



PG Diploma in ML/AI

Course : Machine Learning
Lecture On : Intro To NN
Instructor : Manish Kumar

KEY TOPICS

- 1 Perceptron
- 2 Weights, Bias & Activation Function
- 3 Feed Forward
- 4 Gradient Descent
- 5 Backpropagation

Deep learning—a machine learning technique—is an efficient way of learning that relies on big data, where features that can help a machine map an input to an output is automatically extracted from layers of “neurons”.

Deep learning is the main technology behind:

- Driverless cars
- Large-scale recommendation engines like Spotify, YouTube, and Amazon
- Language translation services like Google Translate
- Chatbots like Siri and Google assistant.

Artificial Intelligence

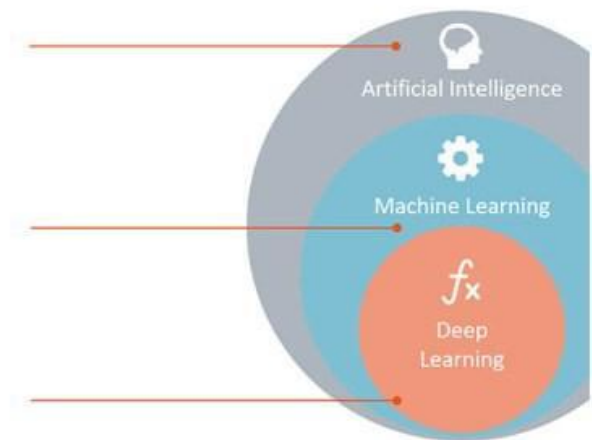
Any technique which enables computers to mimic human behavior.

Machine Learning

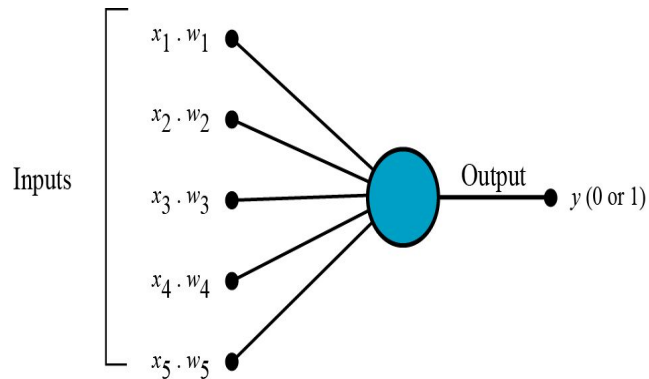
Subset of AI techniques which use statistical methods to enable machines to improve with experiences.

Deep Learning

Subset of ML which make the computation of multi-layer neural networks feasible.

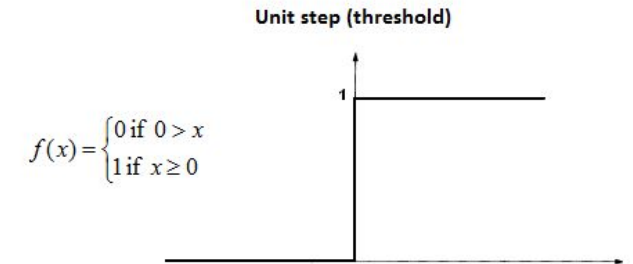


The perceptron works on these simple steps



All the inputs x are multiplied with their weights w . Let's call it k .

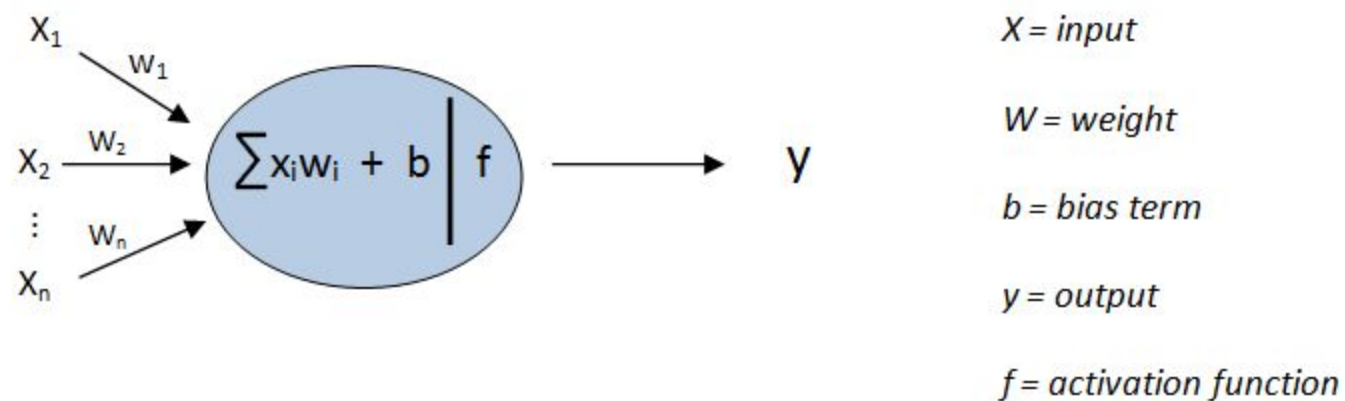
Add all the multiplied values and call them **Weighted Sum**



Apply that weighted sum to the correct Activation Function.

