(k)}}{\partial z_j^{(k)}} = \frac{\partial L^{(n)}}{\partial a_j^{(k)}} \cdot act'(z_j^{(k)})$$

$$= \left[\sum_s \frac{\partial L^{(n)}}{\partial z_s^{(k+1)}} \cdot \frac{\partial z_s^{(k+1)}}{\partial a_j^{(k)}} \right] \cdot act'(z_j^{(k)})$$

$$= \left[\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)} \right] \cdot act'(z_j^{(k)})$$

details can be seen in Listing 2

```
def back_propagate(self, y, output):
        current_batch_size = y.shape[0]

        dZ3 = output - y.T
        dW3 = (1/current_batch_size) *
              np.matmul(dZ3, self.cache["A2"].T)
        db3 = (1/current_batch_size) *
              np.sum(dZ3, axis=1, keepdims=True)

        dA2 = np.matmul(self.params["W3"].T, dZ3)
        dZ2 = dA2 * self.activation(self.cache["Z2"],
                                    derivative=True)
        dW2 = (1/current_batch_size) *
              np.matmul(dZ2, self.cache["A1"].T)
        db2 = (1/current_batch_size) *
              np.sum(dZ2, axis=1, keepdims=True)

        dA1 = np.matmul(self.params["W2"].T, dZ2)
        dZ1 = dA1 * self.activation(self.cache["Z1"],
                                    derivative=True)
        dW1 = (1/current_batch_size) *
              np.matmul(dZ1, self.cache["X"])
        db1 = (1/current_batch_size) *
              np.sum(dZ1, axis=1, keepdims=True)

        self.grads = {"W1":dW1, "b1":db1, "W2":dW2, "b2":db2,
                      "W3":dW3, "b3":db3}
        return self.grads
```

Listing 2: Backward Pass

2.3 Training and Validation

模型參數更新的方式是採用 mini-batch 加上 Momentum，演算法如下：

$$v^{(t+1)} \leftarrow \beta v^{(t)} - (1-\beta)\nabla L(y,\hat{y})$$
$$w^{(t+1)} \leftarrow w^{(t)} - \eta v^{(t+1)}$$

(β,η) 都是 hyperparameter，Momentum 相較於一般的 mini-batch SGD，比較不會錯誤的收斂在 local minimum or saddle points。

訓練模型的途中可以藉由觀察 training 和 validation dataset 各自的 Accuracy 和 Cross-entropy loss 得知模型是否有 overfitting 的現象，藉此來挑整 hyperparameter，例如：Figure 2 中可以看到在 Epoch 超過 200 後 validation loss 開始逐漸上升，代表模型在此時已經有 overfitting 的情況發生了，這種情況可以提早停止我們模型的 training，這也是爲什麼我們一開始需要從 train data 中額外多分割出一份 validation data 的原因，因爲只觀察 train data 的話，是無法發現 overfitting 的現象。其餘的 hyperparameter 也是用類似的同時觀察 train and validation loss 所得，在此就不多加贅述。

