

```
In [21]: """
import things required
"""

import numpy as np
import matplotlib.pyplot as plt
import cv2
import os
```

```
In [22]: """
Helper functions to load data
"""

def load_mnist_dataset(dataset, path):
    """
    returns samples and lables specified by path and dataset params

    loop through each label and append image to X and label to y
    """
    labels = os.listdir(os.path.join(path, dataset))

    X = []
    y = []

    for label in labels:
        image_counter = 0
        for file in os.listdir(os.path.join(path, dataset, label)):
            image = cv2.imread(os.path.join(path, dataset, label, file), cv2.IMREAD_UNCHANGED)
            X.append(image)
            y.append(label)

    return np.array(X), np.array(y).astype('uint8')

def create_data_mnist(path):
    """
    returns train X, y and test X and y
    """
    X, y = load_mnist_dataset('train', path)
    X_test, y_test = load_mnist_dataset('test', path)

    return X, y, X_test, y_test
```

```
In [3]: """
get train and test data
scale data to be in range of -1 to 1 i.e. centered around 0
"""
X, y, X_test, y_test = create_data_mnist('fashion_mnist_images')

# Scale features
X = (X.astype(np.float32) - 127.5) / 127.5
X_test = (X_test.astype(np.float32) - 127.5) / 127.5

# print(X.min(), X.max())
# print(X.shape)
```

```
In [4]: print(X.shape)
print(y.shape)
```

```
(60000, 28, 28)
(60000,)
```

In [18]:

```
"""
pixel size - 28 x 28 - 784 features
images are reshaped to be next to each other
"""

# Reshape to vectors
X = X.reshape(X.shape[0], -1)
X_test = X_test.reshape(X_test.shape[0], -1)
X.shape[1]
```

Out[18]:

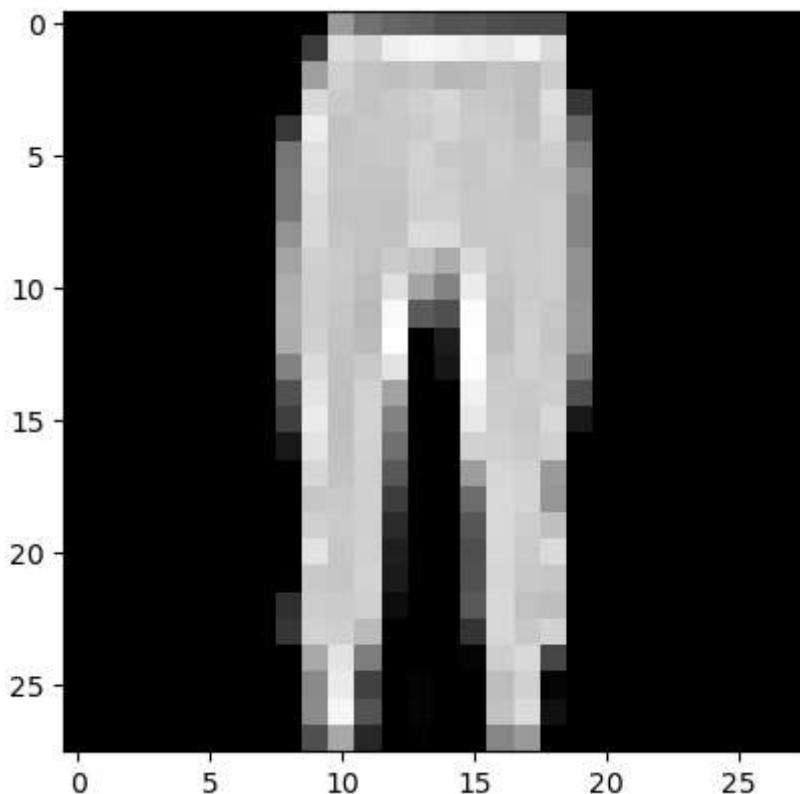
784

In [15]:

```
"""
shuffle the keys to remove biase that might be caused towards one label
"""

keys = np.array(range(X.shape[0]))
print(keys[:10])
np.random.shuffle(keys)
X = X[keys]
y = y[keys]
plt.imshow((X[4].reshape(28, 28)), cmap='gray')
plt.show()
```

[0 1 2 3 4 5 6 7 8 9]



