Computer Vision

Introduction to neural networks

Course Instructor:
Dr. Suman Kumar Maji

Building Blocks: Neurons

- Neurons, also called perceptrons, the basic unit of a neural network.
- A neuron takes inputs, does some math with them and produces one output.
- Here's what a 2-input neuron (perceptron) looks like:

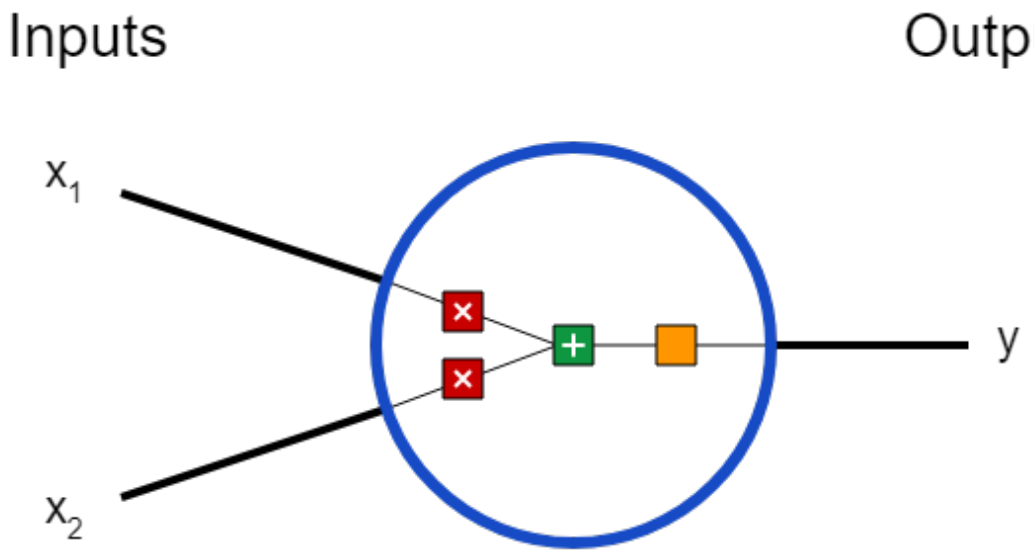


Image courtesy: `https://victorzhou.com/`

cont...

Three things are happening here.
First, each input is multiplied by a weight:

$$x_1 \rightarrow x_1 \cdot w_1$$

$$x_2 \rightarrow x_2 \cdot w_2$$

Next, all the weighted inputs are added together with a bias b:

$$(x_1 \cdot w_1) + (x_2 \cdot w_2) + b$$

Finally, the sum is passed through an activation function:

$$y = f(x_1 \cdot w_1 + x_2 \cdot w_2 + b)$$

Sigmoid function

The activation function turns an unbounded input into an output with a nice, predictable form. A commonly used activation function is the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

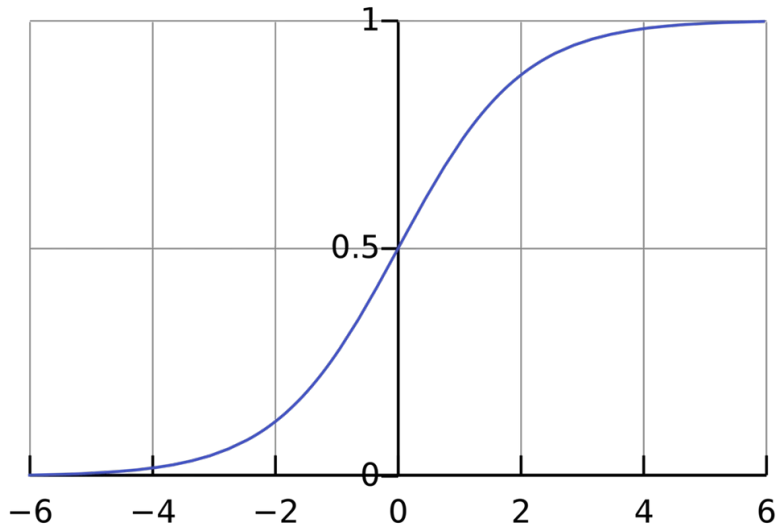


Image courtesy: `https://victorzhou.com/`

cont...

The sigmoid function only outputs numbers in the range $(0, 1)$. You can think of it as compressing $(-\infty, +\infty)$ to $(0, 1)$—big negative numbers become approximately 0, and big positive numbers become approximately 1.
The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

This function smoothly maps any real-valued number to a value between 0 and 1, making it useful for probability-based applications in machine learning.

A Simple Example

Assume we have a 2-input neuron that uses the sigmoid activation function and has the following parameters:

$$w = [0, 1]$$

$$b = 4$$

The weight vector $w = [0, 1]$ is just a way of writing:

$$w_1 = 0, \quad w_2 = 1$$

In vector form. Now, let's give the neuron an input of:

$$x = [2, 3]$$

We'll use the dot product to write things more concisely:

$$(w \cdot x) + b = ((w_1 \cdot x_1) + (w_2 \cdot x_2)) + b$$

$$= 0 \cdot 2 + 1 \cdot 3 + 4 = 7$$

cont...

Now, applying the activation function:

$$y = f(w \cdot x + b) = f(7) = 0.999$$

Thus, the neuron outputs **0.999** given the inputs $x = [2, 3]$.
That's it! This process of passing inputs forward to get an output is known as **feedforward**.

Some other activation functions

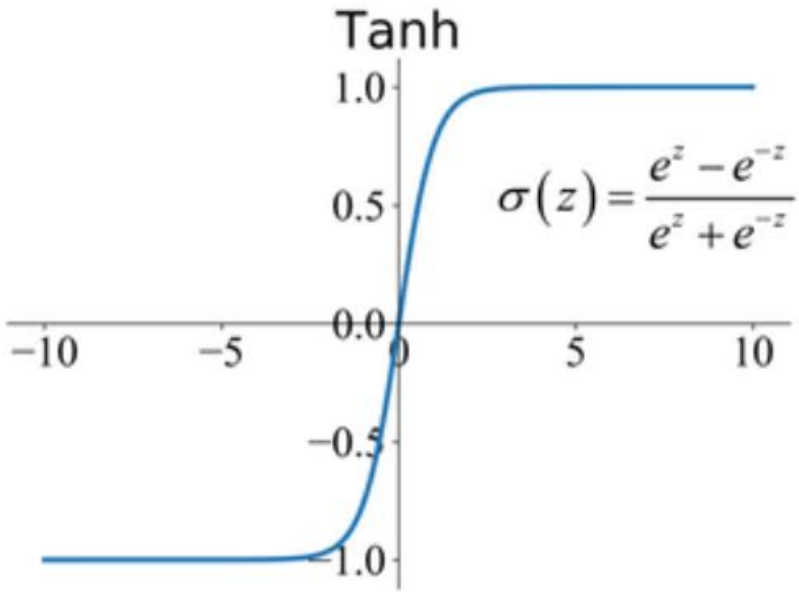
Some other activation functions are:

1. tanh
2. Relu

Tanh Function (Hyperbolic Tangent)

- Tanh function is very similar to the sigmoid/logistic activation function, and even has the same S-shape with the difference in output range of -1 to 1.
- In Tanh, the larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.

cont...



Mathematically it can be represented as:

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

---

image courtesy:
`https://www.v7labs.com/blog/neural-networks-activation-functions`

Advantages of using this activation function

- The output of the tanh activation function is zero-centered; hence we can easily map the output values as strongly negative, neutral, or strongly positive.
- Usually used in hidden layers of a neural network as its values lie between -1 to; therefore, the mean for the hidden layer comes out to be 0 or very close to it.
- It helps in centering the data and makes learning for the next layer much easier.