Weights shows the strength of the particular node.

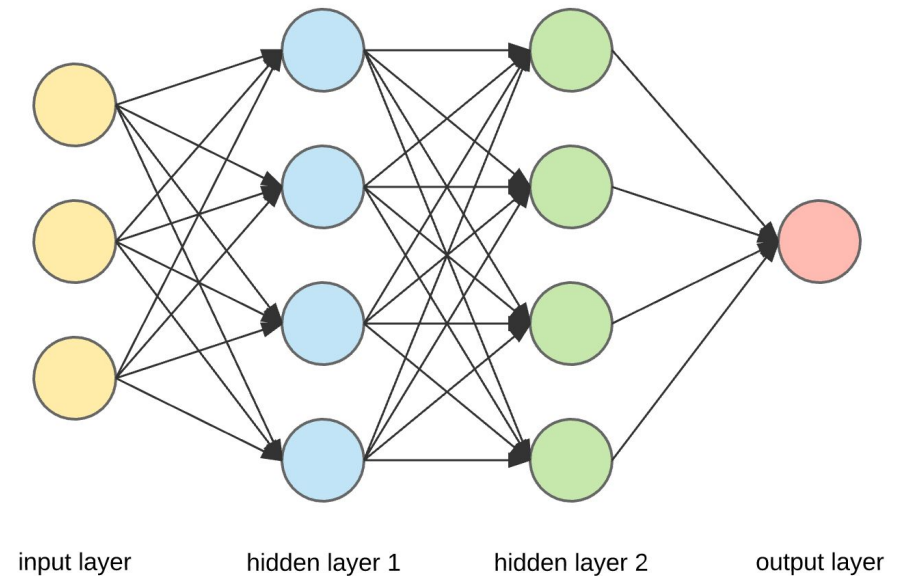
A bias value allows you to shift the activation function curve up or down.



$$y = f(\sum(\text{weight} * \text{input}) + \text{bias})$$

3 types of layers in neural network:

- **Input layer** — It is used to pass in our input (an image, text or any suitable type of data for NN).
- **Hidden Layer** — These are the layers in between the input and output layers. These layers are responsible for learning the mapping between input and output. (i.e. in the dog and cat gif above, the hidden layers are the ones responsible to learn that the dog picture is linked to the name dog, and it does this through a series of matrix multiplications and mathematical transformations to learn these mappings).
- **Output Layer** — This layer is responsible for giving us the output of the NN given our inputs.



Poll

Statement 1: It is possible to train a network well by initializing all the weights as 0.

Statement 2: It is possible to train a network well by initializing biases as 0

Which of the statements given above is true?

- A) Statement 1 is true while Statement 2 is false
- B) Statement 2 is true while statement 1 is false
- C) Both statements are true
- D) Both statements are false

Even if all the biases are zero, there is a chance that neural network may learn. On the other hand, if all the weights are zero; the neural neural network may never learn to perform the task.

Poll

Statement 1: It is possible to train a network well by initializing all the weights as 0

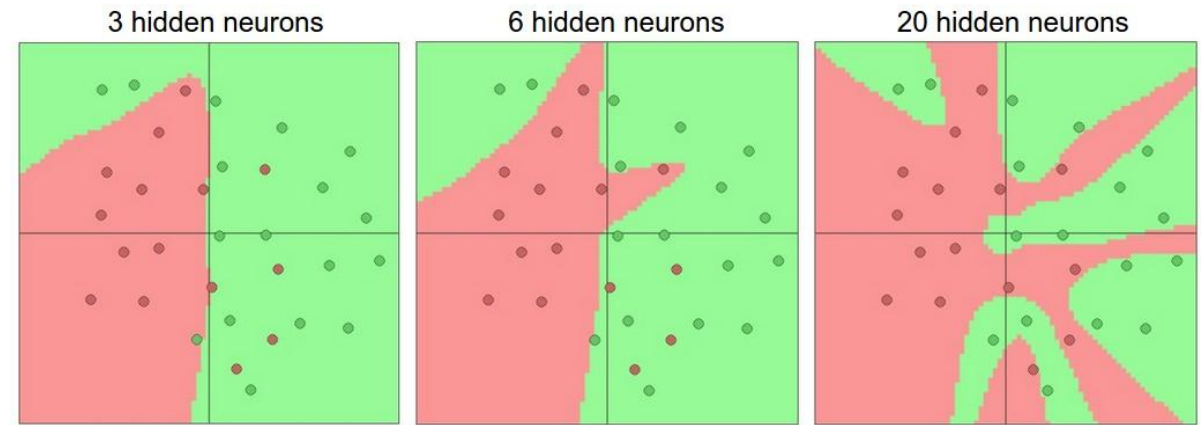
Statement 2: It is possible to train a network well by initializing biases as 0

Which of the statements given above is true?