In [7]:

```
class Layer_Dense:
    def __init__(self, n_inputs, n_neurons,
                 weight_regularizer_l1=0, weight_regularizer_l2=0,
                 bias_regularizer_l1=0, bias_regularizer_l2=0):
        """
        n_inputs represents the no of features
        n_neurons represents the no of neurons we want in this particular layer
    
```

```

weights are inputs x no of neurons - helps avoid transpose operation in dot pr
weights range between -1 to +1 so tht they are close to each other and NN does

biases is set to zero initially and if we encounter error where the entore out
NN is zero we can intialize it to some value to avoid dead ANN

rest four parameters refers to lambda that will be used for
L1 and L2 regularization
"""

self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
self.biases = np.zeros((1, n_neurons))

# Set regularization strength
self.weight_regularizer_l1 = weight_regularizer_l1
self.weight_regularizer_l2 = weight_regularizer_l2
self.bias_regularizer_l1 = bias_regularizer_l1
self.bias_regularizer_l2 = bias_regularizer_l2

def forward(self, inputs):
    self.inputs = inputs
    self.output = np.dot(inputs, self.weights) + self.biases

def backward(self, dvalues):
    # Gradients on parameters
    self.dweights = np.dot(self.inputs.T, dvalues)
    self.dbiases = np.sum(dvalues, axis=0, keepdims=True)

    # Gradients on regularization
    # L1 on weights
    if self.weight_regularizer_l1 > 0:
        dL1 = np.ones_like(self.weights)
        dL1[self.weights < 0] = -1
        self.dweights += self.weight_regularizer_l1 * dL1

    # L2 on weights
    if self.weight_regularizer_l2 > 0:
        self.dweights += 2 * self.weight_regularizer_l2 * \
            self.weights

    # L1 on biases
    if self.bias_regularizer_l1 > 0:
        dL1 = np.ones_like(self.biases)
        dL1[self.biases < 0] = -1
        self.dbiases += self.bias_regularizer_l1 * dL1

    # L2 on biases
    if self.bias_regularizer_l2 > 0:
        self.dbiases += 2 * self.bias_regularizer_l2 * \
            self.biases

    # Gradient on values
    self.dinputs = np.dot(dvalues, self.weights.T)

```

In [8]:

```

class Activation_ReLU:
    def forward(self, inputs):
        self.inputs = inputs
        self.output = np.maximum(0, inputs)

```

```

def backward(self, dvalues):
    self.dinputs = dvalues.copy()
    # input negative - gradient zero
    self.dinputs[self.inputs <= 0] = 0

class Activation_Softmax:
    def forward(self, inputs):
        self.inputs = inputs

        exp_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))
        probabilities = exp_values / np.sum(exp_values, axis=1, keepdims=True)
        self.output = probabilities

    def backward(self, dvalues):
        # Create uninitialized array
        self.dinputs = np.empty_like(dvalues)

        # Enumerate outputs and gradients
        for index, (single_output, single_dvalues) in enumerate(zip(self.output, dvalues)):
            # Flatten output array
            single_output = single_output.reshape(-1, 1)
            # Calculate Jacobian matrix of the output and
            jacobian_matrix = np.diagflat(single_output) - np.dot(single_output, single_output.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.dinputs[index] = np.dot(jacobian_matrix, single_dvalues)

```

In [9]:

```

class Loss:
    def calculate(self, output, y):
        sample_losses = self.forward(output, y)
        mean_loss = np.mean(sample_losses)
        return mean_loss

    def regularization_loss(self, layer):

        regularization_loss = 0

        # L1 regularization - weights
        # calculate only when factor greater than 0
        if layer.weight_regularizer_l1 > 0:
            regularization_loss += layer.weight_regularizer_l1 * np.sum(np.abs(layer.weights))

        # L2 regularization - weights
        if layer.weight_regularizer_l2 > 0:
            regularization_loss += layer.weight_regularizer_l2 * np.sum(layer.weights * layer.weights)

        # L1 regularization - biases
        # calculate only when factor greater than 0
        if layer.bias_regularizer_l1 > 0:
            regularization_loss += layer.bias_regularizer_l1 * np.sum(np.abs(layer.biases))

        # L2 regularization - biases
        if layer.bias_regularizer_l2 > 0:
            regularization_loss += layer.bias_regularizer_l2 * np.sum(layer.biases * layer.biases)

        return regularization_loss

class Loss_CategoricalCrossentropy(Loss):
    def forward(self, y_pred, y_true):

