νral Networks

Pratik T

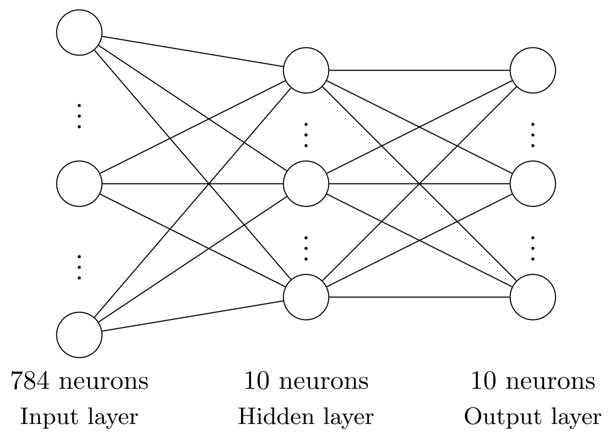
June 2025

1 Project Aim

Creating a neural network from scratch to predict what number a given handwritten input is.

2 Structure

1. The input will be a vector of $28 \times 28 = 784$ elements. This will represent 784 pixels in the input image. Each pixel will have a grayscale value. The data is from MNIST.
2. There will be 1 hidden layer of 10 elements that initially takes on random values.
3. The output layer will also have 10 elements representing the numbers 0 through 9. This layer is also initialized with random values.



3 Forward Propagation : Perceptron

1. For an input, a perceptron computes its output by applying a linear transform on the input and then passing it through an activation function to

introduce nonlinearity. Any multilayer network without an activation will be essentially useless, as it will still be a linear combination.

2. **Linear combination step**

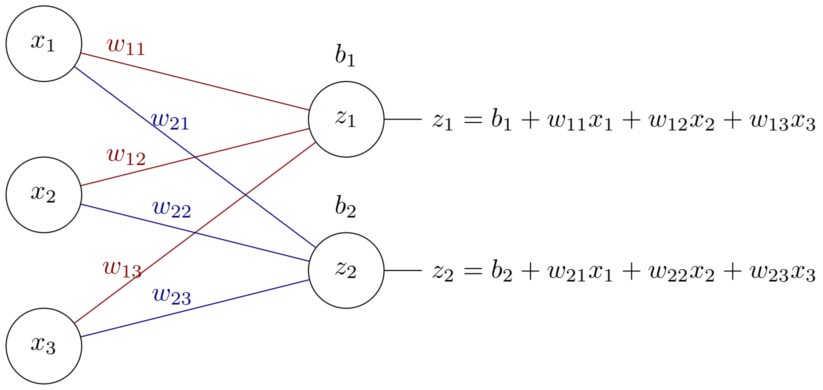
   (a) Each neuron in the hidden layer first computes $b + (x_1 \times w_1) + (x_2 \times w_2) + ...$ Here b is the bias associated with the neuron, and w_i are the weights associated with the connections between this neuron and a neuron of the preceding layer x_i. For a $i \times 1$ column vector $\mathbf{x}$, the linear combination output would look like

   $$\mathbf{W}^T \cdot \mathbf{x} + b$$

   where $\mathbf{W}$ is also a vector of dimensions $[i \times 1]$ and b is a number. This is the linear combination output of 1 neuron.

   (b) For multiple neurons then the output of the i^{th} neuron for m neurons in the previous x layer is

   $$z_i = b_i + \sum_{j=1}^{m} w_{ij} x_j$$



x_1
x_2
x_3
w_{11}
w_{21}
w_{12}
w_{22}
w_{13}
w_{23}
b_1
b_2
z_1
z_2
$z_1 = b_1 + w_{11}x_1 + w_{12}x_2 + w_{13}x_3$
$z_2 = b_2 + w_{21}x_1 + w_{22}x_2 + w_{23}x_3$

   $$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} + \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

   $$\vec{z} = \vec{b} + \overset{\leftrightarrow}{W}^T \vec{x}$$

   (c) W and b are called the weights and biases of the network. Since all neurons are connected, it's called a dense network. The result of the neuron is thus a linear combination of all preceding neurons.