- A) Statement 1 is true while Statement 2 is false
- B) Statement 2 is true while statement 1 is false**
- C) Both statements are true
- D) Both statements are false

Even if all the biases are zero, there is a chance that neural network may learn. On the other hand, if all the weights are zero; the neural neural network may never learn to perform the task.

A linear equation is easy to solve but they are limited in their complexity and have less power to learn complex functional mappings from data. A Neural Network without Activation function would simply be a **Linear regression Model**, which has limited power and does not performs good most of the times.



*That is why we use Artificial Neural network techniques such as **Deep learning** to make sense of something complicated ,high dimensional, non-linear -big datasets, where the model has lots and lots of hidden layers in between and has a very complicated architecture which helps us to make sense and extract knowledge form such complicated big datasets.*

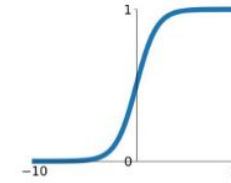
<https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

Activation functions are really important for a Artificial Neural Network to learn and make sense Non-linear complex functional mappings between the inputs and response variable.

They *introduce non-linear properties to our Network*. Their **main purpose is to convert a input signal of a node in a A-NN to an output signal**. That output signal now is used as a input in the next layer in the stack.

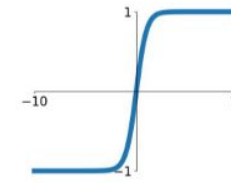
Sigmoid

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



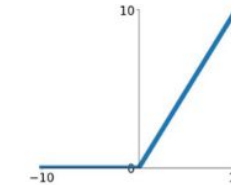
tanh

$$\tanh(x)$$



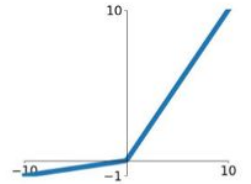
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

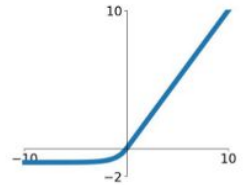


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

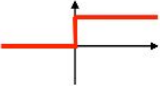
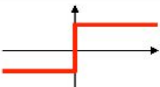
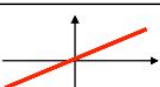
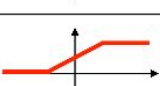




$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



<https://ai.stackexchange.com/questions/5493/what-is-the-purpose-of-an-activation-function-in-neural-networks>

Nowadays we should use **ReLU** which should only be applied to the hidden layers. And if your model suffers from dead neurons during training we should use **leaky ReLU** or **Maxout** function.

It's just that *Sigmoid and Tanh* should not be used nowadays due to the **vanishing Gradient Problem** which causes a lots of problems to train, degrades the accuracy and performance of a **deep Neural Network Model**.

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = \max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Copyright © Sebastian Raschka 2016
(<http://sebastianraschka.com>)

Poll 3

Which of following activation function can't be used at output layer to classify an image ?

- A) sigmoid
- B) Tanh
- C) ReLU
- D) $\text{If}(x > 5, 1, 0)$
- E) None of the above

Poll 3

Which of following activation function can't be used at output layer to classify an image ?

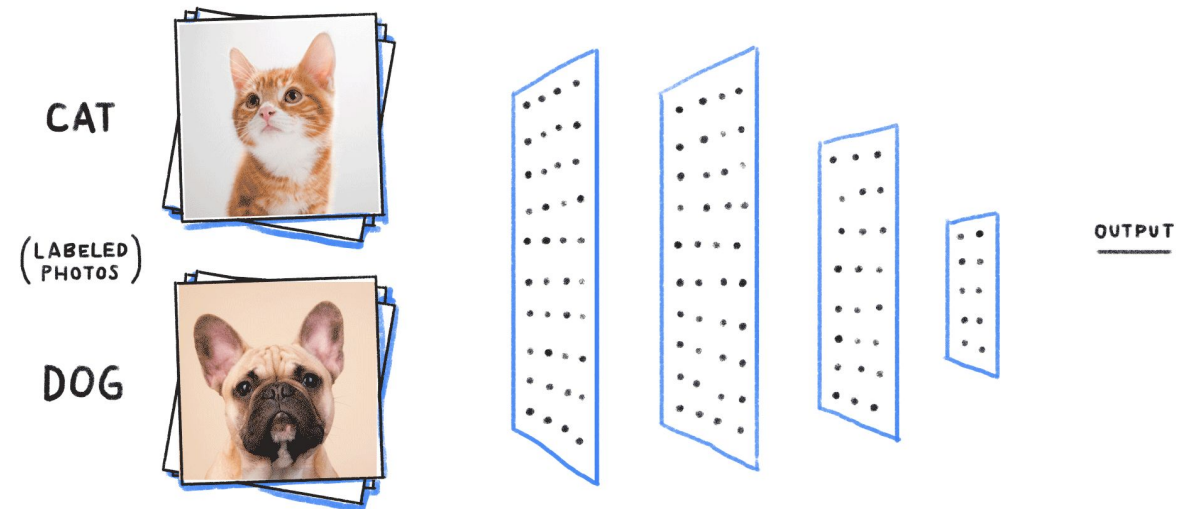
- A) sigmoid
- B) Tanh
- C) ReLU**
- D) $\text{If}(x > 5, 1, 0)$
- E) None of the above

Solution: C

ReLU gives continuous output in range 0 to infinity. But in output layer, we want a finite range of values. So option C is correct.

the output from one layer is used as input to the next layer. Such networks are called **feedforward neural networks**.

