(1) Softmax Reg.

$$W = [\, w^{(1)} \; \cdots \; w^{(C)} \,]$$

$$\hat{y}_k = \frac{\exp z_k}{\sum_{k'=1}^{C} \exp z_{k'}}$$

$$z_k = x^T w^k + b_k$$

$$f_{CE}(W, b) = -\frac{1}{n} \sum_{p=1}^{n} \sum_{k=1}^{C} y_k^{(p)} \cdot \log \hat{y}_k^p$$

$$\nabla_{w^{(l)}} f_{CE}(W, b) = -\frac{1}{n} \sum_{p=1}^{n} \sum_{k=1}^{C} y_k^p \nabla_{w^l} \log \hat{y}_k^p$$

$$= -\frac{1}{n} \sum_{p=1}^{n} \sum_{k=1}^{C} y_k^p \left(\frac{\nabla_{w^l} \hat{y}_k^l}{\hat{y}_k^p} \right)$$

Case 1 for $l = k$

$$\nabla_{w(l)} \hat{y}_k = \nabla_{w(l)} \frac{\exp z_k}{\sum_{k'=1}^{c} \exp z_{k'}}$$

$$\frac{\partial}{\partial x}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x) \cdot f'(x) - f(x)\, g'(x)}{[g(x)]^2}$$

$$\therefore \sum_{k'=1}^{c} \exp z_{k'} \cdot \nabla_{w(l)} \exp z_k -$$

$$\exp z_k \cdot \nabla_{w(l)} \sum_{k'=1}^{c} \exp z_{k'}$$

$$\left(\sum_{k'=1}^{c} \exp z_{k'}\right)^2$$

$$\frac{\sum_{k'=1}^{c} \exp^{z_{k'}} \cdot \exp^{z_k} \cdot x^{(p)} - \exp^{z_k} \cdot x^{(p)} \cdot \sum_{k'=1}^{c} \exp^{z_{k'}}}{\sum_{k'=1}^{c} \exp^{z_{k'}} \quad \sum_{k'=1}^{c} \exp^{z_{k'}}}$$

$$\left(\sum_{k'=1}^{c} \exp^{z_{k'}} \right)^2$$

$$= x^{(i)} \frac{\exp^{z_k}}{\sum_{k'=1}^{c} \exp^{z_{k'}}} \left[1 - \frac{\exp^{z_k}}{\sum_{k'=1}^{c} \exp^{z_{k'}}} \right]$$

$$= x^{(i)} \hat{y}_k^i (1 - \hat{y}_k^i) \quad \cdots (1)$$

Now for case (2) $l \neq k$ we get

$$= \frac{\sum_{k'=1}^{c} \exp^{z_{k'}} \cdot (0) - e^{z_k} \cdot x^{(i)} \cdot e^{z_{k'}}}{\sum_{k'=1}^{c} \exp^{z_{k'}}}$$

$$= x^{(i)} \cdot \hat{y}_k^i \cdot \hat{y}_l^i \quad \cdots (2)$$

Now to find total gradient of f_{CE}.

$$\nabla_{w^{(l)}} f_{CE}(W, b) = \frac{-1}{n} \sum_{i=1}^{n} \sum_{k=1}^{c} y_k^i \nabla_{w^l} \log \hat{y}_k^i$$

$$= \frac{-1}{n} \sum_{i=1}^{n} \sum_{k=1}^{c} y_k^i \frac{1}{\hat{y}_k^i} \cdot \nabla_{w^l} \hat{y}_k^i$$

$$= \frac{-1}{n} \sum_{i=1}^{n} \sum_{k=1}^{c} y_k^i \cdot \frac{1}{\hat{y}_k^i} \left[x^i \hat{y}_l^i (1 - \hat{y}_l^i) - x^i \hat{y}_l^i \hat{y}_k^i \right]$$

$\sum_k y_k = 1$ given & $\sum_k \hat{y}_k = 1$ (Softmax layer sums to 1)

$$= \frac{-1}{n} \sum_{i=1}^{n} \sum_{k=1}^{c} y_k^i \cdot \frac{1}{\hat{y}_k^i} \cdot x^i \hat{y}_l^i \left[1 - \hat{y}_l^i - \hat{y}_k^i \right]$$

