notes_18

January 19, 2025

1 Previous Class Definitions

[20]:
```
# imports
import matplotlib.pyplot as plt
import numpy as np
import nnfs
from nnfs.datasets import spiral_data, vertical_data
nnfs.init()
```

[21]:
```
class Layer_Dense:
    def __init__(self, n_inputs, n_neurons):
        # Initialize the weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)  # Normal␣
 ↪distribution of weights
        self.biases = np.zeros((1, n_neurons))

    def forward(self, inputs):
        # Calculate the output values from inputs, weights, and biases
        self.output = np.dot(inputs, self.weights) + self.biases        #␣
 ↪Weights are already transposed

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

class Activation_Softmax:
    def forward(self, inputs):
        # Get the unnormalized probabilities
        # Subtract max from the row to prevent larger numbers
        exp_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))

        # Normalize the probabilities with element wise division
        probabilities = exp_values / np.sum(exp_values, axis=1,keepdims=True)
        self.output = probabilities

# Base class for Loss functions
class Loss:
```

```
    '''Calculates the data and regularization losses given
    model output and ground truth values'''
    def calculate(self, output, y):
        sample_losses = self.forward(output, y)
        data_loss = np.average(sample_losses)
        return data_loss

class Loss_CategoricalCrossEntropy(Loss):
    def forward(self, y_pred, y_true):
        '''y_pred is the neural network output
        y_true is the ideal output of the neural network'''
        samples = len(y_pred)
        # Bound the predicted values
        y_pred_clipped = np.clip(y_pred, 1e-7, 1-1e-7)

        if len(y_true.shape) == 1:      # Categorically labeled
            correct_confidences = y_pred_clipped[range(samples), y_true]
        elif len(y_true.shape) == 2:    # One hot encoded
            correct_confidences = np.sum(y_pred_clipped*y_true, axis=1)

        # Calculate the losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods
```

2 Previous Notes and Notation

The previous notation is clunky and long. From here forward, we will use the following notation for a layer with n inputs and i neurons. The neruon layer has is followed by an activation layer and then fed into a final value y with a computed loss l. There can be j batches of data.

$\vec{X}_j = \begin{bmatrix} x_{1j} & x_{2j} & \cdots & x_{nj} \end{bmatrix}$ -> Row vector for the layer inputs for the j batch of data.

$\overline{\overline{W}} = \begin{bmatrix} \vec{w}_1 \\ \vec{w}_2 \\ \vdots \\ \vec{w}_i \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \cdots & w_{in} \end{bmatrix}$ -> Matrix of weight values. Each row is a neuron's weights and each column is the weights for a given input.

$\vec{B} = \begin{bmatrix} b_1 & b_2 & \cdots & b_i \end{bmatrix}$ -> Row vector for the neuron biases

$\vec{Z}_j = \begin{bmatrix} z_{1j} & z_{2j} & \cdots & z_{ij} \end{bmatrix}$ -> Row vector for the neuron outputs for the j batch of data.

$\vec{A}_j = \begin{bmatrix} a_{1j} & a_{2j} & \cdots & a_{ij} \end{bmatrix}$ -> Row vector for the activation later outputs for the j batch of data.

y_j -> Final layer output for the j batch of data if the layer is the final layer (could be summation, probability, etc).

l_j -> Loss for the j batch of data.

The j is often dropped because we typically only need to think with 1 set of input data.

2.1 Gradient Descent Using New Notation

We will look at the weight that the i neuron applies for the n input.

