notes_13

January 1, 2025

1 Previous Class Definitions

The previously defined Layer_Dense, Activation_ReLU, Activation_Softmax, Loss, and Loss_CategoricalCrossEntropy classes.

[1]:
```
# imports
import matplotlib.pyplot as plt
import numpy as np
import nnfs
from nnfs.datasets import spiral_data, vertical_data
nnfs.init()
```

[2]:
```
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
```

```python
# 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 Backpropagation of a Single Neuron

Backpropagation helps us find the gradient of the neural network with respect to each of the parameters (weights and biases) of each neuron.

Imagine a layer that has 3 inputs and 1 neuron. There are 3 inputs (x0, x1, x2), three weights (w0, w1, w2), 1 bias (b0), and 1 output (z). There is a ReLU activation layer after the neuron output going into a square loss function (loss = z^2).

Loss = (ReLU(sum(mul(x0, w0), mul(x1, w1), mul(x2, w2(, b0)))))^2

$\frac{\delta Loss()}{\delta w0} = \frac{\delta Loss()}{\delta ReLU()} * \frac{\delta ReLU()}{\delta sum()} * \frac{\delta sum()}{\delta mul(x0,w0)} * \frac{\delta mul(x0,w0)}{\delta w0}$

$\frac{\delta Loss()}{\delta ReLU()} = 2 * ReLU(sum(...))$

$\frac{\delta ReLU()}{\delta sum()}$ = 0 if sum(...) is less than 0 and 1 if sum(...) is greater than 0

$\frac{\delta sum()}{\delta mul(x0,w0)} = 1$

$\frac{\delta mul(x0,w0)}{\delta w0} = x0$

This is repeated for w0, w1, w2, b0.

We then use numerical differentiation to approximate the gradient. Then, we update the parameters using small step sizes, such that $w0[i+1] = w0[i] - step * \frac{\delta Loss()}{\delta w0}$

[3]:
```
import numpy as np

# Initial parameters
weights = np.array([-3.0, -1.0, 2.0])
bias = 1.0
inputs = np.array([1.0, -2.0, 3.0])
target_output = 0.0
learning_rate = 0.001

def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return np.where(x > 0, 1.0, 0.0)

for iteration in range(200):
    # Forward pass
    linear_output = np.dot(weights, inputs) + bias
    output = relu(linear_output)
    loss = (output - target_output) ** 2

    # Backward pass to calculate gradient
    dloss_doutput = 2 * (output - target_output)
    doutput_dlinear = relu_derivative(linear_output)
    dlinear_dweights = inputs
    dlinear_dbias = 1.0

    dloss_dlinear = dloss_doutput * doutput_dlinear
    dloss_dweights = dloss_dlinear * dlinear_dweights
    dloss_dbias = dloss_dlinear * dlinear_dbias

    # Update weights and bias
    weights -= learning_rate * dloss_dweights
    bias -= learning_rate * dloss_dbias

    # Print the loss for this iteration
    print(f"Iteration {iteration + 1}, Loss: {loss}")

print("Final weights:", weights)
print("Final bias:", bias)
```

```
Iteration 1, Loss: 36.0
Iteration 2, Loss: 33.872397424621624
Iteration 3, Loss: 31.87054345809546
Iteration 4, Loss: 29.98699091998773
```