```

```

samples = len(y_pred)

# account for zero values -Log(0) = inf and then remove bias caused by its i
y_pred_clipped = np.clip(y_pred, 1e-7, 1-1e-7)

if len(y_true.shape) == 1:
    # y_true is in the form of [1, 0, 1, 1 ....]
    correct_confidences = y_pred_clipped[range(samples), y_true]

elif len(y_true.shape) == 2:
    # y_true is in the form of matrix [[0, 1, 0], [1, 0, 0], [0, 1, 0], [0, 1,
    correct_confidences = np.sum(y_pred_clipped * y_true, axis=1)

negative_log_likelihood = - np.log(correct_confidences)

return negative_log_likelihood

def backward(self, dvalues, y_true):

    samples = len(dvalues)
    labels = len(dvalues[0])
    # convert to hot vector if not
    if len(y_true.shape) == 1:
        y_true = np.eye(labels)[y_true]
    # Calculate gradient
    self.dinputs = -y_true / dvalues
    # Normalize gradient
    self.dinputs = self.dinputs / samples

class Activation_Softmax_Loss_CategoricalCrossentropy():

    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossentropy()

    def forward(self, inputs, y_true):
        # Output Layer's activation function
        self.activation.forward(inputs)
        # Set the output
        self.output = self.activation.output
        # Calculate and return Loss value
        return self.loss.calculate(self.output, y_true)

    def backward(self, dvalues, y_true):
        # Number of samples
        samples = len(dvalues)
        # If Labels are one-hot encoded,
        # turn them into discrete values
        if len(y_true.shape) == 2:
            y_true = np.argmax(y_true, axis=1)
        # Copy so we can safely modify
        self.dinputs = dvalues.copy()
        # Calculate gradient
        self.dinputs[range(samples), y_true] -= 1
        # Normalize gradient
        self.dinputs = self.dinputs / samples

```

In [10]: **class** Optimizer\_SGD:

```

"""
Vanilla option - SGD

```

```

Momentum option - if specified else default 0
"""

def __init__(self, learning_rate=1.0, decay=0, momentum=0.):
    self.learning_rate = learning_rate
    self.current_learning_rate = learning_rate
    self.decay = decay
    self.iterations = 0
    self.momentum = momentum

def pre_update_params(self):
    if self.decay:
        self.current_learning_rate = self.learning_rate * (1. / (1. + self.decay))

    # Update parameters
def update_params(self, layer):
    # If we use momentum
    if self.momentum:

        # create momentum array if not present along with biases
        if not hasattr(layer, 'weight_momentums'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            layer.bias_momentums = np.zeros_like(layer.biases)

        weight_updates = self.momentum * layer.weight_momentums - self.current_learning_rate * layer.dweights
        layer.weight_momentums = weight_updates

        bias_updates = self.momentum * layer.bias_momentums - self.current_learning_rate * layer.dbiases
        layer.bias_momentums = bias_updates

    # Vanilla SGD updates (as before momentum update)
    else:
        weight_updates = -self.current_learning_rate * layer.dweights
        bias_updates = -self.current_learning_rate * layer.dbiases

    # Update weights and biases using either vanilla or momentum update
    layer.weights += weight_updates
    layer.biases += bias_updates

def post_update_params(self):
    self.iterations += 1

```

In [11]:

```

"""
Features = 784
Samples = 60,000
class labels = 10 (0 - 9)

2 hidden layers - ReLU      - 128 neurons each
1 output layer - Softmax - 10 output neurons

Optimizer - SGD with momentum (0.5)
learning rate decay - 0.01
l2 regularization - lambda - 5e-4 (0.0005)
"""

dense1 = Layer_Dense(X.shape[1], 128, weight_regularizer_l2=5e-4, bias_regularizer_l2=5e-4)
activation1 = Activation_ReLU()
dense2 = Layer_Dense(128, 128)
activation2 = Activation_ReLU()
dense3 = Layer_Dense(128, 10)