$\frac{\delta l}{\delta w_{in}} = \frac{\delta l}{\delta y}\frac{\delta y}{\delta a_i}\frac{\delta a_i}{\delta z_i}\frac{\delta z_i}{\delta w_{in}}$

Similarly, for the bias of the i neuron, there is

$\frac{\delta l}{\delta b_i} = \frac{\delta l}{\delta y}\frac{\delta y}{\delta a_i}\frac{\delta a_i}{\delta z_i}\frac{\delta z_i}{\delta b_i}$

For the system we are using, where $l = (y-0)^2$ and the activation layer is ReLU, we have

$\frac{\delta l}{\delta y} = 2y$

$\frac{\delta y}{\delta a_i} = 1$

$\frac{\delta a_i}{\delta z_i} = 1$ if $z_i > 0$ else 0

$\frac{\delta z_i}{\delta w_{in}} = x_n$

$\frac{\delta z_i}{\delta b_i} = 1$

2.2 Matrix Representation of Gradient Descent

We can simplify by seeing that $\frac{\delta l}{\delta y}\frac{\delta y}{\delta a_i}\frac{\delta a_i}{\delta z_i} = \frac{\delta l}{\delta z_i}$ is a common term.

We take $\frac{\delta l}{\delta z_i}$ and turn it into a 1 x i vector that such that

$\frac{\delta l}{\delta \vec{Z}} = \begin{bmatrix}\frac{\delta l}{\delta z_1} & \frac{\delta l}{\delta z_2} & \cdots & \frac{\delta l}{\delta z_i}\end{bmatrix}$

We than can get that the gradient matrix for all weights is a i x n matrix given by

$$\frac{\delta l}{\delta \vec{\vec{W}}} = \begin{bmatrix}\frac{\delta l}{\delta w_{11}} & \frac{\delta l}{\delta w_{12}} & \cdots & \frac{\delta l}{\delta w_{1n}} \\ \frac{\delta l}{\delta w_{21}} & w\frac{\delta l}{\delta w_{22}} & \cdots & \frac{\delta l}{\delta w_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta l}{\delta w_{i1}} & \frac{\delta l}{\delta w_{i2}} & \cdots & \frac{\delta l}{\delta w_{in}}\end{bmatrix} = \begin{bmatrix}\frac{\delta l}{\delta z_1} \\ \frac{\delta l}{\delta z_2} \\ \vdots \\ \frac{\delta l}{\delta z_n}\end{bmatrix}\begin{bmatrix}\frac{\delta z_1}{\delta w_{i1}} & \frac{\delta z_1}{\delta w_{i1}} & \cdots & \frac{\delta z_1}{\delta w_{in}}\end{bmatrix} = \begin{bmatrix}\frac{\delta l}{\delta z_1} \\ \frac{\delta l}{\delta z_2} \\ \vdots \\ \frac{\delta l}{\delta z_n}\end{bmatrix}\begin{bmatrix}x_1 & x_2 & \cdots & x_n\end{bmatrix}$$

Similarly, the gradient vector for the biases is given by $\frac{\delta l}{\delta \vec{B}} = \frac{\delta l}{\delta \vec{Z}}\frac{\delta \vec{Z}}{\delta \vec{B}} = \vec{1}\begin{bmatrix}\frac{\delta l}{\delta z_1} & \frac{\delta l}{\delta z_2} & \cdots & \frac{\delta l}{\delta z_i}\end{bmatrix}$

2.3 Gradients of the Loss with Respect to Inputs

When chaining multiple layers together, we will need the partial derivatives of the loss with respect to the next layers input (ie, the output of the current layer). This involves extra summation because the output of 1 layer is fed into every neuron of the next layer, so the total loss must be found.

The gradient of the loss with respect to the n input fed into i neurons is

$\frac{\delta l}{\delta x_n} = \frac{\delta l}{\delta z_1}\frac{\delta z_1}{\delta x_n} + \frac{\delta l}{\delta z_2}\frac{\delta z_2}{\delta x_n} + ... + \frac{\delta l}{\delta z_i}\frac{\delta z_i}{\delta x_n}$

Noting that $\frac{\delta z_i}{\delta x_n} = w_{in}$ allows us to have

$$\frac{\delta l}{\delta \vec{X}} = \begin{bmatrix}\frac{\delta l}{\delta x_1} & \frac{\delta l}{\delta x_2} & \cdots & \frac{\delta l}{\delta x_n}\end{bmatrix} = \begin{bmatrix}\frac{\delta l}{\delta z_1} & \frac{\delta l}{\delta z_2} & \cdots & \frac{\delta l}{\delta z_n}\end{bmatrix}\begin{bmatrix}w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \cdots & w_{in}\end{bmatrix}$$

2.4 Note With Layer_Dense class

The Layer_Dense class has the weights stored in the transposed fashion for forward propagation. Therefore, the weight matrix must be transposed for the backpropagation.