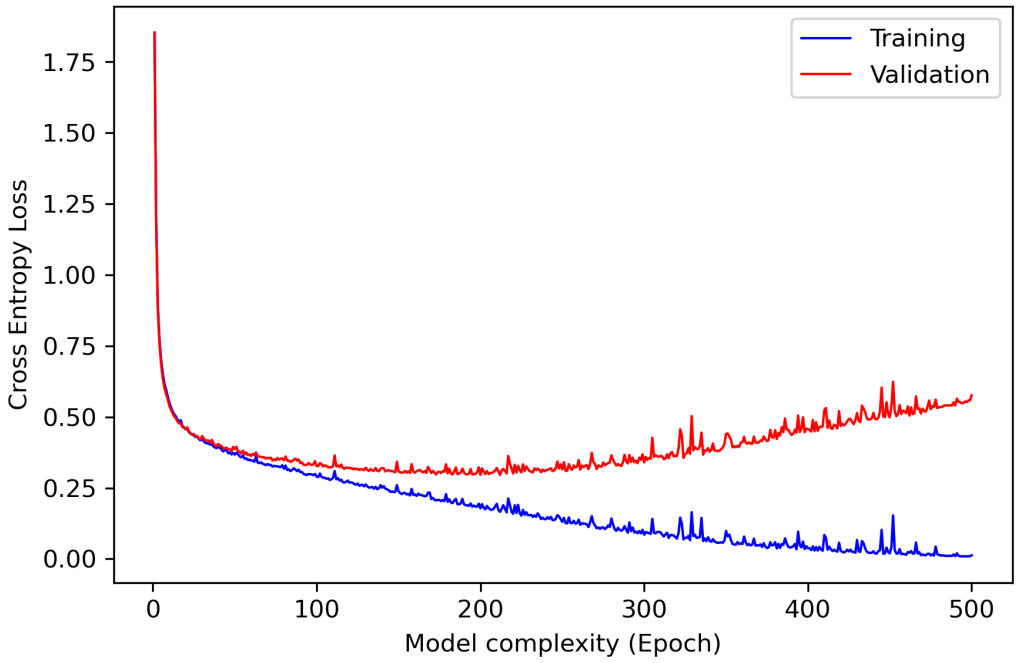

Figure 2: Overfitting Example

2.4 Testing Result

選定的 hyperparameter 如下：

Epoch = 500 , Batch size = 500 , Learning rate $\eta = 0.01$, $\beta = 0.8$

如 Listing 3 所示，可以藉由更改 *train.py* 中的參數設定值更改包括：neuron, epoch, batch size, learning rate, β，各項 hyperparameter，但若想要更改模型 layer 數量，則需改寫 *feed_forward* 和 *back_propagate* 的部分

```
snn = ShallowNeuralNetwork(sezes=[784,600,400,10],
                           epochs=500, activation="relu")
snn.train(x_train, y_train, x_val, y_val, batch_sizes=500,
          optimizer="momentum", l_rate=0.01, beta=0.8)
```

Listing 3: Setting hyperparameter

模型對 train data 以及 validation data 的 accuracy 和 cross-entropy loss 呈現於 Figure 3 & 4，可以看出差不多 500 epochs 的時候，Validation loss 呈現穩定不再明顯下降，Validation accuracy 在此時也已經超過 87%，那麼此時就可以停止訓練模型了，然後再將訓練好的模型對 test data 預測並計算準確率和 loss，可得 Accuracy = 88.84%, Loss = 0.3301，實驗結果可見於 *test.py*。

