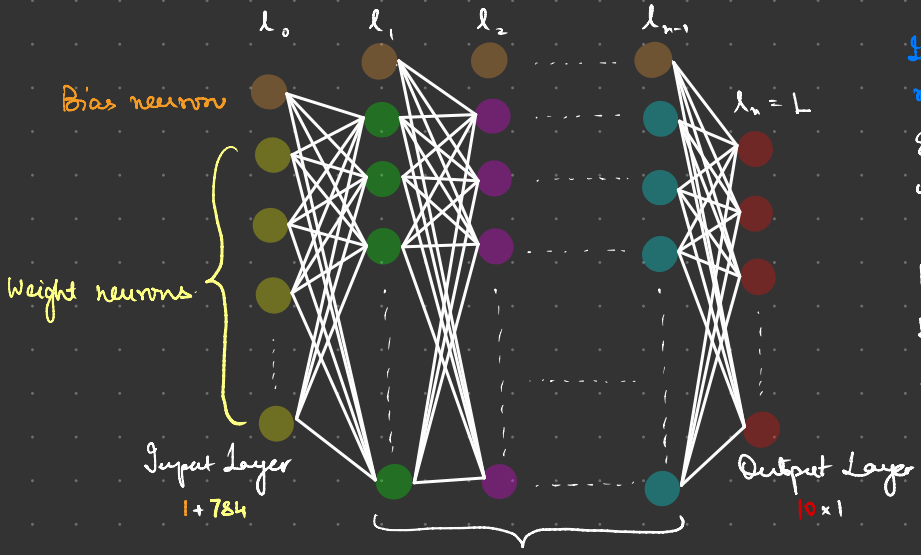
Model Architechtiure

→ On MNIST-handwritten digits dataset $\xrightarrow{\text{image}}$ shape = 28×28 $\xrightarrow{\text{flatten}}$ shape = 1×784



Initializing a base architechture (before activations, etc) require us to specify the number of neurons in each layer. Ex: [784, 40, 30, 20, 10], qualifies for us to build the weights & biases list. The bias neuron in l_1 will have 40 connections with l_2, while the bias neuron in l_2 will have 30 connections with l_3 and so on... Therefore the biases look like → [40×1, 30×1, 20×1, 10×1]. Now each of the 784 weight neurons in l_1 will have 40 connections with l_2, while each of the 40 weight neurons in l_2 will have 30 connections with l_3 and so on... Therefore weights look like → [40×784, 30×40, 20×30, 10×20]

Image flattened into a row vector concatenated with a bias term
↳ which only has a forward connection with the next layer.

Softmax over the output which are digits from 0 to 9.

Bias - Weight Initialization Techniques

1. Constant
2. Random — Uniform ⇒ $U[min, max]$
   - Normal ⇒ $\mathcal{N}[mean, std]$
3. Xavier-Glorot — Uniform ⇒ $U\left[-\sqrt{\frac{6}{n_{in}+n_{out}}}, +\sqrt{\frac{6}{n_{in}+n_{out}}}\right]$
   - Normal ⇒ $\mathcal{N}\left[0, \sqrt{\frac{2}{n_{in}+n_{out}}}\right]$
4. He — Uniform ⇒ $U\left[-\sqrt{\frac{3}{n_{in}}}, +\sqrt{\frac{3}{n_{in}}}\right]$
   - Normal ⇒ $\mathcal{N}\left[0, \sqrt{\frac{2}{n_{in}}}\right]$

l_0 l_1 l_2 l_3 l_4

$[784, 40, 30, 20, 10]$

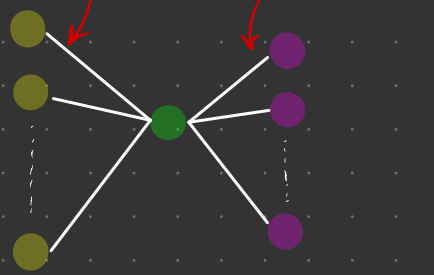
Xavier-glorot ↓ $n_{in} + n_{out}$

$[784+30, 40+20, 30+10, 20+0]$

He initialization → n_{in} → $[784, 40, 30, 20]$

For each neuron in l_1 there are → 784 in connections & 30 out connections

This means W_0 will be of shape 40×784 and the denominator in Xavier-glorot will look like $784 + 30$ while in He it looks like 30

