

```
In [52]: import numpy as np
import nnfs
from nnfs.datasets import spiral_data
import matplotlib.pyplot as plt
```

```
In [53]: import numpy as np
import cv2
import os

# Loads a MNIST dataset
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 [54]: 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)
```

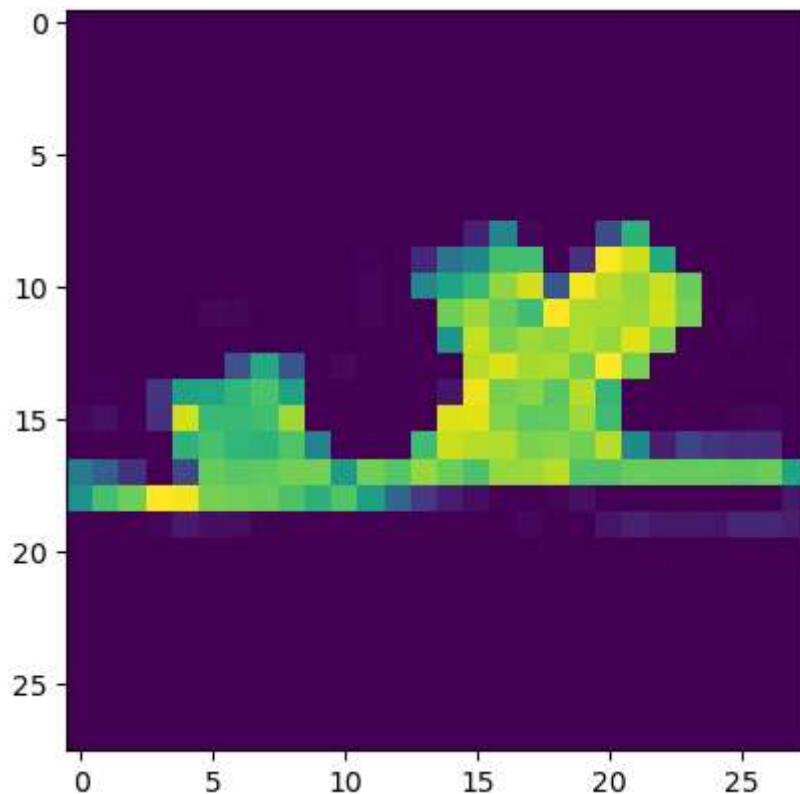
```
In [55]: print(X.shape)
print(y.shape)

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

```
In [56]: # Reshape to vectors
X = X.reshape(X.shape[0], -1)
X_test = X_test.reshape(X_test.shape[0], -1)
```

```
In [57]: 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)))
plt.show()
```

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



```
In [58]: 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 [59]:

```

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

    def backward(self, dvalues):
        # Since we need to modify original variable,
        # let's make a copy of values first
        self.dinputs = dvalues.copy()
        # Zero gradient where input values were negative
        self.dinputs[self.inputs <= 0] = 0

class Activation_Softmax:
    def forward(self, inputs):
        # Remember input values
        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

    # Backward pass

```

```

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 [60]:

```

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 infinity
        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, 0]]
            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):
    # Number of samples
    samples = len(dvalues)
    # Number of Labels in every sample
    # We'll use the first sample to count them
    labels = len(dvalues[0])
    # If Labels are sparse, turn them into one-hot vector
    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():
    # Creates activation and Loss function objects
    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 [61]: 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

    # Call once before any parameter updates

```

```

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:

        # If layer does not contain momentum arrays, create them
        # filled with zeros
        if not hasattr(layer, 'weight_momentums'):
            layer.weight_momentums = np.zeros_like(layer.weights)
            # If there is no momentum array for weights
            # The array doesn't exist for biases yet either.
            layer.bias_momentums = np.zeros_like(layer.biases)

        # Build weight updates with momentum - take previous
        # updates multiplied by retain factor and update with
        # current gradients
        weight_updates = self.momentum * layer.weight_momentums - self.current_learning_rate * layer.dweights
        layer.weight_momentums = weight_updates

        # Build bias 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 updates
    layer.weights += weight_updates
    layer.biases += bias_updates