```
Iteration 5, Loss: 28.214761511794592
Iteration 6, Loss: 26.54726775906168
Iteration 7, Loss: 24.978326552541866
Iteration 8, Loss: 23.5021050739742
Iteration 9, Loss: 22.11313179151597
Iteration 10, Loss: 20.806246424284897
Iteration 11, Loss: 19.576596334671486
Iteration 12, Loss: 18.41961908608719
Iteration 13, Loss: 17.33101994032309
Iteration 14, Loss: 16.306757070164853
Iteration 15, Loss: 15.343027506224132
Iteration 16, Loss: 14.436253786815284
Iteration 17, Loss: 13.583071280700132
Iteration 18, Loss: 12.780312744165439
Iteration 19, Loss: 12.024995767388878
Iteration 20, Loss: 11.314319082257104
Iteration 21, Loss: 10.64564263994962
Iteration 22, Loss: 10.016485041642266
Iteration 23, Loss: 9.424510031713222
Iteration 24, Loss: 8.867521365009814
Iteration 25, Loss: 8.34345204094211
Iteration 26, Loss: 7.850353118483743
Iteration 27, Loss: 7.386397874602818
Iteration 28, Loss: 6.94986173712617
Iteration 29, Loss: 6.539124434950737
Iteration 30, Loss: 6.1526621719118015
Iteration 31, Loss: 5.789039869058961
Iteration 32, Loss: 5.446907999417336
Iteration 33, Loss: 5.124995576577539
Iteration 34, Loss: 4.822108497170647
Iteration 35, Loss: 4.537121521071987
Iteration 36, Loss: 4.268978030723312
Iteration 37, Loss: 4.01668121563854
Iteration 38, Loss: 3.7792956126389763
Iteration 39, Loss: 3.5559389510643094
Iteration 40, Loss: 3.345782865003274
Iteration 41, Loss: 3.1480471758404285
Iteration 42, Loss: 2.961997679823884
Iteration 43, Loss: 2.78694359065541
Iteration 44, Loss: 2.622235303237792
Iteration 45, Loss: 2.467261121418954
Iteration 46, Loss: 2.321446092335641
Iteration 47, Loss: 2.184248486806066
Iteration 48, Loss: 2.0551593804914616
Iteration 49, Loss: 1.9336995852420789
Iteration 50, Loss: 1.8194178573235094
Iteration 51, Loss: 1.7118903069357754
Iteration 52, Loss: 1.6107175940030252
```

```
Iteration 53, Loss: 1.5155241897377694
Iteration 54, Loss: 1.4259567411109748
Iteration 55, Loss: 1.3416826255281136
Iteration 56, Loss: 1.262389208248047
Iteration 57, Loss: 1.1877819791340551
Iteration 58, Loss: 1.1175840765571434
Iteration 59, Loss: 1.0515348500680068
Iteration 60, Loss: 0.9893891461492582
Iteration 61, Loss: 0.930916260625565
Iteration 62, Loss: 0.875899078709395
Iteration 63, Loss: 0.8241334819517507
Iteration 64, Loss: 0.7754271861095672
Iteration 65, Loss: 0.7295994320679934
Iteration 66, Loss: 0.6864801042040583
Iteration 67, Loss: 0.6459091389617334
Iteration 68, Loss: 0.6077358933180028
Iteration 69, Loss: 0.5718187120029812
Iteration 70, Loss: 0.5380242202642829
Iteration 71, Loss: 0.5062269967452033
Iteration 72, Loss: 0.4763089781884024
Iteration 73, Loss: 0.4481591180173807
Iteration 74, Loss: 0.42167291418136477
Iteration 75, Loss: 0.3967520449790852
Iteration 76, Loss: 0.3733039992368791
Iteration 77, Loss: 0.3512417316144445
Iteration 78, Loss: 0.33048334753976116
Iteration 79, Loss: 0.31095177724411444
Iteration 80, Loss: 0.2925745286179104
Iteration 81, Loss: 0.2752833763568879
Iteration 82, Loss: 0.25901412505149535
Iteration 83, Loss: 0.2437063914735247
Iteration 84, Loss: 0.22930333977371198
Iteration 85, Loss: 0.21575151284725816
Iteration 86, Loss: 0.2030006012946216
Iteration 87, Loss: 0.19100326852350488
Iteration 88, Loss: 0.17971497196649536
Iteration 89, Loss: 0.1690938194815031
Iteration 90, Loss: 0.1591003719214838
Iteration 91, Loss: 0.14969754273736763
Iteration 92, Loss: 0.14085041966208015
Iteration 93, Loss: 0.13252615564761738
Iteration 94, Loss: 0.1246938532452423
Iteration 95, Loss: 0.11732446503349986
Iteration 96, Loss: 0.11039058885430607
Iteration 97, Loss: 0.10386649785129919
Iteration 98, Loss: 0.09772798570124883
Iteration 99, Loss: 0.09195226348280558
Iteration 100, Loss: 0.0865178816583512
```