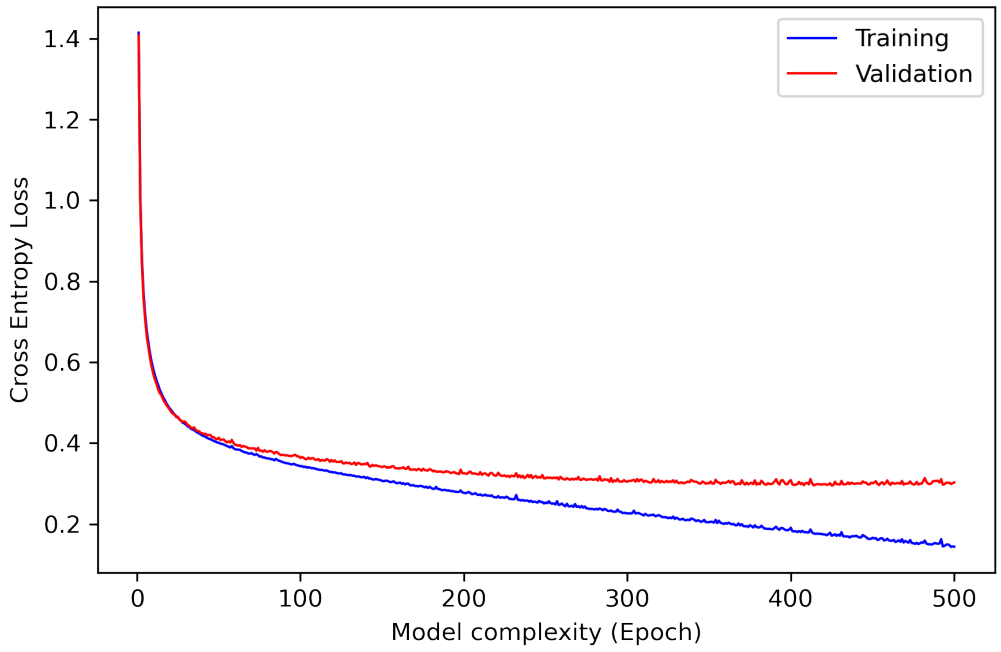

Figure 3: ReLU : Train v.s. Validation loss

3 Some Questions

3.1 What about deep NN?

隨著模型的層數增加，的確會使得 training accuracy 上升，但 validation (testing) accuracy 不一定也會上升，如同 Figure 5 所示，層數從 1 逐漸增加到

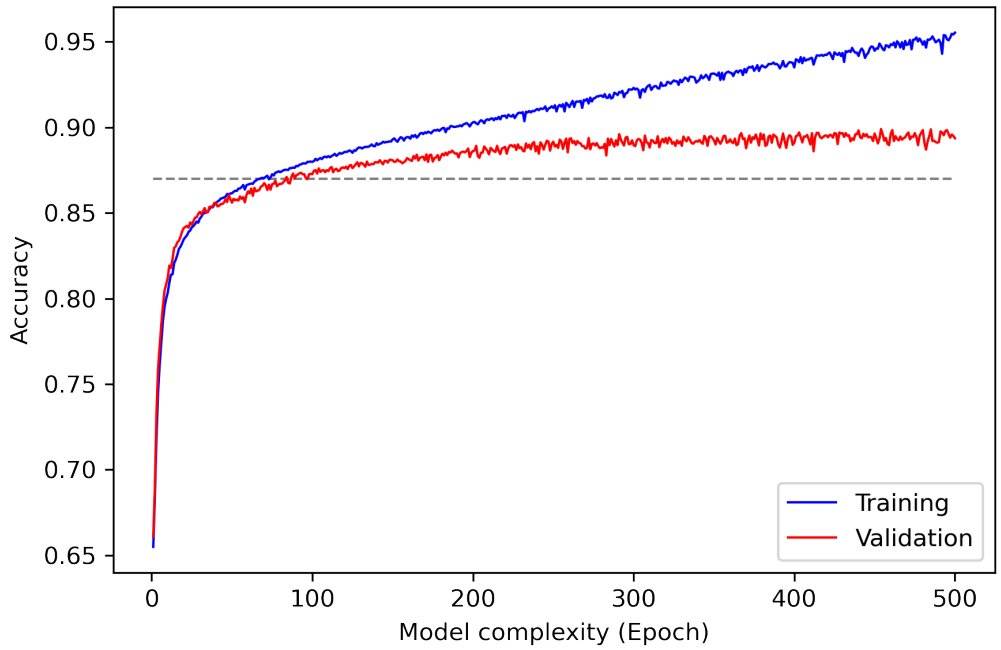

Figure 4: ReLU : Train v.s. Validation accuracy

4，training loss 逐漸下降(training accuracy 逐漸上升)，但當層數>2 時，validation loss 不降反升(validation accuracy 不升反降)，會產生這種現象就是因爲模型太過複雜，產生了 overfitting 的現象。

3.2 Other activation function

在其他 hyperparameter 都不變的情況下，將 activation function 分別改成 sigmoid, tanh, hard tanh 三種，各自去訓練模型，然後觀察其預測值在 test data 上的表現，結果呈現如下表：

Activation	**ReLU**	**Sigmoid**	**tanh**	**Hard tanh**
Test loss	0.3301	0.4657	0.3426	0.3408
Test accuracy	0.8884	0.8331	0.8799	0.8852