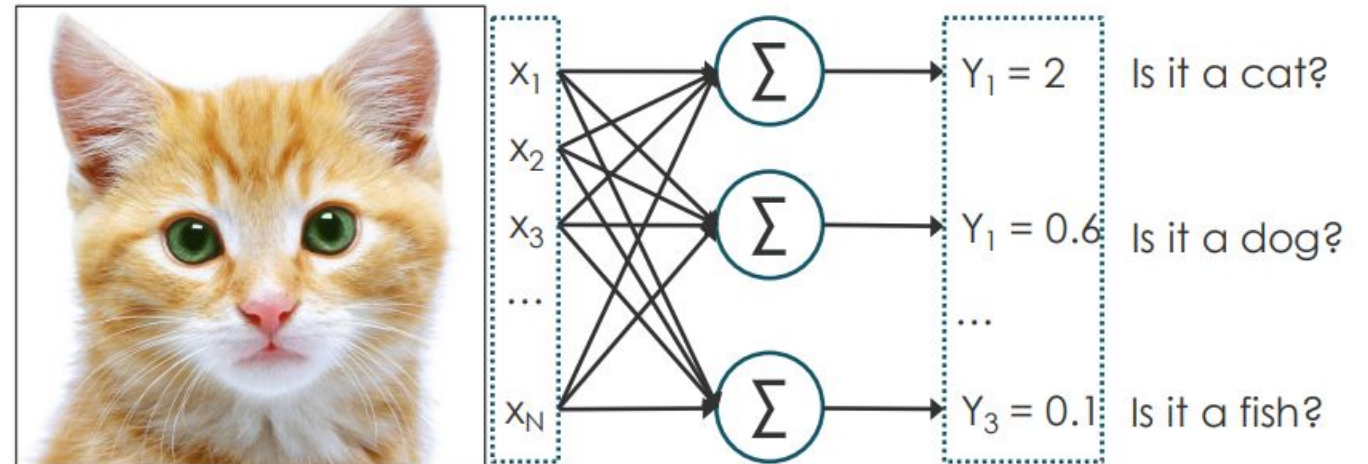
$$\sigma \left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,k} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{j,0} & w_{j,1} & \cdots & w_{j,k} \end{bmatrix} \begin{bmatrix} a_0^0 \\ a_1^0 \\ \vdots \\ a_n^0 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \right)$$



To create a classifier we want the output to look like as set of probabilities

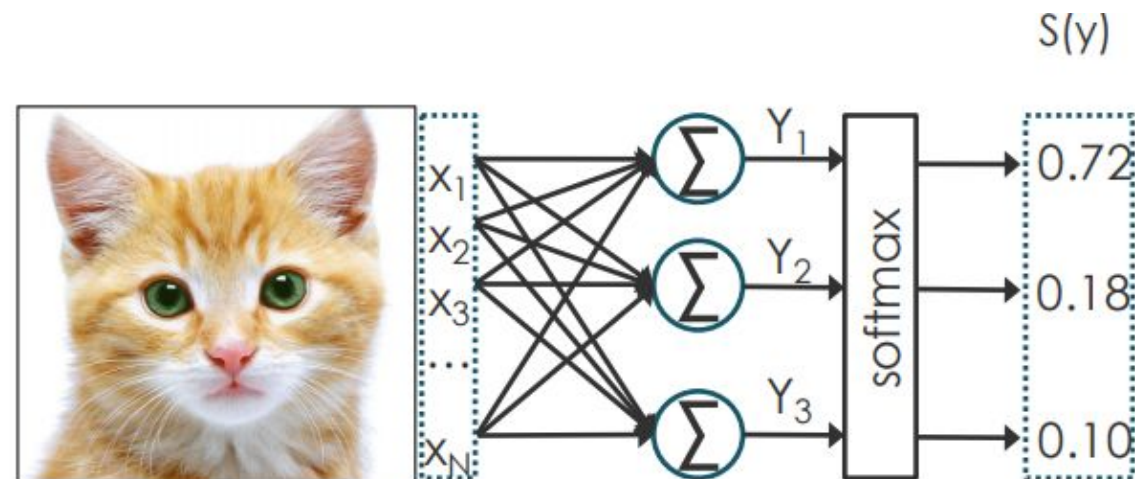
By feeding the input through the network we get a set of scores **Y** called “**logits**” but in here they cannot be used as probabilities

- They are not within the range $[0,1]$
- They do not add up to 1.