   (d) There are weights and bias matrices associated which each layer. Since our network structure has 2 layers (hidden and output), there's going to be 2 weight matrices and 2 bias vectors.

3. **Hidden layer activation**: The linear output of the neuron is then passed through a non-linear function. For now, use ReLU

$$\text{ReLU}(x_i) = \max(0, x_i)$$

This is useful when we don't have/want any negative numbers in the output. If an initial neuron does become negative, then the neuron output becomes 0 and this might propagate and give unsatisfactory results. This is called ReLU dying. Another activation is Sigmoid which also gives a value between 0 and 1. But this will be more useful if we only have 1 output.

4. **Output Activation**: We want the output of the final layer to be 10 values probabilities which add up to 1. Softmax can be used for this.

$$\text{SM}(\text{x}_\text{i}) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

This normalizes the output values so that they add up to 1.

4 Loss

1. To be able to quantify the performance of a DNN, we need a measure of loss. For this problem, since the output is a vector of probability values, with the desired output being 1 for whatever the input number is and 0 for everything else. Our loss can then be a deviation of the predicted probabilities from the actual probability. Again, the actual probability vector will obviously be all 0s except for the input number which will have a probability of 1. To find the deviations between two probability distributions for binary checks, we can use a cross-entropy loss

$$\mathcal{L} = -\sum_i y_i \ln(z_i)$$

where y_i is the actual and z_i is the predicted value. Cross-entropy is good for probability distributions as values too far away from the desired value are penalized more harshly.

2. For outputs which can take on continuous values, losses like mean absolute or mean squared errors make more sense.

5 Gradient Descent

1. We want to find the optimal values of the weights to minimize the loss function. Since the loss is a function of all the weights of our network, we need to find the local slope in our hyper-dimensional loss landscape. By going in the direction of negative slope, we can then 'descent' to a local minimum.

2. The algorithm is thus as follows:

   (a) Initialize weights randomly at step 0

   (b) Do a forward pass and compute the loss $\mathcal{L}$

   (c) Compute the gradient $\partial\mathcal{L}/\partial w$ where w is a weight (either from the weight matrix, or from the bias)

   (d) Update the weights $w = w - \alpha \cdot \frac{\partial \mathcal{L}}{\partial w}$ where α is the step-size (learning rate). The negative comes because we want to go in the opposite direction of the gradient to descent

   (e) Repeat till convergence or for set number of iterations

   (f) Return the weights after completion to get the 'trained' model

3. Computing the gradient in a neural network is called back propagation. This is because we start at the output layer and use the chain rule to compute partial derivatives at each network layer to get the net gradient of a loss with a weight parameter.

6 Back Propagation

1. The gradient which will be used to update weights $w_{i,j}^{(n)}$ is $\partial\mathcal{L}/\partial w_{i,j}^{(n)}$. Let's evaluate this step by step.

2. Let $g(z)$ be the activation function. Then at a layer n (using Einstein summation convention)

$$x_i^{(n)} = g_i(z^{(n)}) = g_i\Big(b_i^{(n)} + \sum_k w_{ik}^{(n)} x_k^{(n-1)}\Big).$$

   Here g can be **any** vector activation, so

$$J_{ij}(z^{(n)}) \equiv \frac{\partial x_i^{(n)}}{\partial z_j^{(n)}} \quad \text{(the full Jacobian, not diagonal).}$$

   (a) **The weight derivative**:

$$\begin{aligned}\frac{\partial x_i^{(n)}}{\partial w_{pq}^{(n)}} &= \sum_l \frac{\partial x_i^{(n)}}{\partial z_l^{(n)}} \frac{\partial z_l^{(n)}}{\partial w_{pq}^{(n)}} = \sum_l J_{il} \frac{\partial}{\partial w_{pq}^{(n)}} \left[b_l^{(n)} + \sum_k w_{lk}^{(n)} x_k^{(n-1)} \right] \\ &= J_{ip}(z^{(n)})\, x_q^{(n-1)}.\end{aligned}$$

   This is a full 3D tensor.