```
Iteration 101, Loss: 0.08140467291758889
Iteration 102, Loss: 0.07659366262828358
Iteration 103, Loss: 0.07206697005843195
Iteration 104, Loss: 0.06780781192053903
Iteration 105, Loss: 0.06380037696069592
Iteration 106, Loss: 0.06002977345222309
Iteration 107, Loss: 0.0564820075507719
Iteration 108, Loss: 0.05314393144118542
Iteration 109, Loss: 0.050003114234231524
Iteration 110, Loss: 0.04704793686603195
Iteration 111, Loss: 0.04426740148833972
Iteration 112, Loss: 0.04165120020443161
Iteration 113, Loss: 0.03918961375201954
Iteration 114, Loss: 0.0368735034129829
Iteration 115, Loss: 0.034694277992582755
Iteration 116, Loss: 0.032643851730490094
Iteration 117, Loss: 0.03071459534999028
Iteration 118, Loss: 0.028899363239415818
Iteration 119, Loss: 0.027191414181739672
Iteration 120, Loss: 0.02558439994540113
Iteration 121, Loss: 0.024072362337913877
Iteration 122, Loss: 0.022649683089386127
Iteration 123, Loss: 0.021311092099735786
Iteration 124, Loss: 0.02005160424149179
Iteration 125, Loss: 0.01886655505507656
Iteration 126, Loss: 0.017751540667355833
Iteration 127, Loss: 0.016702427744061103
Iteration 128, Loss: 0.01571531497821091
Iteration 129, Loss: 0.014786535770396103
Iteration 130, Loss: 0.013912651762769943
Iteration 131, Loss: 0.013090418519936803
Iteration 132, Loss: 0.012316768931710837
Iteration 133, Loss: 0.011588849600126475
Iteration 134, Loss: 0.010903943586632107
Iteration 135, Loss: 0.010259526183227799
Iteration 136, Loss: 0.009653186757193668
Iteration 137, Loss: 0.009082688171817357
Iteration 138, Loss: 0.008545899068542421
Iteration 139, Loss: 0.00804083320361364
Iteration 140, Loss: 0.007565618804557518
Iteration 141, Loss: 0.007118492429622391
Iteration 142, Loss: 0.006697793120481266
Iteration 143, Loss: 0.0063019473730584336
Iteration 144, Loss: 0.005929501997799936
Iteration 145, Loss: 0.005579070290327091
Iteration 146, Loss: 0.005249347396309216
Iteration 147, Loss: 0.004939114136252681
Iteration 148, Loss: 0.004647215154254898
```

```
Iteration 149, Loss: 0.00437256400626425
Iteration 150, Loss: 0.004114139259196158
Iteration 151, Loss: 0.0038709956233987848
Iteration 152, Loss: 0.0036422222163822442
Iteration 153, Loss: 0.0034269635873455254
Iteration 154, Loss: 0.0032244300300798123
Iteration 155, Loss: 0.003033866206344064
Iteration 156, Loss: 0.0028545694817259646
Iteration 157, Loss: 0.0026858615040063873
Iteration 158, Loss: 0.002527124440860861
Iteration 159, Loss: 0.002377772426750458
Iteration 160, Loss: 0.0022372501846465924
Iteration 161, Loss: 0.002105026221950533
Iteration 162, Loss: 0.0019806188966821317
Iteration 163, Loss: 0.001863566163059441
Iteration 164, Loss: 0.0017534302886055876
Iteration 165, Loss: 0.0016498016244949178
Iteration 166, Loss: 0.0015522968336895225
Iteration 167, Loss: 0.0014605572212372654
Iteration 168, Loss: 0.0013742383231737623
Iteration 169, Loss: 0.0012930183418168389
Iteration 170, Loss: 0.0012166008279945002
Iteration 171, Loss: 0.0011447005613673634
Iteration 172, Loss: 0.0010770513341135804
Iteration 173, Loss: 0.001013397095948145
Iteration 174, Loss: 0.0009535029620325111
Iteration 175, Loss: 0.0008971534673183893
Iteration 176, Loss: 0.0008441301639000644
Iteration 177, Loss: 0.0007942435095401501
Iteration 178, Loss: 0.0007473036766382048
Iteration 179, Loss: 0.0007031374518087182
Iteration 180, Loss: 0.0006615806720993984
Iteration 181, Loss: 0.0006224808039162045
Iteration 182, Loss: 0.0005856932236775429
Iteration 183, Loss: 0.0005510780772974099
Iteration 184, Loss: 0.0005185112321657664
Iteration 185, Loss: 0.00048786689510026934
Iteration 186, Loss: 0.00045903387854597503
Iteration 187, Loss: 0.00043190420223823955
Iteration 188, Loss: 0.000406378034681195
Iteration 189, Loss: 0.00038236074013664776
Iteration 190, Loss: 0.0003597649139507893
Iteration 191, Loss: 0.0003385032407062897
Iteration 192, Loss: 0.00031849748027454767
Iteration 193, Loss: 0.00029967346881992795
Iteration 194, Loss: 0.0002819629431575354
Iteration 195, Loss: 0.0002652991815966534
Iteration 196, Loss: 0.00024961903501571355
```