To convert logits into probabilities we can use the softmax function

$$S(y_i) = \frac{e^{y_i}}{\sum e^y}$$



This guarantees all values are between $[0,1]$ and they add up to 1.

A **Cost Function/Loss Function** evaluates the performance of our Machine Learning Algorithm. The **Loss function** computes the error for a single training example while the **Cost function** is the average of the loss functions for all the training examples.

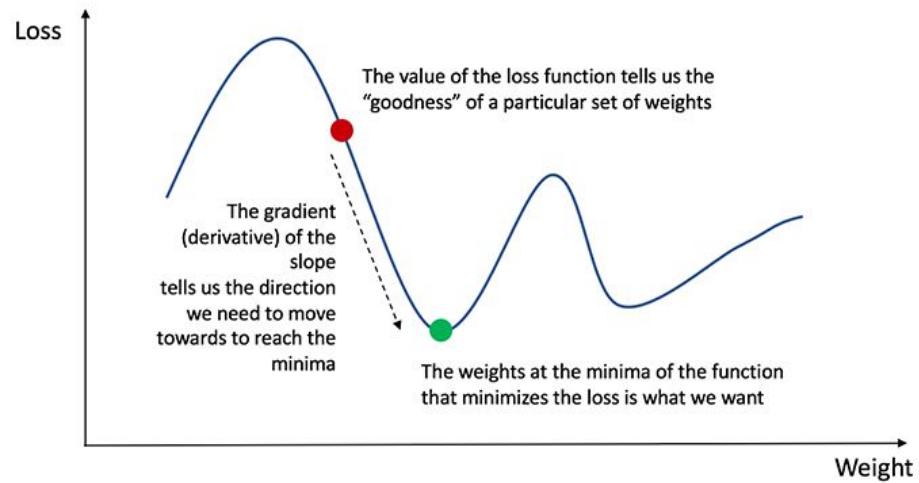
$$Cost = \frac{1}{N} \sum_{i=1}^N (Y' - Y)^2$$

The goal of any Learning Algorithm is to minimize the Cost Function.

lower error between the actual and the predicted values signifies that the algorithm has done a good job in learning.

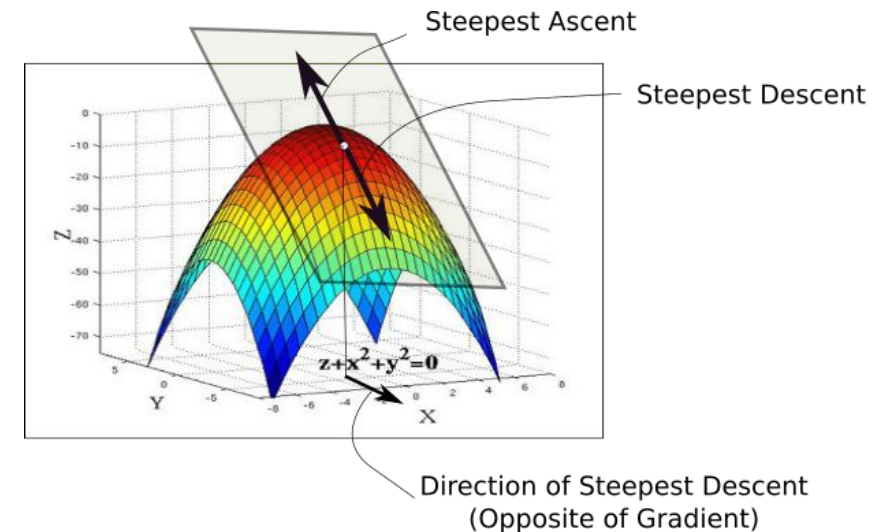
A common measure of the discrepancy between the two values is the "Cross-entropy"