Gradient of Tanh function

Have a look at the gradient of the tanh activation function to understand its limitations.

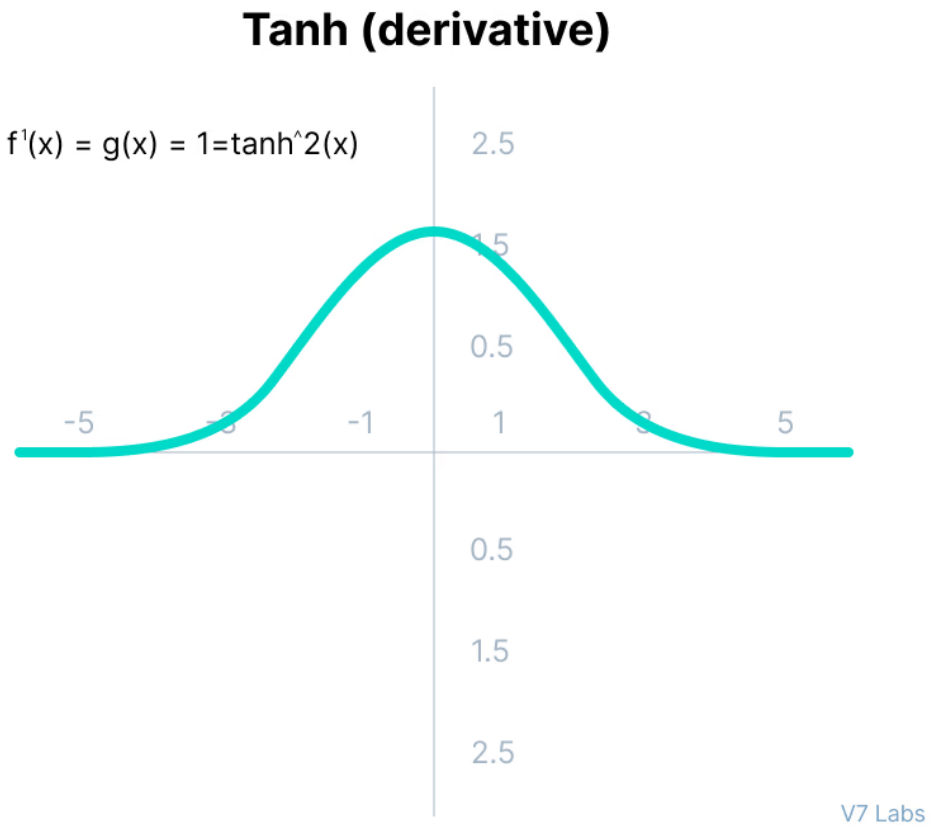


image courtesy:
https://www.v7labs.com/blog/neural-networks-activation-functions

Limiations of Tanh function

- As you can see — it also faces the problem of vanishing gradients similar to the sigmoid activation function.
- Plus the gradient of the tanh function is much steeper as compared to the sigmoid function.

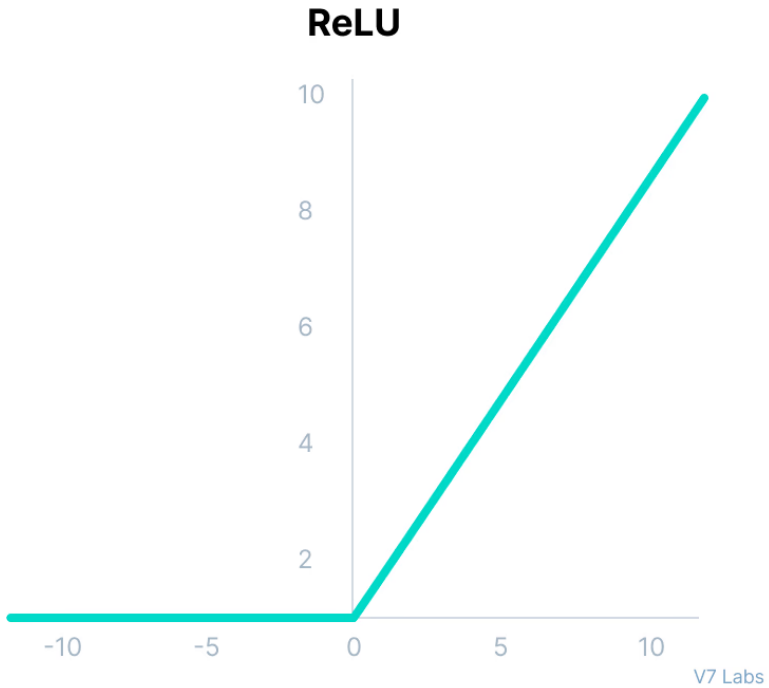
Note

Although both sigmoid and tanh face vanishing gradient issue, tanh is zero centered, and the gradients are not restricted to move in a certain direction. Therefore, in practice, tanh nonlinearity is always preferred to sigmoid nonlinearity.

ReLU Function (Rectified Linear Unit)

- ReLU stands for Rectified Linear Unit.
- Although it gives an impression of a linear function, ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient.
- The main catch here is that the ReLU function does not activate all the neurons at the same time.
- The neurons will only be deactivated if the output of the linear transformation is less than 0.
- If the input to a neuron is greater than or equal to zero, the output is equal to the input.
- If the input is negative, the output is zero.

ReLU graph



ReLU can be mathematically represented as:

$$f(x) = max(0, x);$$

---

image courtesy:
https://www.v7labs.com/blog/neural-networks-activation-functions

Advantages of using ReLU

- Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions.
- ReLU accelerates the convergence of gradient descent towards the global minimum of the loss function due to its linear, non-saturating property.