```
Iteration 197, Loss: 0.00023486641976601822
Iteration 198, Loss: 0.00022098629075865584
Iteration 199, Loss: 0.00020792651372860275
Iteration 200, Loss: 0.00019563773612380077
Final weights: [-3.3990955  -0.20180899  0.80271349]
Final bias: 0.6009044964517248
```

3 Backpropagation of a Layer

Same thing as a single neuron, but now using matrices to keep track of each neuron in the layer.

If there are multiple input arrays (batches), one can take the summation of the loss from each batch as a total loss, and therefore the gradient of the total loss with respect to a weight or bias is the summation of the gradients of each batch's loss with respect to the weight or bias given that batch's input.

In general, the partial derivative of the loss with respect to a specific weight or bias remains the same across all neurons of that layer for that batch. ie, the weight gradient matrix has the same column vector for N number of neurons. The bias gradient matrix is similar but is a single row of N elements for the same value.

[5]:
```python
import numpy as np

# Initial inputs
inputs = np.array([1, 2, 3, 4])

# Initial weights and biases
weights = np.array([
    [0.1, 0.2, 0.3, 0.4],
    [0.5, 0.6, 0.7, 0.8],
    [0.9, 1.0, 1.1, 1.2]
])

biases = np.array([0.1, 0.2, 0.3])

learning_rate = 0.001

# Add the derivative function to the ReLU class
class Activation_ReLU:
    def forward(self, inputs):
        return np.maximum(0, inputs)

    def derivative(self, inputs):
        return np.where(inputs > 0, 1, 0)

relu = Activation_ReLU()

num_iterations = 200
```

```python
# Training loop
# A single layer of 3 neurons, each with 4 inputs
# The neuron layer is then fed into a ReLU activation layer
for iteration in range(num_iterations):
    # Forward pass
    neuron_outputs = np.dot(weights, inputs) + biases
    relu_outputs = relu.forward(neuron_outputs)

    # Calculate the squared loss assuming the desired output is a sum of 0.␣
 ↪Trivial but just an example
    final_output = np.sum(relu_outputs)
    loss = final_output**2

    # Backward pass
    dL_dfinal_output = 2 * final_output
    dfinal_output_drelu_output = np.ones_like(relu_outputs)
    drelu_output_dneuron_output = relu.derivative(neuron_outputs)

    dL_dneuron_output = dL_dfinal_output * dfinal_output_drelu_output *␣
 ↪drelu_output_dneuron_output

    # Get the gradient of the Loss with respect to the weights and biases
    # dL_dW = np.outer(dL_dneuron_output, inputs)
    dL_dW = inputs.reshape(-1, 1) @ dL_dneuron_output.reshape(1, -1)
    dL_db = dL_dneuron_output

    # Update the weights and biases
    # Remove the .T if using dL_dW = np.outer(dL_dneuron_output, inputs)
    weights -= learning_rate * dL_dW.T
    biases -= learning_rate * dL_db