   (b) **The bias derivative**:

$$\frac{\partial x_i^{(n)}}{\partial b_p^{(n)}} = \sum_l J_{il} \frac{\partial z_l}{\partial b_p} = J_{ip}(z^{(n)}).$$

3. Now we evaluate the loss gradient. Again, for a layer n,

   (a) **The weight gradient**:

$$\begin{aligned}\frac{\partial \mathcal{L}\left(x_i^{(N)}\right)}{\partial w_{pq}^{(n)}} &= \frac{\partial \mathcal{L}}{\partial x_i^{(N)}} \cdot \frac{\partial x_i^{(N)}}{\partial w_{pq}^{(n)}} \\ &= \frac{\partial \mathcal{L}}{\partial x_i^{(N)}} \cdot \frac{\partial x_i^{(N)}}{\partial x_j^{(N-1)}} \cdot \frac{\partial x_j^{(N-1)}}{\partial x_k^{(N-2)}} \cdot \frac{\partial x_k^{(N-2)}}{\partial x_l^{(N-3)}} ... \frac{\partial x_l^{(n)}}{\partial w_{pq}^{(n)}}\end{aligned}$$

   i. For the matrix form, let's proceed step by step again. Start at the last layer weights, $n = N$.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{pq}^{(N)}} &= \sum_i \frac{\partial \mathcal{L}}{\partial x_i^{(N)}} \frac{\partial x_i^{(N)}}{\partial w_{pq}^{(N)}} \\ &= \sum_i \frac{\partial \mathcal{L}}{\partial x_i^{(N)}} \, J_{ip}\big(z^{(N)}\big) \, x_q^{(N-1)} \\ &= \left[\sum_i \frac{\partial \mathcal{L}}{\partial x_i^{(N)}} \frac{\partial x_i^{(N)}}{\partial z_p^{(N)}}\right] x_q^{(N-1)} \\ &= \epsilon_p^{(N)} \, x_q^{(N-1)},\end{aligned}$$

   The first term is known as the error-signal vector, $\vec{\epsilon}$ and is of dimensionality $p \times 1$, such that the final product again has dimensionality $p \times q$

$$\frac{\partial \mathcal{L}}{\partial \overset{\leftrightarrow}{W}^{(N)}} = \vec{\epsilon}^{\,(N)} \cdot \vec{x}^{(N-1)^T}$$

   ii. Let's now find the weight gradient for the $N-1$ layer. Remember $x_i^{(N)} = g_i\left(z_i^{(N)}\right)$ and $z_i^{(N)} = b_1^{(N)} + \sum_k w_{ik}^{(N)} x_k^{(N-1)}$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{pq}^{(N-1)}} &= \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial x_i^{(N)}} \cdot \frac{\partial x_i^{(N)}}{\partial x_j^{(N-1)}} \cdot \frac{\partial x_j^{(N-1)}}{\partial w_{pq}^{(N-1)}} \\ &= \sum_i \sum_j \sum_k \frac{\partial \mathcal{L}}{\partial x_i^{(N)}} \cdot \frac{\partial x_i^{(N)}}{\partial z_k^{(N)}} \cdot \frac{\partial z_k^{(N)}}{\partial x_j^{(N-1)}} \cdot J_{jp}\big(z^{(N-1)}\big) \, x_q^{(N-2)} \\ &= \sum_{i,j,k} \frac{\partial \mathcal{L}}{\partial x_i^{(N)}} \cdot J_{ik}^{(N)} \cdot w_{kj}^{(N)} \cdot J_{jp}^{(N-1)} \, x_q^{(N-2)} \\ &= \left[\sum_{j,k} \epsilon_k^{(N)} \cdot w_{kj}^{(N)} \cdot J_{jp}^{(N-1)}\right] x_q^{(N-2)} \\ &= \epsilon_p^{(N-1)} \, x_q^{(N-2)},\end{aligned}$$