3 Adding the Backward Propagation Method to Layer_Dense and ReLU Activation Classes

[28]:
```
class Layer_Dense:
    def __init__(self, n_inputs, n_neurons):
        # Initialize the weights and biases
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)  # Normal␣
 ↪distribution of weights
        self.biases = np.zeros((1, n_neurons))

    def forward(self, inputs):
        # Calculate the output values from inputs, weights, and biases
        self.inputs = inputs
        self.output = np.dot(inputs, self.weights) + self.biases        #␣
 ↪Weights are already transposed

    def backward(self, dvalues):
        '''Calculated the gradient of the loss with respect to the weights and␣
 ↪biases of this layer.
        dvalues is equiavelent to a transposed dl_dZ. It is the gradient
        of the loss with respect to the outputs of this layer.'''
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=0)
        self.dinputs = np.dot(dvalues, self.weights.T)

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

    def backward(self, dvalues):
        '''Calculated the gradient of the loss with respect to this layer's␣
 ↪activation function
        dvalues is equiavelent to a transposed dl_dZ. It is the gradient
        of the loss with respect to the outputs of this layer.'''
        self.dinputs = dvalues.copy()
        self.dinputs[self.inputs <= 0] = 0
```

4 Adding the Backward Propagation Method to the Loss_CategoricalCrossEntropy Class

[23]:
```
class Loss_CategoricalCrossEntropy(Loss):
    def forward(self, y_pred, y_true):
        '''y_pred is the neural network output
        y_true is the ideal output of the neural network'''
        samples = len(y_pred)
        # Bound the predicted values
        y_pred_clipped = np.clip(y_pred, 1e-7, 1-1e-7)

        if len(y_true.shape) == 1:      # Categorically labeled
            correct_confidences = y_pred_clipped[range(samples), y_true]
        elif len(y_true.shape) == 2:    # One hot encoded
            correct_confidences = np.sum(y_pred_clipped*y_true, axis=1)

        # Calculate the losses
        negative_log_likelihoods = -np.log(correct_confidences)
        return negative_log_likelihoods

    def backward(self, dvalues, y_true):
        samples = len(dvalues)

        # Number of lables in each sample
        labels = len(dvalues[0])

        # if the labels are sparse, turn them into a one-hot vector
        if len(y_true.shape) == 1:
            y_true = np.eye(labels)[y_true]

        # Calculate the gradient then normalize
        self.dinputs = -y_true / dvalues
        self.dinputs = self.dinputs / samples
```

5 Combined Softmax Activation and Cross Entropy Loss

[24]:
```
class Activation_Softmax_Loss_CategoricalCrossentropy():
    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss = Loss_CategoricalCrossEntropy()

    def forward(self, inputs, y_true):
        self.activation.forward(inputs)
        self.output = self.activation.output
        return self.loss.calculate(self.output, y_true)
```

```
    def backward(self, dvalues, y_true):
        samples = len(dvalues)

        # if the samples 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 and normalize gradient
        self.dinputs[range(samples), y_true] -= 1
        self.dinputs = self.dinputs / samples
```

[25]:
```
softmax_outputs = np.array([[0.7, 0.1, 0.2],
 [0.1, 0.5, 0.4],
 [0.02, 0.9, 0.08]])
class_targets = np.array([0, 1, 1])
softmax_loss = Activation_Softmax_Loss_CategoricalCrossentropy()
softmax_loss.backward(softmax_outputs, class_targets)
dvalues1 = softmax_loss.dinputs
print('Gradients: combined loss and activation:')
print(dvalues1)
```

```
Gradients: combined loss and activation:
[[-0.1         0.03333333  0.06666667]
 [ 0.03333333 -0.16666667  0.13333333]
 [ 0.00666667 -0.03333333  0.02666667]]
```