```

```
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

optimizer = Optimizer_SGD(decay=0.01, momentum=0.5)

accuracies = []
losses = []
learning_rate = []
```

```
In [12]: for epoch in range(1100):
    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    activation2.forward(dense2.output)
    dense3.forward(activation2.output)
    data_loss = loss_activation.forward(dense3.output, y)

    regularization_loss = loss_activation.loss.regularization_loss(dense1) + loss_acti
        loss_activation.loss.regularization_loss(dense3)

    loss = data_loss + regularization_loss

    predictions = np.argmax(loss_activation.output, axis=1)
    if len(y.shape) == 2:
        y = np.argmax(y, axis=1)
    accuracy = np.mean(predictions==y)

    if not epoch % 10:
        accuracies.append(accuracy)
        losses.append(loss)
        learning_rate.append(optimizer.current_learning_rate)
        print(f'epoch: {epoch}, ' + f'acc: {accuracy:.3f}, ' + f'loss: {loss:.3f}, ' +
            + f'Rr: {regularization_loss}')

    loss_activation.backward(loss_activation.output, y)
    dense3.backward(loss_activation.dinputs)
    activation2.backward(dense3.dinputs)
    dense2.backward(activation2.dinputs)
    activation1.backward(dense2.dinputs)
    dense1.backward(activation1.dinputs)

    optimizer.pre_update_params()
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
    optimizer.update_params(dense3)
    optimizer.post_update_params()
```