We can write this in matrix notation to again give an output with dimensions of $\overset{\leftrightarrow}{W}^{(N-1)}$

$$\frac{\partial \mathcal{L}}{\partial \overset{\leftrightarrow}{W}^{(N-1)}} = \vec{\epsilon}^{\,(N-1)} \cdot \vec{x}^{(N-2)^T}$$

iii. Similarly we can compute this for all the layers, propagating backwards.

iv. So where is the recursion? It's actually in the error vector of the hidden layers. We can write this as

$$\epsilon_p^{(N-1)} = \sum_{j,k} \epsilon_k^{(N)} \cdot w_{kj}^{(N)} \cdot J_{jp}^{(N-1)}$$

$$\epsilon_p^{(n)} = \sum_{j,k} \epsilon_k^{(n+1)} \cdot w_{kj}^{(n+1)} \cdot J_{jp}^{(n)}$$

$$\vec{\epsilon}^{\,(n)} = \overset{\leftrightarrow}{J}\left(\vec{z}^{\,(n)}\right)^T \left[\overset{\leftrightarrow}{W}^{(n+1)^T} \vec{\epsilon}^{\,(n+1)}\right]$$

Where in the last step we took a transpose to convert it into a column vector. The output has dimensions $p \times 1$

(b) **The bias gradient**:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b_p^{(N)}} &= \sum_i \frac{\partial \mathcal{L}}{\partial x_i^{(N)}} \cdot \frac{\partial x_i^{(N)}}{\partial b_p^{(N)}} \\ &= \sum_i \frac{\partial \mathcal{L}}{\partial x_p^{(N)}} \cdot J_{ip}^{(N)} \\ &= \vec{\epsilon}^{\,(N)}\end{aligned}$$

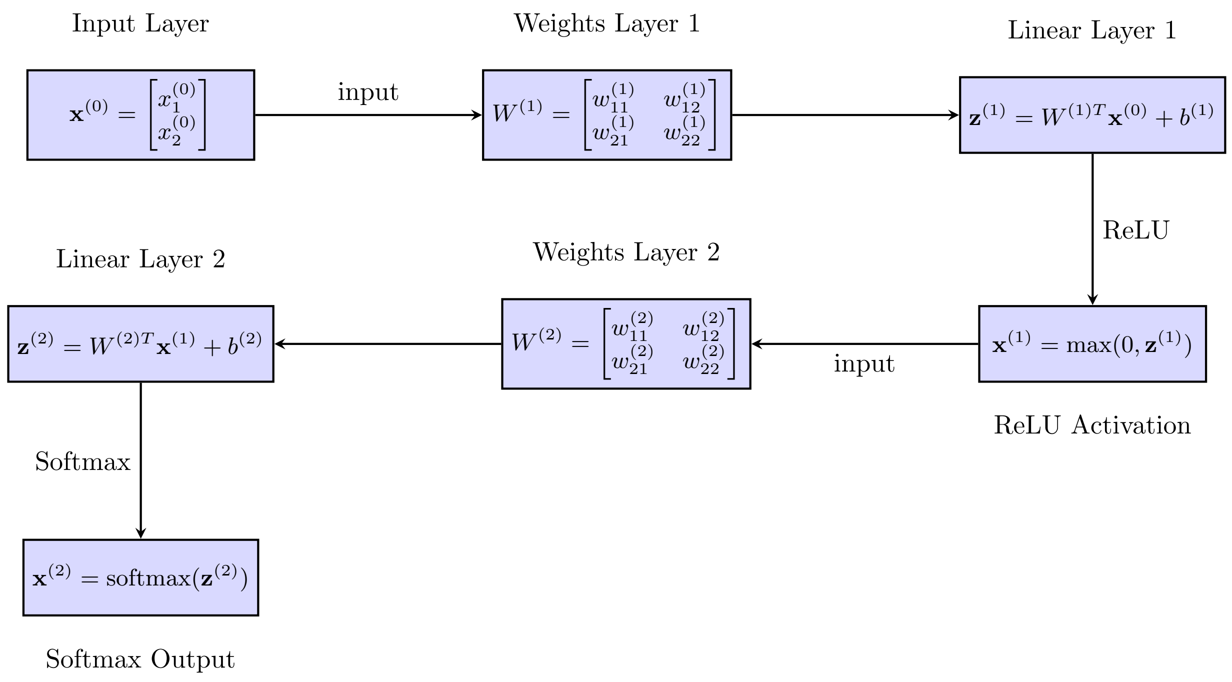
So the bias gradient is just the error vector.