Also $1 - y_l^i = \hat{y}_k^i$

$$= \frac{-1}{n} \sum_{i=1}^{n} \sum_{k=1}^{c} 1 \cdot \frac{1}{1} \cdot x^i \left[1 - \hat{y}_l^i - 1 + y_l^i \right]$$

$$= \frac{-1}{n} \sum_{i=1}^{n} x^i \left(y_i^i - \hat{y}_i^i \right) \quad \text{--- (3)}$$

Also ∇_b for $c \in (1, 2, b)$ will be

$$\nabla_b \left(\frac{-1}{n} \sum_{i=1}^{n} \sum_{k=1}^{c} y_k^i \log \hat{y}_k^i \right)$$

$$= \frac{-1}{n} \sum_{i=1}^{n} \sum_{k=1}^{c} y_k^i \nabla_b \log \left(\frac{\exp^{z_k}}{\sum_{k'=1}^{c} \exp^{z_{k'}}} \right)$$

$$\nabla_b \log \left(\frac{\exp^{z_k}}{\sum_{k'=1}^{c} \exp^{z_{k'}}} \right) = \nabla_b \log \left(\exp^{z_k} \right) - \nabla_b \log \left(\sum_{k'=1}^{c} \exp^{z_{k'}} \right)$$

$$= 1 - \frac{\exp^{z_{k'}}}{\sum_{k'=1}^{c} \exp^{z_{k'}}}$$

$$\therefore \nabla_b \Rightarrow \frac{-1}{n} \sum_{i=1}^{n} \sum_{k=1}^{c} y_k^i \left[1 - \frac{\exp^{z_{k'}}}{\sum_{k'=1}^{c} \exp^{z_{k'}}} \right]$$

$$= \frac{-1}{n} \sum_{i=1}^{n} \left[\sum_{k=1}^{c} y_k^i - \sum_{k=1}^{c} y_k^i \cdot \hat{y}_k^i \right]$$

$$= \frac{-1}{n} \sum_{i=1}^{n} \left(y^i - \hat{y}^i \right) \quad \text{--- (4)}$$

(2) Cross - Entropy as Neg. log-

$\hat{y}_k = P(y_k = 1 \mid x, w, b)$ $\forall k \in (1 \ldots c)$

Now $P(y \mid x, w, b) = P(y_1 = 1 \mid x, w, b)^{y_1} \times \ldots$

$\therefore P(y_c = 1 \mid x, w, b)^{y_c}$

$$P(y \mid x, W, b) = \prod_{k=1}^{C} \hat{y}_k^{y_k}$$

$$D = \{ x^{(q)}, y^{(q)} \}_{q=1}^{n}$$

$$-\log P(D \mid W, b) = -\log P(y \mid x, W, b)$$

$$= -\log \left(\prod_{k=1}^{C} \hat{y}_k^{y_k} \right) = -\sum_{k=1}^{C} \log \left(\hat{y}_k^{y_k} \right)$$

Now $\log a^b = b \cdot \log a$

$\therefore$ we get $-\sum_{k=1}^{C} y_k \log \hat{y}_k$ & to sum over all samples we get

$$-\sum_{q=1}^{n} \sum_{k=1}^{C} y_k^q \log \hat{y}_k^q = \mathcal{L}(D; W, b)$$

(3) $f\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \left[x_1 - 2x_2 + \frac{x_3}{4}\right]$

$$g\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} x_1 + 2x_2 \\ x_2 \\ -x_1 + 3 \end{bmatrix}$$

(4) $g = \sigma(W_1 x + b_1)$ where σ = lpnear or transformation

$f = \sigma(W_2 x + b_2)$ non-lpnear ~~o/tput~~

$$\nabla g(x) = \begin{bmatrix} \nabla_{x_1}(x_1 + 2x_2) & \nabla_{x_2}(x_1 + 2x_2) \\ \nabla_{x_2}(x_2) & \nabla_{x_2} x_2 \\ \nabla_{x_1}(-x_1 + 3) & \nabla_{x_2}(-x_1 + 3) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ -1 & 0 \end{bmatrix} \quad B_{Pad} = \begin{bmatrix} 0 \\ 0 \\ 3 \end{bmatrix}$$

$$\therefore g([x_1\ x_2]) = \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 3 \end{bmatrix}$$

$$= W_1 x + b_1$$

Now for f using similar approach we get

$$= \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \begin{bmatrix} 1 & -2 & \frac{1}{4} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\Rightarrow f([x_1\ x_2\ x_3]) = \begin{bmatrix} 1 & -2 & \frac{1}{4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$= W_2 x + b_2$$