Limitations of ReLU function

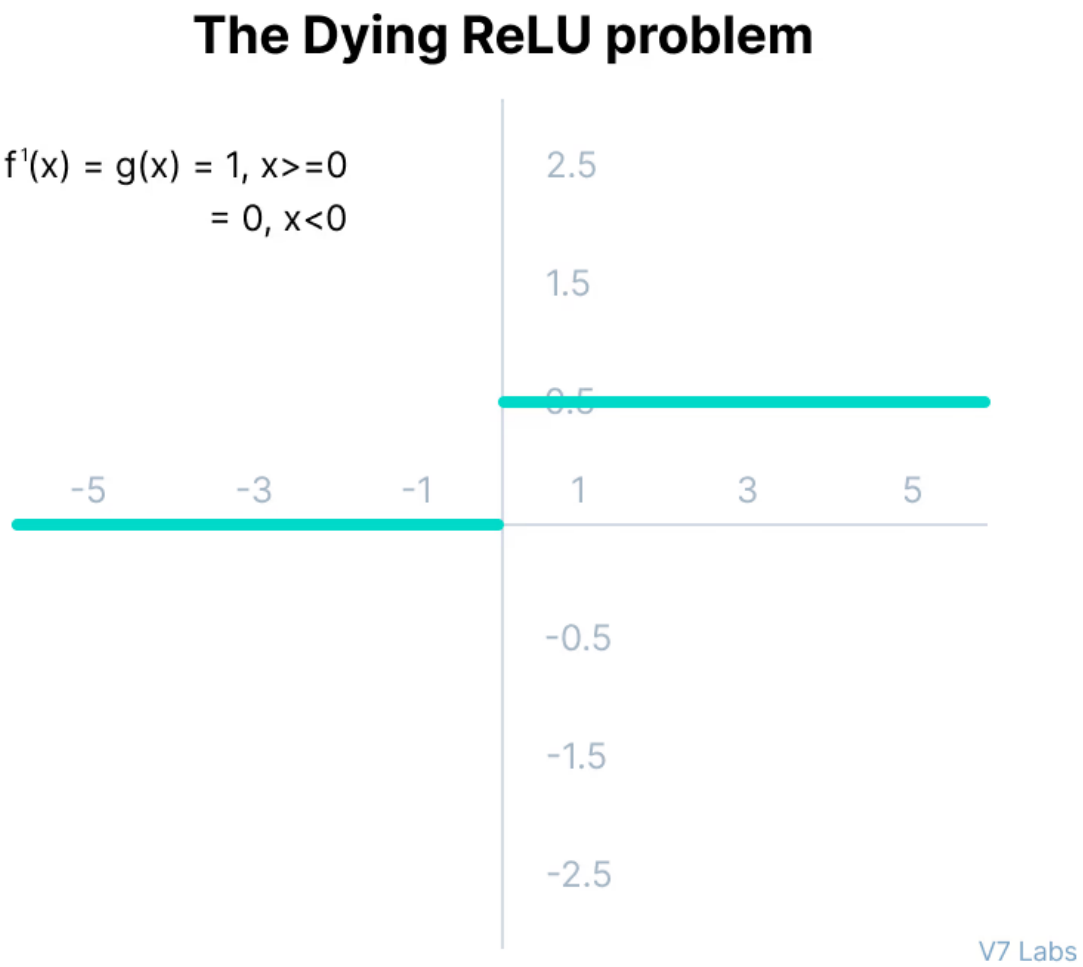


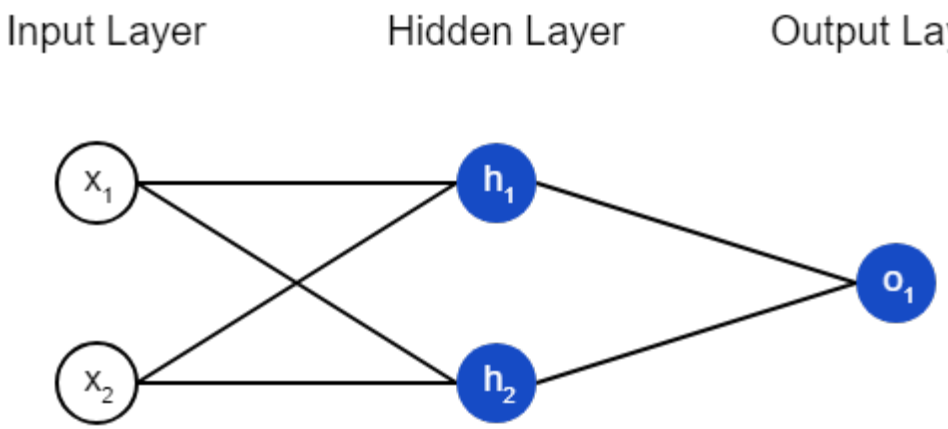
image courtesy:
https://www.v7labs.com/blog/neural-networks-activation-functions

The Dying ReLU problem

- The negative side of the graph makes the gradient value zero.
- Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated.
- This can create dead neurons which never get activated.
- All the negative input values become zero immediately, which decreases the model's ability to fit or train from the data properly.

Combining Neurons into a Neural Network

A neural network is nothing more than a bunch of neurons connected together. Here's what a simple neural network might look like:



- This network has 2 inputs, a hidden layer with 2 neurons (h_1 and h_2), and an output layer with 1 neuron (o_1).
- Notice that the inputs for o_1 are the outputs from h_1 and h_2—that's what makes this a network.
- A **hidden layer** is any layer between the input (first) layer and output (last) layer. There can be multiple hidden layers!

Image courtesy: `https://victorzhou.com/`

Multilayer Perceptron or Feed Forward Neural Network

- The previous network is an example of an MLP with two inputs and a single hidden layer.
- MLP can have multiple hidden layers.
- MLP learns to map inputs to outputs through a process of forward propagation and backpropagation.

MLP characteristics

1. Input Layer
2. Weighted connections and biases
3. Hidden Layers
4. Activation functions
5. Forward propagation
6. Output Layer
7. Backpropagation and learning.

MLP contd

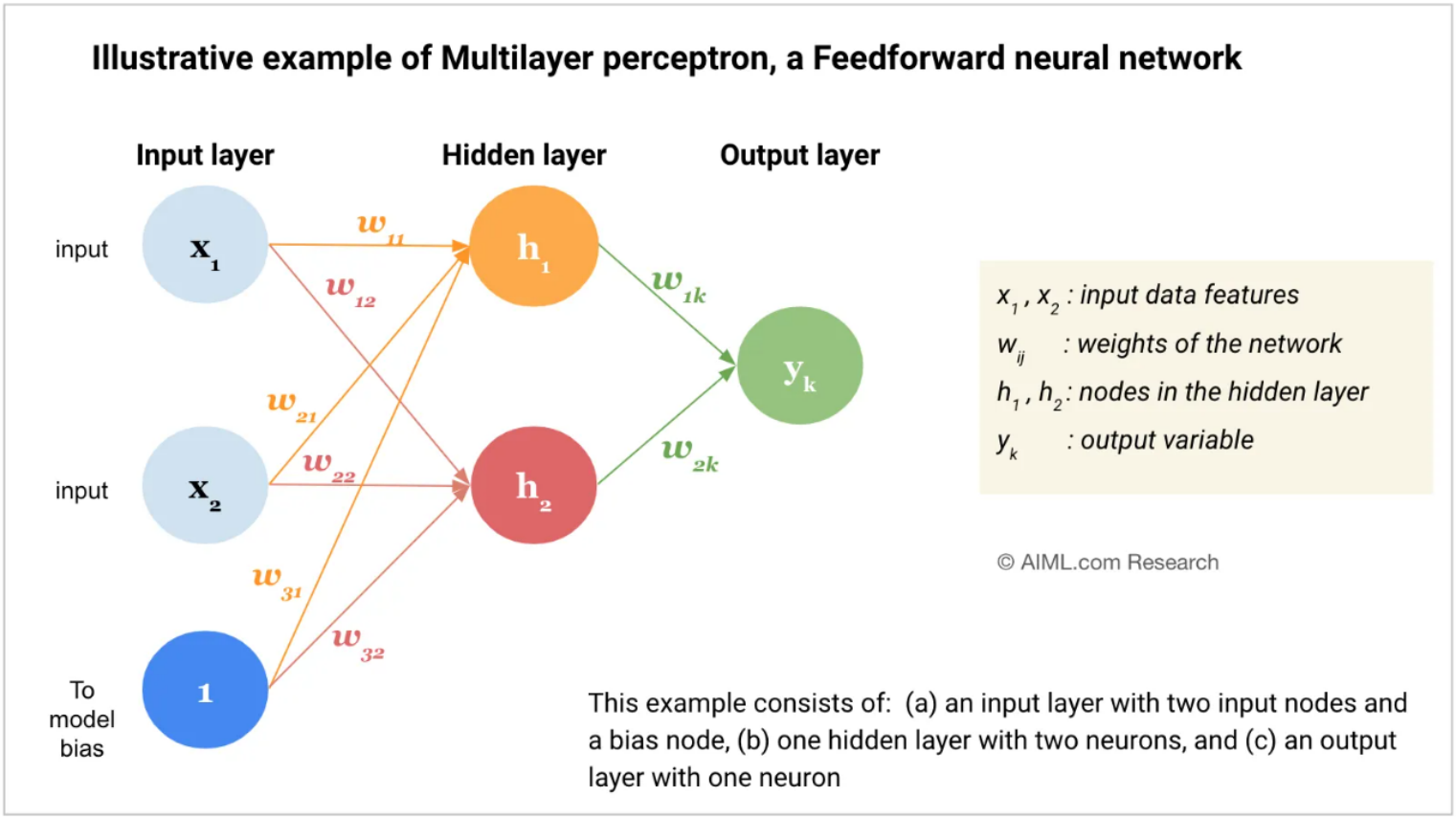
1. Input Layer

- The process starts with the input layer, which receives the input data.
- Each neuron in this layer represents a feature of the input data

2. Weighted Connections and Biases

- Connections between neurons have associated weights, which are learned during the training process.
- These weights determine the strength of the connections and play a crucial role in the networks ability to capture patterns in the data.
- In addition, each neuron, in the hidden and output layers has an associated bias term, which allows for fine-tuning and shifting the activation functions threshold.
- These weights and biases are parameters that the neural network learns during training.

cont...