6 Optimizer_SGD Class

[26]:
```
class Optimizer_SGD():
    def __init__(self, learning_rate=0.5):
        self.learning_rate = learning_rate

    def update_params(self, layer):
        layer.weights += -self.learning_rate * layer.dweights
        layer.biases += -self.learning_rate * layer.dbiases
```

6.1 Optimizer_SGD Class on Spiral Dataset

[31]:
```
# Create dataset
X, y = spiral_data(samples=100, classes=3)

# Create Dense layer with 2 input features and 64 output values
dense1 = Layer_Dense(2, 64)
```

```
# Create ReLU activation (to be used with Dense layer)
activation1 = Activation_ReLU()

# Create second Dense layer with 64 input features (as we take output
# of previous layer here) and 3 output values (output values)
dense2 = Layer_Dense(64, 3)

# Create Softmax classifier's combined loss and activation
loss_activation = Activation_Softmax_Loss_CategoricalCrossentropy()

# Create optimizer
optimizer = Optimizer_SGD()

# Train in loop
for epoch in range(10001):
    # Perform a forward pass of our training data through this layer
    dense1.forward(X)

    # Perform a forward pass through activation function
    # takes the output of first dense layer here
    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer
    # takes outputs of activation function of first layer as inputs
    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function
    # takes the output of second dense layer here and returns loss
    loss = loss_activation.forward(dense2.output, y)

    # Calculate accuracy from output of activation2 and targets
    # calculate values along first axis
    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 % 100:
        print(f'epoch: {epoch}, ' +
              f'acc: {accuracy:.3f}, ' +
              f'loss: {loss:.3f}')

    # Backward pass
    loss_activation.backward(loss_activation.output, y)
    dense2.backward(loss_activation.dinputs)
    activation1.backward(dense2.dinputs)
```

```
    dense1.backward(activation1.dinputs)

    # Update weights and biases
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
```

```
epoch: 0, acc: 0.350, loss: 1.099
epoch: 100, acc: 0.457, loss: 1.090
epoch: 200, acc: 0.460, loss: 1.058
epoch: 300, acc: 0.460, loss: 1.055
epoch: 400, acc: 0.463, loss: 1.053
epoch: 500, acc: 0.463, loss: 1.052
epoch: 600, acc: 0.463, loss: 1.052
epoch: 700, acc: 0.460, loss: 1.052
epoch: 800, acc: 0.463, loss: 1.051
epoch: 900, acc: 0.450, loss: 1.051
epoch: 1000, acc: 0.447, loss: 1.051
epoch: 1100, acc: 0.450, loss: 1.050
epoch: 1200, acc: 0.453, loss: 1.049
epoch: 1300, acc: 0.453, loss: 1.048
epoch: 1400, acc: 0.453, loss: 1.046
epoch: 1500, acc: 0.457, loss: 1.044
epoch: 1600, acc: 0.457, loss: 1.040
epoch: 1700, acc: 0.463, loss: 1.036
epoch: 1800, acc: 0.457, loss: 1.030
epoch: 1900, acc: 0.467, loss: 1.025
epoch: 2000, acc: 0.477, loss: 1.019
epoch: 2100, acc: 0.500, loss: 1.013
epoch: 2200, acc: 0.513, loss: 1.007