$g = \sigma(W_1 x + b_1)$ $\sigma \rightarrow$ linear or non-linear transformation fn

$f = \sigma(W_2 x + b_2)$

i.e $f = \sigma\{W_2 (W_1 x + b_1) + b_2\}$

(3e) $z_2 = f(z_1)$ $\quad \frac{\partial z_2}{\partial b_2}(z_1) = ?$

$z_2 = W_2(z_1) + b_2$ $\quad \therefore \frac{\partial z_2}{\partial b_2} = [0+1] = 1$

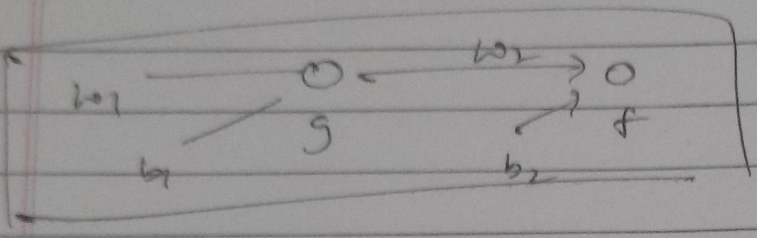
(3f) given $g = W_1 x + b_1$ $\quad f = W_2 x + b_2$

$$\frac{\partial (f \circ g)(x)}{\partial b_2} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial b_2} = 1$$

i.e $f(W_1 x + b_1) = h(x)$

3(d) $\frac{\partial (f \circ g)}{\partial b_1} x = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial b_1}$

$\frac{\partial}{\partial b_1}[w_1 x + b_1] = 1$ & $\frac{\partial}{\partial b_1}(f \circ g)(x) = 0$.



structure

3(e) $\text{unvec}\left[\frac{\partial (f \circ g)(x)}{\partial \text{vec}[W_2]}\right]$

$\partial \text{vec}[W_2] = ?$ $\quad W \Rightarrow w_{n \times m}$ matrix

$\text{vec}[W] = [w_{11} \cdots w_{1m} \cdots w_{n1} \cdots w_{nm}]$

$\frac{\partial (f \circ g)}{\partial \text{vec}[W_2]} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial \text{vec}[W_2]}$ $\quad \frac{\partial f}{\partial g} = p^T$

$\frac{\partial g}{\partial \text{vec}[W_2]} = \left[\frac{\partial g_1}{\partial w^2_{11}} \cdots \frac{\partial g_1}{\partial w^2_{1m}} \cdots \frac{\partial g_1}{\partial w^2_{n1}} \cdots \frac{\partial g_1}{\partial w^2_{nm}}\right]$

$\text{unvec}\left(\frac{\partial (f \circ g)}{\partial \text{vec}[W_2]}\right) = p^T$ ○

3(g) $\text{unvec}\left[\frac{\partial (f \circ g)(x)}{\partial \text{vec}[W_1]}\right] = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial \text{vec}[W_1]}$

Suppose $\frac{\partial f}{\partial g} = p^T$ $\quad \frac{\partial g}{\partial \text{vec}[W_1]} = \left[\frac{\partial g_1}{\partial w^1_{11}} \cdots \frac{\partial g_1}{\partial w^1_{1m}} \cdots \right.$

$\frac{\partial g}{\partial \text{vec}[W_1]} = \begin{bmatrix} x^T & \cdots & 0^T \\ \vdots & & \\ 0^T & \cdots & x^T \end{bmatrix} = x^T \quad \left. \frac{\partial g_1}{\partial w^1_{n1}} \cdots \frac{\partial g_1}{\partial w^1_{nm}}\right]$

$\therefore \text{unvec}\left(\frac{\partial (f \circ g) x}{\partial \text{vec}[W_1]}\right) = p^T \cdot x^T \overset{\text{unvec}}{\Rightarrow} [p_1 x^T \cdots p_n x^T]$