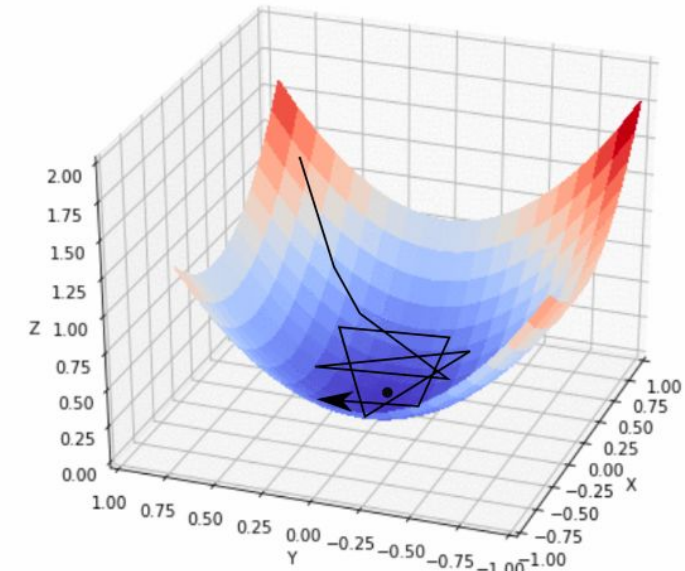
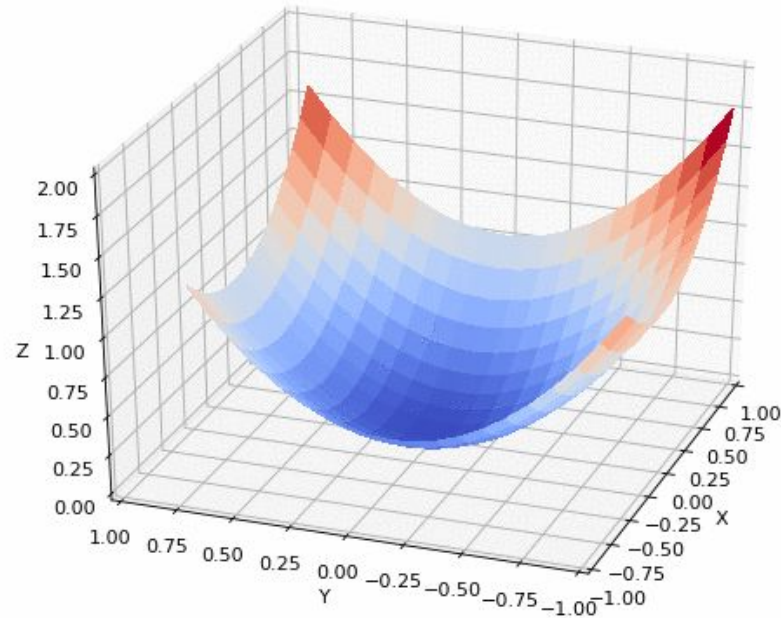
$$D(\hat{S}(y), L) = -\sum_i L_i \log(S(y_i))$$



Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient

While the direction of the gradient tells us which direction has the steepest ascent, its magnitude tells us how steep the steepest ascent/descent is. So, at the minima, where the contour is almost flat, you would expect the gradient to be almost zero. In fact, it's precisely zero for the point of minima.





In practice, we might never *exactly* reach the minima, but we keep oscillating in a flat region in close vicinity of the minima. As we oscillate our this region, the loss is almost the minimum we can achieve, and doesn't change much as we just keep bouncing around the actual minimum. Often, we stop our iterations when the loss values haven't improved in a pre-decided number, say, 10, or 20 iterations. When such a thing happens, we say our training has converged, or convergence has taken place.

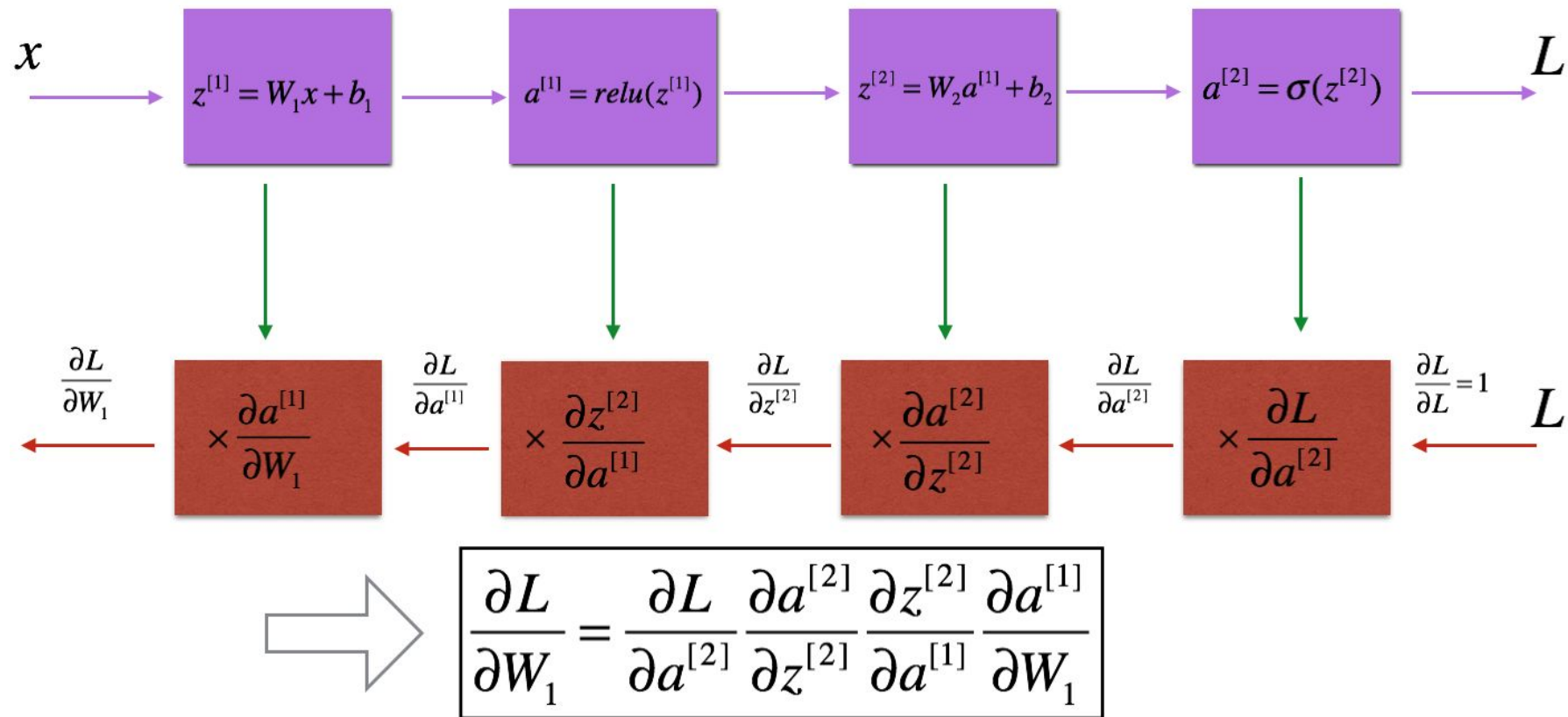
we can't directly calculate the derivative of the loss function with respect to the weights and biases because the equation of the loss function does not contain the weights and biases. Therefore, we need the **chain rule** to help us calculate it.

$$Loss(y, \hat{y}) = \sum_{i=1}^n (y - \hat{y})^2$$

$$\frac{\partial Loss(y, \hat{y})}{\partial W} = \frac{\partial Loss(y, \hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \quad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * z(1-z) * x$$



We know that propagation is used to calculate the gradient of the loss function with respect to the parameters.

The backpropagation algorithm pseudocode is as follows:

$$1. DZ^L = P - Y$$

$$2. \frac{\partial L}{\partial W^L} = \frac{1}{m} DW^L = \frac{1}{m} DZ^L \cdot (H^{L-1})^T$$

$$3. \frac{\partial L}{\partial b^L} = \frac{1}{m} Db^L = \frac{1}{m} DZ^L$$

$$4. dH^{L-1} = (W^L)^T \cdot DZ^L$$

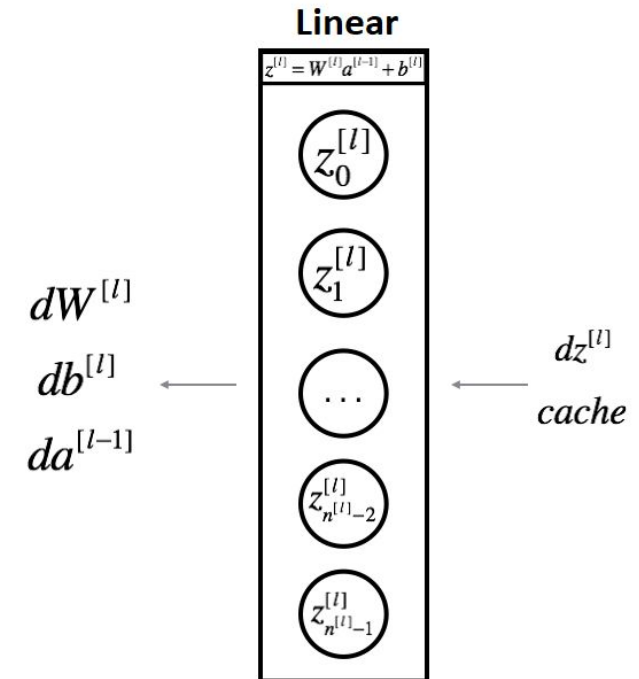
5. for l in $[L - 1, \dots, 1]$:

$$1. DZ^l = dH^l \otimes \sigma'(Z^l)$$

$$2. \frac{\partial L}{\partial W^l} = \frac{1}{m} DW^l = \frac{1}{m} DZ^l \cdot (H^{l-1})^T$$