4. Putting it together:

Weights Gradient	$\frac{\partial \mathcal{L}}{\partial w_{pq}^{(n)}} = \epsilon_p^{(n)}\ x_q^{(n-1)}$
Bias Gradient	$\partial \mathcal{L}/\partial b_p^{(n)} = {\epsilon_p}^{(n)}$
Hidden Layer Error Vector	$\epsilon_p^{(n)} = \sum_{j,k} \epsilon_k^{(n+1)} \cdot w_{kj}^{(n+1)} \cdot J_{jp}^{(n)}$
Boundary Error of Output	${\epsilon_p}^{(N)} = \sum_i \frac{\partial \mathcal{L}}{\partial x_i^{(N)}}\ J_{ip}^{(N)}$

7 A Specific Example

Let's compute the gradients for this example of a $N = 2$ layer neural network


Input Layer
$\mathbf{x}^{(0)} = \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \end{bmatrix}$
input
Weights Layer 1
$W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix}$
Linear Layer 1
$\mathbf{z}^{(1)} = W^{(1)T}\mathbf{x}^{(0)} + b^{(1)}$
ReLU
$\mathbf{x}^{(1)} = \max(0, \mathbf{z}^{(1)})$
ReLU Activation
input
Weights Layer 2
$W^{(2)} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} \end{bmatrix}$
Linear Layer 2
$\mathbf{z}^{(2)} = W^{(2)T}\mathbf{x}^{(1)} + b^{(2)}$
Softmax
$\mathbf{x}^{(2)} = \text{softmax}(\mathbf{z}^{(2)})$
Softmax Output

Loss	$\mathcal{L} = -\sum_i y_i \ln\left(x_i^{(2)}\right)$
Softmax	$x_i^{(2)} = \dfrac{e^{z_i^{(2)}}}{\sum_j e^{z_j^{(2)}}}$
ReLU	$x_i^{(1)} = \max(0, z_i^{(1)})$
Linear	$z_i^{(n)} = b_i^{(n)} + \sum_j w_{ij}^{(n)} x_j^{(n-1)}$

We essentially need to compute the jacobians and loss derivatives. Refer back to the previous table and use $N = 2$.

1. Jacobian of Softmax (drop the z superscript):

$$\begin{aligned} J_{ip}^{(2)} &= \frac{\mathrm{d}x_i^{(2)}}{\mathrm{d}z_p} = \frac{\mathrm{d}}{\mathrm{d}z_p}\left[\frac{e^{z_i}}{\sum_j e^{z_j}}\right] \\ &= \frac{e^{z_i}\delta_{ip}}{\sum_j e^{z_j}} - \frac{e^{z_i}e^{z_p}\delta_{pj}}{\left(\sum_j e^{z_j}\right)^2} \\ &= x_i^{(2)}\delta_{ip} - x_i^{(2)}x_p^{(2)} \\ &= x_i^{(2)}\left(\delta_{ip} - x_p^{(2)}\right) \end{aligned}$$

2. Derivative of Loss:

$$\mathcal{L}' = \frac{\mathrm{d}}{\mathrm{d}x_i^{(2)}}\left[-\sum_j y_j \ln\left(x_j^{(2)}\right)\right] = -\frac{y_i}{x_i^{(2)}}$$