$= p x^T$

```python
# hw_3 problem 3b
import numpy as np
def AffineTransformation(W: np.ndarray,x: np.ndarray,b: np.ndarray) -> np.ndarray:
  return np.dot(W,x)+b
W1: np.ndarray = np.array([[1, 2], [0, 1],[-1, 0]])
b1: np.ndarray = np.array([0, 0, 3])
W2: np.ndarray = np.array([[1, -2, 1/4]])
b2: np.ndarray = np.array([0])
x1: np.ndarray = np.array([1, 2])
x2: np.ndarray = np.array([1, 2, 3])
result_g: np.ndarray = AffineTransformation(W1,x1,b1)
result_f: np.ndarray = AffineTransformation(W2,x2,b2)
print(f"Function g result: {result_g = } & Function f result: {result_f = }")
```

```
Function g result: result_g = array([5, 2, 2]) & Function f result: result_f = array([-2.25])
```

```python
# hw_3 problem 3c
def Composition(L: list, x: np.ndarray) -> list:
  z_values: list = list()
  for W,b in L:
    # get pre-activation
    z = AffineTransformation(W,x,b)
    z_values.append(z)
    # use for the next iteration
    x: np.ndarray = z
  return z_values

L: list = [
  (np.array([[1,2,1]]), np.array([3,4,5])),
  (np.array([[1,-2,1/4]]), np.array([10,20,30])),
]

x: np.ndarray = np.array([100,200,300])
z_values: list = Composition(L,x)
print(f"Composition result: {z_values = }")
```

```
Composition result: z_values = [array([803, 804, 805]), array([-593.75, -583.75, -573.75])]
```

```python
# hw_3 problem 3d
final_output = Composition(L,x)[-1]
# Print the final output
print(f"Final Output (f o g)(x): {final_output = }")
```

```
Final Output (f o g)(x): final_output = array([-593.75, -583.75, -573.75])
```

```python
# hw_3 problem 3h
def ComputeJacobians(L: np.ndarray, x: np.ndarray, z: np.ndarray) -> tuple:
    num_layers = len(L)
    dz_db_list: list = list()
    dz_dW_list: list = list()
    # Compute the Jacobians for each layer in reverse order
    for i in range(num_layers - 1, -1, -1):
        W, b = L[i]
        z_i: np.ndarray = z[i]
        dz_dx: np.ndarray = W
        dz_db: np.ndarray = np.identity(len(b))
        dz_dW: np.ndarray = np.outer(np.ones_like(z_i),x)
        dz_db_list.append(np.dot(dz_dx, dz_db))
        dz_dW_list.append(np.dot(dz_dx, dz_dW))
        x: np.ndarray = AffineTransformation(W,x,b)
    dz_db_list.reverse()
    dz_dW_list.reverse()
    return dz_dW_list,dz_db_list
dz_dW,dz_db = ComputeJacobians(L, x, z_values)
print(f"Jacobians : \nw -> {dz_dW = } \nb -> {dz_db = }")
```

```
Jacobians :
w -> dz_dW = [array([[-860., -820., -780.]]), array([[ -75., -150., -225.]])]
b -> dz_db = [array([[1., 2., 1.]]), array([[ 1.  , -2.  ,  0.25]])]
```

```python
# hw_3 problem 4
import numpy as np
X_tr = np.load("fashion_mnist_train_images.npy")
y_tr = np.load("fashion_mnist_train_labels.npy")
X_te = np.load("fashion_mnist_test_images.npy")
y_te = np.load("fashion_mnist_test_labels.npy")
print(X_tr.shape,y_tr.shape,X_te.shape,y_te.shape)

split_80_20: int = int(X_tr.shape[0]*0.8)
print('80 / 20 Split : ',split_80_20)
X_tr_80, y_tr_80 = X_tr[:split_80_20,:], y_tr[:split_80_20]
X_vl_20, y_vl_20 = X_tr[split_80_20:,:], y_tr[split_80_20:]
del X_tr,y_tr
print(X_tr_80.shape,y_tr_80.shape,X_te.shape,y_te.shape,X_vl_20.shape,y_vl_20.shape)

grid_search_mini_batch_size : list = [100,120,140,160,180,200]
grid_search_learning_rate : list = [3e-7,3e-5,3e-3,3e-1]
grid_search_no_of_epochs : list = [8,16,20,24,32,48,64,80]
grid_search_regularization_val : list = [1e-5,1e-3,1e-1]

# custom helper functions
def get_initialize_weights_and_bias_liner_reg(input_dims:int,output_dims:int) -> tuple:
  return np.random.uniform(low=1e-5,high=1e-2,size=(input_dims,output_dims)),np.random.random(size=output_dims)

def get_y_hat_pred_func(x: np.ndarray, w: np.ndarray, b: np.ndarray) -> np.ndarray:
  return np.dot(x, w) + b

def get_softmax(y_pred: np.ndarray) -> np.ndarray:
    exp_y_pred = np.exp(y_pred - np.max(y_pred, axis=1, keepdims=True))
    return exp_y_pred / np.sum(exp_y_pred, axis=1, keepdims=True)

def get_cross_entropy_regularized_loss(y_true: np.ndarray,y_pred: np.ndarray,W: np.ndarray,alpha_reg: float) -> float:
