Contents

1 Introduction

Three items are presented here. The first is an algorithmic description of the back-propagation process, used to nudge the weights and biases of a neural network so that inputs to the network produce desired outputs to within some arbitrarily small error. This intended for those wishing to hand-code their own neural network with back propagation, and need the associated logic and equations to do so.

This initial presentation is focused on building yourself an 'instructive' neural network, where each neuron is actually a structure containing its own input, activation, delta, and an array of weights that connects it to the neurons in the next forward layer. For this, pseudo-code is presented indicating how both forward and backward passes through such a

a neural network work. Such a neural network is not meant to be fast or 'production ready' (in fact, it's quite slow), but is nonetheless instructive.

The second is a matrix representation of the same. This allows for the use of a matrix library such as Numpy to speed up the calculations required of the neural network.

The third is a "calculus first" derivation of the equations that drive back-propagation. The calculus-first derivation keeps the logic clean by relying only on the chain and product rules of calculus, with simplifications of the results that require a thoughtful interpretation of the internal properties of a neural network.

2 Motivation: Develop an educational neural network

The goal of this work is to develop an "educational" neural network that can be used to understand how back propagation works. By educational we mean there won't be any optimizations, matrices, or linear algebra. The network will run with nested for-loops and really obvious variable names. Back propagation equations will just be stated as givens and used as such. In this educational model, a data structure will be developed for a neuron that will contain information an individual neuron needs to hold, and see how it connects to the larger network. The result will be a complete neural network that you can train and use, but it'll be slow.

2.1 A neuron data structure

In this work the educational neural network was developed in Python. Each neuron is represented by a Python dictionary that looks like this:

```
1 {
2    "desc": desc,
3    "z":0.0,  #input
4    "a": 0.0,  #activation
5    "b": 0.01, #bias
6    "w": [],  #weights
7    "delta": 0.0
8 }
```

Here `z`, `a`, and `b` are the neuron's input, activation and bias. The `w` is a list of initially small random numbers representing each weight that connects this neuron to the neurons in the next forward layer. If for example, the forward layer has 6 neurons in it, this list will contain 6 floating point values. Lastly, the `delta` is the "error" of the neuron's output, which is used in the back propagation process.

A layers of neurons is represented as a list of this basic neuron structure, with one structure per neuron in a layer. A similar implementation in C would likely use a `struct` to

define a neuron.

2.2 Definitions

The neural network (or a section of a larger network) analyzed here is shown in Figure 1. Three alphabetical letters are chosen to refer to the layers, i, j, and k. The output layer is k, which is fed from hidden (or inner) layer j, which is fed from another layer i. Note i may also be the input layer or another hidden layer. More layers can be expanded to the left of i, perhaps layer h (which is referenced below).

All neurons between layers are interconnected. Based on this structure, here are a few definitions used throughout this work (note, we used the generics x and y below, which can refer to i, j, or k).

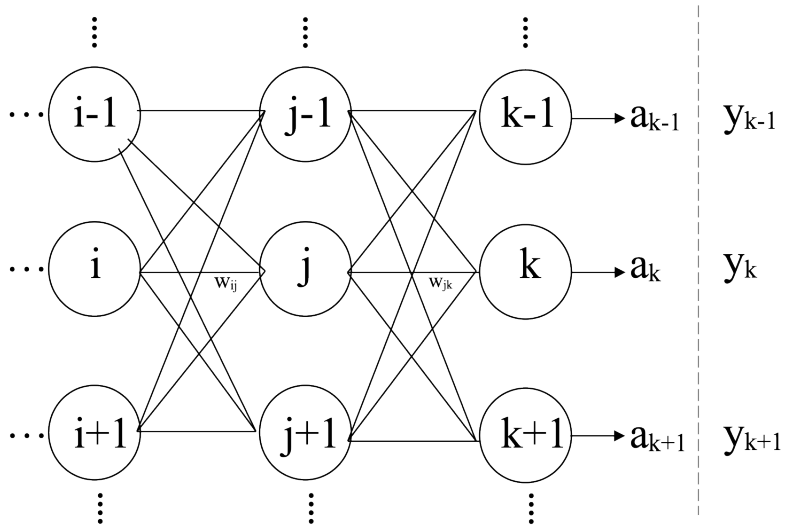

Figure 1: The neural network modeled here. The output layer is referenced with the k index and the hidden layers with j and i. The weights w_{ij} and w_{jk} are highlighted, as they are used in the text.

- w_{xy}. This refers to the weight of neuron x in layer L into neuron y in layer $L+1$. This could be expressed as $w_{L,x\to(L+1),y}$ or $w_x^L \to w_y^{L+1}$, or something similar, but we aim here to simplify the notation. So, w_{ij} will be the weight between the i^{th} neuron in layer i and the j^{th} neuron in layer j, with $j = i + 1$ (always).

  To simplify notation, i, j, and k are used to denote both layers and individual neurons. We hope this balances any conceptual confusion with notational simplicity by the

context in which something is mentioned. So, layer i, or neuron i (implied in layer i) can be referenced, as can w_{ij} as the weight connecting neuron i (in implied layer i) to layer j (in implied layer j).

As used here in w_{xy}, x is always in a layer that feeds input data forward into the layer that contains neuron y, so y is always $x + 1$. In general, x is an index that can run from 1 to the number of neurons in the layer. In words, w_{xy} would be read "the weight from neuron x in implied layer x into neuron y in implied layer y" with the implicit meaning that neuron x feeds neuron y. Symbolically, $w_{from-layer \to to-layer}$.

Note: it turns out that the `from layer→to layer` notation may not be good in the long term, as it'll go counter to some numbering conventions used in he matrix representation (see below). The `from layer→to layer` idea however, is just too tempting to pass up as we try to grasp back propagation. We'll point this out again in the matrix treatment, and even point out the same decision made by other authors.

- z_y. This is the input to neuron y from the backward (or leftward) layer x that feeds it. Here

$$z_y = \sum_x w_{xy} a_x + b_y. \tag{1}$$

  So, the input to neuron y, or z_y, is given by a sum over all of the products of the weights that feed it from layer x (w_{xy}) and the activation of the neuron feeding that weight from layer x, a_x (see below), plus the bias of the destination neuron, b_y (see below).

- $f(x)$ is the squashing function or activation function for a neuron, and $f'(x)$ is the derivative of the activation function. Note, seeing something like $f'(z_k)$ means to find the derivative of f (with respect to its independent variable, x for example) then evaluate the derivative at z_k. f is typically ReLU or the sigmoid function.

- a_x this is the "activation" of neuron x, defined as $a_x = f(z_x)$.

- y_x is the target (or needed) value of an output neuron. The goal of back propagation is to make all a_x values arbitrarily close to a corresponding y_x value.

2.3 The forward pass

2.3.1 Pseudo-code

Here is pseudo-code that will take an input vector `input` and send it forward through the network. A few notes on the code:

1. A neuron is specified in a cartesian (x, y) format like `(neuron,layer)` where `neuron` is the neuron number in layer `layer`.

2. Any other network-wide variable is in parentheses like `output_layer` which is the numerical index to the output layer (and presumably an array of output layer neurons).

3. All for-loop are inclusive of both their starting and ending values.

```
#load up input layer
for neuron = 0 to (number of input neurons):
  z of input neuron (neuron) = input vector[neuron]
  a of input neuron (neuron) = f(z of input neuron (neuron))
end

#propagate from input layer through the entire network
for layer = 0 to output_layer-1
  forward_layer = layer + 1
  for forward_neuron = 0 to neuron in layer (forward_layer)
    z = 0.0
    for neuron =0 to count of neurons in layer (layer)
      z = z + weight connecting (layer,neuron) and (forward_layer,forward_neuron)
          * a of (layer,neuron) + bias of (layer,neuron)
    end
    z of (forward_layer,forward_neuron) = z
    a of (forward_layer,forward_neuron) = f(z)
  end
end
```

2.4 Back-propagation

Here we assume you have a set of input/output vectors on which you'd to train the network. Assuming your output layer is layer k, the layer to the left of it is j, and to the left of it i, here's how to correct the weights via back-propagation. Before beginning, establish some learning rate η for the back-propagation.

1. Send an input vector forward through the network and compute z_x and $a_x = f(z_x)$ for all neurons in the network.

2. For each neuron k in the output layer compute $\delta_k = f'(z_k)(a_k - y_k)$, where y_k is the corresponding component of the output vector.

3. Loop through all weights between layers j and k. It's best to loop through j as the outer loop and k and the inner loop. Correct w_{jk} like this

$$w_{jk,new} = w_{jk,old} - \eta \delta_k a_j. \tag{2}$$

4. Loop through all neurons in layer j and compute δ_j for each neuron from the formula $\delta_j = f'(z_j) \sum_k \delta_k w_{jk}$. It's best to start an outer loop on j through the neurons, an inner loop on k to compute the sum.

5. Loop through all weights between layers i and j. Use i as your outer loop, and j as your inner loop. Correct w_{ij} like this

$$w_{ij,new} = w_{ij,old} - \eta \delta_j a_i. \tag{3}$$

6. Have a layer to the left of i? Call it h. Loop through all neurons in layer i and compute δ_i for each neuron from the formula $\delta_i = f'(z_i) \sum_j \delta_j w_{ij}$. It's best to start an outer loop on i through the neurons, an inner loop on j to compute the sum.

7. Loop through all weights between layers h and i. Use h as your outer loop, and i as your inner loop. Correct w_{hi} like this

$$w_{hi,new} = w_{hi,old} - \eta \delta_i a_j. \tag{4}$$

You may have more hidden weights to correct, in which case the pattern evolving pattern here can be used. So, instead of continuing with explicit formulas for correcting the rest of your weights, let's come up with a systematic pattern.

Let's suppose the last set of weights you corrected were between layers b and c, where c is always $b + 1$. So you corrected weights w_{bc}. Now you want to keep going and correct weights w_{ab}, where a is $b - 1$. Here's what you'd do

1. Since you corrected weights w_{bc}, you must have all of the δ_c values available.

2. Compute δ_b from $\delta_b = f'(z_b) \sum_c \delta_c w_{bc}$.

3. Correct w_{ab} with $w_{ab,new} = w_{ab,old} - \eta(\delta_b a_a)$.

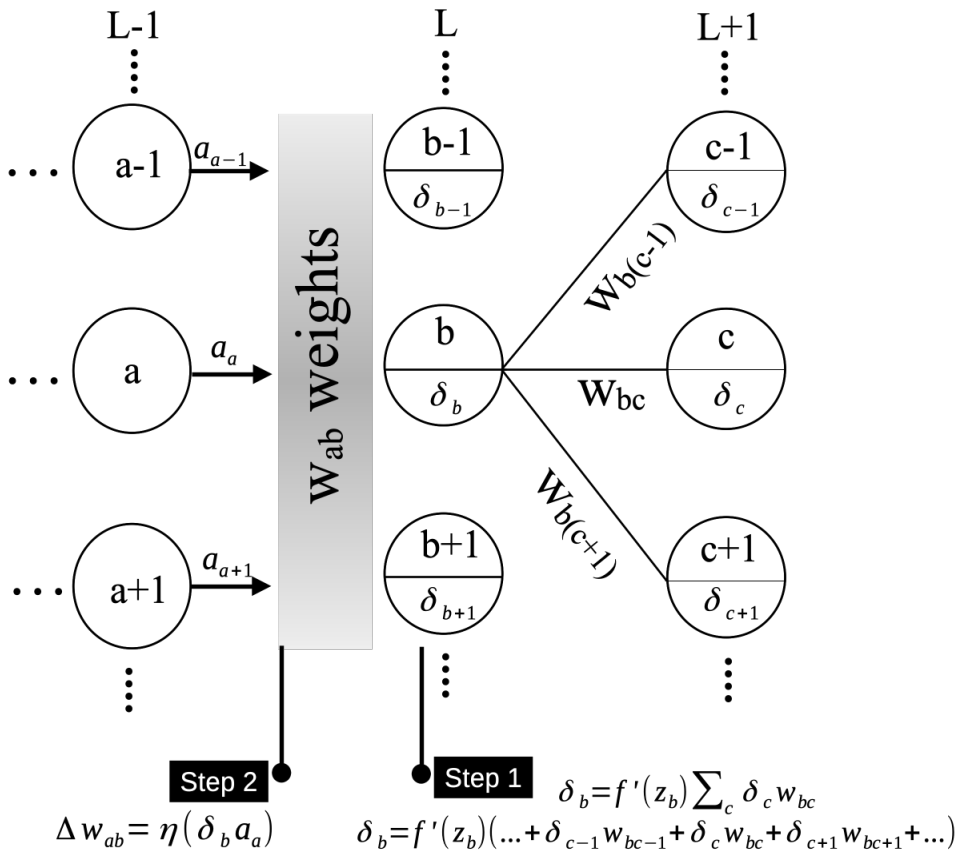

Figure 2: The back propagation plan. Step 1: Compute δ_j for the first layer to the left of the last one that has been corrected (i.e. the last one whose δs are known). Step 2: Adjust the weights by Δw_{ab}.

2.4.1 Pseudo-code

Here's pseudo code (illustrated in Figure 2) that will traverse a network backward, correcting its weights and neuron biases. The following assumptions are made:

1. You have appropriate data structures for a neural network all set up.
2. A forward pass of some input has been applied to the network and all neuron inputs (z) and activations $a = f(z)$ have been computed.
3. You know the corresponding output for a given input.

4. You can retrieve the number of neurons in a given layer, as well as a and z for any neuron.

5. You can retrieve the weight connecting any two neurons and modify it.

6. You can retrieve the bias of a neuron and modify it.

7. You've defined some learning rate `eta` and some learning rate for the biases `eta_b`.

```
#full backpropagation pass through a neural network

function fp(x)
 return(derivative of activation function evaluated at x)
end

output_layer = index of the output layer

#compute all of the deltas for the output layer
#and correct bias of all neurons in the output layer
for neuron=1 to number of neurons in (output_layer)
  z = input to (neuron,output_layer)
  a = activation of (neuron,output_layer)
  dL = a - target value needed of (neuron,output_layer)
  delta of (neuron,output_layer) = dL * fp(z)
  bias of (neuron,output_layer) -= eta_b * delta of (neuron,output_layer)
end

#compute deltas and correct biases for all hidden layer neurons
for hidden_layer=(output_layer)-1 to 0, step -1
  forward_layer = hidden_layer + 1
  for neuron = 0 to number of neurons in (hidden_layer)
    dsum = 0.0
    for forward_neuron = 0 to number of neurons in layer (forward_layer)
      w=weight connecting (neuron,hidden_layer) and (forward_neuron,hidden_layer)
      delta_forward_layer = delta of neuron (forward_neuron,forward_layer)
      dsum = dsum + delta_forward_layer * w
      z = input to (neuron,hidden_layer)
      delta of (neuron,hidden_layer) = fp(z) * dsum
    end
  end
end

#correct all weights in the network
```

```
35 for layer=output_layer-1 to 0, step=-1
36   for neuron=1 to number of neurons in layer (layer)
37     bias of neuron at (layer,neuron) += eta_b * delta of neuron at (layer,neuron)
38       a = activation of neuron at (layer,neuron)
39       for forward_neuron=1 to number of neurons in layer (layer+1)
40         delta = delta of neuron at (layer+1,forward_neuron)
41         weight connecting (layer,neuron) and (layer+1,forward_neuron) -= eta *
               delta * a
42       end
43   end
44 end
```

After you've reached the leftmost layer, the back propagation pass for this input/output pair is complete. In practice you don't correct the weights directly as described. For each input/output pair, you accumulate the sum of all needed weight changes for a given weight over all needed input/output pairs. Then the average weight change is computed by dividing the sum by the number of input/output pairs. The average weight change is used to actually adjust a weight, so

$$w_{xy,new} = w_{xy,old} + \frac{\sum \Delta w_{xy}}{N}, \tag{5}$$

where N is the number of input/output pairs you are training, $\Delta w_{xy} = w_{xy,new} - w_{xy,old}$, and the sum is over all weight changes a given input/output pair requires for a given weight, w_{xy}.

3 Build an optimized neural network model using matrices

The codebase above works, which means it can be trained on vector pairs, and the loss function (target - output) summed across all output neurons will get arbitrarily small. The code is slow though. Looking at the pseudo-code, you'll see at least one place where there are three nested loops. Typically educational code like this won't be fast.

It turns out that it's better to approach modeling neural networks using matrices from linear algebra to handle all of the numbers. This falls under the purview of linear algebra, for which one should use a professional library's built-in matrix manipulation routines (matrix multiplication, dot product, etc.) wherever possible. In this work, Python and Numpy will be used for this, as shown here. Below it will be shown how both the forward and backward passes can be (re)implemented with matrices.

3.1 Forward pass

Analyzing the forward pass will set the idea for using matrices. Consider Equation 1, which is stated again here for clarity

$$z_y = \sum_x w_{xy} a_x + b_y \tag{6}$$

which the input to neuron y as fed from layer x. Let's write out a few terms of this sum for a few values of y. For $y = 0$ we get

$$z_0 = w_{00} a_0 + w_{10} a_1 + w_{20} a_2 + b_0, \tag{7}$$

$y = 1$ gives

$$z_1 = w_{01} a_0 + w_{11} a_1 + w_{21} a_2 + b_1, \tag{8}$$

and $y = 2$

$$z_2 = w_{02} a_0 + w_{12} a_1 + w_{22} a_2 + b_2. \tag{9}$$

Looking at it, the z's can all be organized into a column vector like this

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \end{pmatrix}. \tag{10}$$

The weights can be contained in a matrix, like this

$$\begin{pmatrix} w_{00} & w_{10} & w_{20} \\ w_{01} & w_{11} & w_{21} \\ w_{02} & w_{12} & w_{22} \end{pmatrix}, \tag{11}$$

the a's in another column vector like this

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix}, \tag{12}$$

and finally, the b's like this

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}. \tag{13}$$

Thus, Equations 7-9 can be written as

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} w_{00} & w_{10} & w_{20} \\ w_{01} & w_{11} & w_{21} \\ w_{02} & w_{12} & w_{22} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}. \tag{14}$$

This is a matrix equation that shows how the activations of the neurons in one layer (a_0, a_1, a_2) feed the inputs of the neurons in the next layer (z_0, z_1, z_2), via the interconnecting weights (the w's), including the direct and additive effect of the biases on each neuron. A single matrix multiplication between the weight matrix and activation vector, plus the biases will yield the inputs to the forward layer. This is contrasted with the two nested for-loops doing the same in the pseudo-code above. Thus, our neural network might be thought of as that shown in Figure 3.

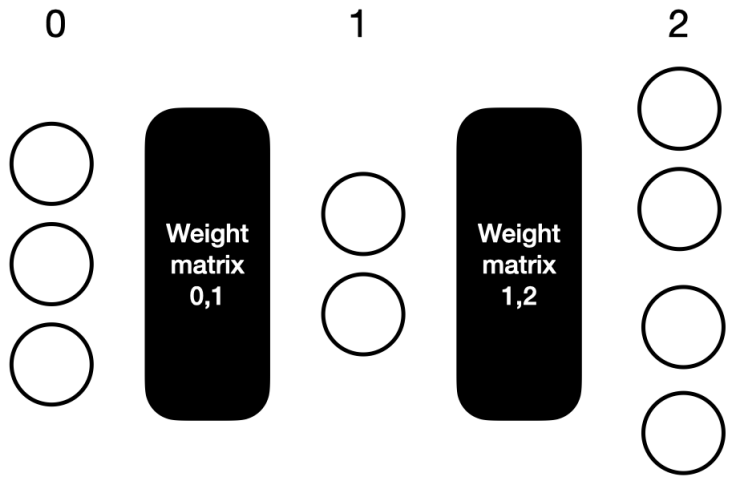

Figure 3: A neural network with 3 input neurons, 2 neurons in the single hidden layer, and 4 neurons in the output layer.

3.1.1 Matrix sizes

Here's another linear algebra consideration for our matrix-style network: the size (that is, the width and height, or rows and columns) of each weight matrix.

As you can tell from the figure, the input layer has 3 neurons, which feed 2 neurons in the hidden layer, which in turn feed 4 neurons in the output layer. Thus, a goal of each weight matrix is to "transform" the activation vector of one layer (an a-vector), into the input of the next layer (the z-vector). In the case of the input to hidden layer here, a 3 row a-vector

from the input layer needs to be transformed into a 2 row z-vector for the hidden layer. Then the 2 row a-vector of the hidden layer needs to be transformed into a 4 row z-vector for the output layer. Here's how it might look for the input to hidden layer transformation

$$\begin{pmatrix} z_0 \\ z_1 \end{pmatrix} = \begin{pmatrix} w_{00} & w_{10} & w_{20} \\ w_{01} & w_{11} & w_{21} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}. \tag{15}$$

Here you can see the 3-row a vector of the input layer being transformed into the 2 row z vector of the hidden layer. This is done with the 3 column, 2 row weight matrix.

As matrix multiplication goes, the weight matrix between two layers must have the same number of columns as neurons in the activation (or source) layer, and the same number of rows as neurons in the destination layer. This will handle any "reshaping" needed of the a-vectors from one layer getting properly shaped into z-vectors for the next layer.

Thus, in Figure 4, we see the 2-row, 3-column weight vector reshaping the 3 neuron input vector into a 2 neuron hidden layer vector. Then, a 4-row, 2-column weight vector reshapes the 2 neuron hidden layer into a 4 neuron output layer.

So, the forward pass of a neural network is a series of matrix multiplications, that transform the input vector into the output vector, with potentially many multiplications in between. Each weight matrix scales a vector with the its numbers, and reshapes it with its size as needed by the next layer. So all we'll need to run a forward pass is a good matrix multiplication routine, not any clunky nested for-loops.

3.1.2 Pseudo-code

Here is pseudo-code that will take a vector `input` and propagate it through the network.

```
1  function forward(input)
2    layer_count = count of layers in network (input + hidden layers + output)
3
4    #load up the input layer
5
6    #input and bias are both vectors
7    z = input + bias (input layer)
8
9    #f is vectorized, meaning it'll operate on all elements of vector z
10   a = f(z)
11
12   #load up the rest of the network
13   for layer = 0 to layer_count-1
14     # dot is a professional matrix library function that
```

```
15      # will multiply a weight matrix by an activation vector
16      # in Numpy it's called "dot."
17      z = dot(weight matrix for layer to layer+1,a (layer)) + bias (layer+1)
18      a = f(z)
19    end
20    #return the last activation, which will be that for the output layer
21    return a
```

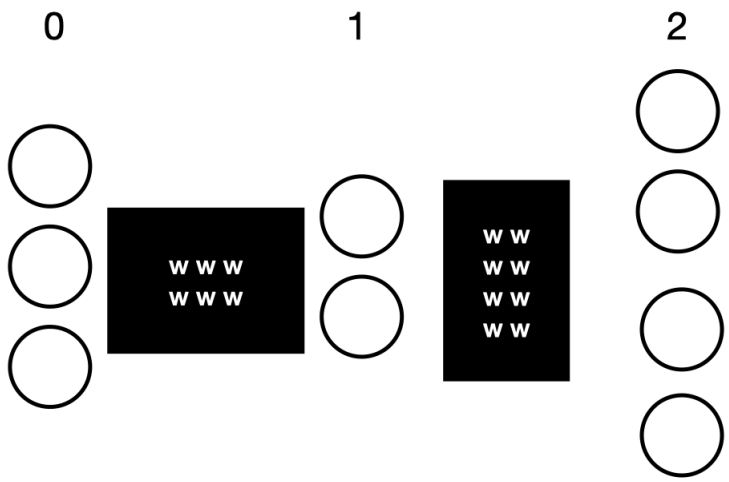

Figure 4: Showing how the weight matrices must vary in size to reshape a source a-vector into the destination z-vector of the next layer.

3.2 Back propagation (the backward pass)

The first consideration for a backward pass is in knowing two things for all neurons in the network, which are 1) the inputs to each (the z's), and the error each neuron is introducing into the overall loss function of the network, which are the δ parameters.

3.2.1 The z-vectors

For later use in the error correcting (i.e. backward) pass, the z-values for all neurons must be captured and stored in as the forward pass progresses [7, 8]. This is not shown in the pseudo-code above, but essentially after each `z=dot(...)` line, the matrix multiplication will return the z-values (or collectively z-vector) for that layer. These vectors must be stored with each iteration of the `layer` forward pass, so they'll be available for later. (In this work, each z-vector is appended to a Python list.)

3.2.2 The δ-vectors

Next is the contribution of each neuron to the overall loss of the network. These are the δ parameters. Here, we will call a given error δ_k. In our simplified notation, this mean δ of the kth neuron in later k.

For the output layer

The δ's for the output layer are straightforward to compute, since the desired target for each output neuron is known. (They are sometimes called the "boundary condition" for the network.) The formula is

$$\delta_k = f'(z_k)(a_k - y_k), \tag{16}$$

where k is the kth neuron in the output layer, $f'(z_k)$ is the derivative of the activation function evaluated at z_k, a_k is the activation (i.e. output) of the kth neuron in the output layer, and y_k is the target value for the kth neuron in the output layer. Thus, a δ-vector can be compiled for the output layer, which will resemble (here show for 4 neurons)

$$\begin{pmatrix} \delta_0 \\ \delta_1 \\ \delta_2 \\ \delta_3 \end{pmatrix}_k = \begin{pmatrix} f'(z_0)(a_0 - y_0) \\ f'(z_1)(a_1 - y_1) \\ f'(z_2)(a_2 - y_2) \\ f'(z_3)(a_3 - y_3) \end{pmatrix}, \tag{17}$$

where the subscript k denotes the output layer of the network. We may also write this as

$$\vec{\delta}_{\text{output layer}} = \begin{pmatrix} f'(z_0)(a_0 - y_0) \\ f'(z_1)(a_1 - y_1) \\ f'(z_2)(a_2 - y_2) \\ f'(z_3)(a_3 - y_3) \\ \vdots \end{pmatrix}, \tag{18}$$

For the hidden layers

The δ parameters for the hidden layers are more complicated to compute, since the target values for such neurons are not known. They are, however, related to the target values of the network as a whole. The δ value for a hidden layer is given by

$$\delta_j = f'(z_j) \sum_k \delta_k w_{jk}. \tag{19}$$

This equation will hold in general (meaning for any two adjacent layers in a network), but for the sake of instruction, let's consider the case of wanting to compute the δ-vector for the first hidden layer that directly feeds the output layer. This will mean that k denotes the output layer and j the first hidden layer.

Looking closely at the sum, one can see that finding a δ_j for a hidden layer neuron consists of propagating the known values of δ_k from the output layer, through all weights that connect them to a given neuron j in the first hidden layer. This is what the sum does. The final step will be to evaluate the derivative of the activation function of the target neuron, at it's output level z_j. Let's look more closely.

Writing out a few terms of the sum (recalling that k refers to a neuron in the output layer), we get

$$\sum_k \delta_k w_{jk} = \delta_0 w_{j0} + \delta_1 w_{j1} + \delta_2 w_{j2} + \delta_3 w_{j3} \ldots . \tag{20}$$

Next, recall that j refers to a neuron in the hidden layer. The first neuron will have $j = 0$, then $j = 1$, etc. So let's look at few more sequences here. For $j = 0$ we get

$$\sum_k \delta_k w_{0k} = \delta_0 w_{00} + \delta_1 w_{01} + \delta_2 w_{02} + \delta_3 w_{03} \ldots . \tag{21}$$

For $j = 1$ we get

$$\sum_k \delta_k w_{1k} = \delta_0 w_{10} + \delta_1 w_{11} + \delta_2 w_{12} + \delta_3 w_{13} \ldots . \tag{22}$$

These patterns reveal the operation needed to compute the sum: the sum is the *dot product* between the δ-vector of the output layer and the weight vector that connects the output layer to neuron j (in the hidden layer). This is illustrated in Figure 5.

Intuitively, in order to compute δ_0 in the hidden layer, we need all of the δ values in the output layer to "feed" their size through each's interconnecting weight to neuron 0 in the hidden layer. This portion of the formula is $\delta_{0,\text{hidden layer}} \sim \delta_0 w_{00} + \delta_1 w_{01} + \delta_2 w_{02} + \delta_3 w_{03}$.

This can all be computed from two things we already have, 1) the weight matrix connecting the two layers, and 2) the δ values from the output layer. Let's look first at the weight matrix connecting the two layers, which is (shaped properly to go from a 2 neuron hidden layer, to a 4 neuron output layer)

$$\begin{pmatrix} w_{00} & w_{10} \\ w_{01} & w_{11} \\ w_{02} & w_{12} \\ w_{03} & w_{13} \end{pmatrix} . \tag{23}$$

Notice the weights we need are in the first column, w_{00}, w_{01}, w_{02}, and w_{03}. We also have the just-computed δ-vector for the output layer, which was

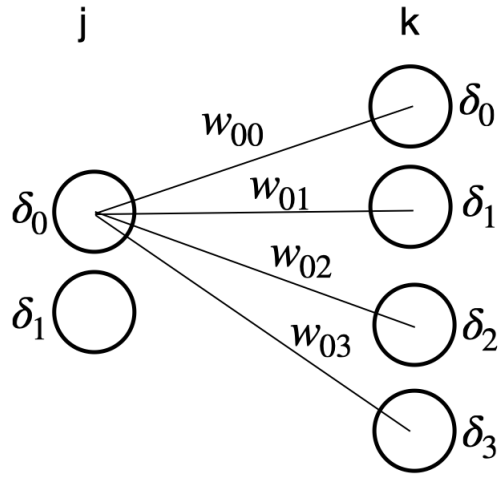

Figure 5: Showing how δ for a neuron in the hidden layer is computed: the δ-values of the *output layer* are fed backwards into a given neuron in the *hidden layer*, through the weights leading into the hidden layer neuron. Here $\delta_{0,\text{hidden layer}} \sim \delta_0 w_{00} + \delta_1 w_{01} + \delta_2 w_{02} + \delta_3 w_{03}$.

$$\begin{pmatrix} \delta_0 \\ \delta_1 \\ \delta_2 \\ \delta_3 \end{pmatrix}_{k.} \tag{24}$$

So, the sum in Equation 19 can be computed via a *dot product* between the first column of the weight matrix and the δ-vector for the output layer. The full result will be

$$\delta_0 = f'(z_0)\vec{w}_{0\text{th col}} \cdot \vec{\delta}_{\text{output layer}}. \tag{25}$$

The δ values for the other neurons will come similarly as in

$$\delta_1 = f'(z_1)\vec{w}_{1\text{st col}} \cdot \vec{\delta}_{\text{output layer}}, \tag{26}$$

$$\delta_2 = f'(z_2)\vec{w}_{2\text{nd col}} \cdot \vec{\delta}_{\text{output layer}}, \tag{27}$$

and

$$\delta_3 = f'(z_3)\vec{w}_{3\text{rd col}} \cdot \vec{\delta}_{\text{output layer}}. \tag{28}$$

Hopefully you see how the δ-vector for the first hidden layer can be constructed and that there is a pattern to the δ-calculating pattern: for the jth neuron in a hidden layer, multiply the derivative of the jth neuron's activation function (evaluated at it's output z_j), by the

dot product between the jth column of the weight matrix and the δ-vector of the forward layer.

As far as matrix shapes go, the number of rows in the weight matrix will always match that of the length of the δ-vector, since the number of rows in the weight matrix is what transforms the forward-going input vector into the number of eventual output neurons. (So, the dot product will always be evaluatable.)

The reason for pushing for mathematical conciseness (as in using a dot product) is that once again (for speed), the dot product function will come from a professional matrix library like NumPy.

As you might guess, there is a pattern for computing and using δ-values. Once they are known for a given layer, they are used in a similar manner to compute the δ-values for the next (hidden layer) to the left of it, etc., all the way down to the input layer.

3.2.3 Correcting the weights

Let's now look at how these δ-vectors are used to correct the weights in a weight matrix. For orientation, let's assume we want to correct the weights in a given weight matrix, and know the δ-vector for the layer to the right of the weight matrix (or the layer that the weight matrix feeds into during a forward pass). So for example, suppose we want to correct the w_{jk} weights and know the δ-vector for layer k. The individual weights in the matrix need to be corrected, in order to nudge the all of the output neurons of the network to within some small distance of the desired target values of the network.

The weight correction formula is

$$\Delta w_{jk} = -\eta \delta_k a_j, \tag{29}$$

where we again focus on the weight matrix between the last hidden layer j and the output layer k. Here, η is a hyperparameter of the network, called the "learning rate," and two more items are needed:

1. a_j, which is the activation of the hidden layer. This was found during the forward pass. As mentioned in the forward pass discussion above, we need to store the z-vectors for each layer during the forward pass (the z_j values). The activation can be found from these since $a_j = f(z_j)$, where f is the activation function.

2. δ_k, which is the δ of the kth neuron in the output layer (its in δ-vector of the output layer).

Ideally, we'd like to once again exploit the speed of a matrix library, and avoid using any for-loops to do all of the weight corrections. Our goal then is to construct a Δw *matrix*, that we can have the library add to the weight matrix, to nudge the weight values. In other words, if we have a weight matrix like this

$$\begin{pmatrix} w_{00} & w_{10} \\ w_{01} & w_{11} \\ w_{02} & w_{12} \\ w_{03} & w_{13} \end{pmatrix}, \tag{30}$$

we'd like to construct

$$\begin{pmatrix} \Delta w_{00} & \Delta w_{10} \\ \Delta w_{01} & \Delta w_{11} \\ \Delta w_{02} & \Delta w_{12} \\ \Delta w_{03} & \Delta w_{13} \end{pmatrix}, \tag{31}$$

and then have the library do an operation like

$$\begin{pmatrix} w_{00} & w_{10} \\ w_{01} & w_{11} \\ w_{02} & w_{12} \\ w_{03} & w_{13} \end{pmatrix} + \begin{pmatrix} \Delta w_{00} & \Delta w_{10} \\ \Delta w_{01} & \Delta w_{11} \\ \Delta w_{02} & \Delta w_{12} \\ \Delta w_{03} & \Delta w_{13} \end{pmatrix}, \tag{32}$$

to correct the weights. Since we know a weight's correction is $\Delta w_{jk} = -\eta \delta_k a_j$, the Δw matrix can be constructed by indexing through values of j and k (here j is the column and k is the row).

Note: In Section 2.2, in the description of w_{xy}, we noted that the $w_{from-layer \to to-layer}$ notation might not have been the best choice, despite its natural fit in this work when referring to the weights between layers. Here is the reason why: in the standard notation of matrix algebra, when one writes something like w_{jk}, the row always comes first, then the column. So in this notation j should be the row and k the column. This isn't the case in this work. So for example in the matrix above, where you see the element w_{01}, that should read w_{10} since it's on row 1, column 0.

$$\begin{pmatrix} \delta_0 a_0 & \delta_0 a_1 & \delta_0 a_2 \\ \delta_1 a_0 & \delta_1 a_1 & \delta_1 a_2 \\ \delta_2 a_0 & \delta_2 a_1 & \delta_2 a_2 \end{pmatrix}. \tag{33}$$

You can see the δ values are ordered vertically in each column, and the a values are ordered horizontally in each row [11]. As tempting as it may seem with this matrix, there's no dot product to be taken here. Rather it's just a matrix construction we need to do, again while trying to avoid using any of our own for-loops. We'll have a matrix library make two matrices, one that looks like this

$$\begin{pmatrix} \delta_0 & \delta_0 & \delta_0 \\ \delta_1 & \delta_1 & \delta_1 \\ \delta_2 & \delta_2 & \delta_2 \end{pmatrix}, \tag{34}$$

and the other like this

$$\begin{pmatrix} a_0 & a_1 & a_2 \\ a_0 & a_1 & a_2 \\ a_0 & a_1 & a_2 \end{pmatrix}. \tag{35}$$

Then, the matrix library construct a third matrix which is the element-by-element product of the two matrices, which will result in the matrix of Equation 33 above.

Numpy

Numpy allows for manipulation and creation of the matrices as discussed above. First, the function `np.broadcast_to` will take the activation vector (i.e. (a_0, a_1, a_2, a_3)) and "broadcast it" into a new shape. Here, we'd like to shape the activation vector into a matrix that has a number of rows equal to the number of neurons in the forward layer, and columns equal to the length of the activation vector itself.

As an example, if an activation vector is `as_vector=[0.5, 0.2, 0.3, 0.1]`, then

```
np.broadcast_to(as_vector,(5,len(as_vector)))
```

will result in

```
[[0.5 0.2 0.3 0.1]
 [0.5 0.2 0.3 0.1]
 [0.5 0.2 0.3 0.1]
 [0.5 0.2 0.3 0.1]
 [0.5 0.2 0.3 0.1]].
```

In other words, 5 rows of the activation vector. So we can make a matrix out of the activation vector by doing

```
mat_activations = np.broadcast_to(as_vector,(len(deltas[ahead]),len(as_vector)))
```

Here `len(deltas[ahead])` is the number of neurons (and δ values) in the forward layer. This will create `mat_activations` as a matrix with the activation values all lined up in the rows of a matrix, as in Equation 35.

The delta values are handled similarly. First, we take our δ-vector and reshape it into a column array using

```
ds_vector = deltas[ahead].reshape((len(deltas[ahead]),1))
```

Next, this is broadcast into a matrix using

```
mat_deltas = np.broadcast_to(ds_vector,(len(deltas[ahead]),len(as_vector)))
```

The `len(as_vector)` is used to ensure the δ matrix has as many columns as the activity matrix. For example, if our delta vector was `delta = np.array([1,2,3,4])`, then `delta = np.reshape(delta,(len(delta),1))` would give

```
[[1]
 [ 2]
 [ 3]
 [ 4]]
```

with the `np.broadcast_to` line making this into

```
[[1, 1, 1, 1],
 [2, 2, 2, 2],
 [3, 3, 3, 3],
 [4, 4, 4, 4]]
```

as needed. With the activation values in rows, and the δ values in columns, the Δw matrix can be found from `np.multiply(mat_activations,mat_deltas)`. So, we've constructed the needed Δw matrix using only the fast routines in the matrix library, and without any of our own for-loops.

4 Derivation of all back-propagation formulas

The total loss of the network, L_T is defined as

$$L_T = \frac{1}{2}\sum_n = (a_n - y_n)^2, \tag{36}$$

where the n is summed over the k neurons in the output layer. It would have been more natural just to use k instead of n for in Equation 36, but this can lead to confusion later on (which we will note), in deriving the equations for backpropagation.

4.1 Correcting weights feeding the output layer

We begin with a left-to-right look at the network's architecture. To begin we'll look at how the weights to the left of the output layer would need to be adjusted to nudge all a_k values closer to their corresponding y_k values. For this, we'd like to know how the w_{jk} weights affect L_T or

$$\frac{\partial L_T}{\partial w_{jk}}, \tag{37}$$

where w_{jk} are the weights connecting neurons in the leftward layer j to neurons in the rightward layer k. Recall that the output layer is special, since its required outputs in the values of y_n are known. Proceeding, we get that

$$\frac{\partial L_T}{\partial w_{jk}} = \sum_n (a_n - y_n) \frac{\partial a_n}{\partial w_{jk}}. \tag{38}$$

Finding $\partial a_n / \partial w_{jk}$ requires some careful consideration, and is related to our choosing of n for the index to the sum (i.e. not k). Here n will run over all neurons in the output layer k, while j refers to a specific neuron in the preceding layer and k a specific neuron in the output layer.

Upon careful inspection, we see that there is only one neuron in the output layer that has an activation with an explicit dependence on w_{jk}: is it the a_n neuron when $n = k$, or neuron a_k. With this single neuron depending w_{jk}, only one term in the sum will be non-zero since $\partial a_n / \partial w_{jk} = 0$ for all $n \neq k$. All a_n terms with $n \neq k$ do not have any dependence on w_{jk}, so will all have derivatives with respect to w_{jk} of zero. Thus, Equation 38 becomes

$$\frac{\partial L_T}{\partial w_{jk}} = (a_k - y_k) \frac{\partial a_k}{\partial w_{jk}}. \tag{39}$$

Finding $\partial a_k / \partial w_{jk}$ is fairly straightforward since $a_k = f(z_k)$ and $z_k = \sum_n w_{nk} a_n + b_k$, so

$$\frac{\partial a_k}{\partial w_{jk}} = \frac{\partial a_k}{\partial z_k} \frac{\partial z_k}{\partial w_{jk}}. \tag{40}$$

First, we work on $\partial a_k / \partial z_k$, again noting tht $a_k = f(z_k)$, with $f(x)$ the activation function in use. In general f will be a function of one variable, like x, so $f = f(x)$. For the sigmoid function as an example, $f(x) = 1/(1 + e^{-x})$. For ReLU, $f(x) = \max(0, x)$. So, $f'(z_k)$ means to differentiate the activation function with respect to x and then evaluate the derivative at $x = z_k$. Since $a_k = f(z_k)$ we get

$$\frac{\partial a_k}{\partial z_k} = \left. \frac{\partial f}{\partial x} \right|_{x = z_k} = f'(z_k) \tag{41}$$

Next,

$$\frac{\partial z_k}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \sum_n w_{nk} a_n + b_k. \tag{42}$$

which becomes

$$\frac{\partial z_k}{\partial w_{jk}} = \sum_n \frac{\partial w_{nk}}{\partial w_{jk}} a_n + w_{nk} \frac{\partial a_n}{\partial w_{jk}} + \frac{\partial b_k}{\partial w_{jk}}. \tag{43}$$

Here we must examine all three terms in the sum, remembering that n is summing over the j neurons. First the term that has a w_{jk} in it to differentiate

- First term: All derivatives of w_{nk} with respect to w_{jk} are zero unless $n = j$, so all of these terms will be zero except for $\partial w_{jk}/\partial w_{jk}$ which is 1. So this term becomes a_j. Weights are independent parameters, and one will not depend on another, hence all of the zero derivatives (unless the weight is differentiated with respect to itself).
- Second term: No activation a_n in the j layer will depend on a weight w_{jk}, since the a_j neurons all feed into the w_{jk} weights. So this term is zero for all n.
- Third term: No bias depends on any weight so this term is zero.

Thus Equation 43 becomes

$$\frac{\partial z_k}{\partial w_{jk}} = a_j. \tag{44}$$

Collecting these results, we arrive at

$$\frac{\partial L_T}{\partial w_{jk}} = (a_k - y_k) f'(z_k) a_j. \tag{45}$$

This is our first equation for back-propagation and we note it's left-to-right nature: It tells us how any changing any weight to the left of the output layer, w_{jk} will affect L_T, which is computed from the rightward (more forward) output layer.

As per standard treatment of back-propagation, we can define δ_k to be

$$\delta_k = f'(z_k)(a_k - y_k), \tag{46}$$

so Equation 45 becomes

$$\frac{\partial L_T}{\partial w_{jk}} = \delta_k a_j, \tag{47}$$

which is the same as Equation BP4 in Ref.[1].

One last reminder of the notation: the correction to weight w_{jk}, which is the weight connecting neuron j (in layer j) to neuron k (in layer k) is the δ_k of the kth neuron (in layer k) times the activation of neuron a_j (in layer j). Thishas some intuitive meaning.

The effect of a weight on the loss function or $\partial L_T/\partial w_{jk}$ is the error contributed by the forward layer (δ_k) times the strength of the neuron that feeds it (a_j). So the weight-on-the-loss dependency is related to the ultimate error in the forward layer times the strength of the firing of the neuron that (partly) contributed to the error. Kind of makes sense on two fronts: 1) a small error will mean $\partial L_T/\partial w_{jk}$ is small, and why shouldn't it be? If the error is small, this part of the network's contribution to affecting the loss should. be small. 2) if the neuron firing is small, again its contribution to changing the loss should also be small.

4.1.1 A short numerical example

A tutorial on PyTorch [9] has a nice example that shows the meaning of $\partial L_T/\partial w_{jk} = \delta_k a_j$, and how it works. Let's take a look at it, as it will illustrate the meaning of Equation 47. You can find the original code at Ref. [10].

```
import numpy as np
import math

x = np.linspace(-math.pi, math.pi, 2000)
y = np.sin(x)

a = np.random.randn()
b = np.random.randn()
c = np.random.randn()
d = np.random.randn()

learning_rate = 1e-6
for t in range(2000):
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    loss = np.square(y_pred - y).sum()

    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()

    # Update weights
```

```
26      a -= learning_rate * grad_a
27      b -= learning_rate * grad_b
28      c -= learning_rate * grad_c
29      d -= learning_rate * grad_d
30
31  print(f'Result: y = {a} + {b} x + {c} x^2 + {d} x^3')
```

The goal of this code is to use back propagation to estimate the coefficients of the polynomial $y = a + bx + cx^2 + dx^3$ that would make it approximate $\sin(x)$. The coefficients here are like the weights in a neural network. Here's how the code works:

- The definitions of `x` and `y` set up x and y coordinate space as in $y(x) = \sin(x)$ for $-\pi \leq x \leq \pi$.
- The lines for `a=...`, `b=...`, `c=...` and `d=...` initialize all of the coefficients in the polynomial to (normally distributed) random values.
- Now, a "back propagation loop" is started. The predicted y is computed in `y_pred`, and the loss, which is a measure of how far off the prediction is versus the sin data (i.e. `y`). The loss function is $L = \sum_n (y_{pred(n)} - y_n)^2$.
- Now, the gradients are computed, which are the derivative of the loss function with respect to each coefficient, so the code will compute $\partial L/\partial a$, $\partial L/\partial b$, $\partial L/\partial c$, and $\partial L/\partial d$. Let's run through these.

  1. $\partial L/\partial a$:

  $$\frac{\partial L}{\partial a} = 2\sum_n (y_{pred} - y) \cdot \frac{\partial y_{pred}}{\partial a}. \tag{48}$$

  Now, $\partial y_{pred}/\partial a = 1$, so $\partial L/\partial a = 2\sum_n (y_{pred} - y)$, which all goes into the assignment of `grad_a` in the code.

  2. $\partial L/\partial b$:

  $$\frac{\partial L}{\partial b} = 2\sum_n (y_{pred} - y) \cdot \frac{\partial y_{pred}}{\partial b}. \tag{49}$$

  which is $\partial L/\partial b = 2\sum_n (y_{pred} - y) \cdot x$ (noting that $\partial y_{pred}/\partial b = x$).

  3. The others follow similarly with $\partial L/\partial c = 2\sum_n (y_{pred} - y) \cdot x^2$ and $\partial L/\partial d = 2\sum_n (y_{pred} - y) \cdot x^3$.

  You can see all of these defined in the code for `grad_a=...`, `grad_b=...`, `grad_c=...`, and `grad_d=...`.

- Lastly, all of the coefficients are updated using the general form that Δ`coeff=learning_rate`$\times$`gradients`. Note that the coefficient correction formulas all contain a product of two items: 1) a term reflecting the loss function's behavior and 2) the "strength" of the variable the weight is associated with. Equation 47 has the same elements: δ_k is related to the loss function's behavior and a_j, related to the strength of the neuron firing into the weight.

When the code above is run, it arrives at $y(x) = 0.0237 + 0.8476x + -0.004x^2 - 0.0920x^3$, which is shown compared to $\sin(x)$ in Figure 6. The process did a pretty good job.

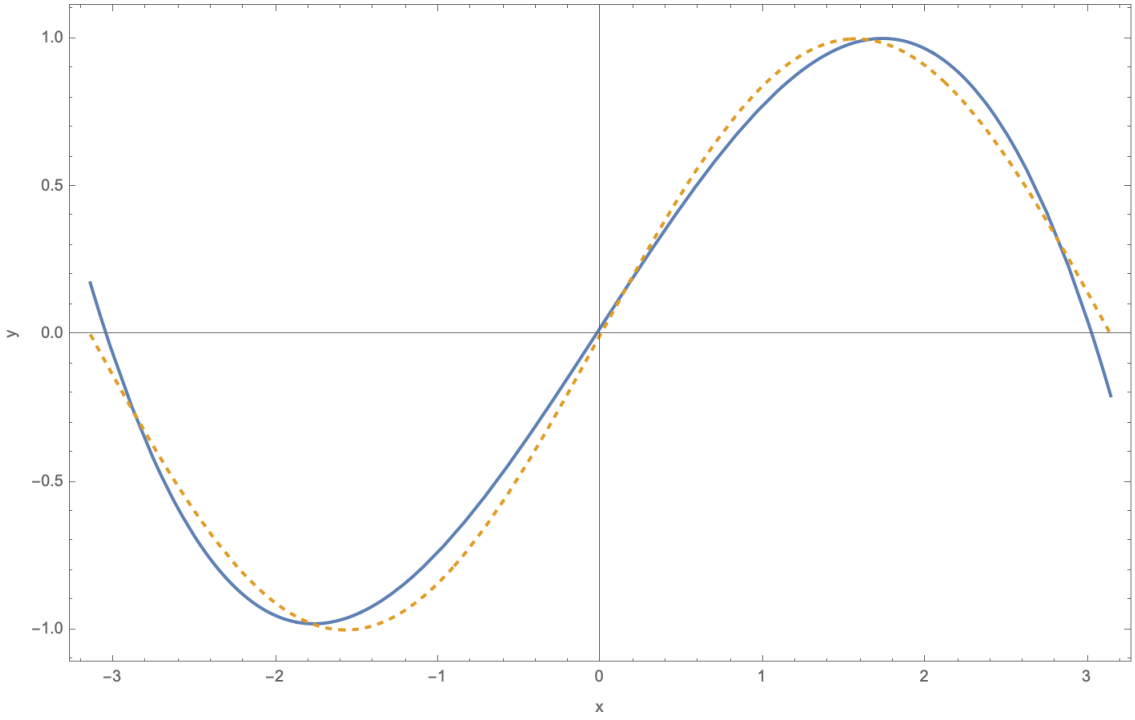

Figure 6: $\sin(x)$ (dotted) vs $y(x)$ (solid) approximated from the gradient descent code above.

4.2 Correcting weights feeding a hidden layer

The challenge with correcting the weights feeding a hidden layer is that the target output of a hidden layer is not known. Recall for an output layer, y_n is known so the error for the output layer $a_n - y_n$ was available. Here we see how to correct the weights w_{ij} for a hidden later, whose error is not known.

As above, we rely on only on calculus and begin by differentiating the loss function with respect to w_{ij}, which are weights feeding the hidden layer as in

$$\frac{\partial L_T}{\partial w_{ij}} = \sum_n (a_n - y_n) \frac{\partial a_n}{\partial w_{ij}}. \tag{50}$$

Recall that a_n is an output layer neuron activation, and there is no explicit dependence of a_n on w_{ij}, as these weights to not directly connect to any of the a_n output neurons. We know there is an *implicit dependence* though, owing to the hyperconnected nature of the neural network; certainly data flow through a hidden layer affects the output of the network.

So, the derivative exists (and is not necessarily zero), but arriving at a proper expression for its derivative will require some careful work. We will rely on the rules of calculus to carry us through. Lastly, the $\sum_n$ persists at the moment, since we cannot state any conditions that make $\partial a_n / \partial w_{ij} = 0$, as a function of n (as we did in the previous section).

Proceeding, we know that $a_n = f(z_n)$ and $z_n = \sum_m w_{mn} a_m + b_n$, where m is summed over the j neurons in the first hidden-layer to the left of the output layer. We again need to use a neutral index like m to avoid confusion with the ij use in the derivative $\partial a_n / \partial w_{ij}$. With this

$$a_n = f\left(\sum_m w_{mn} a_m + b_n \right), \tag{51}$$

which shows the subtle dependence of a_n on w_{ij}: a_n clearly depends on a_m, and a_m depends on the weights to the left of it, which are the w_{ij} weights (again, m is summed over the j neurons). So, we see a_n depends on a_m and a_m depends on w_{ij}, thus a_n depends on w_{ij} (just not with the explicit simplicity of a_n on w_{jk} as in the section above). Looking at Figure 1, you can see from the connectedness of the network that the output neurons a_n would indeed depend on the w_{ij} weights.

Since a_n is a composite function of the activation function evaluated at z_n as in $a_n = f(z_n)$, we apply the chain rule to get

$$\frac{\partial a_n}{\partial w_{ij}} = \frac{\partial a_n}{\partial z_n} \frac{\partial z_n}{\partial w_{ij}}. \tag{52}$$

Now,

$$\frac{\partial a_n}{\partial z_n} = \left. \frac{\partial f}{\partial x} \right|_{x = z_n} = f'(z_n) \tag{53}$$

and using the product rule to allow subtle interdependencies between variables to be handled properly,

$$\frac{\partial z_n}{\partial w_{ij}} = \sum_m \frac{\partial w_{mn}}{\partial w_{ij}} a_m + w_{mn} \frac{\partial a_m}{\partial w_{ij}} + \frac{\partial b_n}{\partial w_{ij}}. \tag{54}$$

Two simplifications are seen. First, weights in different layers are all independent parameters, all $\partial w_{mn}/\partial w_{ij} = 0$ ($w_{mn} \sim w_{jk}$). Second, any bias is independent of any weight, so $\partial b_n/\partial w_{ij} = 0$. Thus, the above reduces to

$$\frac{\partial z_n}{\partial w_{ij}} = \sum_m w_{mn} \frac{\partial a_m}{\partial w_{ij}}. \tag{55}$$

Now, only term in the sum that has a non-zero derivative is when $m = j$, since of all of the ij weights, the jth neuron (a_j) is the only one fed by the one weight w_{ij}. Thus

$$\frac{\partial z_n}{\partial w_{ij}} = w_{jn} \frac{\partial a_j}{w_{ij}}. \tag{56}$$

This puts us in a good position, since we're left with just $\partial a_j/\partial w_{ij}$. We know that the a_j neurons have an explicit dependence on w_{ij}, as these are the weights that feed the a_j neurons directly, so the derivative must have an analytic answer.

As before, we know that $a_j = f(z_j)$ with $z_j = \sum_n w_{nj} a_n + b_j$ (with n summed over the i neurons)

$$\frac{\partial a_j}{\partial w_{ij}} = \frac{\partial a_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}} \tag{57}$$

and

$$\frac{\partial a_j}{\partial z_j} = \left. \frac{\partial f}{\partial x} \right|_{x=z_j} = f'(z_j). \tag{58}$$

Lastly,

$$\frac{\partial z_j}{w_{ij}} = \sum_n \frac{\partial w_{nj}}{\partial w_{ij}} a_n + w_{nj} \frac{\partial a_n}{\partial w_{ij}} + \frac{\partial b_j}{\partial w_{ij}}. \tag{59}$$

We know $\partial b_j/\partial w_{ij}$ is zero since no biases depend on any weights, but we are still left with two non-obvious derivatives, $\partial w_{nj}/\partial w_{ij}$ and $\partial a_n/\partial w_{ij}$.

First, we see that $\partial w_{nj}/\partial w_{ij} = 1$ if $n = i$ and 0 otherwise, since two different weights are independent of one another.

Second, at $n = i$, $\partial a_i/\partial w_{ij} = 0$ since the the activation of a neuron does not depend on the weights that it feeds forward into. So Equation 59 becomes

$$\frac{\partial z_j}{w_{ij}} = a_i. \tag{60}$$

With these results, Equation 57 becomes

$$\frac{\partial a_j}{\partial w_{ij}} = f'(z_j)a_i, \tag{61}$$

allowing Equation 56 to become

$$\frac{\partial z_n}{\partial w_{ij}} = w_{jn}f'(z_j)a_i, \tag{62}$$

and Equation 52

$$\frac{\partial a_n}{\partial w_{ij}} = f'(z_n)w_{jn}f'(z_j)a_i. \tag{63}$$

Plugging this into Equation 50, we get

$$\frac{\partial L_T}{\partial w_{ij}} = \sum_n (a_n - y_n)f'(z_n)w_{jn}f'(z_j)a_i. \tag{64}$$

With possible confusion over the sum indices behind us, we can put $n \to k$ into this equation to get

$$\frac{\partial L_T}{\partial w_{ij}} = \sum_k (a_k - y_k)f'(z_k)w_{jk}f'(z_j)a_i. \tag{65}$$

Using the definition of δ_k in Equation 46 this becomes

$$\frac{\partial L_T}{\partial w_{ij}} = \sum_k \delta_k w_{jk}f'(z_j)a_i. \tag{66}$$

Additionally, to see a form consistent with Equation 47, we define δ_j as

$$\delta_j = f'(z_j)\sum_k \delta_k w_{jk}, \tag{67}$$

so $\partial L_T/\partial w_{ij}$ becomes

$$\frac{\partial L_T}{\partial w_{ij}} = \delta_j a_i. \tag{68}$$

This result agrees with the equation in Section 4.5 of Ref. [2].

4.2.1 One more time

It appears that a pattern is emerging on the effect of weights on the network loss: $\partial L/\partial w_{xy} = \delta_y a_x$. Note again, w_{xy} is the weight connecting neuron x (in layer x) to neuron y (in layer y), with y always being the layer that x feeds into (in the forward data-flow sense of the network). The reversal of the indices across the equal sign means we need δ of the forward (or rightward) layer and the activation of the backward (or leftward) layer.

In Figure 1, we'll imagine one more layer h to the left of layer i, and consider $\partial L/\partial w_{hi}$. Without going through the steps, which are all have similar considerations to the above, we arrived at

$$\frac{\partial L_T}{\partial w_{hi}} = \sum_k (a_k - y_k) f'(z_k) \sum_j w_{jk} f'(z_j) w_{ij} f'(z_i) a_h. \tag{69}$$

Let's rearrange this some to get

$$\frac{\partial L_T}{\partial w_{hi}} = \sum_j w_{ij} f'(z_j) \sum_k (a_k - y_k) f'(z_k) w_{jk} f'(z_i) a_h. \tag{70}$$

Remembering that $\delta_k = f'(z_k)(a_k - y_k)$ we get

$$\frac{\partial L_T}{\partial w_{hi}} = \sum_j w_{ij} f'(z_j) \sum_k \delta_k w_{jk} f'(z_i) a_h. \tag{71}$$

Also using that $\delta_j = f'(z_j) \sum_k \delta_k w_{jk}$ we get

$$\frac{\partial L_T}{\partial w_{hi}} = \sum_j w_{ij} f'(z_i) \delta_j a_h. \tag{72}$$

Doing some rearranging

$$\frac{\partial L_T}{\partial w_{hi}} = f'(z_i) \sum_j w_{ij} \delta_j a_h, \tag{73}$$

we can now define

$$\delta_i = f'(z_i) \sum_j \delta_j w_{ij} \tag{74}$$

for which we get

$$\frac{\partial L_T}{\partial w_{hi}} = \delta_i a_h \tag{75}$$

Once again, we see a pattern like $\partial L_T/\partial w_{xy} = \delta_y a_x$. This tell us the following about back-propagation:

- To correct the weights connecting two layers, with layer x feeding layer y, you need δ_y from the forward layer and the activations from the backward (x) layer.
- The δ_x parameters all depend on one another, all the way "up" to the original delta that comes from a direct comparison of the output of the network to the needed target for the applied input.
- An unknown δ_x (for neuron x in layer x) that is a layer back from a known δ_{x+1} is found by propagating (summing) the known δ_{x+1} values backward through all weights connecting the known δ_{x+1} and unknown δ_x values. This will be multiplied by the derivative of the activation function evaluated at z_x for backward neuron x to arrive at the value for δ_x.
- It appears that $\partial L_T/\partial w_{xy} = \delta_y a_x$ depends on the forward feed strength of a neuron (its activation, a_x) and the amount of error this caused in the forward layer it feeds (δ_y). To confirm the pattern, let's repeat the derivation one more time.

4.3 Correcting neuron biases

Next, we examine how a the bias of a neuron affects the network loss. Here we still define the total loss, L_T as

$$L_T = \frac{1}{2}\sum_n (a_n - y_n)^2, \tag{76}$$

and would like to know

$$\frac{\partial L_T}{b_n} = \sum (a_n - y_n)\frac{\partial a_n}{\partial b_n}, \tag{77}$$

to see how b_n affects L_T. That is, how the bias of an output neuron, b_n (for neuron a_n) affects L_T. The derivative here is $\partial a_n/\partial b_n$, which is

$$\frac{\partial a_n}{\partial b_n} = \frac{\partial a_n}{\partial z_n}\frac{\partial z_n}{\partial b_n}. \tag{78}$$

Knowing that $z_n = \sum_j w_{jn}a_j + b_n$, we see that $\partial z_n/\partial b_n = 1$ and as seen above $\partial a_n/\partial z_n = f'(z_n)$, thus

$$\frac{\partial L_T}{b_n} = \sum (a_n - y_n) f'(z_n). \tag{79}$$

Now, allowing $n \to k$, we get

$$\frac{\partial L_T}{b_k} = \sum (a_k - y_k) f'(z_k), \tag{80}$$

which is

$$\frac{\partial L_T}{b_k} = \delta_k, \tag{81}$$

as defined in Equation 46. So it appears as if a bias alters the loss directly by the magnitude of δ of a neuron itself.

4.3.1 Another look

Let's look one layer back to confirm this emerging pattern on how loss depends on δ of a neuron.

$$\frac{\partial L_T}{\partial b_j} = \sum_n (a_n - y_n) \frac{\partial a_n}{\partial b_j}. \tag{82}$$

As before

$$\frac{\partial a_n}{\partial b_j} = \frac{\partial a_n}{\partial z_n} \frac{\partial z_n}{\partial b_j}. \tag{83}$$

We've seen that $\partial a_n / \partial z_n = f'(z_n)$. Next, knowing that $z_n = \sum_m w_{mn} a_m + b_n$, we write that

$$\frac{\partial z_n}{\partial b_j} = \sum_m \frac{\partial w_{mn}}{\partial b_j} a_m + w_{mn} \frac{\partial a_m}{\partial b_j} + \frac{\partial b_n}{\partial b_j}. \tag{84}$$

Now, $\partial b_n / \partial b_j = 0$ since no two neuron biases depend on one another. Also, $\partial w_{mn} / \partial b_j = 0$ since no weight depends on any bias. So we are left with

$$\frac{\partial z_n}{\partial b_j} = \sum_m w_{mn} \frac{\partial a_m}{\partial b_j}. \tag{85}$$

Here we see that only $m = j$ has a non-zero $\partial a_m / \partial b_j$, since the activation of a neuron can only depend on its own bias. Thus, we get that

$$\frac{\partial z_n}{\partial b_j} = w_{jn} \frac{\partial a_j}{\partial b_j}. \tag{86}$$

Thus Equation 83 can assembled to be

$$\frac{\partial a_n}{\partial b_j} = f'(z_n) w_{jn} \frac{\partial a_j}{\partial b_j}. \tag{87}$$

And, using Equation 83 with $n = j$, we get

$$\frac{\partial a_j}{\partial b_j} = \frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial b_j}. \tag{88}$$

Here, $\partial a_j/\partial z_j = f'(z_j)$ and $\partial z_j/\partial b_j = 1$

$$\frac{\partial a_j}{\partial b_j} = f'(z_j). \tag{89}$$

Since $z_j = \sum_m w_{mj}a_m + b_j$ and since m is summed over the i neurons, neither w_{mj} or a_m (or a_i) depend on b_j. These can be put into Equation 83 to get

$$\frac{\partial a_n}{\partial b_j} = f'(z_n)f'(z_j). \tag{90}$$

Finally, Equation 82 now becomes

$$\frac{\partial L_T}{\partial b_j} = \sum_n (a_n - y_n)f'(z_n)f'(z_j). \tag{91}$$

Letting $n \to k$, we get

$$\frac{\partial L_T}{\partial b_j} = \sum_k (a_k - y_k)f'(z_k)f'(z_j), \tag{92}$$

which is

$$\frac{\partial L_T}{\partial b_j} = f'(z_j)\sum_k (a_k - y_k)f'(z_k), \tag{93}$$

or

$$\frac{\partial L_T}{\partial b_j} = \delta_j. \tag{94}$$

So indeed, the pattern holds: the dependence of the network loss on a neuron bias is the value of δ of that neuron.

References

[1] "Neural Networks and Deep Learning" by Michael Nielsen, `http://neuralnetworksanddeeplearning.com`.

[2] `https://www.cs.swarthmore.edu/~meeden/cs81/s10/BackPropDeriv.pdf`

[3] `https://dustinstansbury.github.io/theclevermachine/derivation-backpropagation`

[4] http://www.cs.put.poznan.pl/pliskowski/pub/teaching/eio/lab1/eio-supplementary.pdf

[5] https://www.cs.cornell.edu/courses/cs5740/2016sp/resources/backprop.pdf

[6] http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

[7] It's not just something about our approach and needing to store the neuron activations. Listen carefully here for a few moments https://youtu.be/oBkllt KXtDE?si=GadK8tAN98_aUT2M&t=386, where the speaker discusses needing to hold "4096 images and all of their activations in a single forward pass."

[8] This is also similar to PyTorches "history tracking" during a forward pass. See https://youtu.be/IC0_FRiX-sw?si=xKTLav3p9U3hQY1k&t=567.

[9] See https://www.pytorch.org.

[10] See https://pytorch.org/tutorials/beginner/pytorch_with_examples.html.

[11] See https://youtu.be/q8SA3rM6ckI?si=moD3N8lvQPXTKlnw&t=1406 for something similar.