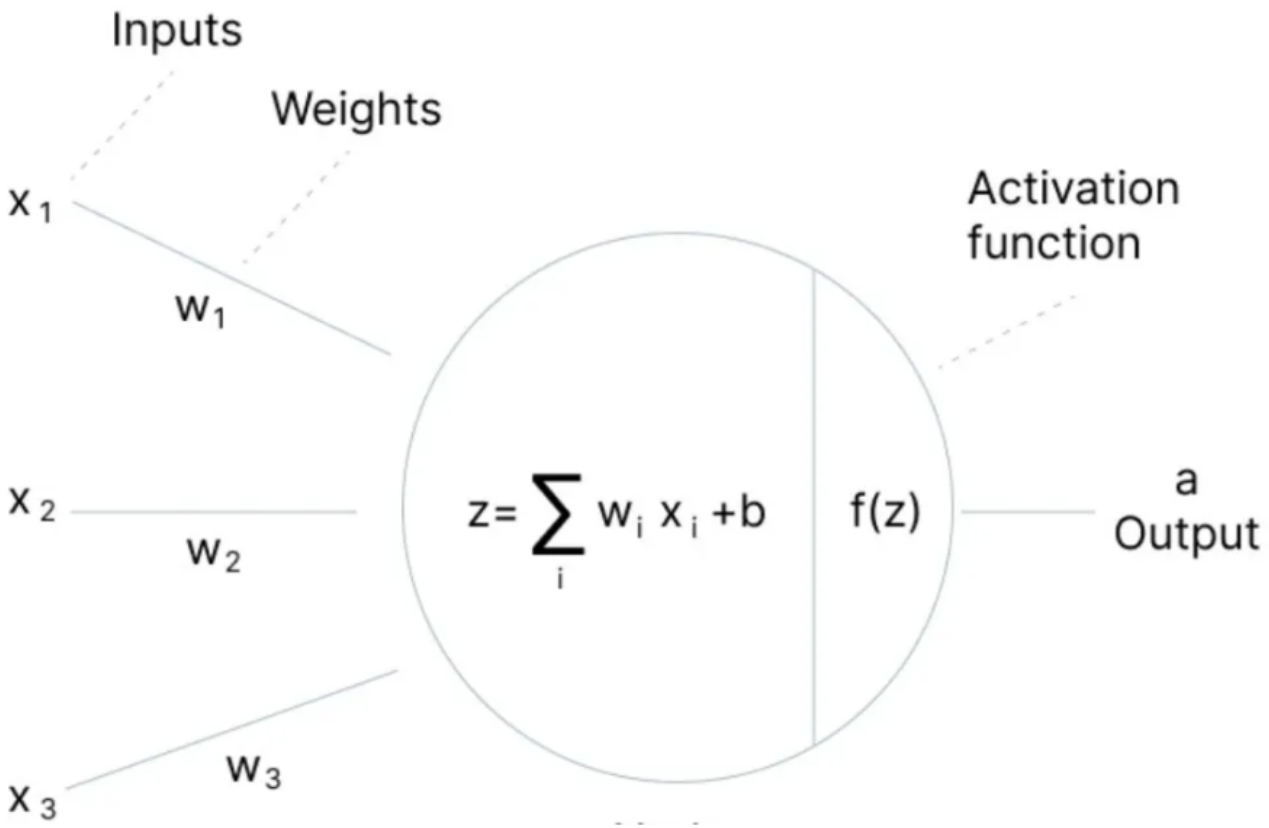
Title: Multilayer perceptron (MLP), a feedforward neural network
Source: AIML.com Research

image courtesy: `https://aiml.com/what-is-a-multilayer-perceptron-mlp/`

3. Hidden Layer

- After the input layer, there are one or more hidden layers.
- The neurons in these layers perform computations on the inputs.
- The output of each neuron is calculated by applying a weighted sum of its inputs (from the previous layer), adding a bias, and then passing this sum through an activation function.

cont...

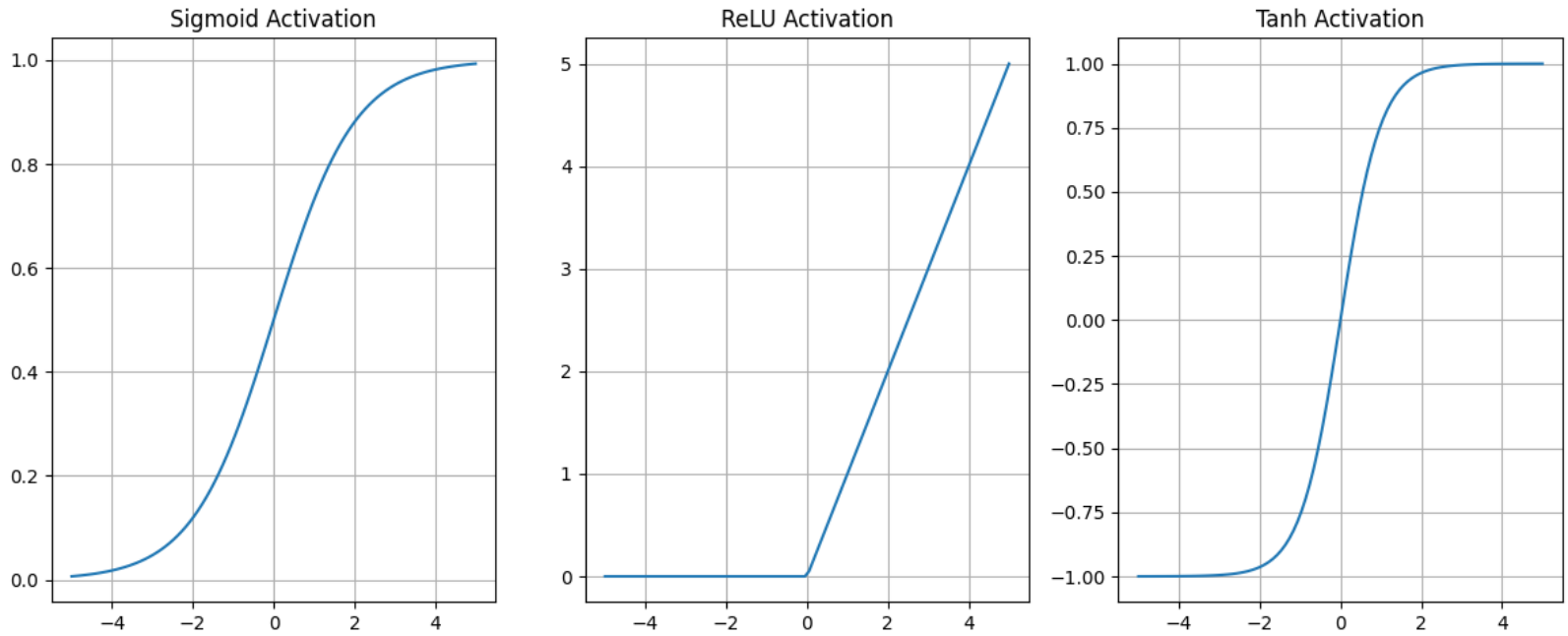


Depicting input and output to a neuron in hidden layers

image courtesy: `https://aiml.com/what-is-a-multilayer-perceptron-mlp/`

4. Activation functions

- The activation function is crucial as it introduces non-linearity into the model, allowing it to learn more complex patterns.
- Common activation functions include sigmoid, tanh, and ReLU (Rectied Linear Unit).