# Call once after any parameter updates
def post_update_params(self):
    self.iterations += 1

```

In [73]:

```

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 [91]:

```

for epoch in range(500):
    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_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}')

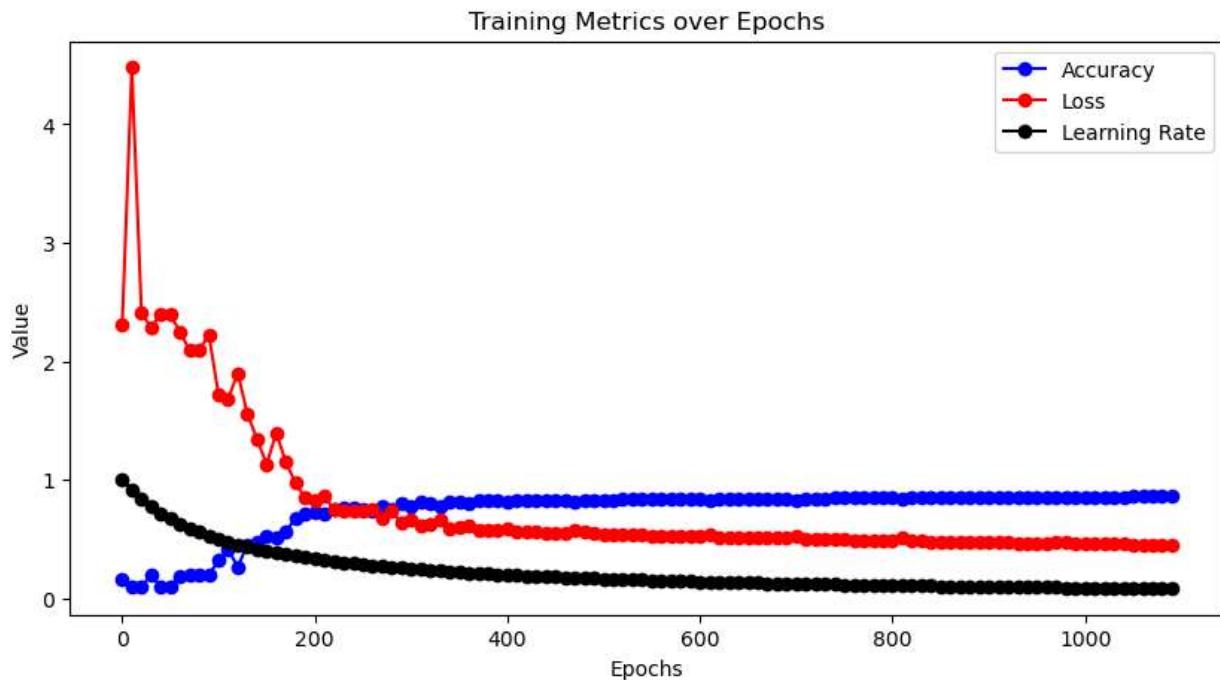
loss_activation.backward(loss_activation.output, y)
dense3.backward(loss_activation.dinputs)
activation2.backward(dense3.dinputs)
dense2.backward(activation2.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)

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

epoch: 0, acc: 0.839, loss: 0.523, lr: 0.14306151645207438Rr: 0.0728328941363401
 epoch: 10, acc: 0.832, loss: 0.539, lr: 0.14104372355430184Rr: 0.07249486494390431
 epoch: 20, acc: 0.840, loss: 0.519, lr: 0.13908205841446453Rr: 0.07220264282913928
 epoch: 30, acc: 0.841, loss: 0.517, lr: 0.13717421124828533Rr: 0.07187326258439371
 epoch: 40, acc: 0.842, loss: 0.515, lr: 0.13531799729364005Rr: 0.0715458592218554
 epoch: 50, acc: 0.842, loss: 0.513, lr: 0.13351134846461948Rr: 0.07122387632568698
 epoch: 60, acc: 0.843, loss: 0.512, lr: 0.13175230566534915Rr: 0.07090775334519181
 epoch: 70, acc: 0.843, loss: 0.510, lr: 0.13003901170351104Rr: 0.07059741898960634
 epoch: 80, acc: 0.844, loss: 0.508, lr: 0.12836970474967907Rr: 0.07029324495361015
 epoch: 90, acc: 0.843, loss: 0.508, lr: 0.12674271229404308Rr: 0.06999605535443824
 epoch: 100, acc: 0.832, loss: 0.531, lr: 0.1251564455569462Rr: 0.06972342506256138
 epoch: 110, acc: 0.845, loss: 0.504, lr: 0.12360939431396786Rr: 0.0694688196460336
 epoch: 120, acc: 0.846, loss: 0.502, lr: 0.12210012210012208Rr: 0.06918904041393766
 epoch: 130, acc: 0.846, loss: 0.500, lr: 0.12062726176115804Rr: 0.06891211244341199
 epoch: 140, acc: 0.847, loss: 0.498, lr: 0.11918951132300357Rr: 0.06863923809994117
 epoch: 150, acc: 0.848, loss: 0.496, lr: 0.11778563015312131Rr: 0.06837057418432571
 epoch: 160, acc: 0.848, loss: 0.495, lr: 0.11641443538998836Rr: 0.06810630467110856
 epoch: 170, acc: 0.849, loss: 0.493, lr: 0.1150747986191024Rr: 0.06784662732417243
 epoch: 180, acc: 0.849, loss: 0.491, lr: 0.11376564277588169Rr: 0.0675913599441077
 epoch: 190, acc: 0.850, loss: 0.490, lr: 0.11248593925759279Rr: 0.06734068184599792
 epoch: 200, acc: 0.849, loss: 0.492, lr: 0.11123470522803114Rr: 0.06709623900376309
 epoch: 210, acc: 0.839, loss: 0.514, lr: 0.11001100110011001Rr: 0.06687653755630929
 epoch: 220, acc: 0.851, loss: 0.485, lr: 0.1088139281828074Rr: 0.06666106815561916
 epoch: 230, acc: 0.851, loss: 0.484, lr: 0.1076426264800861Rr: 0.06642801796171957
 epoch: 240, acc: 0.852, loss: 0.482, lr: 0.10649627263045792Rr: 0.06619845840528867
 epoch: 250, acc: 0.853, loss: 0.480, lr: 0.1053740779768177Rr: 0.0659723244968744
 epoch: 260, acc: 0.853, loss: 0.479, lr: 0.10427528675703858Rr: 0.06574952083087482
 epoch: 270, acc: 0.854, loss: 0.477, lr: 0.10319917440660475Rr: 0.06552977676733339
 epoch: 280, acc: 0.854, loss: 0.476, lr: 0.10214504596527067Rr: 0.06531338502787754
 epoch: 290, acc: 0.854, loss: 0.476, lr: 0.10111223458038422Rr: 0.06510087256352358
 epoch: 300, acc: 0.853, loss: 0.478, lr: 0.10010010010010009Rr: 0.06489425033260518
 epoch: 310, acc: 0.851, loss: 0.481, lr: 0.09910802775024777Rr: 0.0646995218724174
 epoch: 320, acc: 0.855, loss: 0.472, lr: 0.09813542688910697Rr: 0.06450387769457161
 epoch: 330, acc: 0.857, loss: 0.469, lr: 0.09718172983479105Rr: 0.06430287487283957
 epoch: 340, acc: 0.857, loss: 0.468, lr: 0.09624639076034648Rr: 0.06410375340141714
 epoch: 350, acc: 0.857, loss: 0.467, lr: 0.09532888465204957Rr: 0.06390718814754749
 epoch: 360, acc: 0.857, loss: 0.466, lr: 0.09442870632672333Rr: 0.06371353793745388
 epoch: 370, acc: 0.856, loss: 0.470, lr: 0.09354536950420955Rr: 0.06352511081027587
 epoch: 380, acc: 0.855, loss: 0.472, lr: 0.09267840593141798Rr: 0.06334816062847623
 epoch: 390, acc: 0.858, loss: 0.463, lr: 0.09182736455463728Rr: 0.06316669502805544
 epoch: 400, acc: 0.859, loss: 0.461, lr: 0.09099181073703366Rr: 0.06298134987317147
 epoch: 410, acc: 0.859, loss: 0.460, lr: 0.09017132551848513Rr: 0.06279830383425015
 epoch: 420, acc: 0.859, loss: 0.460, lr: 0.08936550491510277Rr: 0.06261805855716635
 epoch: 430, acc: 0.858, loss: 0.463, lr: 0.08857395925597873Rr: 0.0624426786301628
 epoch: 440, acc: 0.859, loss: 0.461, lr: 0.08779631255487269Rr: 0.06227186759097666
 epoch: 450, acc: 0.860, loss: 0.457, lr: 0.08703220191470844Rr: 0.062099684438484425
 epoch: 460, acc: 0.860, loss: 0.456, lr: 0.08628127696289906Rr: 0.06192744116744557
 epoch: 470, acc: 0.860, loss: 0.456, lr: 0.0855431993156544Rr: 0.06175787132019155
 epoch: 480, acc: 0.860, loss: 0.456, lr: 0.08481764206955046Rr: 0.06159192508070434
 epoch: 490, acc: 0.861, loss: 0.454, lr: 0.08410428931875526Rr: 0.061427073152394214

```
In [97]: 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()
```



In [98]: # 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: 0.843, loss: 0.453
```

In []: