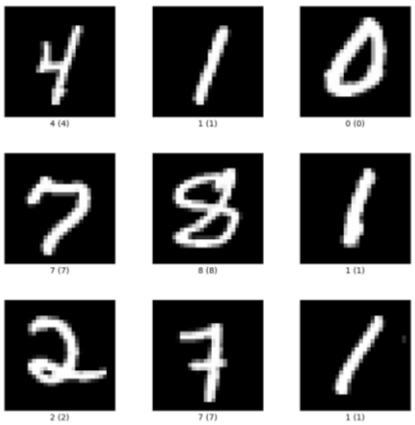
Image Recognition – A Neural Network From Scratch

This project is basically just coding a neural network from scratch, that can read an image. Specifically, a 28x28 pixel greyscale image, of a handwritten digit, like the ones shown below:

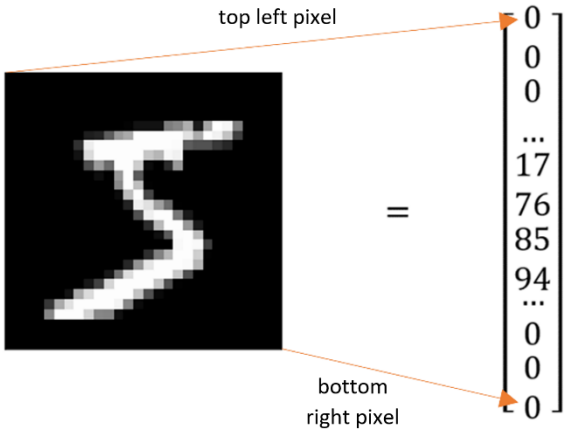


Note: The images that the neural network will be trained off of, come from the MNIST dataset.

So, firstly, how can a neural network even "read", one of these images?

Well, each one of these images, is just 784 pixels, and each one of these pixels, is just a number, ranging from 0 to 255 (how white it is). So an image is just 784 separate numbers.

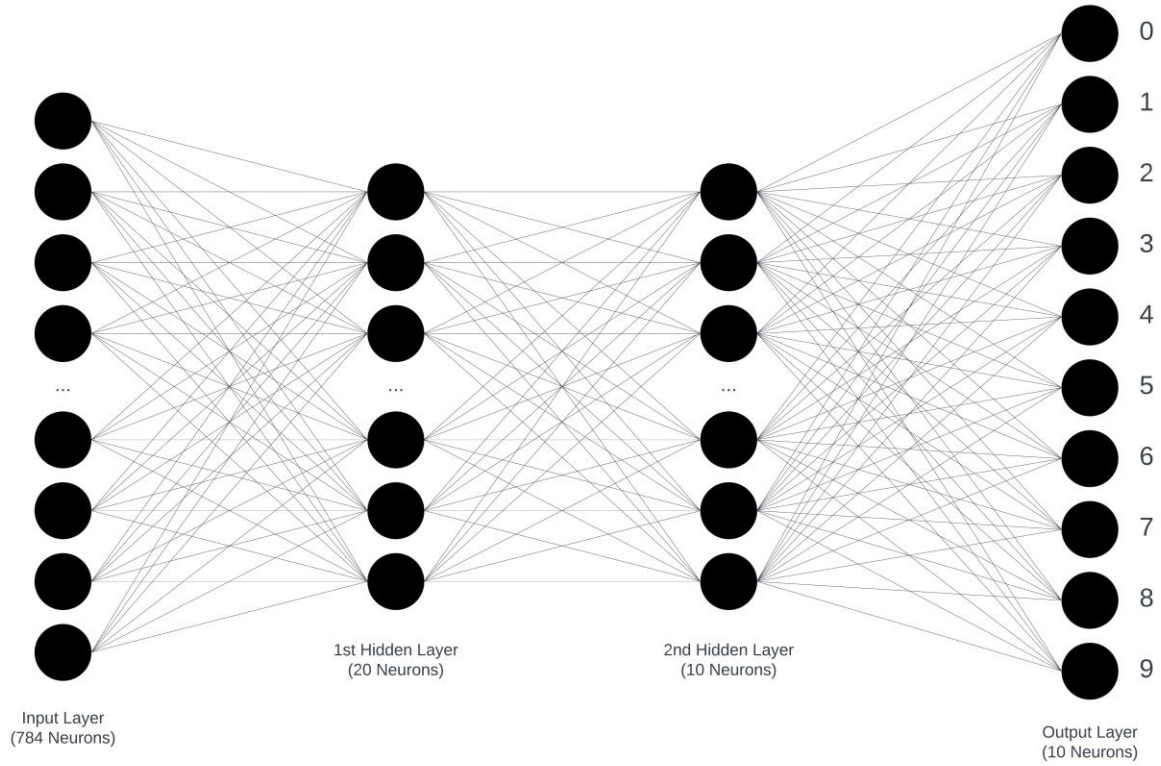
We can put these 784 separate numbers into a column vector, and thus we've turned an IMAGE, into a VECTOR. So it looks like this:



Note: The numbers in the vector, represent the pixels in the image, in the order of top left, to bottom right (same order as when reading a book).

So, this image/vector, will be what we INPUT into the neural network, and then the neural network will do some calculations, and then OUTPUT what digit it sees.

Sounds good, but what does the neural network actually look like? Well, it's just layers of "neurons", connected via "weights", and it'll ultimately end up looking like this:



There's quite a lot going on here, so we'll go through it bit by bit, starting with the first layer (the input layer).

The input layer is just what's being inputted into the neural network. So in our specific case, we're inputting into our network, an image (AKA a vector with 784 numbers), so our input layer would simply be 784 neurons, and each of these neurons would just be a number between 0 and 255, that represents a pixel from the inputted image.

Great! That's one layer down, now, to keep things simple, we'll see if our neural network can have NO middle layers, and instead just the input and output layer.

So, what would the output layer look like?

Well, the output layer is just what's being outputted from the neural network. So in this case, the output would be what the network sees in the image, which could be anything from a 0, to a 9.

Now ideally, the neural network would just be given an image, and then output a SINGLE number, like a "7", or a "2", to tell you what digit it saw in the image.

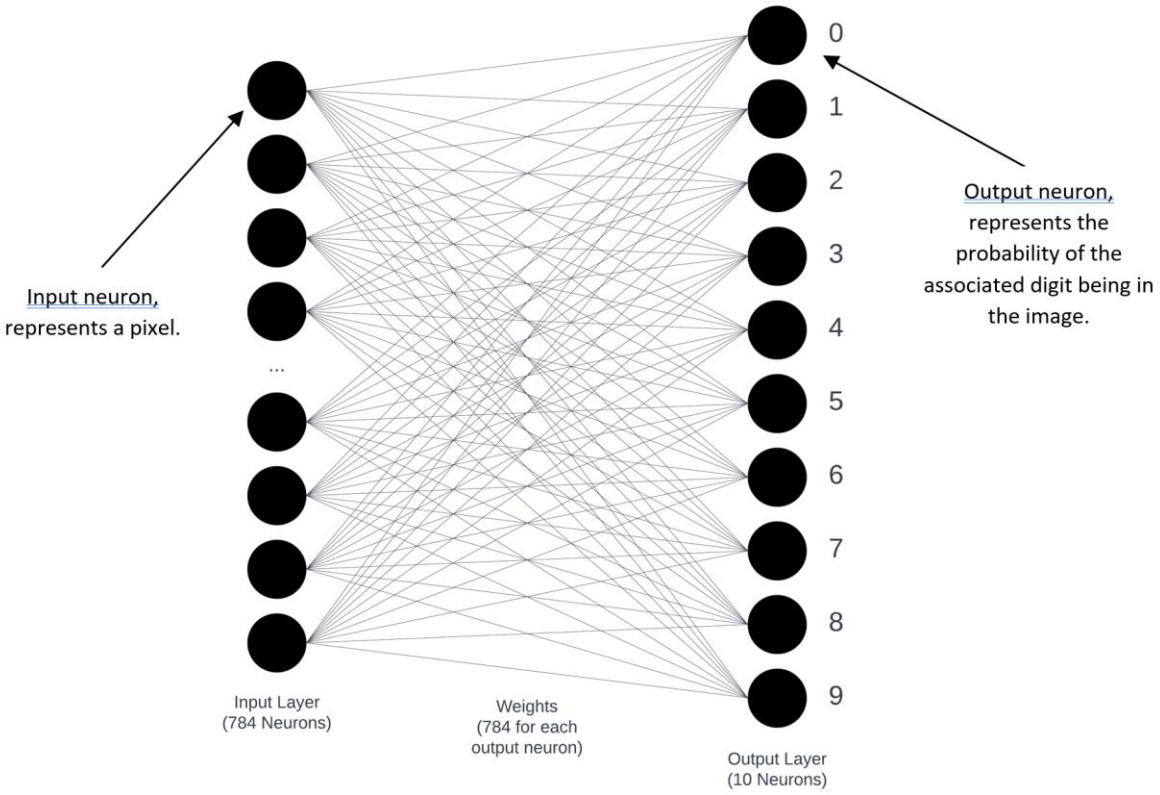
Unfortunately, that approach makes the underlying mathematics noticeably more complicated *(all partial derivatives will contain piecewise functions, due to the derivative of the max function).*

So instead, to make the underlying math simpler, the neural network will tell us what's in an image, by outputting TEN numbers instead of one, and each number is just the probability of a particular digit being in that image. So it'll look like this:

$$Example\_Output \quad = \quad \begin{bmatrix} 0.4 \\ 0.3 \\ 0.7 \\ 0.1 \\ 0.0 \\ 0.2 \\ 0.9 \\ 0.1 \\ 0.5 \\ 0.4 \end{bmatrix} \quad = \quad \begin{bmatrix} Probability\ that\ its\ a\ 0 \\ Probability\ that\ its\ a\ 1 \\ Probability\ that\ its\ a\ 2 \\ Probability\ that\ its\ a\ 3 \\ Probability\ that\ its\ a\ 4 \\ Probability\ that\ its\ a\ 5 \\ Probability\ that\ its\ a\ 6 \\ Probability\ that\ its\ a\ 7 \\ Probability\ that\ its\ a\ 8 \\ Probability\ that\ its\ a\ 9 \end{bmatrix}$$

Now, as we can see in the example answer/output above, 0.9 is the highest probability, representing the digit "6", and thus the network is saying the inputted image, is most likely a 6.

So, now that we know how our neural network provides answers (in the form of 10 separate numbers), we can simply see that our output layer must contain 10 neurons, one neuron for each of these numbers. Thus, we've got this so far:

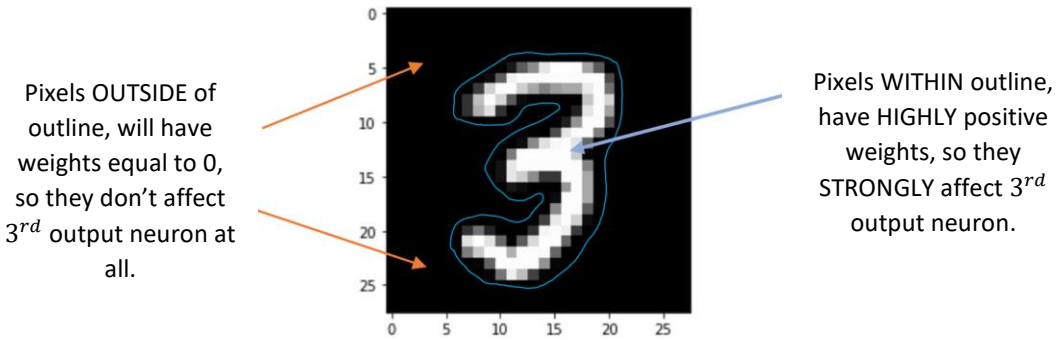


Now, could this simply be our neural network? No middle layers, each output neuron, is just this:

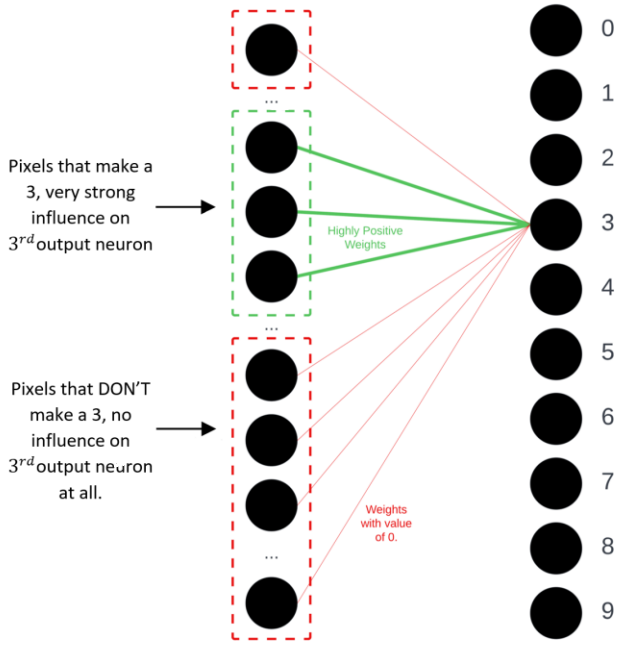
$$OutputNeuron_i = \sigma\left(\left(\sum_{j=1}^{784} Weight_{ij} * InputNeuron_j\right) + Bias_i\right)$$

Well, as we can see, each output neuron, is just the sum of all input neurons, so no information about the image is lost, which is good. Additionally, each output neuron, has weights connected to it, from ALL input neurons.

This seems incredibly useful, because now for example, we could take the output neuron designed to detect "3" (the 4^{th} output neuron in the layer), and set its weights for pixels that make a 3, to highly positive values, and the weights for pixels that DON'T make a 3, to values of zero. So basically this:



So, the weights of the 4^{th} output neuron, will look like this:



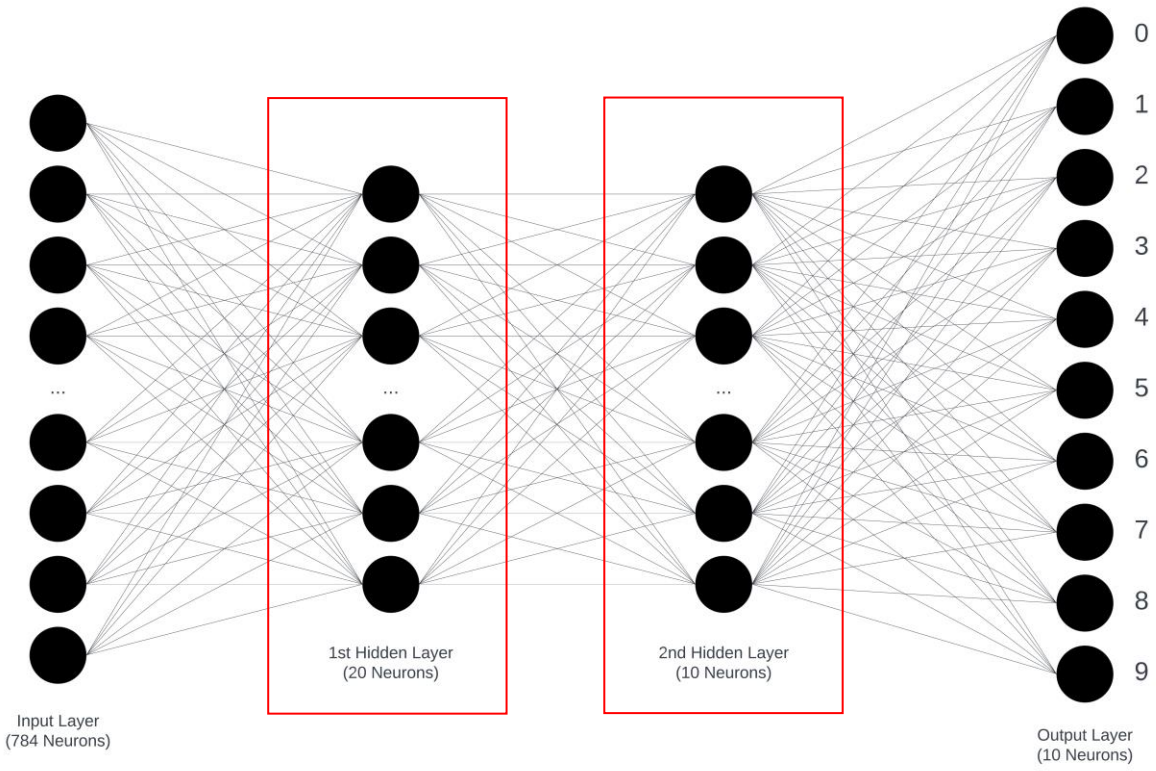
This means, if we give the neural network an image of a 3 (like the one above), then the 4^{th} output neuron will end up becoming a high probability value, because all the pixels that have highly positive weights (so they strongly influence the 4^{th} output neuron), will all be white (have high values), and all the pixels that have weights of zero (so they don't affect the 4^{th} output neuron at all), will all be black (have values equal to 0).

Thus 3 would be detected quite accurately! This seems to work perfectly, right?

Well unfortunately, if we were to give this network say an image of an 8, then all the pixels that are cared about, are still white, and the extra pixels that make up the 8, are ignored (because they have weights equal to 0), thus tricking the neural networking into thinking an 8, is actually a 3. This is true for any digit that overlaps another digit, so a 4 could be seen as a 1, or an 8 could be seen as a 6.

Additionally anyone who writes a 3 that's a little a little oddly shaped or rotated, will cause a lot of the pixels that are cared about (the ones with highly positive weights), to be black, and thus the 3^{rd} output neuron would have a low probability value.

Therefore, the neural network would fail in quite a lot of cases. So unfortunately, just having an input and output layer, isn't enough, we'll need extra layers of neurons, specially the two middle layers, shown earlier:



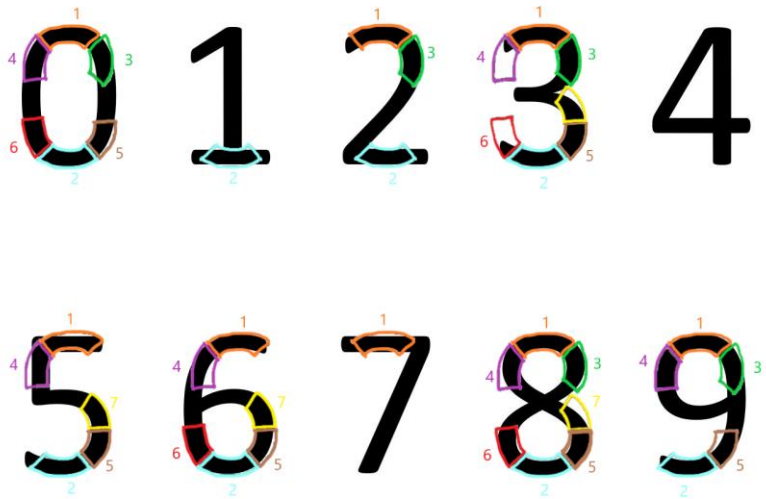
Now, how do these two additional layers (conventionally called "hidden layers"), help?

Well, we'll start with the first hidden layer. As previously mentioned, it's quite easy for our network to misidentify images, like thinking an 8 is actually a 3, due to the extra pixels that separate an 8 from a 3, being ignored.

So, we need to make sure that the extra pixels aren't ignored.

So what we'll do, is create a new layer of neurons, and each neuron in this "1st hidden layer", will represent a small group of pixels, a "chunk", that can be found amongst the digits.

Let's say this new layer had 7 neurons, and thus we have 7 new, common chunks, let's say the ones shown just below:



Note: Chunks that don't fit to digits very well aren't drawn, to decrease visual clutter.

As we can see, we can use these common chunks (marked 1 through to 7), to differentiate the 10 digits from each other, including the 3 and the 8, as the purple and red chunks, are clearly a part of "8, but are barely a part of 3. So, we can just make the purple and red chunks, NEGATIVELY influence the 3^{rd} output neuron (by giving them negative weights).

Thus, this solves the misidentification problem between our 3 and 8!

Unfortunately, these chunks are a little big, and there's only 7 of them, so we can't precisely cover the entirety of all 10 digits, just using these 7 chunks. For example, virtually none of the chunks fit "4" very well at all, and only one chunk fits quite well to "1" and "7". There's also a fair bit of white space when matching some chunks to some digits, like the yellow chunk, to digit "8".

So, if we really wanted to accurately differentiate digits from each other, we would need all of these chunks to be a lot smaller, and we'd need a lot more of them, to precisely cover the entirety of all 10 digits. Unfortunately, more chunks, means more neurons, which means it'll take longer to train/run the neural network, so we'll go with around 20 chunks.

Additionally, if we simply make each of these new, smaller chunks, a bit of a looser fit (the 7 above are quite tight fitting), then they'll also take care of the slightly oddly shaped/rotated issue, that comes with handwritten digits. This is because, if each chunk is tightly fitting (like the 7 above), then a small rotation/odd shaping, of a digit, will cause white pixels to go outside of the tight fitting chunks, but loose fitting chunks would still reasonably catch them.

Note: We can't make the chunks too loosely fitting though, otherwise they'll no longer be identifying distinct pieces of digits.

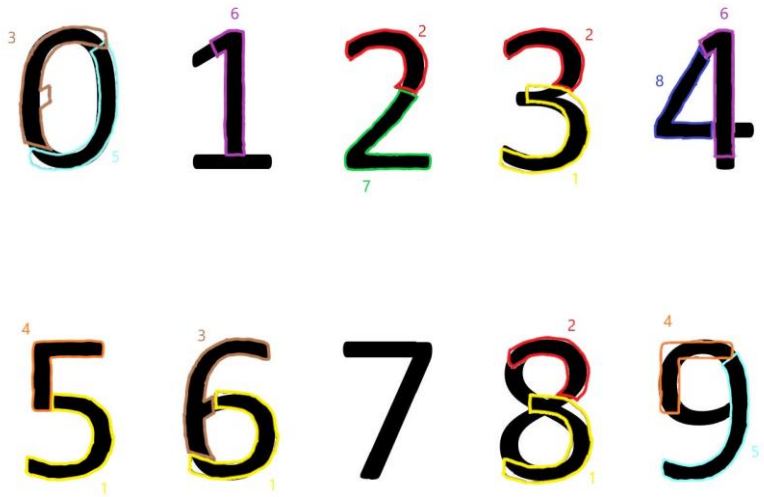
Now that we have our 1st hidden layer, what is the point of the 2nd hidden layer?

Well, having the 1st hidden layer, means we now have lots of loose fitting, very small chunks, but unfortunately it's quite tricky for the neural network to group a lot of them together at once, in just the right way, to get an entire digit, that happens to be the correct one. For us and our human eyes it would be easy, since we can just see which chunks are in the image, and instantly see which digit they combine to make.

The network unfortunately, doesn't have this luxury, in this case, it only has 20 probability values, one for each small chunk, to discern ENTIRE DIGITS from each other. The neural network could still work with just the 1st hidden layer of 20 neurons, but accuracy would definitely quite a hit.

Thus, to further improve accuracy, we make it easier on the neural network, so instead of having to identify entire digits, from just a bunch of small chunks, there'll be a stepping stone, it'll only need to identify "large chunks", using these smaller chunks, such as the top loop of an 8, from the bottom loop of a 6. Intuitively this is like giving it a 5 piece jigsaw puzzle, instead of a 20 piece jigsaw puzzle.

So, what would our "large chunks", look like in general, in the 2nd hidden layer? Well, something like this:



Note: Each large chunk, is made up of multiple smaller chunks. They haven't been drawn, to avoid visual clutter.

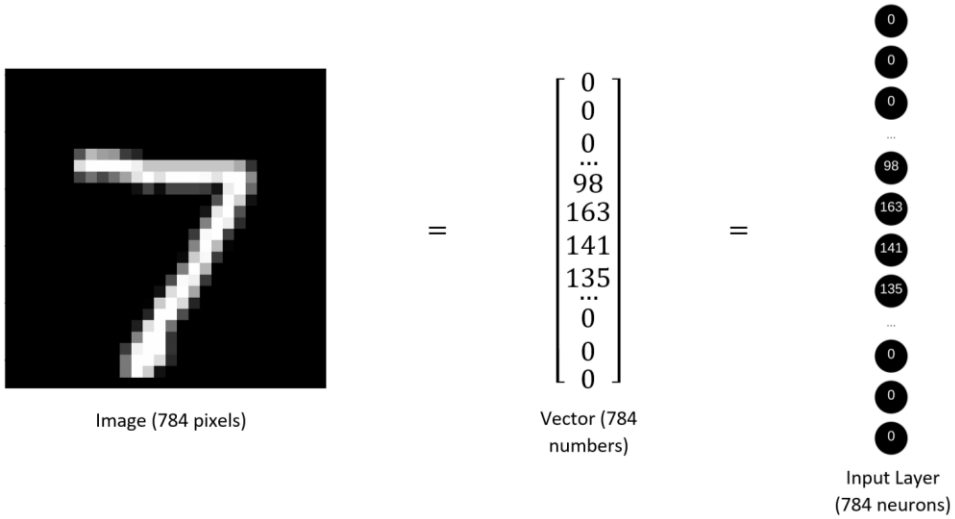
In the example just above, we're able to reasonably cover almost all digits, with only 8 large chunks, so about 10 should do it, which means our 2nd hidden layer will have 10 neurons.

Note: If the 1st hidden layer had more neurons, then so to could the 2nd hidden layer, to improve accuracy, but again we don't want to increase training/runtime too significantly, due to diminishing returns in accuracy gained.

So, there we have it! The structure of our neural network, finalized.

Now that we know what our neural network looks like, and what each layer is responsible for, we can walk through what the network does, when it's actually given an image.

So, first, an image is turned into a vector, that contains 784 numbers, with each number representing a pixel. That vector is then inputted into the neural network, so the 784 neurons in the input layer, become those 784 numbers.



Now, the 1st hidden layer is calculated. Each neuron in the 1st hidden layer, is simply the sum of the previous layers neurons, with their weights, then adding a bias onto that summation, and finally inputting the overall number, into the sigmoid function (to scale it between 0 and 1, the reason why even middle layer neurons need the sigmoid function, will be explained soon).

$$Neuron\ 1st\ Hidden\ Layer_i = \sigma\left(\left(\sum_{j=1}^{784} Weight_{ij} * InputNeuron_j\right) + Bias_i\right)$$

Now, we can vectorize this mathematical expression, to not only make the notation far more compact, but to also make the code of the neural network far more optimized, when we build the network in Python, later on.

We'll use A to represent an input neuron, P to represent a 1st hidden layer neuron, W to represent a weight, and B to represent a bias.

$$A_i = Input\ Neuron_i$$

$$P_i = 1^{st}\ Hidden\ Layer\ Neuron_i$$

$$W_{i,j} = Weight\ ending\ at\ P_i\ from\ N_j$$

$$B_i = Bias\ for\ P_i$$

$$InputLayer = \begin{bmatrix} A_1 \\ A_2 \\ \cdots \\ A_{784} \end{bmatrix}$$

$$Weights = \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} & \cdots & W_{1,784} \\ W_{2,1} & W_{2,2} & W_{2,3} & \cdots & W_{2,784} \\ W_{3,1} & W_{3,2} & W_{3,3} & \cdots & W_{3,784} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ W_{20,1} & W_{20,2} & W_{20,3} & \cdots & W_{20,784} \end{bmatrix}$$

$$Biases = \begin{bmatrix} B_1 \\ B_2 \\ \cdots \\ B_{20} \end{bmatrix}$$

$$HiddenLayer_1 = \sigma(Weights * InputLayer + Biases) = \begin{bmatrix} P_1 \\ P_2 \\ \cdots \\ P_{20} \end{bmatrix}$$

There we have it, the 1st hidden layer, expressed in a quite compact form. Now, as it turns out, ALL layers of neurons, are calculated the exact same way. Any layer, whether it be the final layer, 1st hidden layer, or 100th hidden layer, is simply equal to the previous layer, with weights, then adding on the bias, and finally applying the sigmoid function.

Therefore, we can express any neuron, as just:

$$Neuron_i^l = Neuron\ at\ layer\ l\ position\ i$$

$$Bias_i^l = Bias\ at\ layer\ l\ position\ i$$

$$Weight_{i,j}^l = Weight\ ending\ at\ layer\ l\ position\ i, starting\ at\ position\ j\ of\ previous\ layer$$

$$h = number\ of\ neurons\ in\ previous\ layer$$

$$Neuron_i^l = \sigma\left(\left(\sum_{j=1}^{h} Weight_{i,j}^l * Neuron_j^{l-1}\right) + Bias_i^l\right)$$

The indices are getting pretty messy, thankfully we can vectorize this expression too, which now has the additional benefit of neater indices:

$$A^l = The\ l^{th}\ layer\ of\ neurons$$
$$W^l = The\ weights\ ending\ at\ A^l$$
$$B^l = The\ biases\ of\ A^l$$

$$\boldsymbol{A^l = \sigma(W^l * A^{l-1} + B^l)}$$

There we have it! A way of expressing/calculating any layer we want, in the neural network.

Now, real quick, why do all the middle layers, need the sigmoid function as well? Since the neurons in those layers, don't need to be probability values, like the final/output layer. Well, simply put, if none of the middle layers had the sigmoid function, then the final layer, would simply be a linear combination of the input/initial layer, meaning all the middle layers would be useless. I'll demonstrate this mathematically:

$$A^1 = InputLayer$$
$$A^2 = 1stHiddenLayer = W^2 * A^1 + B^2$$
$$A^3 = 2ndHiddenLayer = W^3 * A^2 + B^3$$
$$A^4 = FinalLayer = W^4 * A^3 + B^4$$

Now if we substitute the previous layers, into the final layer, we get:

$$FinalLayer = W^4 * A^3 + B^4 = W^4 * (W^3 * A^2 + B^3) + B^4$$
$$FinalLayer = W^4 * (W^3 * (W^2 * A^1 + B^2) + B^3) + B^4$$

$$FinalLayer = W^4 * (W^3 * (W^2 * InputLayer + B^2) + B^3) + B^4$$

As we can see, the final/output layer, is just the input layer, repetitively multiplied by weights, and added onto by biases. There's no point multiplying the $InputLayer$, by THREE different weight matrices (W^2, W^3, W^4), and adding onto it, THREE different bias vectors (B^2, B^3, B^4), as ultimately, that would end up being the same thing, as just multiplying on a single weight matrix, and single bias vector. We can verify this by using distribution:

$$FinalLayer = W^4 * (W^3 * (W^2 * InputLayer + B^2) + B^3) + B^4$$
$$FinalLayer = W^4 * (W^3 * W^2 * InputLayer + W^3 * B^2 + B^3) + B^4$$
$$FinalLayer = W^4 * W^3 * W^2 * InputLayer + W^4 * W^3 * B^2 + W^4 * B^3 + B^4$$

$$W^4 * W^3 * W^2 = W^5$$

$$W^4 * W^3 * B^2 + W^4 * B^3 + B^4 = B^5$$

$$FinalLayer = W^5 * InputLayer + B^5$$

As we can see, if the middle layers don't have sigmoid functions, then the final layer, can be expressed directly in terms of the input layer, skipping all the middle layers, because they weren't actually doing anything. Thus, every middle layer MUST have a sigmoid function.

Now, we pretty much understand how a neural network, works. The neural network will receive an input, calculate the 1st layer of neurons, then the 2nd layer, and so on, all the way up to the final layer (the output layer), using this expression:

$$A^l = \sigma(W^l * A^{l-1} + B^l)$$

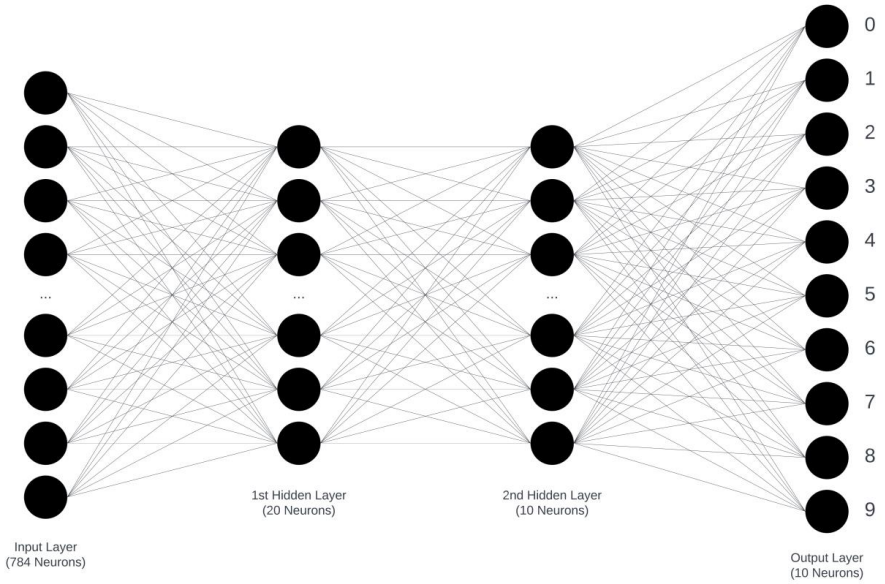
Once the final layer is calculated, it's outputted, and thus we finally get our output/answer from the neural network, done! (Hopefully the output/answer provided by the neural network, is correct)

However, there is one final thing. What determines the accuracy of our neural network?

Well, as we can see from the expression for A^l just above, it's the weights and the biases (technically our choice of "activation" function as well, sigmoid in this case, but that's beyond the scope of this project).

So, if the neural network has good weights and biases, then it'll read these images of handwritten digits, correctly, almost every single time.

So then, how do we figure out what the good weights and biases are? Well first let's determine how many there are. Looking at our neural network again:



We can see there are 784 weights, for each of the 20 neurons in the 1st hidden layer. There's also 1 bias for each of these 20 neurons, so we have:

$$size(W^2) = 784 * 20 = 15{,}680$$
$$size(B^2) = 1 * 20 = 20$$

Repeating the same process for the subsequent layers:

$$size(W^3) = 20 * 10 = 200$$
$$size(B^3) = 1 * 10 = 10$$
$$size(W^4) = 10 * 10 = 100$$
$$size(B^4) = 1 * 10 = 10$$
$$Weight\ Total = 15{,}680 + 200 + 100 = 15{,}980$$
$$Bias\ Total = 20 + 10 + 10 = 40$$

So, we have to figure out good values, for about $16{,}000$ weights/biases. Tricky, but it can certainly be done.

Just like in my other project, covering "Multivariable Linear Regression", good values for the weights/biases, are found, by creating a "cost" function, and then using the negative gradient of that cost function, to approach/converge on good values (this is known as "training" the neural network).

Now, as this document is already 12 pages long, I will skip a thorough explanation of the concept of "cost", and an intuitive explanation of "negative gradient". Instead, I'll get straight into creating a cost function for this neural network, then working out the gradient of it.

$$n = number\ of\ training\ instances/images$$
$$_i output = what\ the\ neural\ network\ sees\ in\ the\ i^{th}\ inputted\ image$$
$$_i actual = what\ was\ ACTUALLY\ in\ the\ i^{th}\ inputted\ image$$

$$C = \frac{\sum_{k=1}^{n} |_k output - _k actual|^2}{2n}$$

Note: The reason why we're taking the absolute magnitude of the difference: "$output - actual$", is because that difference is a vector, and we can't square a vector. Thus, absolute magnitude to turn the difference into a scalar. We're also calling cost "C", since it's a bit shorter.

So, we have our cost function, now we just need to figure out the gradient of it, which is just all the partial derivatives of cost, with respect to the weights/bias, put into a vector. So the gradient of cost will be a vector, containing $\approx 16{,}000$ partial derivatives. It'll look like this:

$$\nabla C = \begin{bmatrix} \frac{dC}{dW^2_{1,1}} \\ \frac{dC}{dW^2_{1,2}} \\ \cdots \\ \frac{dC}{dW^4_{10,10}} \\ \frac{dC}{dB^2_1} \\ \frac{dC}{dB^2_2} \\ \cdots \\ \frac{dC}{dB^4_{10}} \end{bmatrix}$$

Now, how do we work out what one of these partial derivatives is? Say any one of the weight partial derivatives: $\frac{dC}{dW^l_{i,j}}$

Well, we can't just express cost, directly in terms of that particular weight, because the expression would be absolutely tremendous in size. Let's say that weight was at the beginning of the network, then to express just the output layer (a SINGLE $prediction$) in terms of that 1st hidden layer weight, would require 1,570,110 terms, nested amongst multiple sigmoid functions. Even using summation notation, would be extremely messy.

So, what do we do? We basically just have this right now:

$$\frac{dC}{dW^l_{i,j}} = \frac{d\frac{\sum_{k=1}^{n}|_k output - {}_k actual|^2}{2n}}{dW^l_{i,j}}$$

Well, first let's see if we can get around / simplify the summation notation.

Note: We'll use A^L to represent the output, (for this 4 layer neural network, $L = 4$). We'll also use Y to represent the correct answer. We're doing this just to make the notation more compact and generalized, as all this mathematics still works, regardless of how many layers are in our neural network.

$$output = A^L$$

$$actual = Y$$

$$C \;=\; \frac{\sum_{k=1}^{n} |_k output - {}_k actual|^2}{2n} \;=\; \frac{\sum_{k=1}^{n} |_k A^L - {}_k Y|^2}{2n}$$

Now, expanding out the summation:

$$C = \frac{|_1 A^L - {}_1 Y|^2 + |_2 A^L - {}_2 Y|^2 + |_3 A^L - {}_3 Y|^2 + \ldots + |_n A^L - {}_n Y|^2}{2n}$$

Now, if we pull out the factor of n, and distribute the 2, we get:

$$C = \frac{1}{n} * \left(\frac{|_1 A^L - {}_1 Y|^2}{2} + \frac{|_2 A^L - {}_2 Y|^2}{2} + \frac{|_3 A^L - {}_3 Y|^2}{2} + \cdots + \frac{|_n A^L - {}_n Y|^2}{2} \right)$$

Now, let's look closely at this expression… specifically the term: $\frac{|_1 A^L - {}_1 Y|^2}{2}$

This looks a lot like a cost function, that only has one training instance (the very first one), instead of n training instances. Let's double check to make sure:

$$_1 Cost = {}_1 C \;=\; \frac{\sum_{k=1}^{1} |_k output - {}_k actual|^2}{2n} \;=\; \frac{|_k output - {}_k actual|^2}{2 * 1}$$

$$_1 C = \frac{|_1 A^L - {}_1 Y|^2}{2}$$

It is a cost function! Just using a single training instance, instead of n, the same goes for the other terms. C_2 is a cost function using only the second training instance, C_3 is a cost function using only the third training instance, all the way up to C_n, which uses the final training instance.

$$_2 C = \frac{|_2 A^L - {}_2 Y|^2}{2}, \qquad {}_3 C = \frac{|_3 A^L - {}_3 Y|^2}{2}, \qquad \ldots, \qquad {}_n C = \frac{|_n A^L - {}_n Y|^2}{2}$$

This means we can express the OVERALL cost, in terms of these "INDIVIDUAL" costs:

$$C = \frac{1}{n} * ({}_1 C + {}_2 C + \cdots + {}_n C) = \frac{1}{n} * \sum_{k=1}^{n} {}_k C$$

So, we can rewrite the derivative of cost, with respect to some weight, to be:

$$\frac{dC}{dW_{i,j}^{l}} = \frac{d\frac{({}_1C + {}_2C + \cdots + {}_nC)}{n}}{dW_{i,j}^{l}}$$

$$\frac{dC}{dW_{i,j}^{l}} = \frac{1}{n} * \left(\frac{d_1C}{dW_{i,j}^{l}} + \frac{d_2C}{dW_{i,j}^{l}} + \cdots + \frac{d_nC}{dW_{i,j}^{l}}\right)$$

Okay, we have something more manageable here. We just need to work out the derivative of each individual cost, then sum all derivatives together, then divide the sum by n (the number of individual costs), so we're basically averaging over all training instances, to find $\frac{dC}{dW_{i,j}^{l}}$

Fair enough, so then, what's the derivative of one of these individual costs?

Well unfortunately, we still can't express even an INIDIVDUAL cost, directly in terms of some weight "$W_{i,j}^{l}$", because the expression would still be tremendously large (over 1.5 million terms). So, we'll instead need to use a combination of multivariable chain rule, and vectorisation.

Specifically, we'll slowly work our way through the neural network, starting at the end/output, and move backwards until we reach wherever the weight is (this process is called backpropagation).

So basically, we'll be calculating the derivative of an individual cost, with respect to a weight "$W_{i,j}^{l}$", in a way very similar to this:

$$\frac{dC}{dW_{i,j}^{l}} = \frac{dC}{dA^{L}} * \frac{dA^{L}}{dZ^{L}} * \frac{dZ^{L}}{dA^{L-1}} * \frac{dA^{L-1}}{dZ^{L-1}} * \frac{dZ^{L-1}}{dA^{L-2}} \cdots * \frac{dZ^{l}}{dW_{i,j}^{l}}$$

Note: We'll refer to the individual cost using "C", instead of "C_k", so that the indices don't get too confusing and cluttered. Additionally, Z simply refers to a layer of neurons in UNACTIVATED form (sigmoid function not applied yet), this will make more sense later on.

So now that we know what to calculate, we'll get started with the first thing in the "chain" above: the derivative of individual cost with respect to the output:

$$\frac{dC}{dA^{L}}$$

Now, C is a scalar, so we have the derivative of a scalar, with respect to a column vector "A^{L}", which means we need to take the derivative of the scalar, with respect to each of the vectors COMPONENTS, and the result will be a ROW vector.

$$\frac{dC}{dA^L} = \begin{bmatrix} \frac{dC}{dA_1^L} & \frac{dC}{dA_2^L} & \cdots & \frac{dC}{dA_m^L} \end{bmatrix}$$

Note: A^L is the output layer, containing 10 neurons, thus m = 10.

So, let's try work out what these components are. We'll start with the first one, $\frac{dC}{dA_1^L}$

$$\frac{dC}{dA_1^L} = \frac{d\frac{|A^L - Y|^2}{2}}{dA_1^L} = \frac{1}{2} * \frac{d|A^L - Y|^2}{dA_1^L}$$

$$|A^L - Y|^2 = (A_1^L - Y_1)^2 + (A_2^L - Y_2)^2 + \cdots + (A_m^L - Y_m)^2$$

$$\frac{d|A^L - Y|^2}{dA_1^L} = \frac{d\ (A_1^L - Y_1)^2 + (A_2^L - Y_2)^2 + \cdots + (A_m^L - Y_m)^2}{dA_1^L}$$

$$\frac{d|A^L - Y|^2}{dA_1^L} = \frac{d(A_1^L - Y_1)^2}{dA_1^L} + \frac{d(A_2^L - Y_2)^2}{dA_1^L} + \cdots + \frac{d(A_m^L - Y_m)^2}{dA_1^L}$$

All derivatives except for $\frac{d(A_1^L - Y_1)^2}{dA_1^L}$, are 0, because A_1^L isn't present in those binomials:

$$\frac{d|A^L - Y|^2}{dA_1^L} = \frac{d(A_1^L - Y_1)^2}{dA_1^L} = \frac{d(A_1^L)^2 + -2A_1^L Y_1 + (Y_1)^2}{dA_1^L}$$

$$\frac{d|A^L - Y|^2}{dA_1^L} = 2A_1^L + -2Y_1 = 2(A_1^L - Y_1)$$

$$\frac{dC}{dA_1^L} = \frac{1}{2} * \frac{d|A^L - Y|^2}{dA_1^L} = A_1^L - Y_1$$

So, we've found "$\frac{dC}{dA_1^L}$", but what "$\frac{dC}{dA_2^L}$", "$\frac{dC}{dA_3^L}$", etc?

Well, the mathematics above is the same for any component, so it should be "$A_2^L - Y_2$", "$A_3^L - Y_3$", etc. We'll prove it just in-case, via demonstration for any component $\frac{dC}{dA_?^L}$

$$\frac{dC}{dA_?^L} = \frac{d\frac{|A^L - Y|^2}{2}}{dA_?^L} = \frac{1}{2} * \frac{d|A^L - Y|^2}{dA_?^L}$$

$$\frac{d|A^L - Y|^2}{dA_?^L} = \frac{d(A_1^L - Y_1)^2}{dA_?^L} + \frac{d(A_2^L - Y_2)^2}{dA_?^L} + \cdots + \frac{d(A_m^L - Y_m)^2}{dA_?^L}$$

Again, same thing, all these derivatives are 0, except for whichever one contains $dA_?^L$, thus:

$$\frac{d|A^L - Y|^2}{dA_?^L} = \frac{d\left(A_?^L - Y_?\right)^2}{dA_?^L} = \frac{d(A_?^L)^2 + -2A_?^L Y_? + (Y_?)^2}{dA_?^L} = 2(A_?^L - Y_?)$$

$$\frac{dC}{dA_?^L} = \frac{1}{2} * \frac{d|A^L - Y|^2}{dA_?^L} = A_?^L - Y_?$$

Thus, we've worked out $\frac{dC}{dA^L}$

$$\frac{dC}{dA^L} = \begin{bmatrix} \frac{dC}{dA_1^L} & \frac{dC}{dA_2^L} & \cdots & \frac{dC}{dA_m^L} \end{bmatrix} = [A_1^L - Y_1 \quad A_2^L - Y_2 \quad \cdots \quad A_m^L - Y_m]$$

$$\boldsymbol{\frac{dC}{dA^L} = \left(A^L - Y\right)^T}$$ ← *Transposed*

Fantastic, we can now move onto the next derivative in the "chain", which is $\frac{dA^L}{dZ^L}$

Now, Z^L is just the exact same layer of neurons as A^L, except the neurons haven't been activated yet (no activation function, specifically the sigmoid function, has been applied yet). So basically:

$$A^L = \sigma(Z^L)$$

Now, as for "$\frac{dA^L}{dZ^L} = \frac{d\sigma(Z^L)}{dZ^L}$", this is a derivative of a VECTOR, with respect to another vector. Which means we take the derivative of top vectors components, with respect to all components of the bottom vector:

$$\frac{dA^L}{dZ^L} = \begin{bmatrix} \frac{d\sigma(Z_1^L)}{dZ_1^L} & \frac{d\sigma(Z_1^L)}{dZ_2^L} & \cdots & \frac{d\sigma(Z_1^L)}{dZ_m^L} \\ \frac{d\sigma(Z_2^L)}{dZ_1^L} & \frac{d\sigma(Z_2^L)}{dZ_2^L} & \cdots & \frac{d\sigma(Z_2^L)}{dZ_m^L} \\ \vdots & \vdots & & \vdots \\ \frac{d\sigma(Z_m^L)}{dZ_1^L} & \frac{d\sigma(Z_m^L)}{dZ_2^L} & \ddots & \frac{d\sigma(Z_m^L)}{dZ_m^L} \\ & & \cdots & \end{bmatrix}$$

That's a lot of derivatives, thankfully, almost all of them equate to zero, as an activated neuron doesn't change, if a DIFFERENT, unactivated neuron changes. So only the main diagonal row is nonzero:

$$\frac{d\sigma(Z_m^L)}{dZ_m^L} = \sigma`(Z_m^L) = \frac{d\frac{1}{1+e^{-Z_m^L}}}{dZ_m^L} = \frac{e^{-Z_m^L}}{\left(1+e^{-Z_m^L}\right)^2}$$

The notation for this derivative "$\frac{d\sigma(Z_m^L)}{dZ_m^L}$", is pretty messy, so we'll leave it as $\sigma`(Z_m^L)$. Thus $\frac{dA^L}{dZ^L}$ is simply:

$$\frac{dA^L}{dZ^L} = \begin{bmatrix} \sigma`(Z_1^L) & 0 & \cdots & 0 \\ 0 & \sigma`(Z_2^L) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma`(Z_m^L) \end{bmatrix}$$

Thus, so far in our chain of derivatives, we've figured out both "$\frac{dC}{dA^L}$", and "$\frac{dA^L}{dZ^L}$", now multiplying them together, we get:

$$\frac{dC}{dA^L} * \frac{dA^L}{dZ^L} = [A_1^L - Y_1 \quad A_2^L - Y_2 \quad \cdots \quad A_m^L - Y_m] * \begin{bmatrix} \sigma`(Z_1^L) & 0 & \cdots & 0 \\ 0 & \sigma`(Z_2^L) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma`(Z_m^L) \end{bmatrix}$$

$$\frac{dC}{dA^L} * \frac{dA^L}{dZ^L} = \begin{bmatrix} (A_1^L - Y_1) * \sigma`(Z_1^L) \\ (A_2^L - Y_2) * \sigma`(Z_2^L) \\ \cdots \\ (A_m^L - Y_m) * \sigma`(Z_m^L) \end{bmatrix}^T$$

Now, the matrix $\frac{dA^L}{dZ^L}$ is equal to, is a bit messy to represent in Microsoft Word (and won't be as efficient to code in later on), so instead, we can leverage the fact that "$\frac{dA^L}{dZ^L}$", is similar to an identity matrix, and then use the "Hadamard product", to represent "$\frac{dC}{dA^L} * \frac{dA^L}{dZ^L}$", like this:

Equation 1

$$\boldsymbol{\frac{dC}{dA^L} * \frac{dA^L}{dZ^L}} = \begin{bmatrix} A_1^L - Y_1 \\ A_2^L - Y_2 \\ \cdots \\ A_m^L - Y_m \end{bmatrix}^T \odot \begin{bmatrix} \sigma`(Z_1^L) \\ \sigma`(Z_2^L) \\ \cdots \\ \sigma`(Z_m^L) \end{bmatrix}^T = \boldsymbol{\left(A^L - Y\right)^T \odot \sigma`\left(Z^L\right)^T}$$

This expression: "$\boldsymbol{\left(A^L - Y\right)^T \odot \sigma`\left(Z^L\right)^T}$", will be incredibly useful when we work out the derivative of cost, with respect to any weights/biases of the FINAL layer.

Now, we can move onto the next derivative in the chain, "$\frac{dZ^L}{dA^{L-1}}$".

Note: p = number of neurons in the second last layer

$$\frac{dZ^L}{dA^{L-1}} = \begin{bmatrix} \frac{dZ_1^L}{dA_1^{L-1}} & \frac{dZ_1^L}{dA_2^{L-1}} & \cdots & \frac{dZ_1^L}{dA_p^{L-1}} \\ \frac{dZ_2^L}{dA_1^{L-1}} & \frac{dZ_2^L}{dA_2^{L-1}} & \cdots & \frac{dZ_2^L}{dA_p^{L-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dZ_m^L}{dA_1^{L-1}} & \frac{dZ_m^L}{dA_2^{L-1}} & \cdots & \frac{dZ_m^L}{dA_p^{L-1}} \end{bmatrix}$$

Now, we need to workout what these derivates are, that are inside this matrix. We'll work out "$\frac{dZ_m^L}{dA_p^{L-1}}$", as that's in general form, so if we solve that one, we solve all of them.

$$Z_m^L = W_{m,1}^L * A_1^{L-1} + W_{m,2}^L * A_2^{L-1} + \cdots + W_{m,p}^L * A_p^{L-1} + B_m^L$$

$$\frac{dZ_m^L}{dA_p^{L-1}} = \frac{dW_{m,1}^L * A_1^{L-1} + W_{m,2}^L * A_2^{L-1} + \cdots + W_{m,p}^L * A_p^{L-1} + B_m^L}{A_p^{L-1}}$$

$$\frac{dZ_m^L}{dA_p^{L-1}} = \frac{dW_{m,1}^L * A_1^{L-1}}{dA_1^{L-1}} + \frac{dW_{m,2}^L * A_2^{L-1}}{dA_2^{L-1}} + \cdots + \frac{dW_{m,p}^L * A_p^{L-1}}{dA_p^{L-1}}$$

All of these individual derivative terms equate to 0, except for the one that contains the neuron "A_p^{L-1}", thus we're left with:

$$\frac{dZ_m^L}{dA_p^{L-1}} = W_{m,p}^L$$

$$\frac{dZ^L}{dA^{L-1}} = \begin{bmatrix} W_{1,1}^L & W_{1,2}^L & \cdots & W_{1,p}^L \\ W_{2,1}^L & W_{2,2}^L & \cdots & W_{2,p}^L \\ \vdots & \vdots & \ddots & \vdots \\ W_{m,1}^L & W_{m,2}^L & \cdots & W_{m,p}^L \end{bmatrix} = W^L$$

Now, the next derivative to be figured out in our chain, is "$\frac{dA^{L-1}}{dZ^{L-1}}$", which can be worked out in the exact same way as "$\frac{dA^L}{dZ^L}$", from earlier. None of the mathematics is different, but I'll demonstrate regardless as proof:

$$\frac{dA^{L-1}}{dZ^{L-1}} = \begin{bmatrix} \frac{d\sigma(Z_1^{L-1})}{dZ_1^{L-1}} & \frac{d\sigma(Z_1^{L-1})}{dZ_2^{L-1}} & \cdots & \frac{d\sigma(Z_1^{L-1})}{dZ_p^{L-1}} \\ \frac{d\sigma(Z_2^{L-1})}{dZ_1^{L-1}} & \frac{d\sigma(Z_2^{L-1})}{dZ_2^{L-1}} & \cdots & \frac{d\sigma(Z_2^{L-1})}{dZ_p^{L-1}} \\ \vdots & \vdots & & \vdots \\ \frac{d\sigma(Z_p^{L-1})}{dZ_1^{L-1}} & \frac{d\sigma(Z_p^{L-1})}{dZ_2^{L-1}} & \cdots & \frac{d\sigma(Z_p^{L-1})}{dZ_p^{L-1}} \end{bmatrix}$$

Again, almost all of the derivatives in this matrix equate to zero, due to an activated neuron not changing at all, if a DIFFERENT, unactivated neuron changes. Thus again, only main diagonal row is nonzero:

$$\frac{d\sigma(Z_p^{L-1})}{dZ_p^{L-1}} = \sigma`\left(Z_p^{L-1}\right) = \frac{d\frac{1}{1+e^{-Z_p^{L-1}}}}{dZ_p^{L-1}} = \frac{e^{-Z_p^{L-1}}}{\left(1+e^{-Z_p^{L-1}}\right)^2}$$

$$\frac{dA^{L-1}}{dZ^{L-1}} = \begin{bmatrix} \sigma`(Z_1^{L-1}) & 0 & \cdots & 0 \\ 0 & \sigma`(Z_2^{L-1}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma`\left(Z_p^{L-1}\right) \end{bmatrix}$$

So, we've worked out enough derivatives in the chain, to reach the second last layer now:

$$\frac{dC}{dA^L} * \frac{dA^L}{dZ^L} * \frac{dZ^L}{dA^{L-1}} * \frac{dA^{L-1}}{dZ^{L-1}} = ((A^L - Y)^T \odot \sigma`(Z^L)^T) * W^L * \begin{bmatrix} \sigma`(Z_1^{L-1}) & 0 & \dots & 0 \\ 0 & \sigma`(Z_2^{L-1}) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma`(Z_p^{L-1}) \end{bmatrix}$$

Again, just like with "$\frac{dA^L}{dZ^L}$", this matrix is a bit clunky, for the exact same reasons, so we'll use the Hadamard product again, to make things neater:

$$\boldsymbol{\frac{dC}{dZ^{L-1}} = \frac{dC}{dA^L} * \frac{dA^L}{dZ^L} * \frac{dZ^L}{dA^{L-1}} * \frac{dA^{L-1}}{dZ^{L-1}} = \left((A^L - Y)^T \odot \sigma`(Z^L)^T\right) * W^L \odot \sigma`(Z^{L-1})^T}$$

This expression: "$((A^L - Y)^T \odot \sigma`(Z^L)^T) * W^L \odot \sigma`(Z^{L-1})^T$", will be incredibly useful for when we work out the derivative of cost, with respect to any weights/biases of the SECOND-LAST layer.

Now, there's a pattern here, as we go further back in our neural network, we'll encounter the exact same types of derivatives, over and over:

$$\frac{dZ^{L-1}}{dA^{L-2}} * \frac{dA^{L-2}}{dZ^{L-2}} * \frac{dZ^{L-2}}{dA^{L-3}} * \frac{dA^{L-3}}{dZ^{L-3}} \frac{dZ^{L-3}}{dA^{L-4}} * \frac{dA^{L-4}}{dZ^{L-4}} \dots$$

All derivatives like "$\frac{dA^{L-2}}{dZ^{L-2}}$", "$\frac{dA^{L-3}}{dZ^{L-3}}$", etc, are calculated the exact same way as "$\frac{dA^L}{dZ^L}$", so we'll keep getting the same matrix that is similar to an identity matrix:

$$h = number\ of\ neurons\ in\ layer\ l$$

$$\frac{dA^l}{dZ^l} = \begin{bmatrix} \sigma`(Z_1^l) & 0 & \dots & 0 \\ 0 & \sigma`(Z_2^l) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma`(Z_h^l) \end{bmatrix}$$

And all derivatives like "$\frac{dZ^{L-1}}{dA^{L-2}}$", "$\frac{dZ^{L-2}}{dA^{L-3}}$", etc, are calculated the same way as "$\frac{dZ^L}{dA^{L-1}}$", so we'll keep getting a weight matrix:

$$\frac{dZ^l}{dA^{l-1}} = W^l$$

Since we keep getting the same results as we go further back through more and more layers, we can also just keep using the Hadamard product, to provide a nicer, more compact way of representing the chain of derivatives:

Equation 2

$$\frac{\boldsymbol{dC}}{\boldsymbol{dZ^l}} = \frac{\boldsymbol{dC}}{\boldsymbol{dZ^{l+1}}} * \boldsymbol{W^{l+1}} \odot \boldsymbol{\sigma`}\left(\boldsymbol{Z^l}\right)^{\boldsymbol{T}}$$

And thus, we have a way of figuring out the derivative o f cost, with respect to any layer of neurons!

Now, the only thing left in our original chain, is the final derivative:

$$\frac{dC}{dW_{i,j}^l} = \frac{dC}{dA^L} * \frac{dA^L}{dZ^L} * \frac{dZ^L}{dA^{L-1}} * \frac{dA^{L-1}}{dZ^{L-1}} * \frac{dZ^{L-1}}{dA^{L-2}} \dots * \frac{dZ^l}{dW_{i,j}^l}$$

$$h = number\ of\ neurons\ in\ layer\ l$$

$$\frac{dZ^l}{dW_{i,j}^l} = \begin{bmatrix} \frac{dZ_1^l}{dW_{i,j}^l} \\ \frac{dZ_2^l}{dW_{i,j}^l} \\ \dots \\ \frac{dZ_h^l}{dW_{i,j}^l} \end{bmatrix}$$

Now, all of the individual derivatives inside this column vector, equate to zero, except for "$\frac{dZ_i^l}{dW_{i,j}^l}$", because a neuron does not change, if a weight NOT connected it, changes. So we only have to work out $\frac{dZ_i^l}{dW_{i,j}^l}$

$$q = number\ of\ neurons\ in\ layer\ "l-1"$$

$$\frac{dZ_i^l}{dW_{i,j}^l} = \frac{d\ W_{i,1}^l * A_1^{l-1} + W_{i,2}^l * A_2^{l-1} + \dots + W_{i,q}^l * A_q^{l-1} + B_i^l}{dW_{i,j}^l}$$

$$\frac{dZ_i^l}{dW_{i,j}^l} = \frac{dW_{i,1}^l * A_1^{l-1}}{dW_{i,j}^l} + \frac{dW_{i,2}^l * A_2^{l-1}}{dW_{i,j}^l} + \dots + \frac{dW_{i,q}^l * A_q^{l-1}}{dW_{i,j}^l} + \frac{dB_i^l}{dW_{i,j}^l}$$

All of these terms equate to zero, except for the one that contains the particular weight: "$W_{i,j}^{l}$"

$$\frac{dZ_i^l}{dW_{i,j}^l} = \frac{dW_{i,j}^l * A_j^{l-1}}{dW_{i,j}^l} = A_j^{l-1}$$

$$\frac{dZ^l}{dW_{i,j}^l} = \begin{bmatrix} \frac{dZ_1^l}{dW_{i,j}^l} \\ \frac{dZ_2^l}{dW_{i,j}^l} \\ \vdots \\ \frac{dZ_i^l}{dW_{i,j}^l} \\ \vdots \\ \frac{dZ_h^l}{dW_{i,j}^l} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ A_j^{l-1} \\ \vdots \\ 0 \end{bmatrix}$$

Done! We can now figure out what "$\frac{dC}{dW_{i,j}^l}$" is:

$$\frac{dC}{dW_{i,j}^l} = \frac{dC}{dZ^l} * \frac{dZ^l}{dW_{i,j}^l} = \left(\frac{dC}{dZ^{l+1}} * W^{l+1} \odot \sigma`\left(Z^l\right)^T\right) * \begin{bmatrix} 0 \\ 0 \\ \vdots \\ A_j^{l-1} \\ \vdots \\ 0 \end{bmatrix}$$

This looks confusing, but $\frac{dC}{dZ^l}$ is just a row vector, and $\frac{dZ^l}{dW_{i,j}^l}$ is just a column vector, that is actually full of ZEROES, except for its i^{th} component. Thus, when we distribute the row vectors elements, onto the corresponding column vectors elements, all terms in $\frac{dC}{dZ^l}$ will be multiplying onto a 0, and thus becoming a 0 themselves, EXCEPT for the i^{th} component in $\frac{dC}{dZ^l}$

So, the only component that matters (doesn't become 0), in both vectors, is the i^{th} component, thus the vector product simplifies to:

$$\boldsymbol{\frac{dC}{dW_{i,j}^l}} = \left(\frac{dC}{dZ^{l+1}} * W^{l+1} \odot \sigma`\left(Z^l\right)^T\right)_i * \begin{bmatrix} 0 \\ 0 \\ \vdots \\ A_j^{l-1} \\ \vdots \\ 0 \end{bmatrix}_i = \boldsymbol{\frac{dC}{dZ_i^l} * A_j^{l-1}}$$

And voila! We have finally managed to work out the derivative of cost, with respect to any weight we want, in the entire neural network!

But what about the derivative of cost, with respect to any BIAS we want? Well that's extremely similar, we have the exact same chain of derivatives, except for the final one, which is worked out in a similar way to $\frac{dZ^l}{dW^l_{i,j}}$

$$\frac{dC}{dB^l_i} = \frac{dC}{dA^L} * \frac{dA^L}{dZ^L} * \frac{dZ^L}{dA^{L-1}} * \frac{dA^{L-1}}{dZ^{L-1}} * \frac{dZ^{L-1}}{dA^{L-2}} \dots * \frac{dZ^l}{dB^l_i}$$

$$h = number\ of\ neurons\ in\ layer\ l$$

$$\frac{dZ^l}{dB^l_i} = \begin{bmatrix} \frac{dZ^l_1}{dB^l_i} \\ \frac{dZ^l_2}{dB^l_i} \\ \dots \\ \frac{dZ^l_h}{dB^l_i} \end{bmatrix}$$

Again, all the derivatives within this column vector equate to 0, except for "$\frac{dZ^l_i}{dB^l_i}$", as the bias "B^l_i", only affects a single neuron in this layer, "Z^l_i"

$$q = number\ of\ neurons\ in\ layer\ "l-1"$$

$$\frac{dZ^l_i}{dB^l_i} = \frac{d\ W^l_{i,1} * A^{l-1}_1 + W^l_{i,2} * A^{l-1}_2 + \dots + W^l_{i,q} * A^{l-1}_q + B^l_i}{dB^l_i}$$

$$\frac{dZ^l_i}{dB^l_i} = \frac{dW^l_{i,1} * A^{l-1}_1}{dB^l_i} + \frac{dW^l_{i,2} * A^{l-1}_2}{dB^l_i} + \dots + \frac{dW^l_{i,q} * A^{l-1}_q}{dB^l_i} + \frac{dB^l_i}{dB^l_i}$$

$$\frac{dZ^l_i}{dB^l_i} = \frac{dB^l_i}{dB^l_i} = 1$$

$$\frac{dZ^l}{dB_i^l} = \begin{bmatrix} \frac{dZ_1^l}{dB_i^l} \\ \frac{dZ_2^l}{dB_i^l} \\ \vdots \\ \frac{dZ_i^l}{dB_i^l} \\ \vdots \\ \frac{dZ_h^l}{dB_i^l} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

We've figured out what "$\frac{dZ^l}{dB_i^l}$" is, and thus we can now figure out what $\frac{dC}{dB_i^l}$ is:

$$\frac{dC}{dB_i^l} = \frac{dC}{dZ^l} * \frac{dZ^l}{dB_i^l} = \left(\frac{dC}{dZ^{l+1}} * W^{l+1} \odot \sigma`\left(Z^l\right)^T\right) * \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Again, this looks complicated, but it can be simplified, as $\frac{dC}{dZ^l}$ is a row vector, and "$\frac{dZ^l}{dB_i^l}$" is just a column vector full of zeroes, except for the i^{th} component, so when we multiply corresponding elements together between both vectors, all components of $\frac{dC}{dZ^l}$ will multiply onto a 0, and thus become 0, leaving just the i^{th} component of both vectors left, in the product:

$$\boldsymbol{\frac{dC}{dB_i^l}} = \left(\frac{dC}{dZ^{l+1}} * W^{l+1} \odot \sigma`\left(Z^l\right)^T\right)_i * \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}_i = \boldsymbol{\frac{dC}{dZ_i^l}}$$

Voila! We've finally managed to figure out derivative of an individual cost, with respect to any bias we want, and any weight we want. The two expressions we got, are below:

Note: These are individual costs, I've just been suppressed the subscript notation $_kC$ to make the indices less confusing:

$$\boldsymbol{\frac{dC}{dW_{i,j}^l} = \frac{dC}{dZ_i^l} * A_j^{l-1}} \qquad \textit{Equation 3}$$

$$\boldsymbol{\frac{dC}{dB_i^l} = \frac{dC}{dZ_i^l}} \qquad \textit{Equation 4}$$

Now, we can calculate the gradient of the overall cost, ∇C, for our network, which was just this:

$$\nabla C = \begin{bmatrix} \frac{dC}{dW^2_{1,1}} \\ \frac{dC}{dW^2_{1,2}} \\ \cdots \\ \frac{dC}{dW^4_{10,10}} \\ \frac{dC}{dB^2_1} \\ \frac{dC}{dB^2_2} \\ \cdots \\ \frac{dC}{dB^4_{10}} \end{bmatrix}$$

First, we'll work out what the partial derivatives of the overall cost is, with respect to the weights:

$$\frac{dC}{dW^l_{i,j}} = \frac{1}{n} * \left(\frac{d_1 C}{dW^l_{i,j}} + \frac{d_2 C}{dW^l_{i,j}} + \cdots + \frac{d_n C}{dW^l_{i,j}} \right) = \frac{1}{n} * \left(\sum_{k=1}^{n} \frac{d_k C}{dW^l_{i,j}} \right)$$

We've already managed to calculate what the derivative of an individual cost is, with respect to a weight, so "$\frac{dC}{dW^l_{i,j}}$" is simply:

$$\frac{d_k C}{dW^l_{i,j}} = \frac{d_k C}{dZ^l_i} * A^{l-1}_j$$

$$\frac{dC}{dW^l_{i,j}} = \frac{1}{n} * \left(\sum_{k=1}^{n} \frac{d_k C}{dZ^l_i} * A^{l-1}_j \right)$$

Now, we just need to calculate the partial derivatives of overall cost, with respect to the biases:

$$\frac{dC}{dB^l_i} = \frac{1}{n} * \left(\frac{d_1 C}{dB^l_i} + \frac{d_2 C}{dB^l_i} + \cdots + \frac{d_n C}{dB^l_i} \right) = \frac{1}{n} * \left(\sum_{k=1}^{n} \frac{d_k C}{dB^l_i} \right)$$

$$\frac{d_k C}{dB^l_i} = \frac{d_k C}{dZ^l_i}$$

$$\frac{dC}{dB_i^l} = \frac{1}{n} * \left(\sum_{k=1}^{n} \frac{d_k C}{dZ_i^l} \right)$$

And thus, we now know the derivative of overall cost, with respect to any weight, and any bias, via these two expressions:

$$\frac{dC}{dW_{i,j}^l} = \frac{1}{n} * \left(\sum_{k=1}^{n} \frac{d_k C}{dZ_i^l} * A_j^{l-1} \right) \quad \textit{Equation 5}$$

$$\frac{dC}{dB_i^l} = \frac{1}{n} * \left(\sum_{k=1}^{n} \frac{d_k C}{dZ_i^l} \right) \quad \textit{Equation 6}$$

Both summations are divided by n (the number of individual costs), and thus the derivative of overall cost, with respect to any weight/bias, is actually just the AVERAGE amongst the derivatives of all the individual costs.

So, once we calculate the derivatives of all the individual costs, we can just average that sum, and we'll have the derivative of OVERALL cost, with respect to any weight/bias we want.

Thus, we can now calculate the gradient of cost, "∇C"!

Using the negative gradient of cost, "$-\nabla C$", we'll be able to work out the direction of steepest descent of the cost function, and thus be able to continuously train our neural network, until it hopefully finds some good weights and biases, and thus become reasonably accurate.

With the mathematics behind backpropagation now covered, we can implement our neural network in Python, from scratch. That concludes this explanation!

Credit

Credit to 3Blue1Brown, for the intuitive explanation of the overall structure of a neural network, found in the playlist here: https://www.youtube.com/watch?v=aircAruvnKk

Additional credit to Michael Nielsen, for a more in-depth explanation on the mathematics/code, and recommendations for future improvements, such as changing the cost function (currently mean-squared-error method) to a logarithm based one (cross-entropy), as well as exploring other activation functions, such as tanh, softmax, and rectified linear unit (ReLU).

Note – Vectorizing equation 3, on page 25:

The expression for the partial derivative of an individual cost, with respect to any weight, shown at the end of page 25 (equation 3):

$$\frac{\boldsymbol{dC}}{\boldsymbol{dW^l_{i,j}}} = \frac{\boldsymbol{dC}}{\boldsymbol{dZ^l_i}} * \boldsymbol{A^{l-1}_j}$$

Is correct, if we're evaluating a single weight at a time "$\mathrm{W^l_{i,j}}$". However, since we're implementing vectorisation to make the neural network faster, then the vectorized form of this equation will look slightly different, we can't just cut the subscript indices off everything. If we did that, then we'd get:

$$\frac{\boldsymbol{dC}}{\boldsymbol{dW^l}} = \frac{\boldsymbol{dC}}{\boldsymbol{dZ^l}} * \boldsymbol{A^{l-1}}$$

$\frac{\mathrm{dC}}{\mathrm{dZ^l}}$ is a row vector, and its length, is equal to the number of neurons in layer l. $\mathrm{A^{l-1}}$ is a column vector, and its length is equal to the number of neurons in layer $\mathrm{l}-1$.

You WILL NOT be able to multiply $\frac{\mathrm{dC}}{\mathrm{dZ^l}}$ and $\mathrm{A^{l-1}}$ together, UNLESS layer l and layer $\mathrm{l}-1$ have the exact same number of neurons (or in other words, $\mathrm{i}=\mathrm{j}$), and the odds of that are pretty low.

So, to fix this issue, you simply need to transpose the $\frac{\mathbf{dC}}{\mathbf{dZ^l}}$ into a column vector, and $\mathrm{A^{l-1}}$ into a row vector:

$$\frac{\boldsymbol{dC}}{\boldsymbol{dW^l}} = \left(\frac{\boldsymbol{dC}}{\boldsymbol{dZ^l}}\right)^{\boldsymbol{T}} * \left(\boldsymbol{A^{l-1}}\right)^{\boldsymbol{T}} \qquad \textit{Equation 7}$$

This procedure is not applicable to the derivative of cost with respect to biases (equation 4 at the end of page 25), because there are no two different indices i, j. So you CAN just cut off the subscript in that case, to vectorize:

$$\frac{\boldsymbol{dC}}{\boldsymbol{dB^l}} = \frac{\boldsymbol{dC}}{\boldsymbol{dZ^l}} \qquad \textit{Equation 8}$$