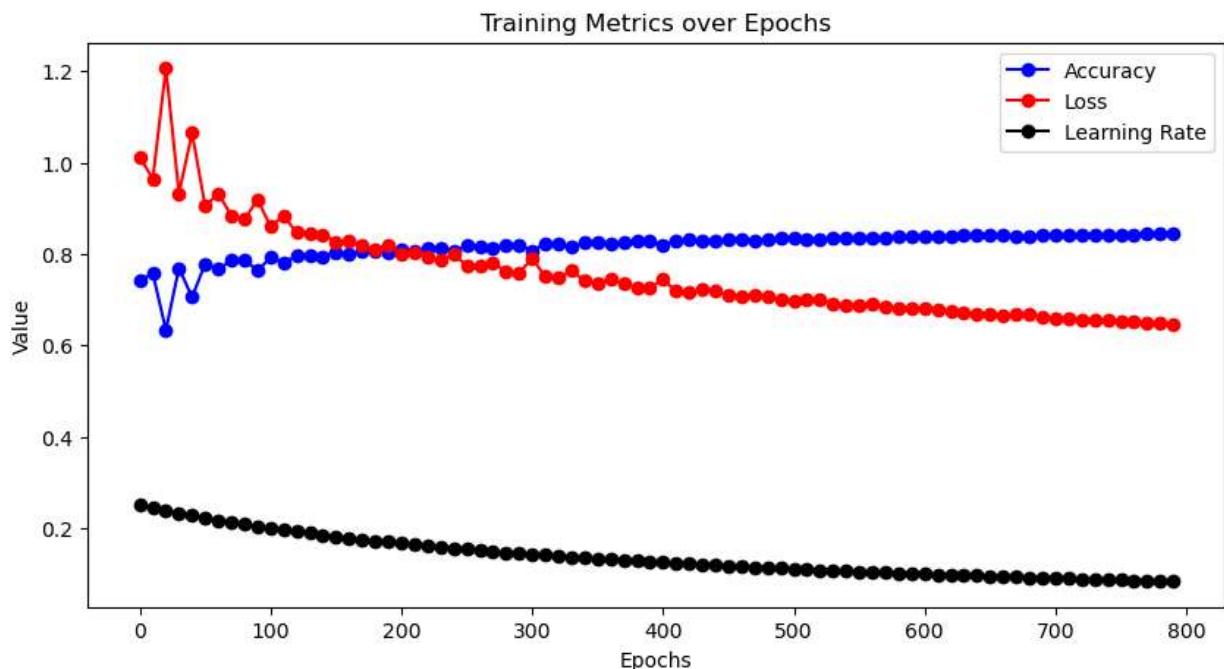
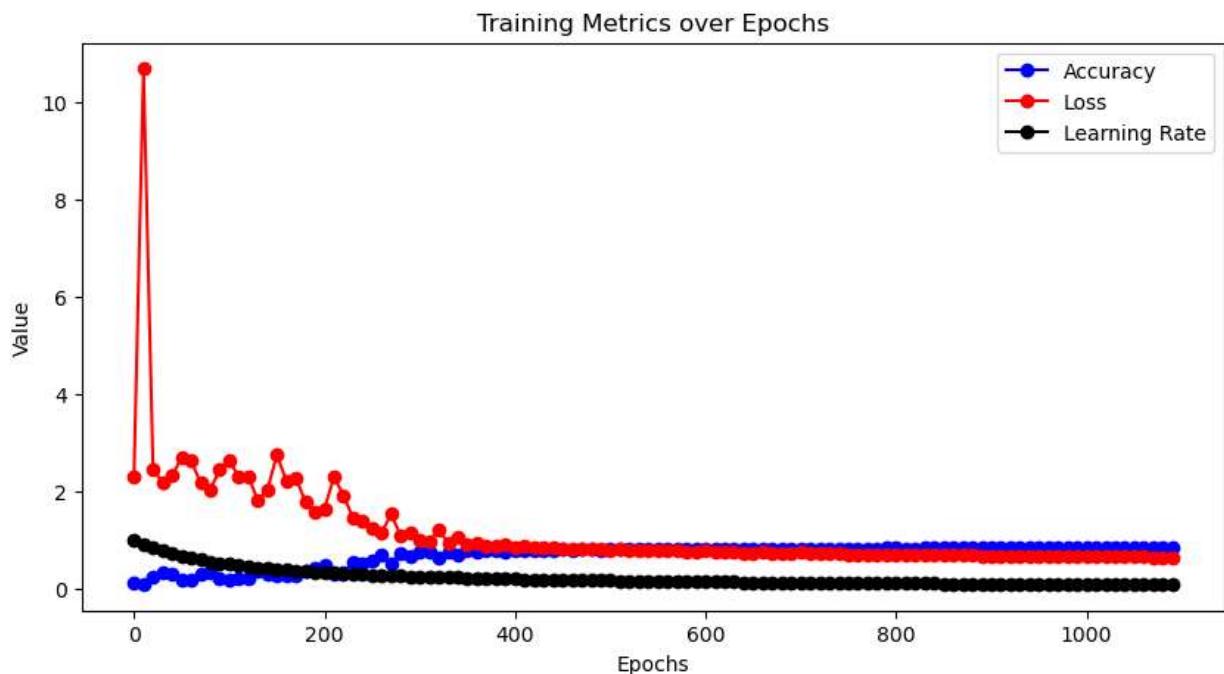
epoch: 0, acc: 0.112, loss: 2.308, lr: 1.0Rr: 0.00503451968865524  
epoch: 10, acc: 0.100, loss: 10.680, lr: 0.9174311926605504Rr: 0.00838227200825124  
epoch: 20, acc: 0.238, loss: 2.456, lr: 0.8403361344537815Rr: 0.4002204596130479  
epoch: 30, acc: 0.335, loss: 2.184, lr: 0.7751937984496123Rr: 0.406407055266603  
epoch: 40, acc: 0.313, loss: 2.323, lr: 0.7194244604316546Rr: 0.41817819724849087  
epoch: 50, acc: 0.185, loss: 2.690, lr: 0.6711409395973155Rr: 0.42146232069142076  
epoch: 60, acc: 0.169, loss: 2.628, lr: 0.628930817610063Rr: 0.4136935871644017  
epoch: 70, acc: 0.298, loss: 2.170, lr: 0.591715976331361Rr: 0.4059028981217895  
epoch: 80, acc: 0.341, loss: 2.039, lr: 0.5586592178770949Rr: 0.3975730039535512  
epoch: 90, acc: 0.203, loss: 2.439, lr: 0.5291005291005291Rr: 0.3899865106487447  
epoch: 100, acc: 0.174, loss: 2.643, lr: 0.5025125628140703Rr: 0.3868441508671544  
epoch: 110, acc: 0.216, loss: 2.289, lr: 0.47846889952153115Rr: 0.3797515158791232  
epoch: 120, acc: 0.213, loss: 2.312, lr: 0.4566210045662101Rr: 0.372698010463934  
epoch: 130, acc: 0.369, loss: 1.812, lr: 0.4366812227074236Rr: 0.3661404638099571  
epoch: 140, acc: 0.313, loss: 2.024, lr: 0.41841004184100417Rr: 0.36015653418287336  
epoch: 150, acc: 0.273, loss: 2.745, lr: 0.4016064257028112Rr: 0.354834598073053  
epoch: 160, acc: 0.268, loss: 2.216, lr: 0.3861003861003861Rr: 0.363729285424908  
epoch: 170, acc: 0.285, loss: 2.277, lr: 0.3717472118959108Rr: 0.3602163278317301  
epoch: 180, acc: 0.376, loss: 1.781, lr: 0.35842293906810035Rr: 0.3555415370514555  
epoch: 190, acc: 0.424, loss: 1.571, lr: 0.3460207612456747Rr: 0.350705861404834  