3. Boundary error term:

$$\begin{aligned} \epsilon_p^{(2)} &= \sum_i -\frac{y_i}{x_i^{(2)}} \cdot x_i^{(2)}\left(\delta_{ip} - x_p^{(2)}\right) \\ &= -y_p + x_p^{(2)}\sum_i y_i \\ &= x_p^{(2)} - y_p \end{aligned}$$

   Where since y_i are normalised probability values, they add up to 1

4. Gradients for layer 2:
   (a) Weights $\Rightarrow \nabla w_{pq}^{(2)} = \epsilon_p^{(2)}\, x_q^{(1)}$
   (b) Biases $\Rightarrow \nabla b_p^{(2)} = \epsilon_p^{(2)}$

5. Jacobian for ReLU:

$$\begin{aligned} J_{jp}^{(1)} &= \frac{\mathrm{d}x_j^{(1)}}{\mathrm{d}z_p} = \frac{\mathrm{d}}{\mathrm{d}z_p}\left[\max(0, z_j)\right] \\ &= \delta_{jp} \iff z_j > 0 \end{aligned}$$

6. Hidden layer error:

$$\begin{aligned} \epsilon_p^{(1)} &= \sum_{j,k} \epsilon_k^{(2)} \cdot w_{kj}^{(2)} \cdot J_{jp}^{(1)} \\ &= \sum_k \epsilon_k^{(2)} \cdot w_{kp}^{(2)} \times \begin{cases} 1; & z_p^{(1)} > 0 \\ 0; & z_p^{(1)} \le 0 \end{cases} \end{aligned}$$

   Here $z_p^{(1)}$ Is the linear output of the layer before activation.

7. Gradients for layer 1:
   (a) Weights $\Rightarrow \nabla w_{pq}^{(1)} = \epsilon_p^{(1)}\ x_q^{(0)}$
   (b) Biases $\Rightarrow \nabla b_p^{(1)} = \epsilon_p^{(1)}$

Usually it's much more efficient to pass a batch of inputs at once during training. If the batch is of size m, the gradient is an average over all the gradients. So

$$\nabla w^{(2)} = \frac{1}{m}\sum_{i=1}^{m} \epsilon_{p,i}^{(2)} x_{q,i}^{(1)}$$

The sum can be done directly with a matrix product $x^{(1)}\epsilon^{(2)^T}$. It can be seen as follows (dropping the superscripts):

$$\begin{bmatrix} \epsilon_{11} & \epsilon_{12} & \cdots & \epsilon_{1m} \\ \epsilon_{21} & \epsilon_{22} & \cdots & \epsilon_{2m} \\ \vdots & \vdots & \cdots & \vdots \\ \epsilon_{10,1} & \epsilon_{10,2} & \cdots & \epsilon_{10,m} \end{bmatrix} \times \begin{bmatrix} x_{11} & x_{21} & \cdots & x_{10,1} \\ x_{12} & x_{22} & \cdots & x_{10,2} \\ \vdots & \vdots & \cdots & \vdots \\ x_{1,m} & x_{2,m} & \cdots & x_{10,m} \end{bmatrix}$$

$$= \begin{bmatrix} \epsilon_{11}x_{11} + \epsilon_{12}x_{12} + \cdots + \epsilon_{1m}x_{1,m} & \cdots & \epsilon_{11}x_{10,1} + \epsilon_{12}x_{10,2} + \cdots + \epsilon_{1m}x_{10,m} \\ \epsilon_{21}x_{11} + \epsilon_{22}x_{12} + \cdots + \epsilon_{2m}x_{1,m} & \cdots & \epsilon_{21}x_{10,1} + \epsilon_{22}x_{10,2} + \cdots + \epsilon_{2m}x_{10,m} \\ \vdots & \cdots & \vdots \\ \epsilon_{10,1}x_{11} + \epsilon_{10,2}x_{12} + \cdots + \epsilon_{10,m}x_{1,m} & \cdots & \epsilon_{10,1}x_{10,1} + \epsilon_{10,2}x_{10,2} + \cdots + \epsilon_{10,m}x_{10,m} \end{bmatrix}$$