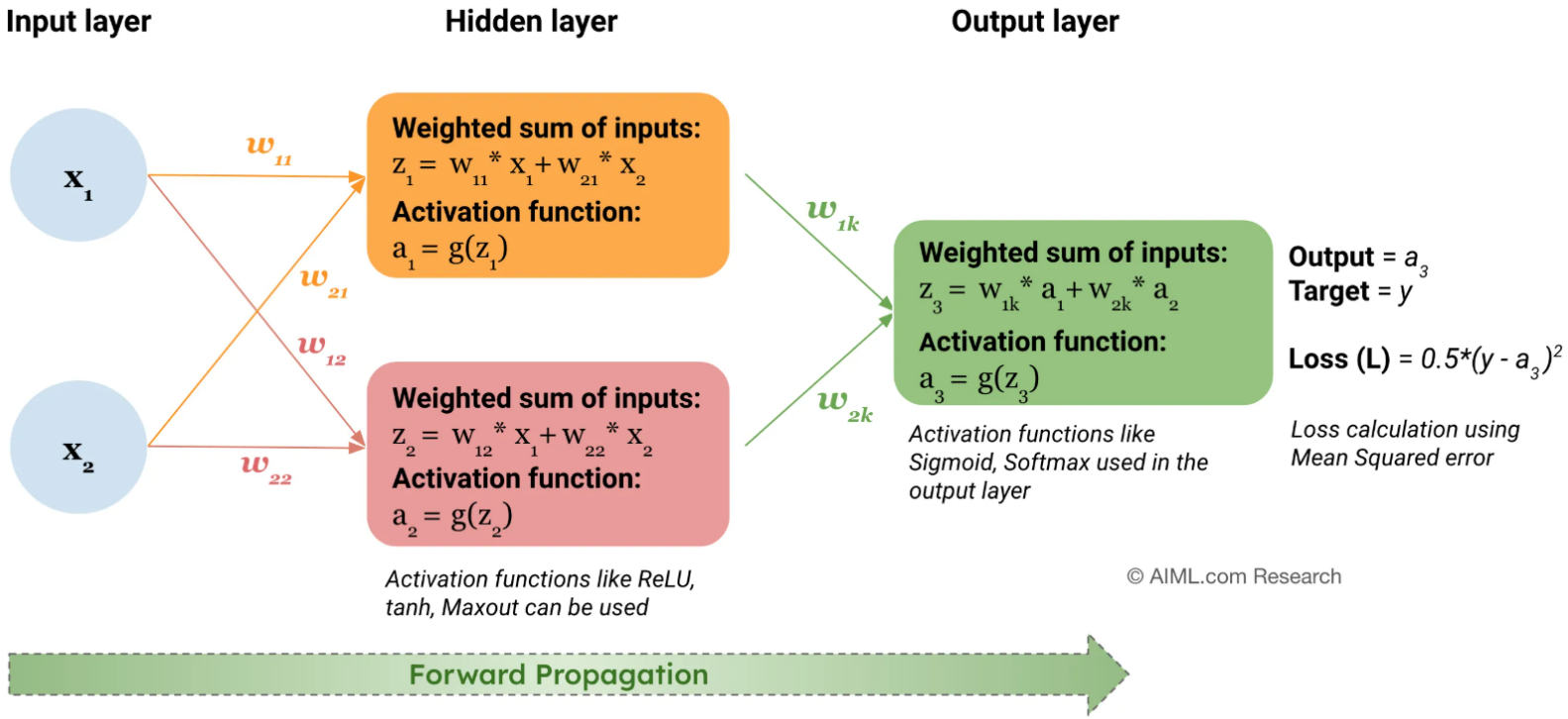
Some activation functions

image courtesy: https://python.plainenglish.io/activation-functions-and-loss-functions-in-deep-learning-differences-and-overl

5. Forward propogation

- The process from the input layer through the hidden layers to the output layer is called forward propagation.
- In each layer, the aforementioned steps (weighted sum, bias addition, activation function) are applied to compute the layers output.
- In an MLP, information flows in one direction, from the input layer through the hidden layers to the output layer.
- There are no feedback loops or recurrent connections (as in the case of RNN), hence the name feedforward architecture.

cont...



Forward propagation in a Neural Network (Source: AIML.com Research)
Note: bias term is not shown in the above diagram

image courtesy: `https://aiml.com/what-is-a-multilayer-perceptron-mlp/`

6. Output layer

- The final layer is the output layer. In a classification task, this layer often uses a softmax function if the task is multi-class classification, or a sigmoid function for binary classification.
- For regression tasks, no activation function might be applied in the output layer.

An Example: Feedforward

Let's use the network pictured above and assume all neurons have the same weights:

$$w = [0, 1]$$

the same bias:

$$b = 0$$

and the same sigmoid activation function. Let h_1, h_2, o_1 denote the outputs of the neurons they represent.

cont...

What happens if we pass in the input $x = [2, 3]$?

$$h_1 = h_2 = f(w \cdot x + b)$$

$$= f((0 \cdot 2) + (1 \cdot 3) + 0)$$

$$= f(3) = 0.9526$$

$$o_1 = f(w \cdot [h_1, h_2] + b)$$

$$= f((0 \cdot h_1) + (1 \cdot h_2) + 0)$$

$$= f(0.9526) = 0.7216$$

The output of the neural network for input $x = [2, 3]$ is **0.7216**.

cont...

- A neural network can have any number of layers with any number of neurons in those layers.
- The basic idea stays the same: feed the input(s) forward through the neurons in the network to get the output(s) at the end.

Understanding Loss functions

Say we have the following measurements:

Name	Weight (lb)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F

Let's train our network to predict someone's gender given their weight and height:

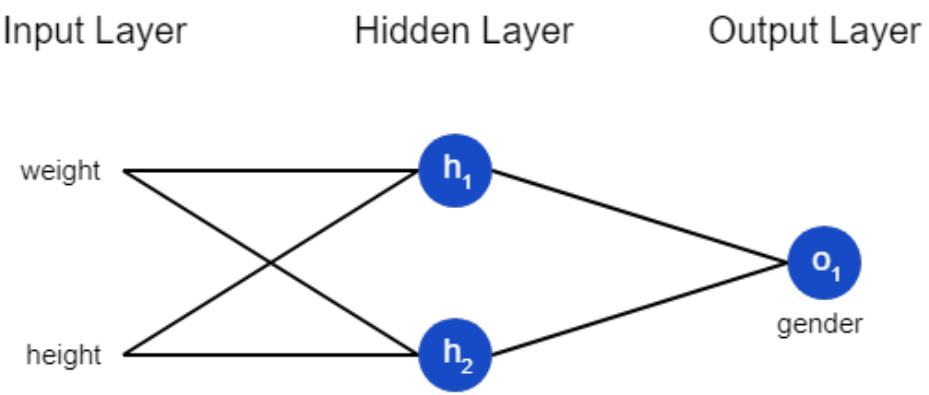


Image courtesy: `https://victorzhou.com/`

cont...

We'll represent Male with a 0 and Female with a 1, and we'll also shift the data to make it easier to use.

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1

Here,we arbitrarily chose the shift amounts (135 and 66) to make the numbers look nice. Normally, we'd shift by the mean.

Image courtesy: `https://victorzhou.com/`

Loss

- Before we train our network, we first need a way to quantify how "good" it's doing so that it can try to do "better".
- That's what the loss is.
- We'll use the mean squared error (MSE) loss:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{true},i} - y_{\text{pred},i})^2$$

Let's break this down:

- n is the number of samples, which is 4 (Alice, Bob, Charlie, Diana).
- y represents the variable being predicted, which is Gender.
- y_{true} is the true value of the variable (the "correct answer"). For example, y_{true} for Alice would be 1 (Female).

cont...

- y_{pred} is the predicted value of the variable. It's whatever our network outputs.
- $(y_{\text{true}} - y_{\text{pred}})^2$ is known as the **squared error**. Our loss function is simply taking the average over all squared errors (hence the name **mean squared error**). The better our predictions are, the lower our loss will be!

Better predictions = Lower loss.
Training a network = Trying to minimize its loss.

An Example Loss Calculation

Let's say our network always outputs 0 - in other words, it's confident all humans are Male. What would our loss be?

Name	y_{true}	y_{pred}	$(y_{true} - y_{pred})^2$
Alice	1	0	1
Bob	0	0	0
Charlie	0	0	0
Diana	1	0	1

$$MSE = \frac{1}{4}(1 + 0 + 0 + 1) = 0.5$$

Understanding Loss functions

- We now have a clear goal: minimize the loss of the neural network.
- For simplicity, let's pretend we only have Alice in our dataset:

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

Then the mean squared error (MSE) loss is just Alice's squared error:

$$MSE = \sum_{i=1}^{1} (y_{\text{true}} - y_{\text{pred}})^2$$

$$= (y_{\text{true}} - y_{\text{pred}})^2$$

$$= (1 - y_{\text{pred}})^2$$

7. Back propagation

Another way to think about loss is as a function of weights and biases. Let's label each weight and bias in our network:

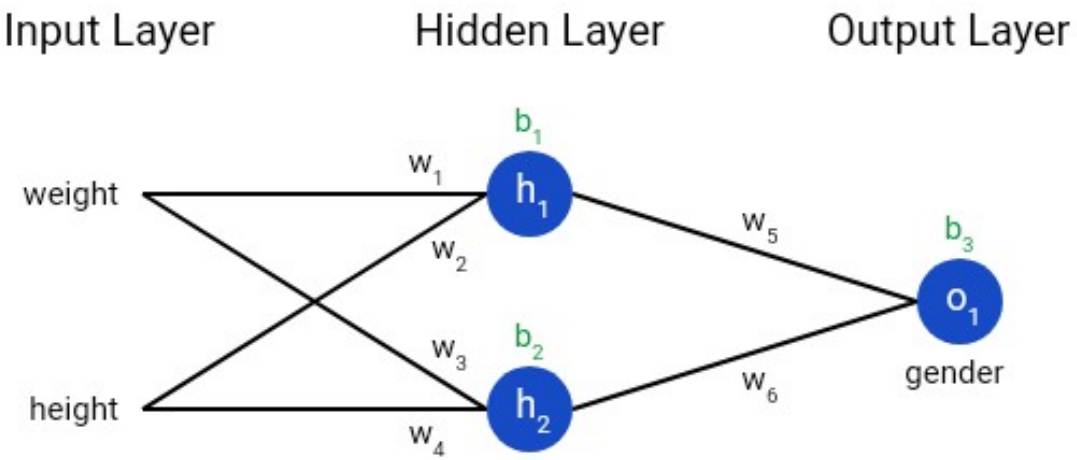


Image courtesy: `https://victorzhou.com/`

cont...

Then, we can write loss as a multi-variable function:

$$L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$$

Imagine we wanted to tweak w_1. How would loss L change if we changed w_1?
That's a question the partial derivative $\frac{\partial L}{\partial w_1}$ can answer.

Calculate $\frac{\partial L}{\partial w_1}$

To start, let's rewrite the partial derivative in terms of $\frac{\partial y_{\text{pred}}}{\partial w_1}$ instead:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{\text{pred}}} \cdot \frac{\partial y_{\text{pred}}}{\partial w_1}$$

This works because of the Chain Rule.
We can calculate $\frac{\partial L}{\partial y_{\text{pred}}}$ because we computed:

$$L = (1 - y_{\text{pred}})^2$$

$$\frac{\partial L}{\partial y_{\text{pred}}} = \frac{\partial (1 - y_{\text{pred}})^2}{\partial y_{\text{pred}}} = -2(1 - y_{\text{pred}})$$

Now, let's figure out what to do with $\frac{\partial y_{\text{pred}}}{\partial w_1}$. Just like before, let h_1, h_2, o_1 be the outputs of the neurons they represent. Then:

$$y_{\text{pred}} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$

where f is the sigmoid activation function.

Since w_1 only affects h_1 (not h_2), we can write:

$$\frac{\partial y_{\text{pred}}}{\partial w_1} = \frac{\partial y_{\text{pred}}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial y_{\text{pred}}}{\partial h_1} = \frac{\partial f}{\partial h_1}(w_5 h_1 + w_6 h_2 + b_3)$$

$$\frac{\partial y_{\text{pred}}}{\partial h_1} = f'(w_5 h_1 + w_6 h_2 + b_3) \cdot \frac{\partial (w_5 h_1 + w_6 h_2 + b_3)}{\partial h_1}$$

$$\frac{\partial y_{\text{pred}}}{\partial h_1} = f'(w_5 h_1 + w_6 h_2 + b_3) \cdot w_5$$

$$\frac{\partial y_{\text{pred}}}{\partial h_1} = w_5 \cdot f'(w_5 h_1 + w_6 h_2 + b_3)$$

By applying the Chain Rule again, we do the same for $\frac{\partial h_1}{\partial w_1}$:

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = \frac{\partial f}{\partial w_1}(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = f'(w_1 x_1 + w_2 x_2 + b_1) \cdot \frac{\partial}{\partial w_1}(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = x_1 \cdot f'(w_1 x_1 + w_2 x_2 + b_1)$$

cont...

Since we've seen $f'(x)$ (the derivative of the sigmoid function) before, let's derive it:

$$f(x) = \frac{1}{1 + e^{-x}}$$

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = f(x) \cdot (1 - f(x))$$

We're done! We've managed to break down $\frac{\partial L}{\partial w_1}$ into several parts we can calculate:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{\text{pred}}} \cdot \frac{\partial y_{\text{pred}}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1}$$

This system of calculating partial derivatives by working backwards is known as **backpropagation**, or "**backprop**".

Example: Calculating the Partial Derivative

We're going to continue pretending only Alice is in our dataset:

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

Let's initialize all the weights to 1 and all the biases to 0. If we do a feedforward pass through the network, we get:

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1) = f(-2 + -1 + 0) = 0.0474$$

$$h_2 = f(w_3 x_1 + w_4 x_2 + b_2) = 0.0474$$

$$o_1 = f(w_5 h_1 + w_6 h_2 + b_3) = f(0.0474 + 0.0474 + 0) = 0.524$$

The network outputs $y_{pred} = 0.524$, which doesn't strongly favor Male (0) or Female (1).

Image courtesy: `https://victorzhou.com/blog/intro-to-neural-networks/#3-training-a-neural-network-part-1`

Now, let's calculate $\frac{\partial L}{\partial w_1}$:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1}$$

We first calculate each part:

1. Calculate $\frac{\partial L}{\partial y_{pred}}$:

$$\frac{\partial L}{\partial y_{pred}} = -2(1 - y_{pred}) = -2(1 - 0.524) = -0.952$$

2. Calculate $\frac{\partial y_{pred}}{\partial h_1}$:

$$\frac{\partial y_{pred}}{\partial h_1} = w_5 \cdot f'(w_5 h_1 + w_6 h_2 + b_3) = 1 \cdot f'(0.0948)$$

cont...