epoch: 200, acc: 0.487, loss: 1.648, lr: 0.33444816053511706Rr: 0.3457487538790493  
epoch: 210, acc: 0.293, loss: 2.296, lr: 0.3236245954692557Rr: 0.3413210042822297  
epoch: 220, acc: 0.321, loss: 1.915, lr: 0.31347962382445144Rr: 0.3377538457315041  
epoch: 230, acc: 0.559, loss: 1.449, lr: 0.303951367781155Rr: 0.33392780742332573  
epoch: 240, acc: 0.524, loss: 1.388, lr: 0.2949852507374631Rr: 0.33096544813019124  
epoch: 250, acc: 0.586, loss: 1.239, lr: 0.28653295128939826Rr: 0.3271496283947017  
epoch: 260, acc: 0.684, loss: 1.153, lr: 0.2785515320334262Rr: 0.3235672674391777  
epoch: 270, acc: 0.507, loss: 1.536, lr: 0.2710027100271003Rr: 0.3202406986157577  
epoch: 280, acc: 0.715, loss: 1.088, lr: 0.2638522427440633Rr: 0.3174255490015097  
epoch: 290, acc: 0.666, loss: 1.139, lr: 0.2570694087403599Rr: 0.3141771709173613  
epoch: 300, acc: 0.742, loss: 1.012, lr: 0.2506265664160401Rr: 0.3112368047446666  
epoch: 310, acc: 0.758, loss: 0.965, lr: 0.24449877750611249Rr: 0.30831208082879535  
epoch: 320, acc: 0.634, loss: 1.206, lr: 0.23866348448687355Rr: 0.30548624722338996  
epoch: 330, acc: 0.769, loss: 0.932, lr: 0.2331002331002331Rr: 0.30288957612238476  
epoch: 340, acc: 0.705, loss: 1.065, lr: 0.2277904328018223Rr: 0.3002569911403449  
epoch: 350, acc: 0.779, loss: 0.905, lr: 0.22271714922048996Rr: 0.2978217357830459  
epoch: 360, acc: 0.766, loss: 0.932, lr: 0.2178649237472767Rr: 0.2953227540733903  
epoch: 370, acc: 0.786, loss: 0.884, lr: 0.21321961620469085Rr: 0.29296502410304565  
epoch: 380, acc: 0.788, loss: 0.875, lr: 0.20876826722338204Rr: 0.29057677395294296  
epoch: 390, acc: 0.763, loss: 0.919, lr: 0.20449897750511245Rr: 0.28834437103459526  