epoch: 2300, acc: 0.523, loss: 0.999
epoch: 2400, acc: 0.533, loss: 0.990
epoch: 2500, acc: 0.540, loss: 0.981
epoch: 2600, acc: 0.537, loss: 0.973
epoch: 2700, acc: 0.543, loss: 0.964
epoch: 2800, acc: 0.537, loss: 0.957
epoch: 2900, acc: 0.547, loss: 0.947
epoch: 3000, acc: 0.553, loss: 0.938
epoch: 3100, acc: 0.507, loss: 0.937
epoch: 3200, acc: 0.523, loss: 0.950
epoch: 3300, acc: 0.523, loss: 0.931
epoch: 3400, acc: 0.537, loss: 0.950
epoch: 3500, acc: 0.480, loss: 0.929
epoch: 3600, acc: 0.490, loss: 0.925
epoch: 3700, acc: 0.540, loss: 0.932
epoch: 3800, acc: 0.530, loss: 0.916
epoch: 3900, acc: 0.507, loss: 0.932
epoch: 4000, acc: 0.487, loss: 0.939
```

```
epoch: 4100, acc: 0.590, loss: 0.950
epoch: 4200, acc: 0.530, loss: 0.932
epoch: 4300, acc: 0.523, loss: 0.921
epoch: 4400, acc: 0.450, loss: 0.907
epoch: 4500, acc: 0.487, loss: 0.932
epoch: 4600, acc: 0.507, loss: 0.906
epoch: 4700, acc: 0.493, loss: 0.908
epoch: 4800, acc: 0.487, loss: 0.911
epoch: 4900, acc: 0.547, loss: 0.908
epoch: 5000, acc: 0.520, loss: 0.913
epoch: 5100, acc: 0.537, loss: 0.909
epoch: 5200, acc: 0.560, loss: 0.907
epoch: 5300, acc: 0.553, loss: 0.914
epoch: 5400, acc: 0.580, loss: 0.893
epoch: 5500, acc: 0.573, loss: 0.927
epoch: 5600, acc: 0.573, loss: 0.910
epoch: 5700, acc: 0.523, loss: 0.886
epoch: 5800, acc: 0.593, loss: 0.894
epoch: 5900, acc: 0.530, loss: 0.904
epoch: 6000, acc: 0.540, loss: 0.876
epoch: 6100, acc: 0.510, loss: 0.900
epoch: 6200, acc: 0.567, loss: 0.881
epoch: 6300, acc: 0.583, loss: 0.920
epoch: 6400, acc: 0.573, loss: 0.896
epoch: 6500, acc: 0.550, loss: 0.865
epoch: 6600, acc: 0.500, loss: 0.899
epoch: 6700, acc: 0.590, loss: 0.874
epoch: 6800, acc: 0.590, loss: 0.936
epoch: 6900, acc: 0.547, loss: 0.894
epoch: 7000, acc: 0.570, loss: 0.890
epoch: 7100, acc: 0.560, loss: 0.876
epoch: 7200, acc: 0.583, loss: 0.866
epoch: 7300, acc: 0.550, loss: 0.876
epoch: 7400, acc: 0.527, loss: 0.874
epoch: 7500, acc: 0.533, loss: 0.854
epoch: 7600, acc: 0.540, loss: 0.847
epoch: 7700, acc: 0.540, loss: 0.842
epoch: 7800, acc: 0.540, loss: 0.841
epoch: 7900, acc: 0.553, loss: 0.882
epoch: 8000, acc: 0.577, loss: 0.961
epoch: 8100, acc: 0.500, loss: 0.914
epoch: 8200, acc: 0.510, loss: 0.872
epoch: 8300, acc: 0.513, loss: 0.869
epoch: 8400, acc: 0.597, loss: 0.898
epoch: 8500, acc: 0.553, loss: 0.927
epoch: 8600, acc: 0.520, loss: 0.884
epoch: 8700, acc: 0.533, loss: 0.852
epoch: 8800, acc: 0.540, loss: 0.844
```

```
epoch: 8900, acc: 0.540, loss: 0.840
epoch: 9000, acc: 0.537, loss: 0.840
epoch: 9100, acc: 0.543, loss: 0.851
epoch: 9200, acc: 0.560, loss: 0.880
epoch: 9300, acc: 0.590, loss: 0.914
epoch: 9400, acc: 0.610, loss: 0.926
epoch: 9500, acc: 0.533, loss: 0.932
epoch: 9600, acc: 0.503, loss: 0.915
epoch: 9700, acc: 0.503, loss: 0.894
epoch: 9800, acc: 0.510, loss: 0.882
epoch: 9900, acc: 0.513, loss: 0.873
epoch: 10000, acc: 0.513, loss: 0.872
```