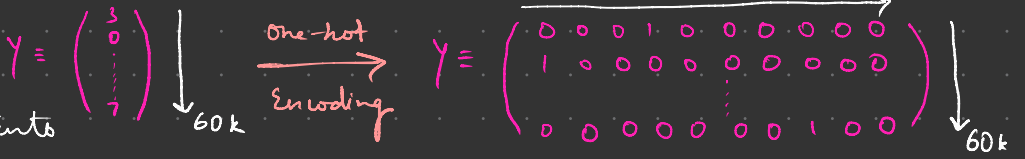


Input Normalization & Output Encoding

Images — Because pixels range from $[0, 255]$ → Divide by 255 → $[0, 1]$

Digits (output) range from 0 to 9

These class values will be encoded into a vector of length = number of classes, which is greater than equal to 3. This is done to support matrix product during "softmax" at the final layer. If we have 2 classes, then using sigmoid at last layer doesn't require us to encode the output classes.



Activation Functions

① Sigmoid — $s(z) = \frac{1}{1+\exp(-z)}$

$$1 - s(z) = 1 - \frac{1}{1+\exp(-z)} = \frac{\exp(-z)}{1+\exp(-z)}$$

↓ Derivative

$$\frac{\partial}{\partial z} s(z) = \frac{\partial}{\partial z}\left(\frac{1}{1+\exp(-z)}\right) = (-1)\left(\frac{1}{1+\exp(-z)}\right)^2 \frac{\partial}{\partial z}(1+\exp(-z)) = (-1)\left(\frac{1}{1+\exp(-z)}\right)^2 (0 + (-1)\exp(-z)) = \frac{\exp(-z)}{(1+\exp(-z))^2}$$

$$= \left(\frac{1}{1+\exp(-z)}\right)\left(\frac{\exp(-z)}{1+\exp(-z)}\right) = s(z) \odot (1 - s(z))$$

② Leaky ReLU (leak = 0 ⇒ ReLU) — $\text{relu}(z) = \begin{cases} z & \text{if } z \geq 0 \\ \alpha z & \text{if } z < 0 \text{ where } \alpha \geq 0 \end{cases}$

↓ Derivative

$$\frac{\partial}{\partial z}\text{relu}(z) = \begin{cases} \partial_z z \\ \partial_z \alpha z \end{cases} = \begin{cases} 1 & \text{if } z \geq 0 \\ \alpha & \text{otherwise} \end{cases}$$

③ Tanh — $\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$

↓ Derivative

$$\partial_z \tanh(z) = \frac{[\exp(z)+\exp(-z)]\,\partial_z[\exp(z)-\exp(-z)] - [\exp(z)-\exp(-z)]\,\partial_z[\exp(z)+\exp(-z)]}{[\exp(z)+\exp(-z)]^2} = \frac{[\exp(z)+\exp(-z)]^2 - [\exp(z)-\exp(-z)]^2}{[\exp(z)+\exp(-z)]^2}$$

$$= 1 - \tanh^2(z)$$

$$\partial_z \tanh(z) = [1 - \tanh(z)] \odot [1 + \tanh(z)]$$

③ Softmax — $\varphi(z) = \frac{\exp(z)}{\sum_i^C \exp(z_i)} = \frac{\exp(z - \max(z))}{\sum_i^C \exp(z_i - \max(z))}$ ← out of all logits

where $C \equiv$ number of classes

Numerically stable (when $\exp(z)$ is large)

Derivative of Softmax

Given that softmax depends on all of the input raw logits, taking the derivative becomes not so straightforward. Therefore, the derivative is partial when calculating for a single logit, so we have to consider the Jacobian Matrix while talking about the first order partials.

$$J \equiv \begin{pmatrix} \partial_{z_1}\varphi_1 & \partial_{z_2}\varphi_1 & \cdots & \partial_{z_c}\varphi_1 \\ \partial_{z_1}\varphi_2 & \partial_{z_2}\varphi_2 & & \\ \vdots & & \ddots & \vdots \\ \partial_{z_1}\varphi_c & \cdots & & \partial_{z_c}\varphi_c \end{pmatrix} \quad \text{where } \varphi_i = \frac{\exp(z_i - \max(z))}{\sum_k^c \exp(z_k - \max(z))}$$

constant, so ignoring this for derivation of the Derivative.

Derivative of 'Log' $\Rightarrow \partial_{z_j} \log(\varphi_i) = \frac{1}{\varphi_i}\partial_{z_j}\varphi_i$

$$\partial_{z_j}\varphi_i = \varphi_i \cdot \partial_{z_j}\log(\varphi_i)$$

$$\frac{\partial}{\partial z_j}\left(\exp z_1 + \cdots + \exp z_j + \cdots + \exp z_c\right) = (0 + \cdots + \exp z_j + \cdots + 0) = \exp z_j$$

Now, $\log(\varphi_i) = z_i - \log\left(\sum_k^c \exp(z_k)\right)$

$$\rightarrow \partial_{z_j}\log(\varphi_i) = \partial_{z_j} z_i - \partial_{z_j}\log\left(\sum_k^c \exp(z_k)\right) = \delta_{ij} - \frac{1}{\sum_k^c \exp(z_k)}\left(\partial_{z_j}\sum_k^c \exp(z_k)\right) = \delta_{ij} - \frac{e^{z_j}}{\sum_k^c e^{z_k}}$$

φ_j

$$\therefore \partial_{z_j}\varphi_i = \varphi_i \cdot (\delta_{ij} - \varphi_j)$$

This will come in handy while backpropagation, when the softmax layer stands trial against the actual encoded output classes by means of Categorical Cross Entropy as the loss/cost function

Loss Functions

① Mean Squared Error — $C(a^L) = \frac{1}{2n}\sum_i\left[(y-a^L)^2\right]_i$

↓ Derivative

$$\partial_{a^L} C = \frac{1}{2n}\sum_i^n\left[2(y-a^L)(-1)\right]_i = \frac{-1}{n}\sum_i^n\left[y-a^L\right]_i$$

$$\delta^L = \partial_{a^L} C \odot \partial_{z^L} a^L = \frac{-1}{n}\sum_i^n\left[(y-a^L)\odot\partial_{z^L}a^L\right]_i$$

② Binary Cross Entropy — $C(a^L) = \frac{-1}{n}\sum_i^n\left[y\ln a^L + (1-y)\ln(1-a^L)\right]_i$, where n = number of training instances

↓ Derivative

$$\partial_{a^L} C = \frac{-1}{n}\sum_i^n\left[\frac{y}{a^L}(1) + \frac{1-y}{1-a^L}(-1)\right]_i = \frac{-1}{n}\sum_i^n\left[\frac{y}{a^L} - \frac{1-y}{1-a^L}\right]_i = \frac{-1}{n}\sum_i^n\left[\frac{y - ya^L - a^L + a^Ly}{a^L(1-a^L)}\right]_i$$

$$= \frac{-1}{n}\sum_i^n\left[\frac{y-a^L}{\partial_{z^L}a^L}\right]_i, \quad \text{here } a^L \equiv \text{Sigmoid}$$

$$\delta^L = \partial_{a^L} C \odot \partial_{z^L} a^L = \frac{-1}{n}\sum_i^n\left[y-a^L\right]_i$$

③ Categorical Cross Entropy — $C(a^L) = \frac{-1}{n}\sum_i^n\left[\sum_k^c y_k \ln a^L_k\right]_i$

↓ Derivative

$$\partial_{a^L_j} C = \frac{-1}{n}\sum_i^n\left[\sum_k^c \frac{y_k}{a^L_k}\,\partial_{z^L_j} a^L_k\right], \quad \text{here } a^L \equiv \text{Softmax}$$

$$\delta^L_j = \partial_{a^L_j} C \cdot \partial_{z^L_j} a^L_j = \frac{-1}{n}\sum_i^n\left[\sum_k^c \frac{y_k}{a^L_k}\partial_{a^L_j}a^L_k \cdot \partial_{z^L_j}a^L_j\right]_i = \frac{-1}{n}\sum_i^n\left[\sum_k^c \frac{y_k}{a^L_k}\,\partial_{z^L_j}a^L_k\right]_i = \frac{-1}{n}\sum_i^n\left[\sum_k^c \frac{y_k}{a^L_k}\,a^L_k(\delta_{kj}-a^L_j)\right]_i = \frac{-1}{n}\sum_i^n\left[\sum_k^c y_k\delta_{kj} - a^L_j\sum_k^c y_k\right]_i = \frac{-1}{n}\sum_i^n\left[y_j - a^L_j\right]_i$$

$$\delta^L = \frac{-1}{n}\sum_i^n\left[y-a^L\right]_i$$

Initialization Step

Given architechture → [784, 40, 30, 20, 10]

$\text{biases}_{\text{shape}} = [40\times1,\ 30\times1,\ 20\times1,\ 10\times1]$

$\text{weights}_{\text{shape}} = [40\times784,\ 30\times40,\ 20\times30,\ 10\times20]$

Forward Propagation

Assume a batch contains 100 instances of flattened images

After processing the data & getting the batches, the forward pass starts. We need to store the logits & activations of each layer (to be used in BP) & update this memory at each epoch.

$\text{logits\_history} = [\_, \_, \_, \_]$

$\text{activations\_history} = [\_, \_, \_, \_]$

We have the input, weights & biases, choice of activation and a place to store the computations.

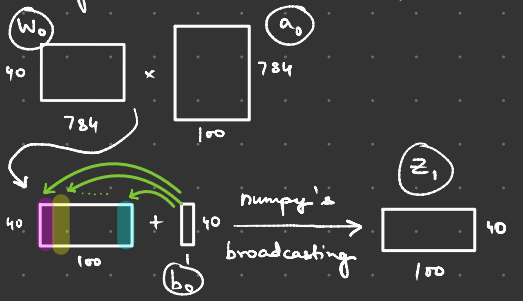
"Better Shape is all you need !!"

Input = $x \to (100, 784) \xrightarrow[x^T]{\text{reshape}} (784, 100)$ → a_0

Weight = $W_0 \to (40, 784)$

Bias = $b_0 \to (40, 1)$

Logit = $z_1 = W_0 a_0 + b_0 \to (40, 100)$



Activation = $a_1 = \sigma(z_1) \to (40, 100)$

activations & logits history = $[(40,100), \_, \_, \_]$

$z_2 = W_1 a_1 + b_1 \to (30, 100)$

$(30, 40) \times (40, 100) + (30, 1)$

$(30, 100) + (30, 1)$

$(30, 100)$

so on → $[(40,100), (30,100), (20,100), (10,100)]$ = logit_history

for all layers

The input batch

$[(784,100)] + [(40,100), (30,100), (20,100), (10,100)]$ = activation_history

Activation is just an element-wise operation over the logit tensors.

This is useful in completing last step of Back propagation

At the last layer, softmax will be used

$a(z_4) \rightarrow (10, 100)$

$$a(z) = \frac{\exp(z - \max(z))}{\sum \exp(z - \max(z))}$$

→ The 'max' & 'sum' operations are performed for each column separately, because each column is a different data vector in the instance space of all images

This operation would reduce the dimension if used by default. So set → keepdims = True

axis = 0

Backpropagation

Four Equations of Backpropagation → $\cdot \delta^L = \nabla_a C \odot \sigma'(z^L)$ $\cdot \delta^l = \left((w^{l+1})^T \delta^{l+1}\right) \odot \sigma'(z^l)$ $\cdot \frac{\partial C}{\partial b^l} = \delta^l$ $\cdot \frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^T$

(Element-wise) Hadamard product

To store the quantities: $\frac{\partial C}{\partial b^l}$ & $\frac{\partial C}{\partial w^l}$, we need to initialize empty lists while making sure to generate tensors of shape just like the bias & weight so that we can cross check any shape errors in matrix operations.

grad_biases = [40×1, 30×1, 20×1, 10×1] of zeros

grad_weights = [40×784, 30×40, 20×30, 10×20] of zeros

Now, the process starts in reverse, ∴ we start with actual output (encoded) vs last activation (prediction)

(100, 10) (10, 100)

delta = −(actual_output.T − activation_history[-1]) → (10, 100)

grad_biases[-1] = $\begin{pmatrix} \Sigma \text{ delta}[0] \\ \vdots \\ \Sigma \text{ delta}[9] \end{pmatrix}$ → (10, 1) ≡ biases[-1]

Normalize these gradients by the Batch-size or the Accuracy during fit would not improve !!

grad_weights[-1] = delta (10, 100) * activation_history[-2].T (100, 20) → (10, 20) ≡ weights[-1]

(20, 100)

update delta to update at [-2]

delta = (weights[-1].T (20, 10) * delta (10, 100)) ⊙ σ'(logits[-2]) (20, 100) → (20, 100)

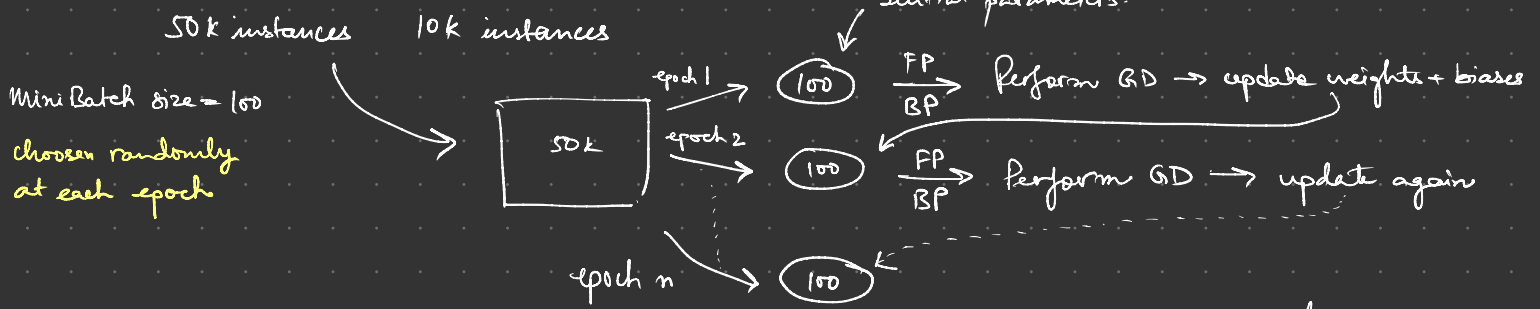
→ & so on till the input layer

Data Splitting and Mini-Batch Gradient Descent

$X_{shape} = (60k, 784)$, $Y_{shape} = (60k,)$

Train → 50k instances

Validation → 10k instances



To add momentum to the optimizer, we need to initialize a new list that remembers what were the gradients (scaled with momentum) in the previous epoch.

→ $velocity = [(momentum) * velocity] + [(1 - momentum) * grad]$

$weight/bias = weight/bias - [lr * velocity]$