Since $f(x) = \frac{1}{1+e^{-x}}$, we compute $f'(x) = f(x)(1 - f(x))$.
For $f(0.0948) = 0.5237$, we have:

$$f'(0.0948) = 0.5237 \cdot (1 - 0.5237) = 0.249$$

Thus:

$$\frac{\partial y_{pred}}{\partial h_1} = 0.249$$

3. Calculate $\frac{\partial h_1}{\partial w_1}$:

$$\frac{\partial h_1}{\partial w_1} = x_1 \cdot f'(w_1 x_1 + w_2 x_2 + b_1) = -2 \cdot f'(-3)$$

Since $f(-3) = 0.0474$, we have:

$$f'(-3) = 0.0474 \cdot (1 - 0.0474) = 0.0452$$

cont...

Thus:

$$\frac{\partial h_1}{\partial w_1} = -2 \cdot 0.0452 = -0.0904$$

Now, putting everything together:

$$\frac{\partial L}{\partial w_1} = (-0.952) \cdot (0.249) \cdot (-0.0904) = 0.0214$$

Reminder: We derived $f'(x) = f(x) \cdot (1 - f(x))$ for our sigmoid activation function earlier.
This tells us, if we were to increase w_1, L would increase by a small amount.

Training: Stochastic Gradient Descent

- We'll use an optimization algorithm called *stochastic gradient descent (SGD)* that tells us how to change our weights and biases to minimize the loss.
- It's basically just this update equation:

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

where η is a constant called the *learning rate* that controls how fast we train.

cont...

All we're doing is subtracting $\eta \frac{\partial L}{\partial w_1}$ from w_1:

- If $\frac{\partial L}{\partial w_1}$ is positive, w_1 will decrease, which makes L decrease.

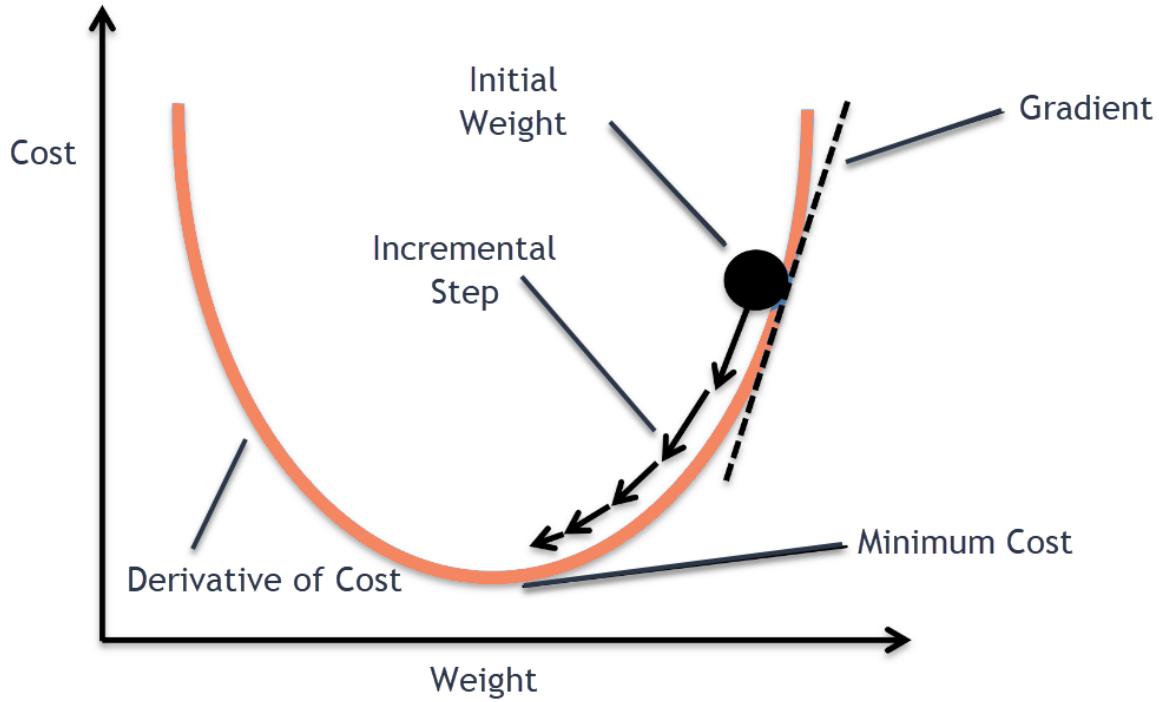
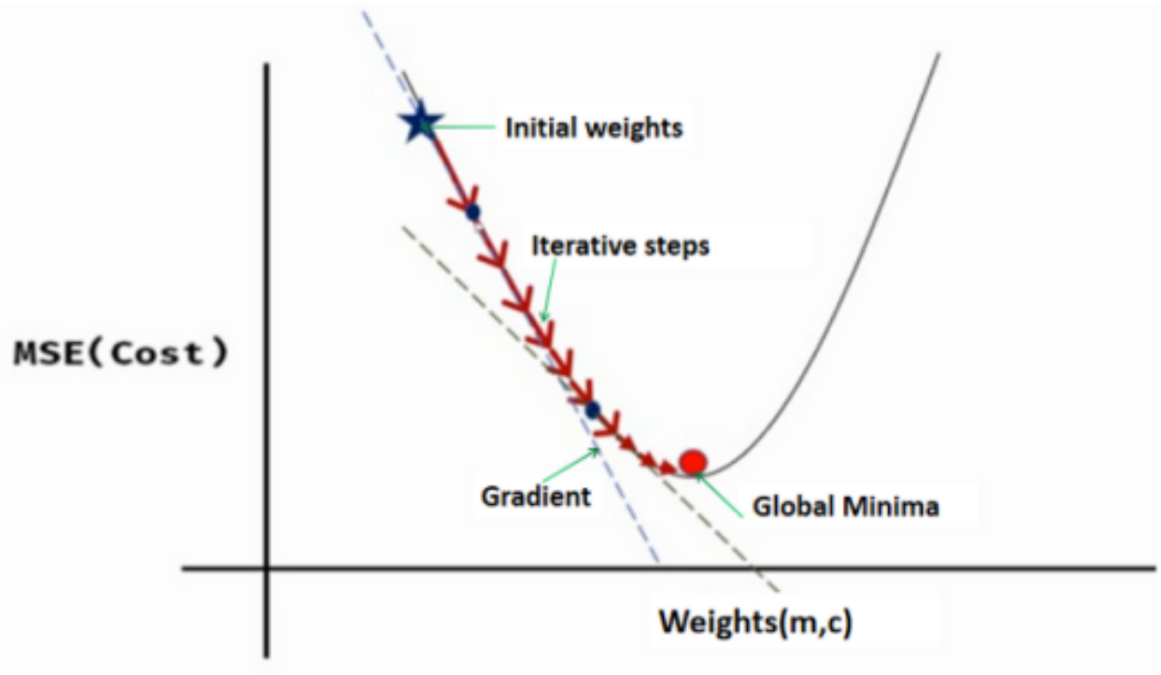


Image courtesy:https://medium.com/analytics-vidhya/stochastic-gradient-descent-1ab661fabf89

cont...

- If $\frac{\partial L}{\partial w_1}$ is negative, w_1 will increase, which makes L decrease.



If we do this for every weight and bias in the network, the loss will slowly decrease and our network will improve.

Image source:https://samuel-ozechi.medium.com/stochastic-gradient-descent-for-deep-learning-8d911b6b625a

cont...