epoch: 400, acc: 0.793, loss: 0.860, lr: 0.2004008016032064Rr: 0.28615070258906034  
epoch: 410, acc: 0.781, loss: 0.883, lr: 0.19646365422396858Rr: 0.2839843248900268  
epoch: 420, acc: 0.796, loss: 0.847, lr: 0.19267822736030826Rr: 0.28190963366499644  
epoch: 430, acc: 0.795, loss: 0.846, lr: 0.1890359168241966Rr: 0.27983635092609416  
epoch: 440, acc: 0.795, loss: 0.843, lr: 0.1855287569573284Rr: 0.2778848358064881  
epoch: 450, acc: 0.803, loss: 0.827, lr: 0.18214936247723132Rr: 0.27591801748527817  
epoch: 460, acc: 0.801, loss: 0.828, lr: 0.17889087656529518Rr: 0.273988602461581  
epoch: 470, acc: 0.805, loss: 0.819, lr: 0.17574692442882248Rr: 0.27220016713795486  
epoch: 480, acc: 0.808, loss: 0.811, lr: 0.17271157167530224Rr: 0.2703865720905306  
epoch: 490, acc: 0.802, loss: 0.820, lr: 0.16977928692699493Rr: 0.26861669483552764  
epoch: 500, acc: 0.810, loss: 0.800, lr: 0.1669449081803005Rr: 0.26690238550154544  
epoch: 510, acc: 0.808, loss: 0.803, lr: 0.16420361247947454Rr: 0.2652118463588427  
epoch: 520, acc: 0.811, loss: 0.793, lr: 0.16155088852988692Rr: 0.2635828164462515  
epoch: 530, acc: 0.814, loss: 0.786, lr: 0.1589825119236884Rr: 0.2619655119386929  
epoch: 540, acc: 0.806, loss: 0.799, lr: 0.1564945226917058Rr: 0.26040678962867503  
epoch: 550, acc: 0.818, loss: 0.774, lr: 0.15408320493066255Rr: 0.25886856412269804  
epoch: 560, acc: 0.816, loss: 0.774, lr: 0.15174506828528073Rr: 0.25735475951600567  
epoch: 570, acc: 0.812, loss: 0.781, lr: 0.14947683109118085Rr: 0.2559052643244058  
epoch: 580, acc: 0.820, loss: 0.761, lr: 0.14727540500736377Rr: 0.25445608755202737  
epoch: 590, acc: 0.820, loss: 0.759, lr: 0.14513788098693758Rr: 0.25303260182852055