使用 Sigmoid, tanh, Hard tanh 訓練時的 train (validation) loss and accuracy 各自呈現於 Figure 6 ∼ 11，可以看出像是 Sigmoid, tanh 這種連續性的 activation function 模型的收斂速度會慢於 ReLU, Hard tanh 這種 piecewise linear activation function，在相同的 epochs 下，對 test data 的預測準確率也較差。

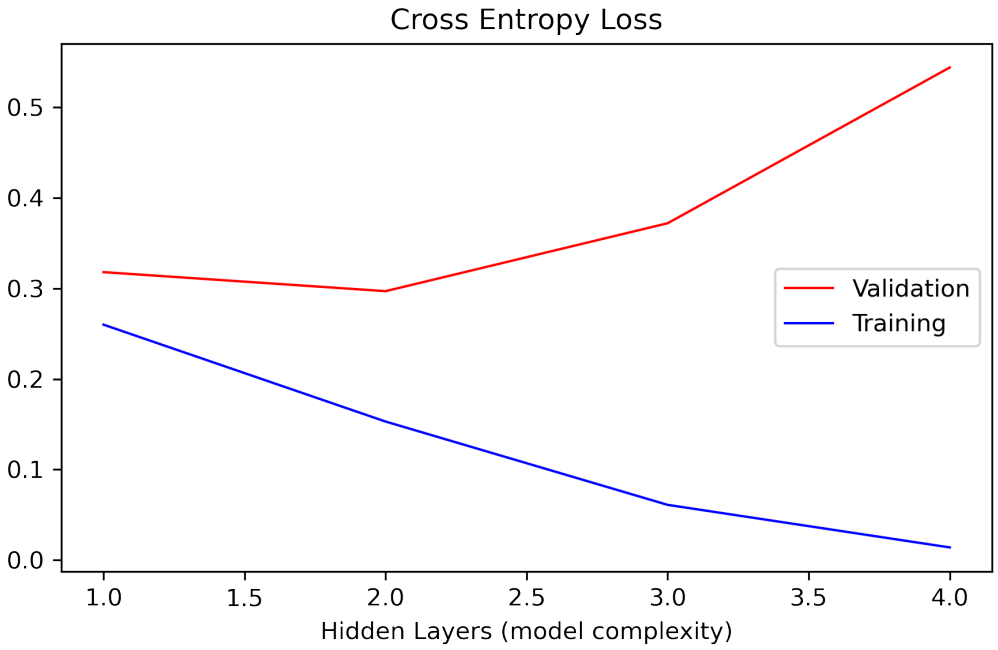

Figure 5: Loss in different Layers

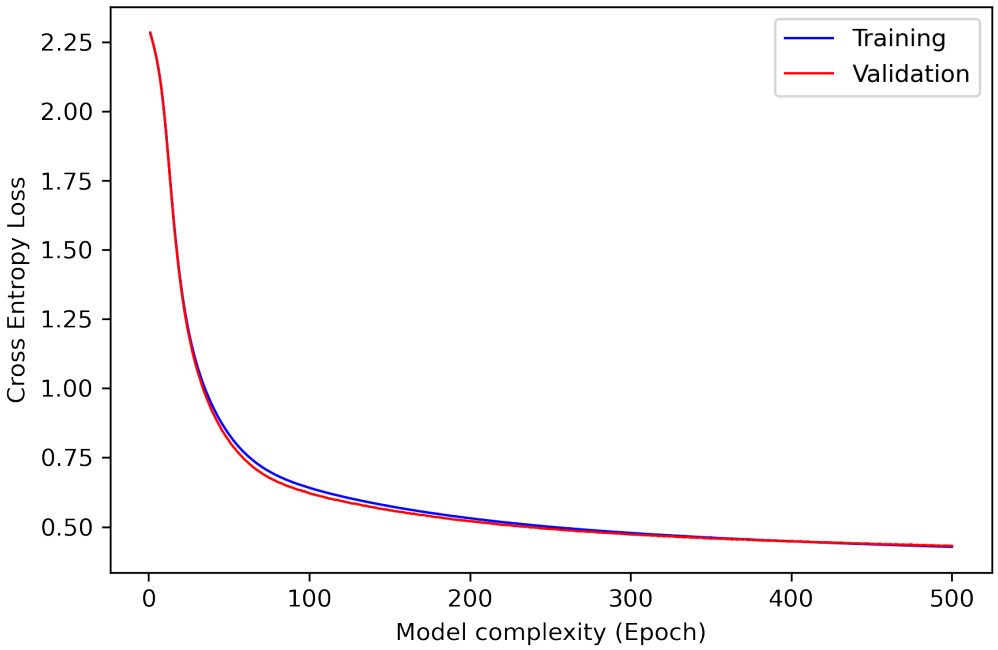

Figure 6: Sigmoid : Train v.s. Validation loss

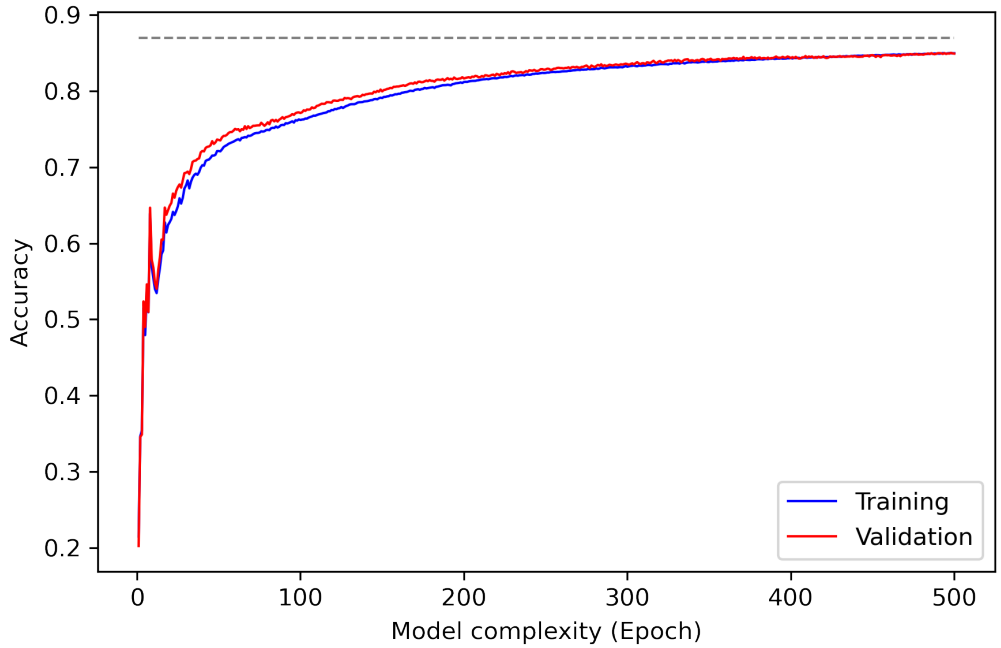