$$= \sum_{i=1}^{m} \begin{bmatrix} \epsilon_{1,i}x_{1,i} & \cdots & \epsilon_{1,i}x_{m,i} \\ \epsilon_{2,i}x_{1,i} & \cdots & \epsilon_{2,i}x_{m,i} \\ \vdots & \cdots & \vdots \\ \epsilon_{10,i}x_{1,i} & \cdots & \epsilon_{10,i}x_{m,i} \end{bmatrix} = \sum_{i=1}^{m} \epsilon_{p,i}^{(2)} x_{q,i}^{(1)}$$

This was also obvious as

$$\sum_{i=1}^{m} x_{q,i}^{(1)} \epsilon_{i,p}^{(2)}$$

is the definition of a matrix product. And we're finally done. Computing these quantities is much simpler as matrices, so that's what we'll write them as. We could also use an input batch which is a matrix where column vectors are joined. This is shown in the next section.

8 Algorithm

1. All input will be stored column wise into a matrix. The x input is a 784×1 array of pixel data and the y input is a one hot encoded 10×1 array which is 1 for the label corresponding to the input x. From this, a single input batch $x^{(0)}$ will be of size $784 \times m$ where m is the batch size

2. Initialize the weight matrices $w^{(1)}$ and $w^{(2)}$ with random values between -0.5 and 0.5. The size of the $w^{(1)}$ is $784 \times l$ and $w^{(2)}$ is $l \times l$ where l is the layer size, in our case 10
3. Initialize the bias matrices $b^{(1)}$ and $b^{(2)}$ of size $l \times 1$ and $l \times 1$.
4. Perform the first pass to layer 1, which will be a linear transform $z^{(1)} = {w^{(1)}}^T \cdot x^{(0)} + b^{(1)}$
5. Pass this through ReLU to get $x^{(1)}$
6. Apply the linear transform again to get $z^{(2)} = {w^{(2)}}^T \times x^{(1)} + b^{(2)}$
7. Pass this through Softmax to get $x^{(2)}$. Normalize the input to softmax so the exponent doesn't blow up.
8. Now we compare $x^{(2)}$ to our actual value of a one hot encoded column wise $784 \times m$ $y^{(0)}$ matrix of y values. This is done via the boundary error term which we found for the case of the output layer being Softmax and Loss being cross entropy as ${\epsilon^{(2)}}_{784\times m} = x^{(2)} - y^{(0)}$
9. Using this find the weight gradient for layer 2: $\nabla w^{(2)} = \frac{1}{m} x^{(1)} {\epsilon^{(2)}}^T$
10. And the bias gradient for layer 2: $\nabla b^{(2)} = \sum_{i=1}^{m} \frac{1}{m} \epsilon_i^{(2)}$
11. Find the hidden layer error $\epsilon^{(1)} = {w^{(2)}}^T \epsilon^{(2)} \odot (1 \iff z^{(1)} > 0, 0 \text{ otherwise})$. Here $\odot$ is the Hadamard or element-wise product
12. Find the weight gradient for layer 1: $\nabla w^{(1)} = \frac{1}{m} x^{(0)} \epsilon^{(1^T)}$
13. And the bias gradient for layer 1: $\nabla b^{(1)} = \sum_{i=1}^{m} \frac{1}{m} \epsilon_i^{(1)}$
14. Update the weights and biases for a set learning rate α

$$\begin{aligned} w^{(1)} &= w^{(1)} - \alpha \nabla w^{(1)} & \qquad b^{(1)} &= b^{(1)} - \alpha \nabla b^{(1)} \\ w^{(2)} &= w^{(2)} - \alpha \nabla w^{(2)} & \qquad b^{(2)} &= b^{(2)} - \alpha \nabla b^{(2)} \end{aligned}$$

15. Repeat for a set number of epochs.