$$3. \frac{\partial L}{\partial b^l} = \frac{1}{m} Db^l = \frac{1}{m} DZ^l$$

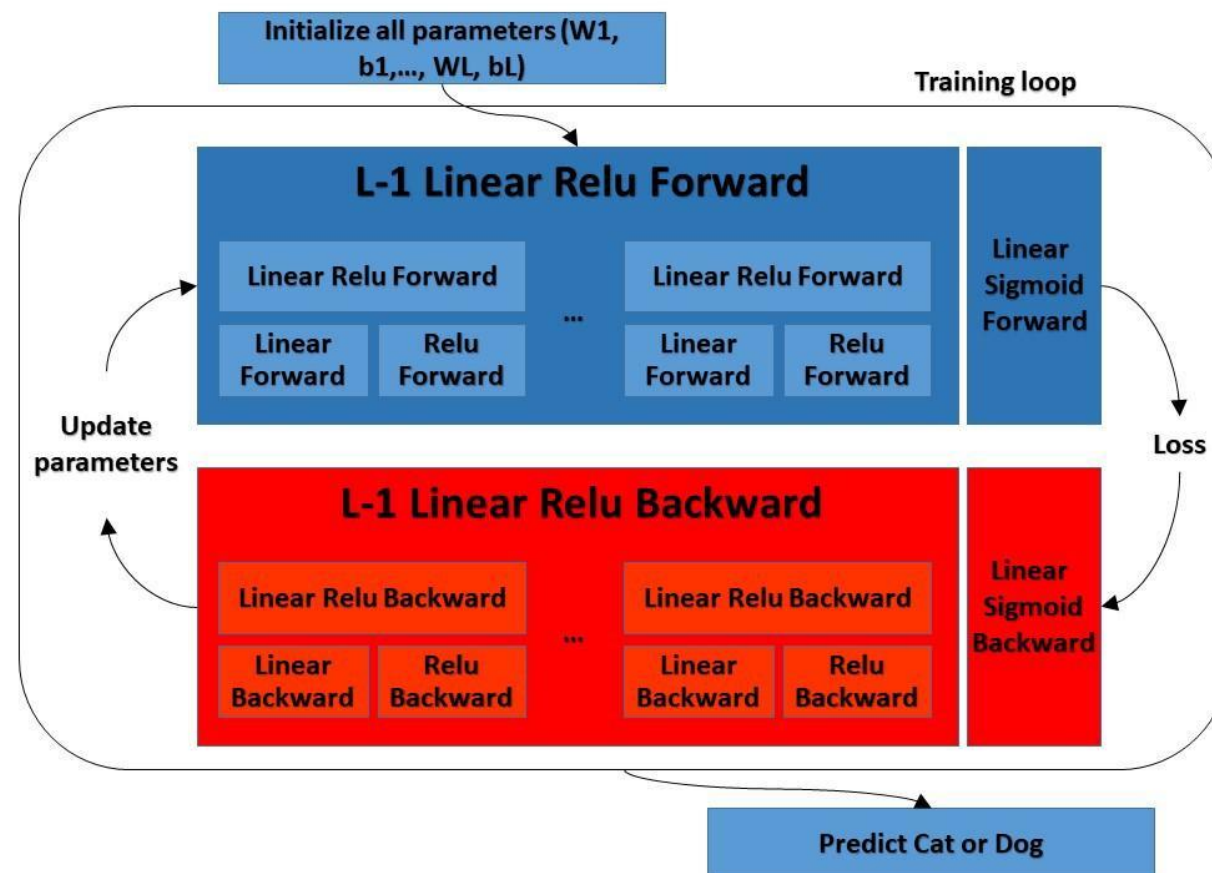
$$4. dH^{l-1} = (W^l)^T \cdot DZ^l$$

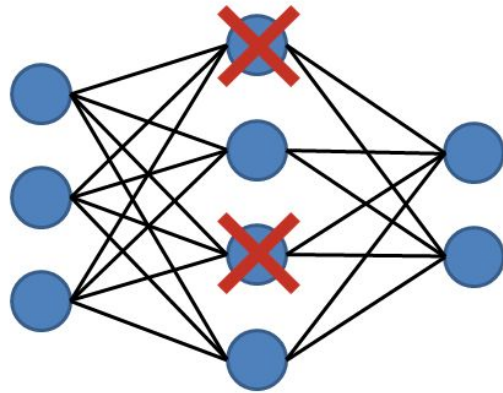


dZ — Gradient of the cost with respect to the linear output (of current layer l).

Training refers to the task of finding the optimal combination of weights and biases to minimize the total loss. The optimization is done using the familiar **gradient descent** algorithm. In gradient descent, the parameter being optimized is iterated in the direction of reducing cost according to the following rule

$$W_{new} = W_{old} - \alpha \cdot \frac{\partial L}{\partial W}.$$





Dropout is a regularization technique. On each iteration, we randomly shut down some neurons (units) on each layer and don't use those neurons in both forward propagation and back-propagation. Since the units that will be dropped out on each iteration will be random, the learning algorithm will have no idea which neurons will be shut down on every iteration; therefore, force the learning algorithm to spread out the weights and not focus on some specific features (units)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

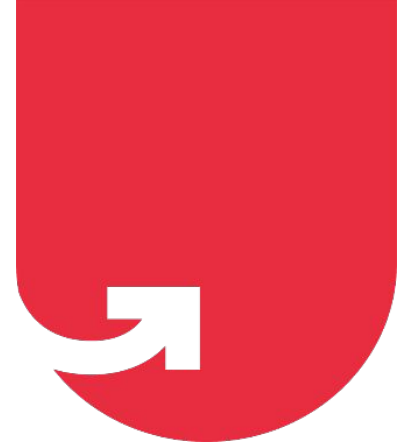
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

In **Batch Normalization** We normalize the input layer by adjusting and scaling the activations. For example, when we have features from 0 to 1 and some from 1 to 1000, we should normalize them to speed up learning. If the input layer is benefiting from it, why not do the same thing also for the values in the hidden layers, that are changing all the time, and get 10 times or more improvement in the training speed.



Thank You!