Figure 7: Sigmoid : Train v.s. Validation accuracy

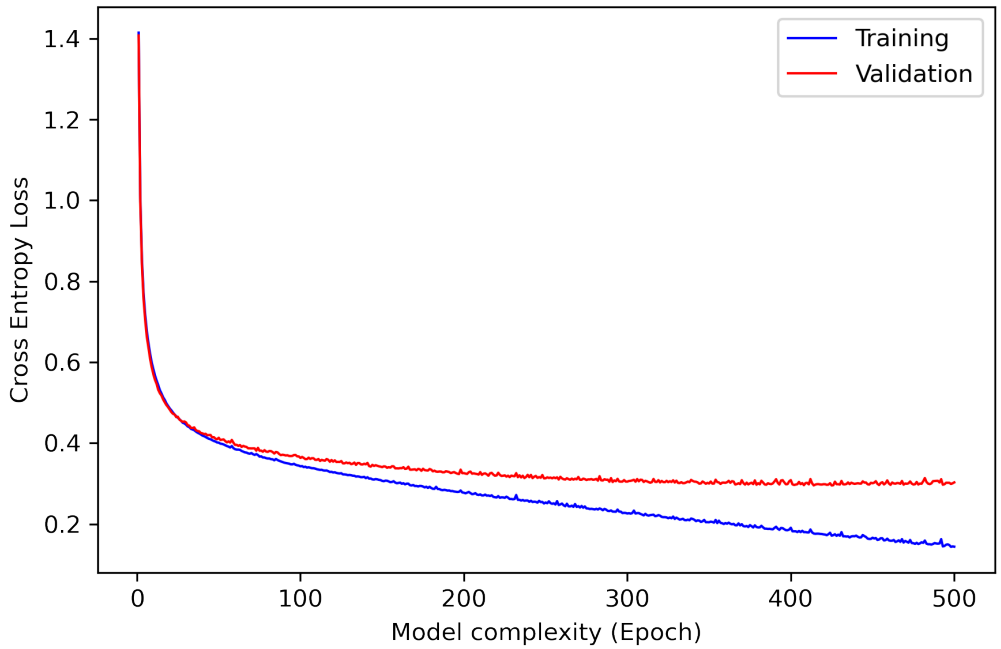

Figure 8: tanh : Train v.s. Validation loss

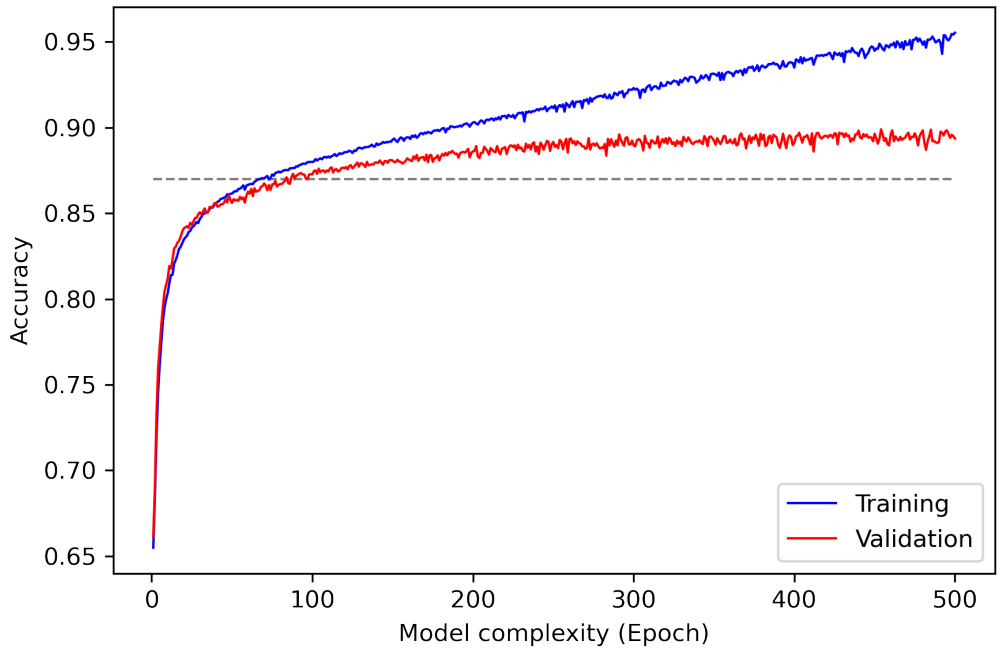