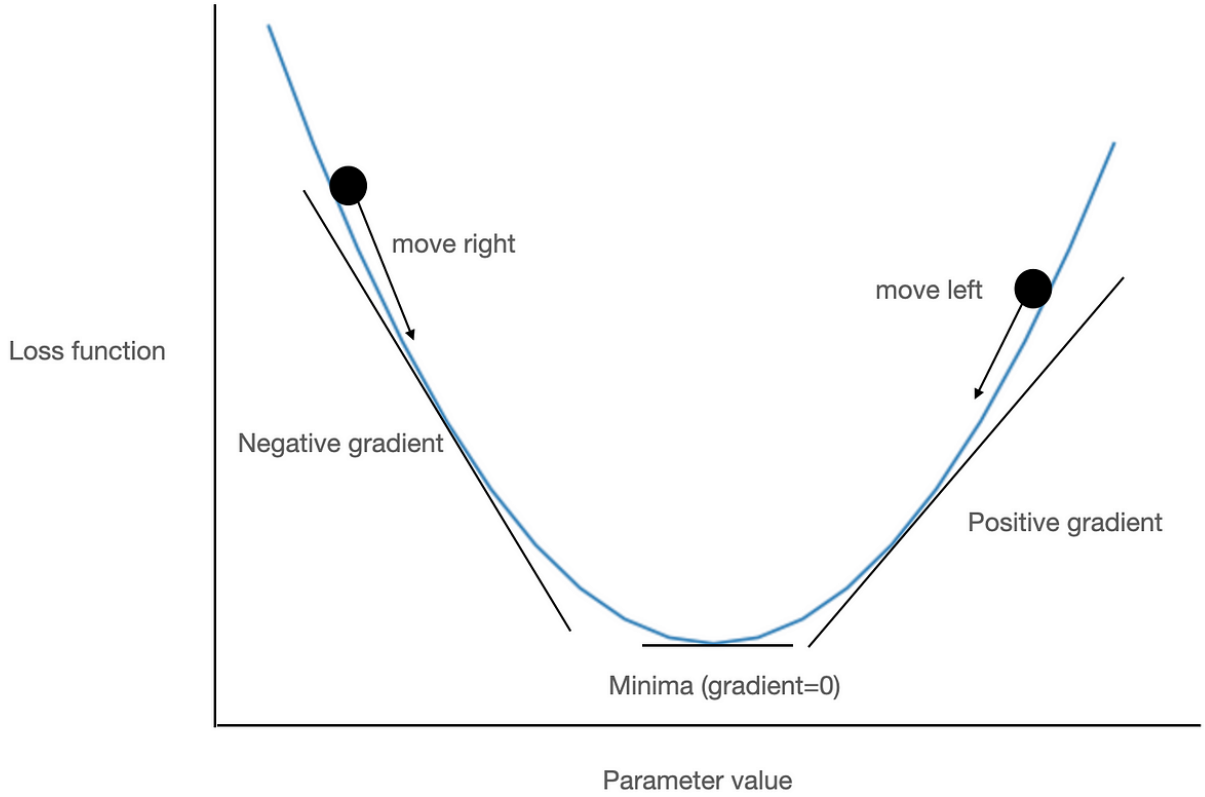


Image courtesy: https://medium.com/@sharadjoshi/why-stochastic-gradient-descent-works-75e6a1806c80

Training Process

Our training process will look like this:

1. Choose one sample from our dataset. This is what makes it *stochastic gradient descent* - we only operate on one sample at a time (or a small set). Gradient descent calculates the gradient based on the entire dataset.
2. Calculate all the partial derivatives of loss with respect to weights or biases (e.g. $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots$).
3. Use the update equation to update each weight and bias.
4. Go back to step 1.

Loss reduction

Our loss steadily decreases as the network learns:

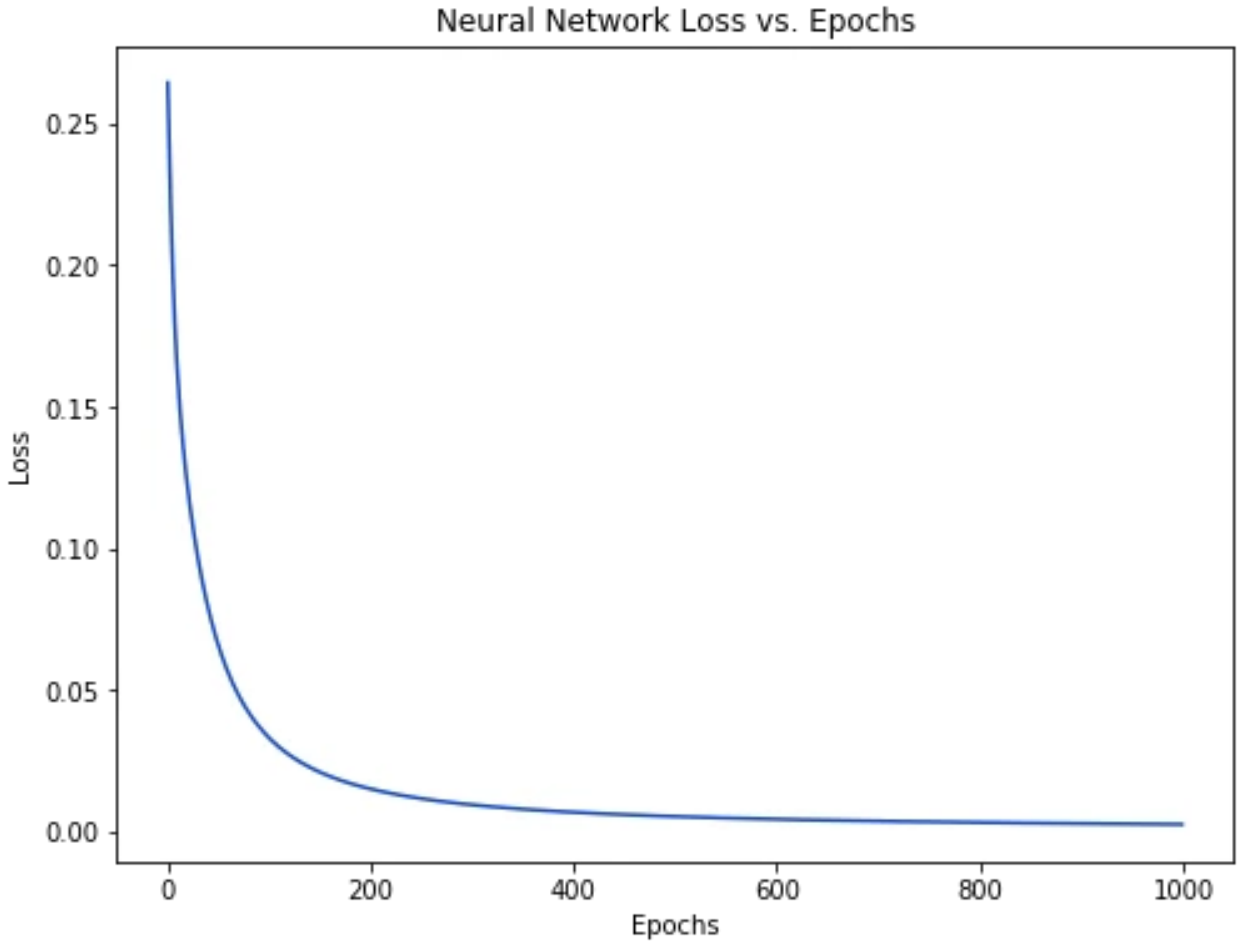


Image courtesy: `https://victorzhou.com/blog/intro-to-neural-networks/#3-training-a-neural-network-part-1`