    # Print the loss every 20 iterations
    if iteration % 20 == 0:
        print(f"Iteration {iteration}, Loss: {loss}")

# Final weights and biases
print("Final weights:\n", weights)
print("Final biases:\n", biases)
```

```
Iteration 0, Loss: 466.56000000000006
Iteration 20, Loss: 5.329595763793193
Iteration 40, Loss: 0.41191524253483786
Iteration 60, Loss: 0.03183621475376345
Iteration 80, Loss: 0.002460565405431671
Iteration 100, Loss: 0.0001901729121621426
Iteration 120, Loss: 1.4698120139337557e-05
```

```
Iteration 140, Loss: 1.1359948840900371e-06
Iteration 160, Loss: 8.779778427447647e-08
Iteration 180, Loss: 6.785903626216421e-09
Final weights:
 [[-0.00698895 -0.0139779  -0.02096685 -0.0279558 ]
 [ 0.25975286  0.11950571 -0.02074143 -0.16098857]
 [ 0.53548461  0.27096922  0.00645383 -0.25806156]]
Final biases:
 [-0.00698895 -0.04024714 -0.06451539]
```

[ ]:

3.1 Change of 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.

3.1.1 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

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

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

3.1.2 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 $\frac{\delta l}{\delta \vec{B}} = \frac{\delta l}{\delta \vec{Z}}\frac{\delta \vec{Z}}{\delta \vec{B}} = \vec{1}\begin{bmatrix} \frac{\delta l}{\delta z_1} & \frac{\delta l}{\delta z_2} & \cdots & \frac{\delta l}{\delta z_i} \end{bmatrix}$

[13]:
```python
# Code changed to match new notation
import numpy as np

# Initial inputs
X = np.array([1, 2, 3, 4])

# Initial weights and biases
W = np.array([
    [0.1, 0.2, 0.3, 0.4],
    [0.5, 0.6, 0.7, 0.8],
    [0.9, 1.0, 1.1, 1.2]
])

B = np.array([0.1, 0.2, 0.3])

learning_rate = 0.001

# Add the derivative function to the ReLU class
class Activation_ReLU:
    def forward(self, inputs):
        return np.maximum(0, inputs)

    def derivative(self, inputs):
        return np.where(inputs > 0, 1, 0)

relu = Activation_ReLU()
```

```
num_iterations = 200

# Training loop
# A single layer of 3 neurons, each with 4 inputs
# The neuron layer is then fed into a ReLU activation layer
for iteration in range(num_iterations):
    # Forward pass
    Z = np.dot(W, X) + B
    A = relu.forward(Z)

    # Calculate the squared loss assuming the desired output is a sum of 0.␣
 ↪Trivial but just an example
    y = np.sum(A)
    l = y**2

    # Backward pass
    dL_dy = 2 * y
    dy_dA = np.ones_like(A)
    dA_dZ = relu.derivative(Z)

    dl_dZ = dL_dy * dy_dA * dA_dZ

    # Get the gradient of the Loss with respect to the weights and biases
    dL_dW = np.outer(X.T, dl_dZ)
    dL_dB = dl_dZ

    # Update the weights and biases
    W -= learning_rate * dL_dW.T
    B -= learning_rate * dL_dB

    # Print the loss every 20 iterations
    if iteration % 20 == 0:
        print(f"Iteration {iteration}, Loss: {l}")

# Final weights and biases
print("Final weights:\n", W)
print("Final biases:\n", B)
```

```
Iteration 0, Loss: 466.56000000000006
Iteration 20, Loss: 5.32959636083938
Iteration 40, Loss: 0.41191523404899866
Iteration 60, Loss: 0.031836212079467595
Iteration 80, Loss: 0.002460565465389601
Iteration 100, Loss: 0.000190172825660145
Iteration 120, Loss: 1.4698126966451542e-05
Iteration 140, Loss: 1.1359926717815175e-06
```

```
Iteration 160, Loss: 8.779889800154524e-08
Iteration 180, Loss: 6.7858241357822796e-09
Final weights:
 [[-0.00698895 -0.01397789 -0.02096684 -0.02795579]
 [ 0.25975286  0.11950572 -0.02074143 -0.16098857]
 [ 0.53548461  0.27096922  0.00645383 -0.25806156]]
Final biases:
 [-0.00698895 -0.04024714 -0.06451539]
```

4 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

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

4.1 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.