Figure 9: tanh : Train v.s. Validation accuracy

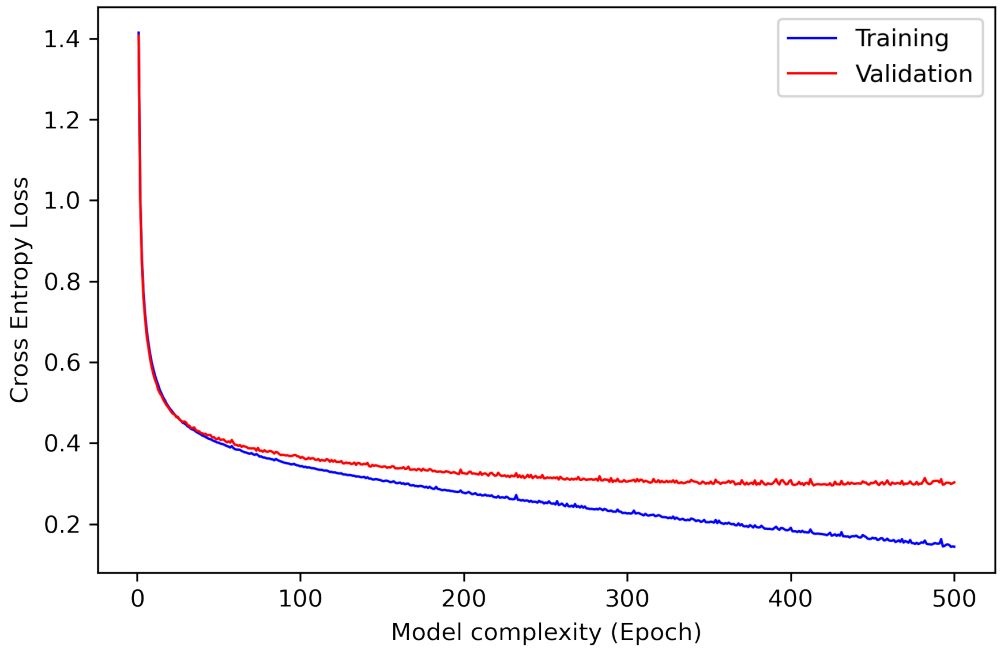

Figure 10: Hard tanh : Train v.s. Validation loss

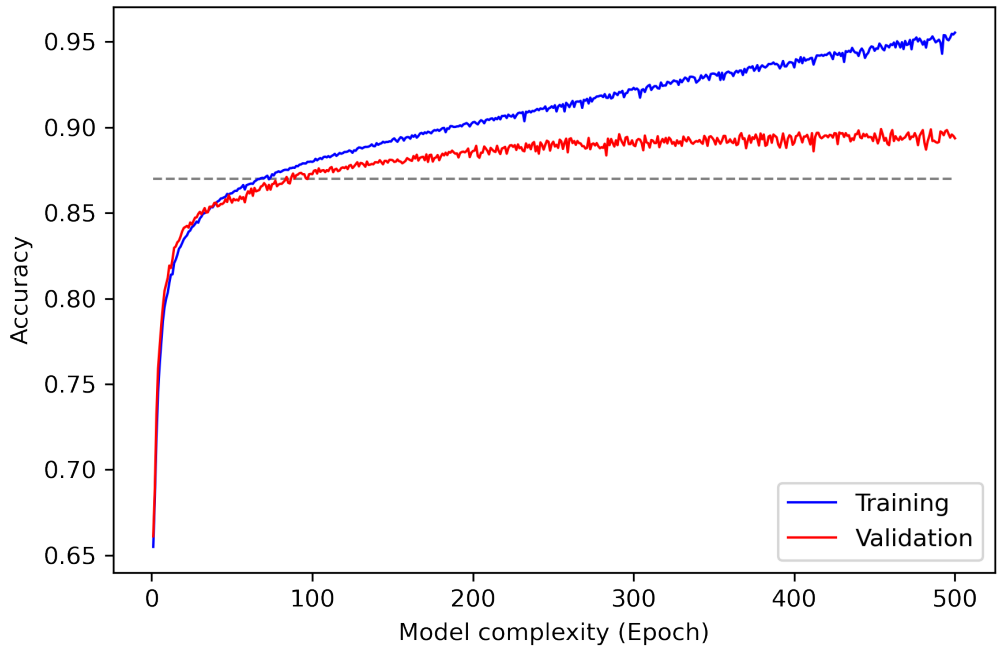

Figure 11: Hard tanh : Train v.s. Validation accuracy