epoch: 600, acc: 0.806, loss: 0.790, lr: 0.14306151645207438Rr: 0.2516672522069906  
 epoch: 610, acc: 0.823, loss: 0.750, lr: 0.14104372355430184Rr: 0.2503171806493208  
 epoch: 620, acc: 0.823, loss: 0.748, lr: 0.13908205841446453Rr: 0.2489739183251788  
 epoch: 630, acc: 0.815, loss: 0.764, lr: 0.13717421124828533Rr: 0.24766286804043333  
 epoch: 640, acc: 0.825, loss: 0.741, lr: 0.13531799729364005Rr: 0.2464009886874537  
 epoch: 650, acc: 0.826, loss: 0.737, lr: 0.13351134846461948Rr: 0.2451360995633058  
 epoch: 660, acc: 0.821, loss: 0.746, lr: 0.13175230566534915Rr: 0.24389374011743867  
 epoch: 670, acc: 0.825, loss: 0.735, lr: 0.13003901170351104Rr: 0.24270128365512475  
 epoch: 680, acc: 0.829, loss: 0.727, lr: 0.12836970474967907Rr: 0.24150960021136902  
 epoch: 690, acc: 0.827, loss: 0.727, lr: 0.12674271229404308Rr: 0.2403316435284347  
 epoch: 700, acc: 0.819, loss: 0.744, lr: 0.1251564455569462Rr: 0.23918524891273077  
 epoch: 710, acc: 0.830, loss: 0.719, lr: 0.12360939431396786Rr: 0.2380717623508691  
 epoch: 720, acc: 0.831, loss: 0.716, lr: 0.12210012210012208Rr: 0.23695576625227158  
 epoch: 730, acc: 0.827, loss: 0.722, lr: 0.12062726176115804Rr: 0.23585843094784978  
 epoch: 740, acc: 0.828, loss: 0.720, lr: 0.11918951132300357Rr: 0.23479644469833075  
 epoch: 750, acc: 0.832, loss: 0.708, lr: 0.11778563015312131Rr: 0.23374401009231477  
 epoch: 760, acc: 0.832, loss: 0.707, lr: 0.11641443538998836Rr: 0.23270052992162304  
 epoch: 770, acc: 0.830, loss: 0.711, lr: 0.1150747986191024Rr: 0.23167483016860638  
 epoch: 780, acc: 0.831, loss: 0.707, lr: 0.11376564277588169Rr: 0.2306769069114837  
 epoch: 790, acc: 0.834, loss: 0.700, lr: 0.11248593925759279Rr: 0.22968870911631437  
 epoch: 800, acc: 0.834, loss: 0.698, lr: 0.11123470522803114Rr: 0.2287106643692249  
 epoch: 810, acc: 0.832, loss: 0.699, lr: 0.11001100110011001Rr: 0.22774802183629586  
 epoch: 820, acc: 0.832, loss: 0.699, lr: 0.1088139281828074Rr: 0.22680670862269828  
 epoch: 830, acc: 0.835, loss: 0.691, lr: 0.1076426264800861Rr: 0.22587869442193545  
 epoch: 840, acc: 0.836, loss: 0.689, lr: 0.10649627263045792Rr: 0.22495956771052214  
 epoch: 850, acc: 0.836, loss: 0.689, lr: 0.1053740779768177Rr: 0.22405349037474043  
 epoch: 860, acc: 0.835, loss: 0.690, lr: 0.10427528675703858Rr: 0.2231637868140928  
 epoch: 870, acc: 0.836, loss: 0.684, lr: 0.10319917440660475Rr: 0.22228897307046677  
 epoch: 880, acc: 0.837, loss: 0.681, lr: 0.10214504596527067Rr: 0.22142303956061293  
 epoch: 890, acc: 0.837, loss: 0.680, lr: 0.10111223458038422Rr: 0.22056818243880688  
 epoch: 900, acc: 0.837, loss: 0.680, lr: 0.10010010010010009Rr: 0.21972637330096356  
 epoch: 910, acc: 0.837, loss: 0.677, lr: 0.09910802775024777Rr: 0.21889786268828346  
 epoch: 920, acc: 0.839, loss: 0.674, lr: 0.09813542688910697Rr: 0.21808011934389276  
 epoch: 930, acc: 0.840, loss: 0.671, lr: 0.09718172983479105Rr: 0.2172717483608713  
 epoch: 940, acc: 0.841, loss: 0.669, lr: 0.09624639076034648Rr: 0.2164731677002239  
 epoch: 950, acc: 0.840, loss: 0.668, lr: 0.09532888465204957Rr: 0.21568463411760205  
 epoch: 960, acc: 0.840, loss: 0.666, lr: 0.09442870632672333Rr: 0.2149063782384027  
 epoch: 970, acc: 0.839, loss: 0.668, lr: 0.09354536950420955Rr: 0.21413904964009467  
 epoch: 980, acc: 0.838, loss: 0.669, lr: 0.09267840593141798Rr: 0.21338735583826487  
 epoch: 990, acc: 0.841, loss: 0.662, lr: 0.09182736455463728Rr: 0.21264512590920423  
 epoch: 1000, acc: 0.842, loss: 0.660, lr: 0.09099181073703366Rr: 0.21190822497789233  
 epoch: 1010, acc: 0.842, loss: 0.658, lr: 0.09017132551848513Rr: 0.21117964117771595  
 epoch: 1020, acc: 0.842, loss: 0.657, lr: 0.08936550491510277Rr: 0.21046016692546368  
 epoch: 1030, acc: 0.842, loss: 0.656, lr: 0.08857395925597873Rr: 0.20974990373548155  
 epoch: 1040, acc: 0.841, loss: 0.655, lr: 0.08779631255487269Rr: 0.20904912304126  
 epoch: 1050, acc: 0.842, loss: 0.653, lr: 0.08703220191470844Rr: 0.20835730436982514  
 epoch: 1060, acc: 0.843, loss: 0.651, lr: 0.08628127696289906Rr: 0.20767388518449204  
 epoch: 1070, acc: 0.843, loss: 0.650, lr: 0.0855431993156544Rr: 0.20699831315887868  
 epoch: 1080, acc: 0.844, loss: 0.648, lr: 0.08481764206955046Rr: 0.2063304252037452  
 epoch: 1090, acc: 0.844, loss: 0.646, lr: 0.08410428931875526Rr: 0.20567010121676432

```
In [20]: epochs = range(0, 1100, 10)
plt.figure(figsize=(10, 5))
plt.plot(epochs, accuracies, label='Accuracy', marker='o', color="blue")
plt.plot(epochs, losses, label='Loss', marker='o', color='red')
plt.plot(epochs, learning_rate, label='Learning Rate', marker='o', color='black')
plt.title('Training Metrics over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Value')
plt.legend()
```

```
plt.savefig('training_metrics_plot.png')
plt.show()

epochs = range(0, 800, 10)
plt.figure(figsize=(10, 5))
plt.plot(epochs, accuracies[30:], label='Accuracy', marker='o', color="blue")
plt.plot(epochs, losses[30:], label='Loss', marker='o', color='red')
plt.plot(epochs, learning_rate[30:], label='Learning Rate', marker='o', color='black')
plt.title('Training Metrics over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Value')
plt.legend()
plt.savefig('training_metrics_plot.png')
plt.show()
```



In [14]: # Validate the model

```
dense1.forward(X_test)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
activation2.forward(dense2.output)
dense3.forward(activation2.output)
data_loss = loss_activation.forward(dense3.output, y_test)

predictions = np.argmax(loss_activation.output, axis=1)
if len(y_test.shape) == 2:
    y_test = np.argmax(y, axis=1)
accuracy = np.mean(predictions==y_test)
print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')

validation, acc: 0.825, loss: 0.645
```

In [ ]: