Neural Networks

- Sasmit Datta

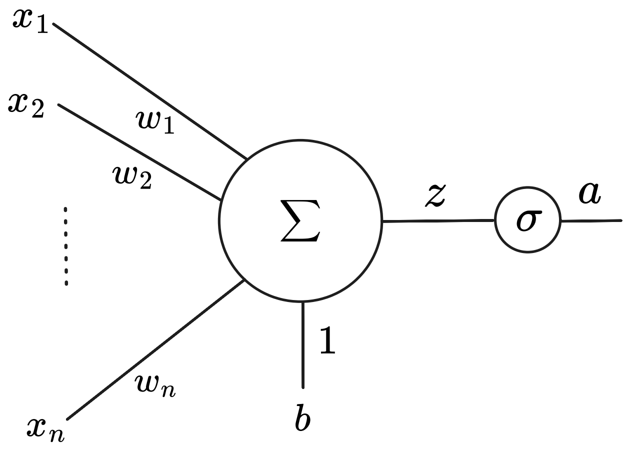
Introduction

Neural networks are powerful machine learning algorithms designed to recognise patterns and predict outcomes by learning from large datasets.

Traditional models like linear and logistic regression rely heavily on feature engineering, where raw data is transformed into meaningful features to improve performance. This involves selecting, modifying, and creating variables to help algorithms understand the data better.

However, neural networks reduce the need for manual feature engineering by automatically learning and extracting relevant features through multiple interconnected layers of neurons, enabling them to capture complex data relationships and improve accuracy in tasks.

Neuron



Very similar to linear and logistic regression, y represents the logits (values of the simple linear combination of the inputs) and hence we can write the equations above as

$$a = \sigma(z) = \sigma\left(\sum_{i=0}^{n} w_i x_i\right) = \sigma(\mathbf{w}^\intercal \mathbf{x})$$

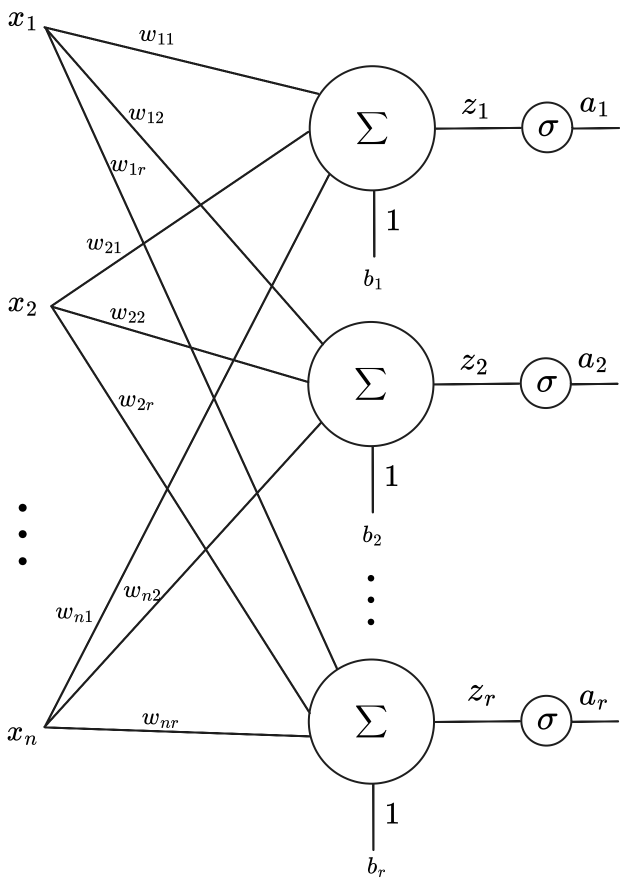
where,

- n is the number features of the input
- $\mathbf{x}$ is the input vector,
- $\mathbf{w}$ is the weight vector,
- i is the index of a particular scaler within a vector,
- b is the bias,
- and σ is the activation function.

So a good way to diagrammatically think about neurons is that:

- The lines are weights that get multiplied with their corresponding input element before streaming into the main summation "head".
- A single neuron outputs only a single number.

Layer of Neurons



The above diagram illustrates a layer of neurons in a neural network. Each neuron in the layer is connected to every input feature via a corresponding weight, and each neuron outputs a single value after processing the inputs through a linear combination followed by an activation function.

Notation

- $\mathbf{x}$: $[x_1, x_2, \ldots, x_n]$ represent the input features to the neural network.
- r : Output dimensions or number of neurons.
- w_{ij} : Each line connecting the input x_i to a neuron is associated with a weight w_{ij}, where i indexes the input and j indexes the neuron. For example, w_{11} is the weight connecting input x_1 to the first neuron, and w_{21} connects x_2 to the first neuron, and so on.
- $\mathbf{b}$: Each neuron has an associated bias term b_j, where j indexes the neuron.

So for an activation z_j

$$a_j = \sigma(z_j) = \sigma\left(\sum_{i=1}^{n} w_{ij}x_i + b_j\right)$$

Since we have r outputs for a single layer, we can simply have an output vector $\mathbf{a}$ with $r$dimensions. Also, instead of a weight vector, like we had in classical regression, we can have an entire weight matrix

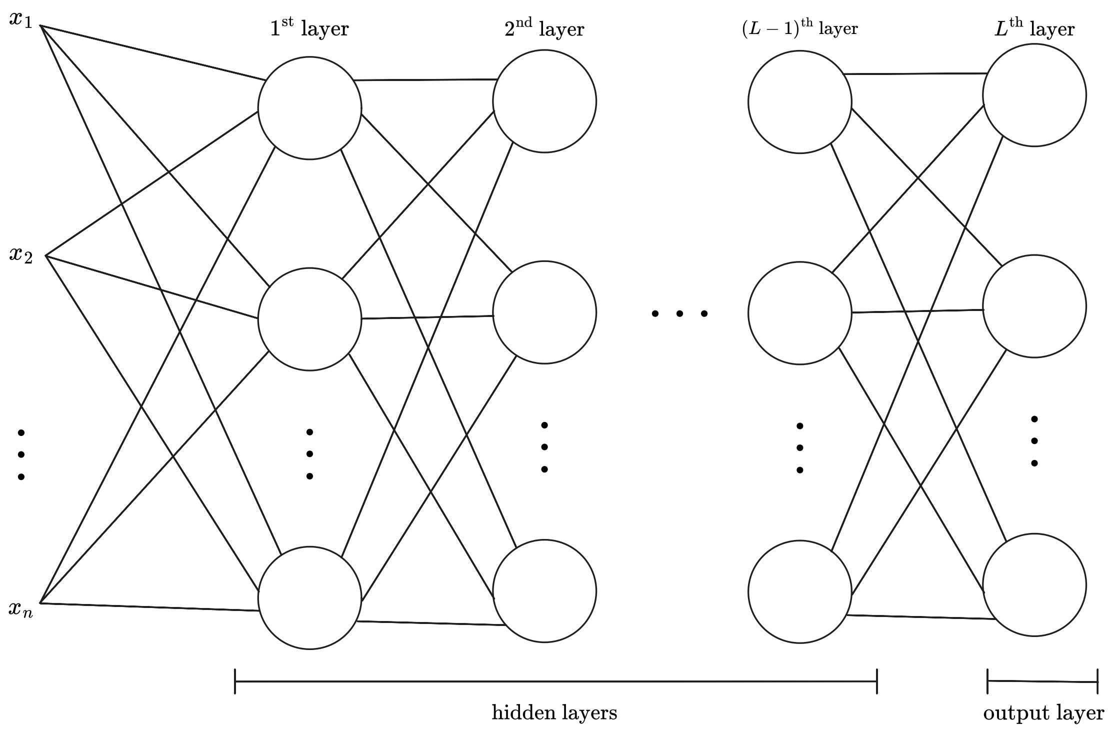
$\mathbf{W} \in \mathbb{R}^{n \times r}$. Furthermore, instead of a single bias term we can now extend it into a vector $\mathbf{b}$ with r dimensions. Hence the vectorised notation for the layer of a neural network can be written as

$$\mathbf{a} = \sigma\left(\mathbf{x}\mathbf{W} + \mathbf{b}\right)$$

where, $\mathbf{W} \in \mathbb{R}^{n \times r}$, $\mathbf{x} \in \mathbb{R}^{1 \times n}$, $\mathbf{b} \in \mathbb{R}^{1 \times r}$and $\mathbf{a} \in \mathbb{R}^{1 \times r}$.

Essentially what we doing through $\mathbf{W}$, $\mathbf{b}$ and σ is mapping an input vector $\mathbf{x}$with n dimensions to an output vector $\mathbf{a}$ with r dimensions. This is what "automatically learning and extracting relevant features through multiple interconnected layers of neurons" means. We, through these interconnected interactions with the different input features make the neural network learn useful intermediate features that might be helpful for the network to make a good prediction.

Neural Network



This diagram represents a multi-layer neural network, also known as a deep neural network. The network consists of multiple layers, including an input layer, several hidden layers, and an output layer. (Assume all the neurons have a bias term being added to the output and also being passed through an activation σ).

A hidden layer is any layer between the input and output layers. The term "hidden" refers to the fact that the true values of these layers are not part of the input or output data. Hidden layers apply transformations to the inputs using weights, biases, and activation functions to extract and learn patterns from the data.

Also note that the output layer can be of any dimension which is dependant on the problem we are trying to solve.

Notation

- $\mathbf{x} \in \mathbb{R}^{1 \times n}$: $x_1, x_2, \ldots, x_n$ represent the input features to the neural network.

- l : The index of the current layer. The input layer is considered the zeroth layer ($l = 0$).
- $\mathbf{W}^{[l]} \in \mathbb{R}^{n_{l-1} \times n_l}$: The weight matrix connecting layer $l - 1$ to layer l. Here, n_{l-1} is the number of neurons in layer $l - 1$ and n_l is the number of neurons in layer l. For instance, $\mathbf{W}^{[1]}$ would connect the input layer (zeroth layer) to the first hidden layer.
- $\mathbf{b}^{[l]} \in \mathbb{R}^{1 \times n_l}$: The bias vector for the l-th layer, where n_l is the number of neurons in layer l.
- $\mathbf{a}^{[l]} \in \mathbb{R}^{1 \times n_l}$: The activation output of the layer l after applying the activation function to the weighted sum of inputs plus the bias.

Therefore, expression for the output of the l-th layer of a neural neural network will look like:

$$\mathbf{a}^{[l]} = \sigma\left(\mathbf{z}^{[l]}\right) = \sigma\left(\mathbf{a}^{[l-1]}\mathbf{W}^{[l]} + \mathbf{b}^{[l]}\right)$$

Working with Batches

Like before, since we will be working with batches of size m. So we can modify the above equations as

$$\mathbf{A}^{[l]} = \sigma\left(\mathbf{Z}^{[l]}\right) = \sigma\left(\mathbf{a}^{[l-1]}\mathbf{W}^{[l]} + \mathbf{b}^{[l]}\right)$$

where,

- $\mathbf{X} \in \mathbb{R}^{m \times n}$
- $\mathbf{W}^{[l]} \in \mathbb{R}^{n_{l-1} \times n_l}$
- $\mathbf{A}^{[l]} \in \mathbb{R}^{m \times n_l}$
- $\mathbf{Z}^{[l]} \in \mathbb{R}^{m \times n_l}$
- $\mathbf{b}^{[l]} \in \mathbb{R}^{1 \times n_l}$

(If you work out the math, the shapes match up).

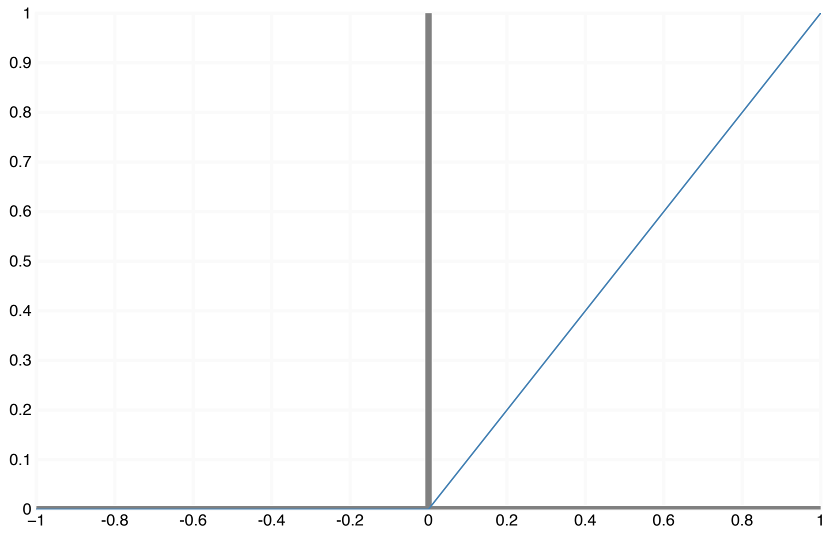
Activation Functions

Neural networks are used when we want to approximate complex functions, which maybe non-linear nature. Hence, just performing simple linear transformations ($\mathbf{A}^{[l]} = \mathbf{Z}^{[l]}$) won't cut it as at the end of the day, even with a neural network with multiple layers we are just performing a fancy linear regression. Hence, we introduce activation functions σ to have a non-linear system.

ReLU (Rectified Linear Unit)

The ReLU activation function is defined as:

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



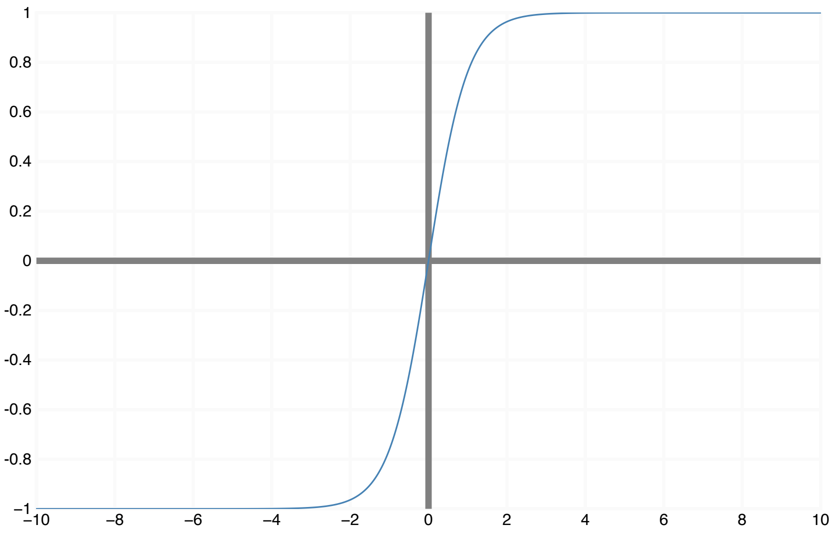
Derivative:

$$\mathrm{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Tanh (Hyperbolic Tangent)

The tanh activation function is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Derivative:

$$\tanh'(x) = 1 - \tanh^2(x)$$

To derive the derivative of the hyperbolic tangent function, we start with its definition:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Let $u = e^x$ and $v = e^{-x}$:

$$\tanh(x) = \frac{u - v}{u + v}$$

Differentiate $\tanh(x)$ with respect to x :

$$\tanh'(x) = \frac{(u' - v')(u + v) - (u - v)(u' + v')}{(u + v)^2}$$

Since $u = e^x$ and $v = e^{-x}$:

$$u' = e^x = u$$

$$v' = -e^{-x} = -v$$

Substitute u and v back into the derivative:

$$\tanh'(x) = \frac{(u - (-v))(u + v) - (u - v)(u - v)}{(u + v)^2}$$

Simplify the numerator:

$$\tanh'(x) = \frac{4uv}{(u + v)^2}$$

Since $u = e^x$ and $v = e^{-x}$:

$$\tanh'(x) = \frac{4}{(e^x + e^{-x})^2}$$

Now we know that:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Let $y = \tanh(x)$, then:

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

We also know that:

$$\operatorname{sech}^2(x) = \frac{4}{(e^x + e^{-x})^2}$$

Since $\operatorname{sech}(x) = \frac{2}{e^x + e^{-x}}$, and therefore:

$$\tanh'(x) = 1 - \tanh^2(x)$$

Forward Propagation

Forward propagation, often simply referred to as "forward pass," is the process by which input data is passed through a neural network to obtain an output. It iteratively follows the equation

$$\mathbf{A}^{[l]} = \sigma\left(\mathbf{Z}^{[l]}\right) = \sigma\left(\mathbf{A}^{[l-1]}\mathbf{W}^{[l]} + \mathbf{b}^{[l]}\right)$$

until $l = L$, where L is the last layer or the output layer of the neural network. Finally, we compute the loss $\mathcal{L}$

$$\mathcal{L}(\mathbf{Y}, \hat{\mathbf{Y}}) \in \mathbb{R}$$

where,

- $\mathbf{Y} \in \mathbb{R}^{m \times n_L}$ is the ground truth labels, n_L is the dimensions of the output layer and m is the batch size.
- $\hat{\mathbf{Y}} \in \mathbb{R}^{m \times n_L} = \mathbf{A}^{[\mathbf{L}]} \in \mathbb{R}^{m \times n_L}$ is the prediction made by the neural network.

Backward Propagation

Backward propagation, often simply referred to as "backward pass," is the process by which we calculate the gradients of our parameters $\mathbf{W}^{[l]} \in \mathbb{R}^{n_{l-1} \times n_l}$ and $\mathbf{b}^{[l]} \in \mathbb{R}^{1 \times n_l}$ to perform gradient descent and optimise them. Before we dive in to the mechanisms of how we do this, a quick refresher of the derivate of matrix multiplication.

Let's consider a matrix multiplication:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}$$

Now let's assume $\mathbf{C} \in \mathbb{R}^{m \times n}$ eventually converges into a loss $\mathcal{L} \in \mathbb{R}$. Therefore there exists $\partial \mathcal{L} / \partial \mathbf{C} \in \mathbb{R}^{m \times n}$. Hence, the gradients of $\mathcal{L}$ with respect to $\mathbf{A}$ and $\mathbf{B}$ are

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \frac{\partial \mathcal{L}}{\partial \mathbf{C}} \mathbf{B}^{\intercal}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{B}} = \mathbf{A}^{\intercal} \frac{\partial \mathcal{L}}{\partial \mathbf{C}}$$

For a given layer l, our goal is to calculate the following gradients:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} \in \mathbb{R}^{n_{l-1} \times n_l}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} \in \mathbb{R}^{1 \times n_l}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}^{[l-1]}} \in \mathbb{R}^{m \times n_{l-1}}$$

We cant to calculate $\partial \mathcal{L} / \partial \mathbf{A}^{[l-1]}$ and not $\partial \mathcal{L} / \partial \mathbf{A}^{[l]}$ is because we already have calculated the latter for the $(l+1)$-th layer (just like we calculated $\partial \mathcal{L} / \partial \mathbf{C}$).

For each layer l:

- Calculate gradients with respect to $\mathbf{Z}^{[l]}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{[l]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{A}^{[l]}} \odot \sigma' \left(\mathbf{Z}^{[l]} \right)$$

- Calculate gradients with respect to $\mathbf{W}^{[l]}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}} = \left(\mathbf{A}^{[l-1]} \right)^{\top} \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{[l]}}$$

- Calculate gradients with respect to $\mathbf{b}^{[l]}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} = \sum_{i=1}^{m} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{[l]}} \right)_i$$

- **Propagate** the gradient **back** to the previous Layer :

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}^{[l-1]}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{[l]}} \left(\mathbf{W}^{[l]} \right)^{\top}$$

Finally, after we propagate through all the previous layers and calculate the gradients of the weights and biases of those layers, we finally make an update:

$$\mathbf{W}^{[l]} := \mathbf{W}^{[l]} - \eta\left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[l]}}\right)$$

$$\mathbf{b}^{[l]} := \mathbf{b}^{[l]} - \eta\left(\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}}\right)$$

Multi-class Classification

Multi-class classification is a type of classification task where the goal is to predict the class label of an instance from three or more possible classes. Unlike binary classification, which deals with two classes, multi-class classification handles more complex scenarios where multiple outcomes are possible. For example: handwritten digit recognition. The key differences in terms of outputs and activations are

- **Binary Classification**: Uses a single neuron with a sigmoid activation function to output a probability between 0 and 1.
- **Multiclass Classification**: Outputs a probability distribution across all classes. So it will use K neurons for the output, where K is the total number of classes. Hence, we chose the class which has the highest probability as the model's prediction. Mathematically we are calculating the following

$$P(y = k|\mathbf{x})$$

where $k = 1, 2, \ldots, K$

Softmax Function

The softmax function is a generalisation of the logistic sigmoid function that is used in multiclass classification. It is used to convert the raw output scores (logits) from a neural network's final layer into a probability distribution over multiple classes.

Definition

Given an input vector $\mathbf{z} = [z_1, z_2, \ldots, z_K]$, where K is the number of classes, the softmax function computes the probability $P(y = k|\mathbf{z})$ that the input belongs to each class k. The softmax function is defined as follows:

$$P(y = k|\mathbf{z}) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

Here, e^{z_k} is the exponential of the k-th logit (score) z_k, and the denominator is the sum of exponentials of all logits. A question arises from here - why exponentiate all the logits before normalising? why not just directly normalise? A good question for the reader to ponder on.

Example

Consider a neural network that outputs logits $\mathbf{z} = [z_1, z_2, z_3]$ for three classes. The softmax function would convert these logits into probabilities as follows:

$$P(y = 1|\mathbf{z}) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y=2|\mathbf{z}) = \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}}$$

$$P(y=3|\mathbf{z}) = \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}}$$

Gradient

Unlike other activation functions, we can't individually calculate the derivatives as all the input elements contribute to the computation of the all output elements. Let's denote the softmax output for class k as s_k:

$$s_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

To find the derivative $\partial S_k/\partial z_i$, we consider two cases: when $k = i$ and when $k \neq i$.

1. **Case 1: $k = i$.**

   $$\frac{\partial s_k}{\partial z_k} = \frac{\partial}{\partial z_k}\left(\frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}\right)$$

   Using the quotient rule:

   $$\frac{\partial s_k}{\partial z_k} = \frac{e^{z_k}(\sum_{j=1}^{K} e^{z_j}) - e^{z_k}e^{z_k}}{(\sum_{j=1}^{K} e^{z_j})^2}$$

   Simplifying the expression:

   $$\frac{\partial s_k}{\partial z_k} = \frac{e^{z_k}\left(\sum_{j=1}^{K} e^{z_j} - e^{z_k}\right)}{(\sum_{j=1}^{K} e^{z_j})^2}$$

   $$\frac{\partial s_k}{\partial z_k} = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}\left(1 - \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}\right)$$

   Using s_k:

   $$\frac{\partial s_k}{\partial z_k} = s_k(1 - s_k)$$

2. **Case 2: $k \neq i$**

   $$\frac{\partial S_k}{\partial z_i} = \frac{\partial}{\partial z_i}\left(\frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}\right)$$

   Using the quotient rule:

   $$\frac{\partial S_k}{\partial z_i} = \frac{0 \cdot (\sum_{j=1}^{K} e^{z_j}) - e^{z_k}e^{z_i}}{(\sum_{j=1}^{K} e^{z_j})^2}$$

   Simplifying the expression:

   $$\frac{\partial S_k}{\partial z_i} = -\frac{e^{z_k}e^{z_i}}{(\sum_{j=1}^{K} e^{z_j})^2}$$

   Using s_k and s_i:

$$\frac{\partial S_k}{\partial z_i} = -s_k s_i$$

Combining the results from both cases, we get:

$$\frac{\partial s_k}{\partial z_i} = \begin{cases} s_k(1 - s_k) & \text{if } k = i \\ -s_k s_i & \text{if } k \neq i \end{cases}$$

Since we are computing the gradient of a vector with respect to another vector, we will get a Jacobian matrix with dimensions $K \times K$.

The Jacobian matrix J of the softmax function can be written in matrix form as follows:

$$J_{ki} = \frac{\partial s_k}{\partial z_i} = s_k(\delta_{ki} - s_i)$$

where δ_{ki} is the Kronecker delta, which is 1 if $k = i$ and 0 otherwise.

Computing $\partial \mathcal{L} / \partial \mathbf{Z}$

Now, let's see how we can get $\partial \mathcal{L} / \partial \mathbf{Z}$ through a softmax activation.

Let $\mathcal{L}$ be a loss function. The derivative of the loss with respect to the logits $\mathbf{Z}$ can be computed as follows. Here, $\mathbf{Z} \in \mathbb{R}^{m \times K}$, where m is the batch size and K is the number of classes. We need to compute:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{S}} \frac{\partial \mathbf{S}}{\partial \mathbf{Z}}$$

where $\mathbf{S} \in \mathbb{R}^{m \times K}$ is the softmax output.

First we compute $\partial \mathcal{L} / \partial \mathbf{S}$. For a loss function $\mathcal{L}(\mathbf{Y}, \mathbf{S})$, where $\mathbf{Y} \in \mathbb{R}^{m \times K}$ is the true label matrix and $\mathbf{S}$ is the softmax output matrix, we can compute the gradient:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{S}}$$

This depends on the specific form of the loss function. For now, let's denote it as $\mathbf{G}$:

$$\mathbf{G} = \frac{\partial \mathcal{L}}{\partial \mathbf{S}} \in \mathbb{R}^{m \times K}$$

Now the Jacobian won't be a matrix of shape $K \times K$ anymore. Since we are working with a batch of size m, our Jacobian will be a tensor $\mathbf{J} \in \mathbb{R}^{m \times K \times K}$.

$$\mathbf{J}_i = \operatorname{diag}(\mathbf{S}_i) - \mathbf{S}_i^\intercal \mathbf{S}_i$$

where:

- $\mathbf{S}_i \in \mathbb{R}^{1 \times K}$ is the vector sample i from the softmax output $\mathbf{S}$.
- $\operatorname{diag}(\mathbf{S}_i) \in \mathbb{R}^{1 \times K}$ places the elements of $\mathbf{S}_i$ in the diagonal of a matrix of shape $K \times K$.
- $\mathbf{J}_i \in \mathbb{R}^{K \times K}$ is the jacobian matrix for sample i.

Finally,

$$\left(\frac{\partial \mathcal{L}}{\partial \mathbf{Z}}\right)_i = \mathbf{G}_i \mathbf{J}_i$$

where,

- $(\partial \mathcal{L}/\partial \mathbf{Z})_i \in \mathbb{R}^{1\times K}$
- $\mathbf{G}_i \in \mathbb{R}^{1\times K}$ is a vector sample i from the softmax gradient output $\mathbf{G}$.

Therefore calculating the the above for all i's in m and concatenating them we get

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}} \in \mathbb{R}^{m\times K}$$

Clipping

While implementing softmax, since we are using exponentiation, values can get very large which can cause numerical overflow errors. So, we make an update to the formula.

$$P(y=k|\mathbf{z}) = \frac{e^{z_k - \max(\mathbf{z})}}{\sum_{j=1}^{K} e^{z_j - \max(\mathbf{z})}}$$

where,

- $\max(\mathbf{z})$ is the max element of the vector $\mathbf{z}$.

Now a few questions aris from here: Does clipping change the value of the softmax? Does it make in any changes in the gradient calculation? A few questions for the reader to ponder.

Cross-Entropy Loss

The cross-entropy loss is a commonly used loss function for classification tasks, particularly in conjunction with the softmax activation function. It measures the dissimilarity between the true labels and the predicted probabilities.

Given the true label vector $\mathbf{y} = [y_1, y_2, \ldots, y_K]$ and the predicted probability vector $\mathbf{s} = [s_1, s_2, \ldots, s_K]$ from the softmax function, the cross-entropy loss for a single instance is defined as:

$$\mathcal{L}(\mathbf{y}, \mathbf{S}) = -\sum_{k=1}^{K} y_k \log(s_k)$$

where:

- y_k is the true label (1 if the class is k, otherwise 0 - one hot encoding).
- s_k is the predicted probability for class k.

For a batch of size m, the average cross-entropy loss is:

$$\mathcal{L}(\mathbf{Y}, \mathbf{S}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} \mathbf{Y}_{ik} \log(\mathbf{S}_{ik})$$

where:

- $\mathbf{Y} \in \mathbb{R}^{m\times K}$ is the matrix of true labels.
- $\mathbf{S} \in \mathbb{R}^{m\times K}$ is the matrix of predicted probabilities from the softmax function.

Derivative

To perform backpropagation, we need to compute the derivative of the cross-entropy loss with respect to the logits $\mathbf{Z}$.

The partial derivative of the cross-entropy loss with respect to the softmax output $\mathbf{S}$ is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{S}} = -\frac{1}{m}\frac{\mathbf{Y}}{\mathbf{S}}$$

Softmax and Cross-Entropy Fusion

For a single example $\mathbf{z}$, the simplified forward calculation will look like

$$-\sum_{k=1}^{K} y_k \left(z_k - \log\left(\sum_{j=1}^{K} e^{z_j}\right)\right)$$

where:

- y_k is the k-th element of the vector $\mathbf{y}$
- z_k is the k-th element of the vector $\mathbf{z}$
  Let's derive the gradient of the cross-entropy loss with softmax activation, similar to the steps shown in the image.

Forward Calculation

$$\mathcal{L} = -z_y + \log\left(\sum_{j=1}^{K} e^{z_j}\right)$$

where z_y is the logit corresponding to the true class. For not the true class, it will 0.

Backward Calculation

We need to compute the gradient of the loss with respect to the logits z_k.

1. **Case 1:** when $k = y$ (true class)

   $$\frac{\partial \mathcal{L}}{\partial z_k} = \frac{\partial}{\partial z_k}\left(-z_k + \log\left(\sum_{j=1}^{K} e^{z_j}\right)\right)$$

   Using the chain rule:

   $$\frac{\partial \mathcal{L}}{\partial z_k} = -1 + \frac{\partial}{\partial z_k}\log\left(\sum_{j=1}^{K} e^{z_j}\right)$$

   Applying the derivative of the log function:

   $$\frac{\partial \mathcal{L}}{\partial z_k} = -1 + \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

   Since $s_k = e^{z_k}/(\sum_{j=1}^{K} e^{z_j})$ (softmax output):

$$\frac{\partial \mathcal{L}}{\partial z_k} = s_k - 1$$

2. **Case 2:** when $k \neq y$

$$\frac{\partial \mathcal{L}}{\partial z_k} = \frac{\partial}{\partial z_k}\left(\log\left(\sum_{j=1}^{K} e^{z_j}\right)\right)$$

   Using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial z_k} = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

   Again, since $s_k = e^{z_k}/(\sum_{j=1}^{K} e^{z_j})$:

$$\frac{\partial \mathcal{L}}{\partial z_k} = s_k$$

   Combining the results for $k = y$ and $k \neq y$, we get:

$$\frac{\partial \mathcal{L}}{\partial z_k} = s_k - y_k$$

   Hence, in matrix form, the gradient we get is

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}} = \frac{1}{m}(\mathbf{S} - \mathbf{Y})$$

   which is quite a simpler expression to compute than going through the gradient calculation in the softmax section.