    n = len(y_true)
    # addition of 1e-10 is to silence - RuntimeWarning: divide by zero encountered in log
    cross_entropy = -1.0 * np.sum(y_pred * np.log(y_pred + 1e-10)) + (0.5 * alpha_reg) * np.sum(W.T @ W)
    cross_entropy = cross_entropy / n
    return cross_entropy
```

```python
def get_cross_entropy_unregularized_loss(y_true: np.ndarray,y_pred: np.ndarray) -> float:
    n = len(y_true)
    cross_entropy = -1.0 * np.sum(y_pred * np.log(y_pred + 1e-10))
    cross_entropy = cross_entropy / n
    return cross_entropy

# hyperparameter tuning
best_params: dict = {
                    'batch_size': 0,
                    'learning_rate': 0,
                    'num_epochs': 0,
                    'regularization_strength': 0,
                    'reg_loss': 0,
                    'unreg_loss': 0,
                    'accuracy': 0,
                }

best_loss: float = 10_000.0
best_accuracy:float = 0.70

# Shuffle training data
shuffling_indices: np.ndarray = np.arange(X_tr_80.shape[0])
np.random.shuffle(shuffling_indices)
X_train_shuffled: np.ndarray = X_tr_80[shuffling_indices]
y_train_shuffled: np.ndarray = y_tr_80[shuffling_indices]

input_dims1,output_dims1 = X_train_shuffled.shape[1],10
```

```python
for initial_mini_batch_size in grid_search_mini_batch_size:
  for initial_no_of_epochs in grid_search_no_of_epochs:
    for initial_learning_rate in grid_search_learning_rate:
      for initial_regularization_strength in grid_search_regularization_val:

        # Initialize weights and bias
        w, b = get_initialize_weights_and_bias_liner_reg(input_dims1,output_dims1)

        # Training Loop
        for epoch in range(initial_no_of_epochs):

          for batch_start in range(0, X_tr_80.shape[0], initial_mini_batch_size):

            # taking mini batches from training data
            X_mini_batch: np.ndarray = X_train_shuffled[batch_start:batch_start+initial_mini_batch_size,:]
            # X_mini_batch: np.ndarray = X_mini_batch / 255.0
            len_X_mini_batch = len(X_mini_batch)
            y_mini_batch: np.ndarray = y_train_shuffled[batch_start:batch_start+initial_mini_batch_size]

            # we compute the y hat predict
            y_pred_mini: np.ndarray = get_softmax(get_y_hat_pred_func(X_mini_batch, w, b))

            # Backpropagation
            # when pred is correct for the class remove it so that false case gets more weight
            y_pred_mini[np.arange(len(y_mini_batch)), y_mini_batch] -= 1

            # we compute gradient of w and b
            grad_w: np.ndarray = np.dot(X_mini_batch.T, y_pred_mini)
            grad_b: np.ndarray = np.sum(y_pred_mini, axis=0)

            # update learning weights and biases
            w -= initial_learning_rate * (grad_w + initial_regularization_strength * w)
            b -= initial_learning_rate * grad_b

        # Validation
        valid_probabilities: np.ndarray = get_softmax(get_y_hat_pred_func(X_vl_20,w,b))
        valid_reg_loss: float = get_cross_entropy_regularized_loss(y_vl_20,valid_probabilities,w,initial_regularization_strength)
        valid_unreg_loss: float = get_cross_entropy_unregularized_loss(y_vl_20,valid_probabilities)
        valid_accuracy: float = np.mean(np.argmax(valid_probabilities, axis=1)==y_vl_20)
```

```
            grad_b: np.ndarray = np.sum(y_pred_mini, axis=0)

            # update learning weights and biases
            w -= initial_learning_rate * (grad_w + initial_regularization_strength * w)
            b -= initial_learning_rate * grad_b

        # Validation
        valid_probabilities: np.ndarray = get_softmax(get_y_hat_pred_func(X_vl_20,w,b))
        valid_reg_loss: float = get_cross_entropy_regularized_loss(y_vl_20,valid_probabilities,w,initial_regularization_strength)
        valid_unreg_loss: float = get_cross_entropy_unregularized_loss(y_vl_20,valid_probabilities)
        valid_accuracy: float = np.mean(np.argmax(valid_probabilities, axis=1)==y_vl_20)

        if valid_unreg_loss > 0 and valid_reg_loss > 0 and best_accuracy > 0.70:
          if valid_unreg_loss < best_loss and valid_accuracy < best_accuracy:
              best_loss: float = valid_unreg_loss
              best_accuracy: float = valid_accuracy
              best_params = {
                  'batch_size': initial_mini_batch_size,
                  'learning_rate': initial_learning_rate,
                  'num_epochs': initial_no_of_epochs,
                  'regularization_strength': initial_regularization_strength,
                  'valid_reg_loss': valid_reg_loss,
                  'valid_unreg_loss': valid_unreg_loss,
                  'valid_accuracy': valid_accuracy,
              }
          else:
            continue
        else:
          break

# testing dataset
test_probabilities: np.ndarray = get_softmax(get_y_hat_pred_func(X_te,w,b))
test_reg_loss: float = get_cross_entropy_regularized_loss(y_te,test_probabilities,w,initial_regularization_strength)
test_unreg_loss: float = get_cross_entropy_unregularized_loss(y_te,test_probabilities)
test_accuracy: float = np.mean(np.argmax(test_probabilities, axis=1)==y_te)
best_params['test_reg_loss'] = test_reg_loss
best_params['test_unreg_loss'] = test_unreg_loss
best_params['test_accuracy'] = test_accuracy
print(f"Computing Loss on Test Data Set ->\nBest Test Reg Loss: {test_reg_loss},\nBest Test UnReg Loss: {test_unreg_loss},\nBest Accuracy: {test_accuracy}")
```

```python
            valid_reg_loss: float = get_cross_entropy_regularized_loss(y_vl_20,valid_probabilities,w,initial_regularization_strength)
            valid_unreg_loss: float = get_cross_entropy_unregularized_loss(y_vl_20,valid_probabilities)
            valid_accuracy: float = np.mean(np.argmax(valid_probabilities, axis=1)==y_vl_20)

            if valid_unreg_loss > 0 and valid_reg_loss > 0 and best_accuracy > 0.70:
              if valid_unreg_loss < best_loss and valid_accuracy < best_accuracy:
                  best_loss: float = valid_unreg_loss
                  best_accuracy: float = valid_accuracy
                  best_params = {
                      'batch_size': initial_mini_batch_size,
                      'learning_rate': initial_learning_rate,
                      'num_epochs': initial_no_of_epochs,
                      'regularization_strength': initial_regularization_strength,
                      'valid_reg_loss': valid_reg_loss,
                      'valid_unreg_loss': valid_unreg_loss,
                      'valid_accuracy': valid_accuracy,
                  }
              else:
                continue
            else:
              break

    # testing dataset
    test_probabilities: np.ndarray = get_softmax(get_y_hat_pred_func(X_te,w,b))
    test_reg_loss: float = get_cross_entropy_regularized_loss(y_te,test_probabilities,w,initial_regularization_strength)
    test_unreg_loss: float = get_cross_entropy_unregularized_loss(y_te,test_probabilities)
    test_accuracy: float = np.mean(np.argmax(test_probabilities, axis=1)==y_te)
    best_params['test_reg_loss'] = test_reg_loss
    best_params['test_unreg_loss'] = test_unreg_loss
    best_params['test_accuracy'] = test_accuracy
    print(f"Computing Loss on Test Data Set ->\nBest Test Reg Loss: {test_reg_loss},\nBest Test UnReg Loss: {test_unreg_loss},\nBest Accuracy: {test_accuracy}")
```

```
(60000, 784) (60000,) (10000, 784) (10000,)
80 / 20 Split :  48000
(48000, 784) (48000,) (10000, 784) (10000,) (12000, 784) (12000,)

Best Hyperparameters: {'batch_size': 160, 'learning_rate': 3e-05, 'num_epochs': 32, 'regularization_strength': 0.001, 'valid_reg_loss': 0.0005008797915, 'valid_unreg_loss': 0.0005008761, 'valid_accuracy': 0.8

Best Validation Reg Loss: 0.0005008797915,

Best Validation UnReg Loss: 0.0005008761,

Best Accuracy: 0.8065001898765323


Computing Loss on Test Data Set ->
Best Test Reg Loss: 0.0006779493,
Best Test UnReg Loss: 0.000677274824,
Best Accuracy: 0.80671435352
```