Neural Networks and Deep Learning

Introduction

Deep Learning Specialization

Deep learning has changed internet businesses such as search and ads (personalized recommendations). It is also creating new products and opportunities, such as:

- Better diagnostics and interpretations
- Personalized education
- Precision agriculture
- Self driving cars

Over the next 10 years, the society will be more AI powered.

AI is the new electricity

About 100 years ago, electrification changed our society & day-to-day life. There is surprisingly clear path for AI to bring about a similar change. Within AI, the domain of Deep learning has been rising fast.

Specialization

The specialization covers 5 courses:

- **Neural Networks and Deep learning**
- **Improving DNN: Hyperparameter tuning, regularization and optimization.**
- **Structuring Machine learning projects**
- **Convolutional neural networks**
- **Sequence models**

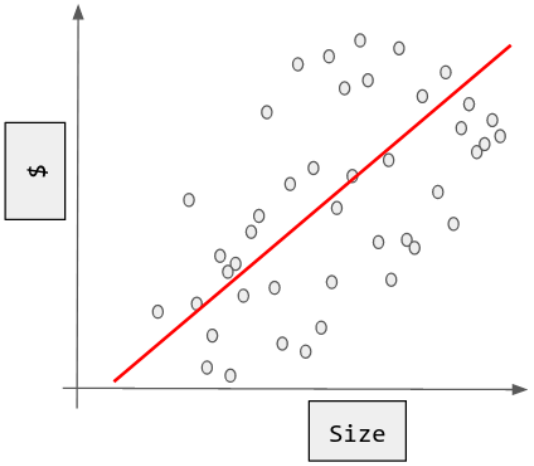
Neural Networks and Deep learning

What is a Neural Network

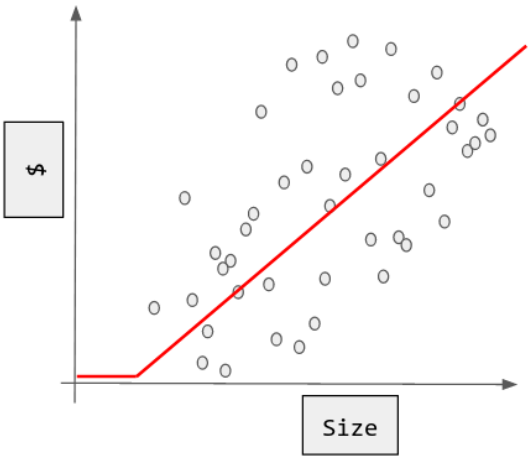
Deep learning involves training large neural networks. Let us first understand the concept of 'Neural Networks'.

Housing Price prediction

Say, we have a dataset of "house-size vs. price" in the real estate market. Can we make price as a function of size i.e. $price = f(size)$? If we draw a regression line through the points, the price would turn negative in the y-axis soon enough.

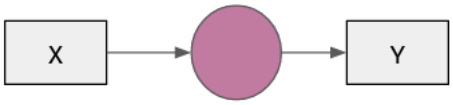


Linear regression



ReLU

Following linear regression though, we draw a line fitting the data, and bend the curve at zero ($price \not< 0$). We get a function $f : size \rightarrow price$. This is the simplest possible neural network implemented by a neuron.



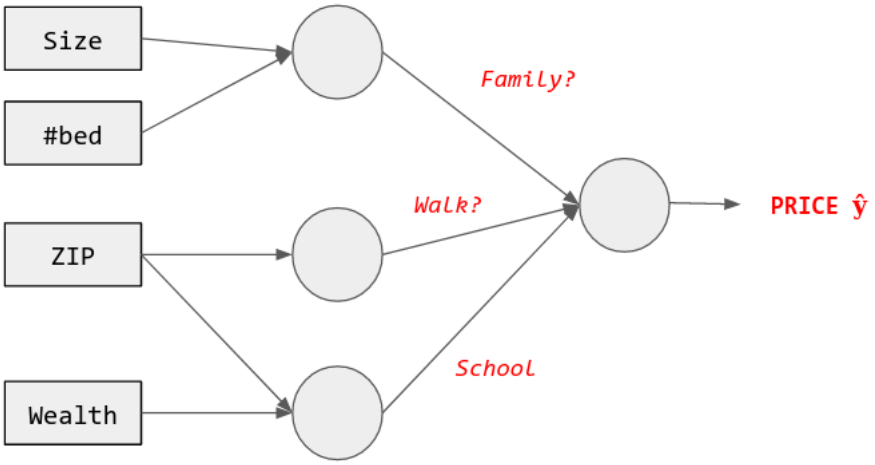
Single node or neuron

The job of neuron is to take an input (size X), compare it against the learned non-linear function (ReLU) and give the output (price Y). Larger networks can be obtained by stacking more such neurons. They are capable of processing more features and complexity. (Better estimation in the presence of more features, variables).

What is Neural Network?

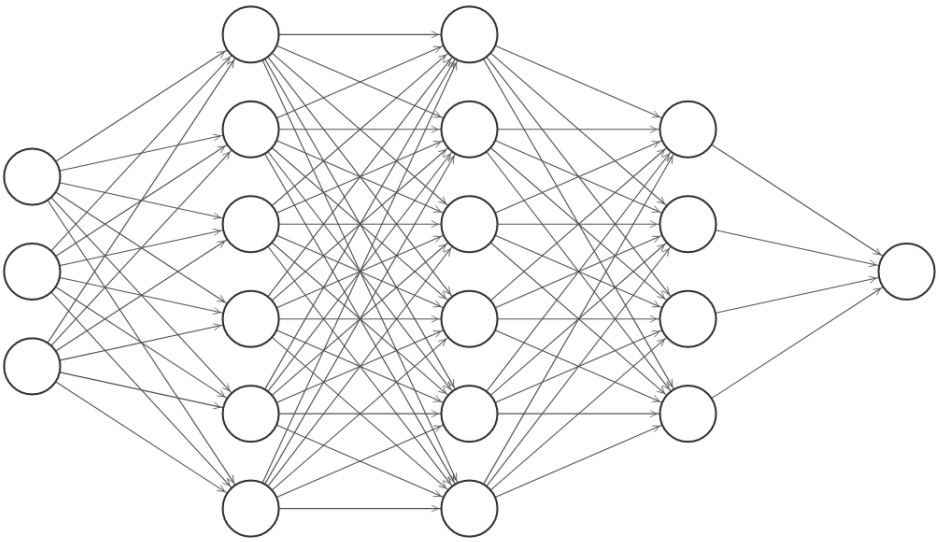
A powerful learning algorithm based on how the brain works.

- Each neuron is capable of providing a non-linear function, leading to better estimates.



Neural network deciding price based on several different inputs, considered as features X.

- We don't have to define the relationships between $\{X, Y\}$. Given enough paired examples $\{X, Y\}$, the neural network is capable of finding the hidden relationships.
- When each input is connected to each of the outputs in the next layer, the network is said to be Densely Connected.



Densely connected neurons

Summary

- **Single Neuron N/W**: Given data of "size vs. price" in real estate, do we have $price = f(size)$? We perform a linear regression, bend the curve at zero ($price \not< 0$) and find $f : size(x) \rightarrow price(\hat{y})$

- **Multiple Neuron N/W**: Implemented by stacking multiple neurons. Role of Neural Net is to predict price($\hat{Y}$), given the different variables stacked as features X. The Neural Net automatically generates relationships. Given enough $\{X, Y\}$ the neural network can establish the relationship i.e. $price = f(size, bedroom, wealth..)$

Supervised Learning

Most economic activity associated with AI is via Supervised Learning. In supervised learning, we train with a dataset and we know the output (y) for training input (x), given the understanding that there could be a relationship underlying X and Y.
For supervised learning,

- Regression - Predicting $\hat{y}$ within continuous output i.e. mapping is a continuous function.
- Classification - Predicting $\hat{y}$ within discrete output i.e. mapping is a discrete function.

X	Y	Note
Home features	price	Standard Neural Net
Ad, User info	click? (T/F)	-d.o-
Image	Label	CNN
Audio	Text	RNN
English	Chinese	-d.o-
Image + Radar	Car Position	Hybrid Network

Problems can be usually broken down into choosing features (X), outputs(Y) and model type.

Nature of data

- Structured Data: "Database of data". Data in row and columns. Each item has a defined unambiguous meaning.
- Unstructured Data: Text, Audio, Images. The average fragment of information has no special correlation with its neighbor.

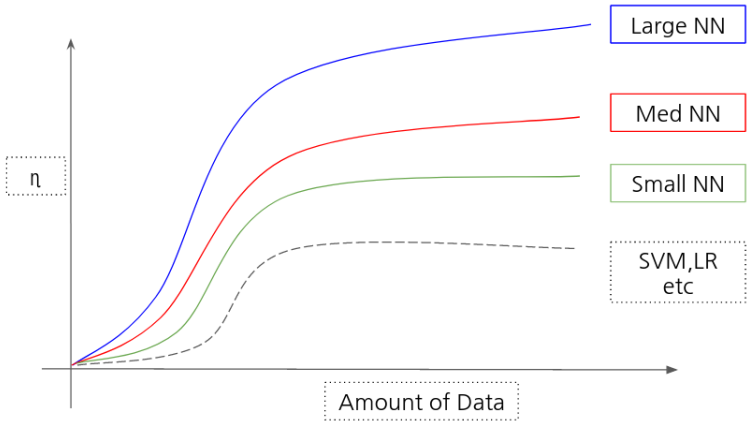
Humans are typically better at making sense from unstructured data. In recent years, Deep learning has made inroads in this domain.

Why is Deep learning taking off?

The ideas for Deep learning have been around for few years. They are taking off now because of specific trends/driving factors. Digitization of society by sensors, I.o.T & cheap storage ⇒ large amount of data. For high performance, we require large network and large amounts of data → "Scale Drives Deep Learning"

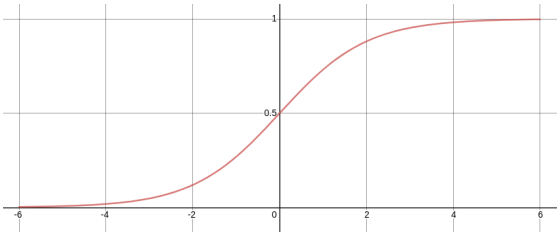
Trends

- Data : larger dataset are available, cheap storage.
- Compute: Faster GPU, CPU
- Algorithms: Better faster algorithms.

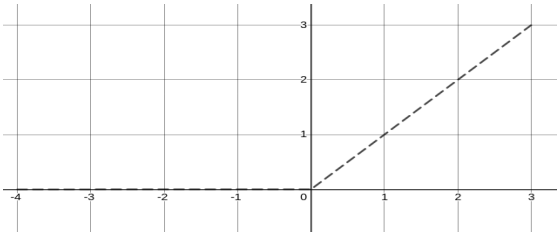


Scale drives deep learning. Relative order not well defined in low regime.

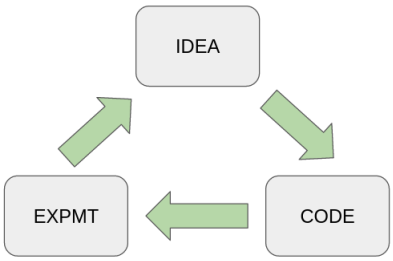
Advent of Deep learning methods have created many algorithmic innovations. One example is the usage of ReLU over Sigmoid activation functions. ReLU provides consistent gradient ($= 1$) $\forall X > 0$ and 0 $\forall X < 0$. This is easier to manage than Sigmoid or Arc-tan activations, which suffer from vanishing gradients once they reach the saturation regions. Also, Faster GPU, SIMD and better algorithms have made development cycles better. Iterating on different ideas have become quicker.



Sigmoid: Vanishing gradients beyond linear region



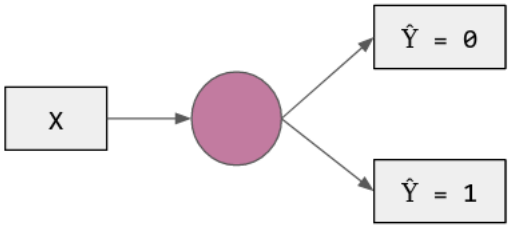
ReLU: Uniform gradient if $X > 0$



Deep learning pipeline/cycle

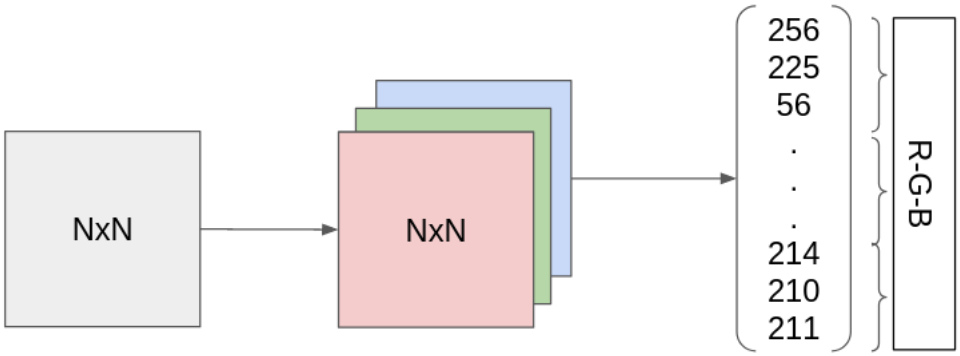
Binary classification

Logistic regression is used for binary classification {0 or 1}



Binary Classification

An image is made of R-G-B pixels. Each pixel location corresponds to a triplet. For an image, x is a feature vector, when all RGB pixels are unrolled into a single column vector. The dimension of the feature vector is $n_x = N \times N \times 3$



Making RGB image to feature vector.

The goal of classification is to take feature vector x and predict $\hat{y}$, as close as possible to real y.

- Training sample is denoted as $\{x, y\} : x \in \mathbb{R}, y \in \{0, 1\}$.
- For m-samples, we have $\{(x^{(1)}, y^{(1)}), ...(x^{(m)}, y^{(m)})\}$

The feature vectors of all images are organized into matrix $\mathbb{X}$ and collection of y-values are made into row-vector $\mathbb{Y}$.

$$\mathbb{X} = \begin{bmatrix} \vdots & \dots & \vdots \\ x^{(1)} & \ddots & x^{(m)} \\ \vdots & \dots & \vdots \end{bmatrix}$$

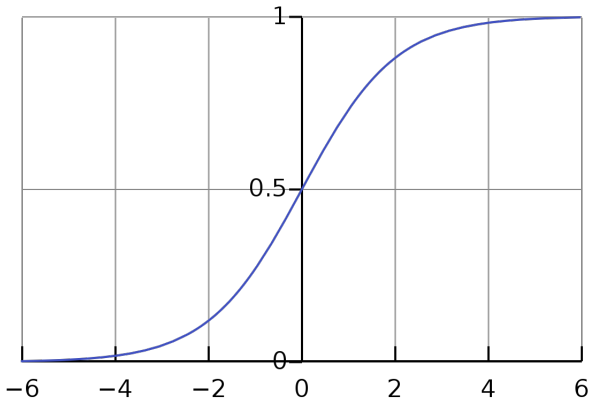
$$\mathbb{Y} = \begin{bmatrix} y^{(1)} & \dots & y^{(m)} \end{bmatrix}$$

Hence the given matrix specification for the image set is as follows: $X \in R^{(n_x, m)}$ and $Y \in R^{(1,m)}$

Logistic Regression

Logistic Regression is used primarily for binary classification.

- Given X, we want $P(y = 1|x) = \hat{y}$, where $\hat{y} \in [0, 1]$ and $X \in R^{n_x}$.
- Parameters are $w \in R^{n_x}$ and $b \in R$
- Output: $\hat{y} = \sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$



Sigmoid Function

Cases:

- Z is large/positive $\rightarrow \sigma(z) \approx 1$.
- Z is small/negative $\rightarrow \sigma(z) \approx 0$
- Z is 0 $\rightarrow \sigma(z) = 0.5$

Logistic regression is a learning algorithm for binary classification. The desired outputs are 0 or 1. The goal of logistic regression is to reduce error between prediction and real values, over the training set. i.e. $\hat{y}^i \approx y^i \forall i$. In doing so, we use the sigmoid function $\sigma(x)$. $(w^T x + b)$ would perform linear regression on the feature vector X. We require $\hat{y} \in [0, 1]$ and hence constrain the output by $\sigma(x)$ function.

Loss & Cost Function

Loss or Error function compares $\{y^{(i)}, \hat{y}^{(i)}\}$. The Cost Function $J(w, b)$ evaluates the fit over the whole training set i.e. find a {w,b} such that it is suitable in general for any input.

We know, $\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$. Given $\{(x^{(1)}, y^{(1)}), ...(x^{(m)}, y^{(m)})\}$, we require $\hat{y}^{(i)} \approx y^{(i)} \forall i$

Loss function: $L(\hat{y}, y) = -[\, y\, log(\hat{y}) + (1 - y)\, log(1 - \hat{y})]$

- If $y = 1$ then $L(\hat{y}, y) = -log(y)$; We want $\hat{y} \rightarrow 1$
- If $y = 0$ then $L(\hat{y}, y) = -log(1 - y)$; We want $\hat{y} \rightarrow 0$

This loss formulation works even when the optimization is a non-convex problem → Cross-entropy loss.

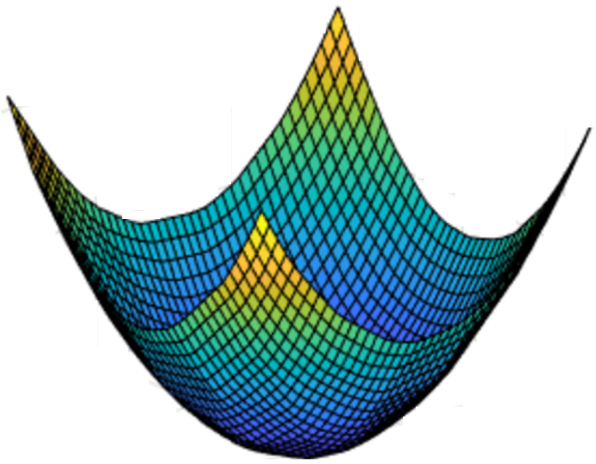
Cost function: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} log\ \hat{y}^{(i)} + (1-y^{(i)}) log\ (1-\hat{y}^{(i)})]$

- Consider this as the average of individual losses → similar to finding a line fitting most of the points on the whole.
- In Logistic regression, LMSE $\mathbb{L} = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$ is usually avoided because it can make the optimization non-convex.

To train $\{w, b\}$ similar to parameters of straight line used for regression, we need a cost function. Loss function measures the discrepancy between individual samples, whereas cost function (averaged loss) works well for most $\{x^{(i)}, y^{(i)}\}$

Gradient Descent

Gradient descent moves $J(w,b)$ towards the direction of minimum. Our goal is to minimize cost function $\mathbb{J}$ over the train set by tuning (w,b). Because of the choice of loss function earlier, the curve is convex.



Loss landscape of typical convex function

We initialize the parameters (w,b) randomly and let it descend to global minimum. Until gradient is non-zero, the value will keep descending. This works for both negative and positive values of gradient → Gradient descent update rule.

```
# Gradient Descent update
while (w!=w_min && b!=b_bmin )
{ w = w - alpha * dJ/dw
  b = b - alpha * dJ/db
}
```

α is called the learning rate. It is the step size of the gradient descent update step i.e. the amount by which the parameter(s) update over each pass of gradient descent.

Gradient Descent on 1 sample

Recall

- $z = w^T x + b$
- $\hat{y} = a = \sigma(z) = \frac{1}{1+e^{-z}}$
- $L(\hat{y}, y) = -[\, y\, log(\hat{y}) + (1-y)\, log(1-\hat{y})]$
- $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} log\ \hat{y}^{(i)} + (1-y^{(i)}) log\ (1-\hat{y}^{(i)})]$

In the case of 1 sample for G.D, $\mathbb{L}(\hat{y}, y)$ is equivalent to $J(w,b)$. Consider a input X of two features,

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

The inputs to logistic regression are x_1, w_1, x_2, w_2, b. Our goal is to reduce $\mathbb{L}(\hat{y}, y)$ w.r.t $\{w_1, w_2, b\}$. Over several passes of G.D, we should adjust them such that $\hat{y}^i \approx y^i \forall i$

$$\boxed{z = w^T x + b} \iff \boxed{a = \hat{y} = \sigma(z)} \iff \boxed{L(\hat{y}, y)}$$

$$\frac{dL}{da} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)$$

$$\begin{aligned}
\frac{dL}{dz} &= \frac{dL}{da}\frac{da}{dz} \\
&= \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)\frac{d}{dz}\,\sigma(z) \\
&= \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)\frac{-1}{(1+e^{-z})^2}\,(-e^{-z}) \\
&= \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)\frac{e^{-z}}{(1+e^{-z})}\frac{1}{(1+e^{-z})} \\
&= \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)\left(1 - \frac{1}{(1+e^{-z})}\right)\frac{1}{(1+e^{-z})} \\
&= \left[\frac{-y(1-a) + a(1-y)}{a(1-a)}\right](1-a)\,a = a - y
\end{aligned}$$

Individual derivatives can be computed further by the chain rule.

$$\begin{aligned}
\frac{dL}{dW_1} &= \frac{dL}{dZ}\frac{dZ}{dW_1} = x_1\,(a-y) \\
\frac{dL}{dW_2} &= \frac{dL}{dZ}\frac{dZ}{dW_2} = x_2\,(a-y) \\
\frac{dL}{db} &= \frac{dL}{dZ}\frac{dZ}{db} = (a-y)
\end{aligned}$$

Perform gradient descent with the computed values:

```
W_1 = W_1 - alpha * dW_1
W_2 = W_2 - alpha * dW_2
b   = b   - alpha * db
```

Gradient Descent on M sample

$J(w, b)$ is minimized by the average of loss functions (from previous section).

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{L}(\hat{y}^{(i)}, y^{(i)})$$
$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \left[L(\hat{y}^{(i)}, y^{(i)})\right]}{\partial w}$$

We know the closed form solution for the derivative in the case of loss function. Hence we have to accumulate the derivatives in each case of i, and average by the number of samples in the end.

Pseudo code

Algorithm: Gradient Descent on m-samples

$W_1, W_2, b \leftarrow rnd$;
while $dW, dB! = 0$ **do**
  $J, dW_1, dW_2, db \leftarrow 0$;
  for $i = 1{:}n$ **do**
    $z^{(i)} = w^T x^{(i)} + b$;
    $a^{(i)} = \sigma(z^{(i)})$;
    $J+ = -[\, y^{(i)} log(\hat{y^{(i)}}) + (1 - y^{(i)})\, log(1 - \hat{y^{(i)}})]$;
    $dZ^{(i)} = a^{(i)} - y^{(i)}$;
    $dW_1+ = x_1\, dZ^{(i)}$;
    $dW_2+ = x_2\, dZ^{(i)}$;
    $db+ = dZ^{(i)}$;
  end
  $J \leftarrow J/m$, $dW_1 \leftarrow dW_1/m$, $dW_2 \leftarrow dW_2/m$, $db \leftarrow db/m$;
  $W_1 = W_1 - \alpha \times dW_1$;
  $W_2 = W_2 - \alpha \times dW_2$;
  $b = b - \alpha \times db$;
end

More on cost function

$\hat{y} = P(y = 1|x)$ is the probability of $y = 1$ event. Conversely, if $y = 1, P(y = 1 \mid x) = \hat{y}$ and $y = 0, P(y = 0 \mid x) = 1 - \hat{y}$. Consider, $P(y \mid x) = \hat{y}^y\,(1-\hat{y})^{(1-y)}$, the Cross entropy function. Taking logarithms on both sides,

$$\begin{aligned} log\ P(y \mid x) &= y\ \log(\hat{y}) + (1-y)\ log\ (1-\hat{y}) \\ &= -\mathbb{L}(\hat{y}, y) \end{aligned}$$

We are minimizing the loss, but maximizing the probability. Assuming i.i.d, we can write

$$P(y_1, y_2 \dots y_m \mid x_1, x_2 \dots x_m) = \prod_{i=1}^{m} P[y^{(i)} \mid x^{(i)}]$$

$$log\ P(y_1, y_2 \dots y_m \mid x_1, x_2 \dots x_m) = \sum_{i=1}^{m} log\ P[y^{(i)}|x^{(i)}]$$

$$= -\sum_{i=1}^{m} \mathbb{L}(y^{(i)}, \hat{y}^{(i)})$$

Combining the two results shown above, we could write

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{L}(y^{(i)}, \hat{y}^{(i)})$$

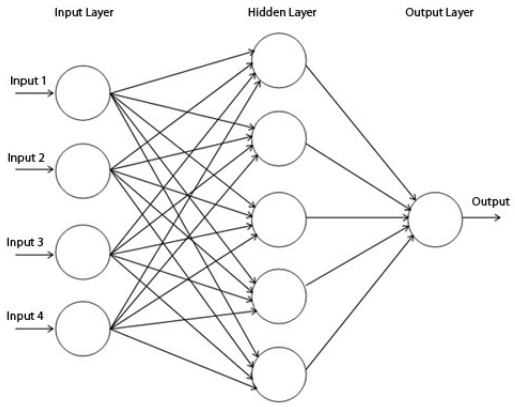
$$= -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} log\ \hat{y}^{(i)} + (1 - y^{(i)}) log\ (1 - \hat{y}^{(i)})]$$

Note:

- $P(y_1, y_2 \dots y_m \mid x_1, x_2 \dots x_m) = \hat{y}^{(1)^{y^{(1)}}} (1 - \hat{y})^{(1)^{(1-y)^{(1)}}} \dots \hat{y}^{(m)^{y^{(m)}}} (1 - \hat{y})^{(m)^{(1-y)^{(m)}}}$
- $\sum_{i=1}^{m} \mathbb{L}(y^{(i)}, \hat{y}^{(i)}) = -\sum_{i=1}^{m} log\ P[y^{(i)}|x^{(i)}] = -\sum_{i=1}^{m} [y^{(i)} log\ \hat{y}^{(i)} + (1 - y^{(i)}) log\ (1 - \hat{y}^{(i)})]$

One hidden-layer neural network

Previously we had input $\mathbb{X}$ going to a neuron. There were 3 parameters for a single neuron: $[x, w, b]$. Z and A were calculated at the node and error was made to backpropagate directly. Now we have an additional layer, which contains multiple nodes. Final layer is a single node, whose output is compared in the loss function.

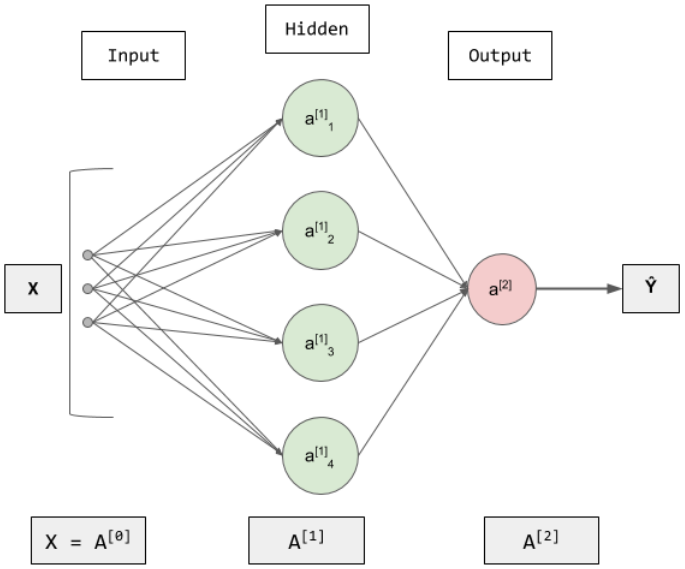


One hidden-layer Neural Net

$$\boxed{Z^{[1]} = W^{[1]}x + b^{[1]}} \iff \boxed{A^{[1]} = g(Z^{[1]})} \iff \boxed{Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}} \iff \boxed{A^{[2]} = g(Z^{[2]})} \Updownarrow$$

$$\boxed{L(A^{[2]}, y)}$$

Representation



Representation of 1-Hidden layer Neural net

- We do not know the middle/intermediate layers → hidden layer.
- We do not count input layer numerically → layer-0
- Activation are values which are passed on to the next layer.
- Each neuron is computing a *linear* (Computing Z) and then *non-linear* ($A = g(Z)$) part
- For simplicity, we would keep $w^T \sim W$

One sample

Each neuron computes

- Linear operation $z = Wx + b$
- Non-linear activation $a = g(z)$

For the hidden layer,

- Linear {1}: $z_1^{[1]} = W_1^{[1]}x + b_1^{[1]}$
- Non-linear {1}: $a_1^{[1]} = g(z_1^{[1]})$

...

- Linear {4}: $z_4^{[1]} = W_4^{[1]}x + b_4^{[1]}$
- Non-linear {4}: $a_4^{[1]} = g(z_4^{[1]})$

The weight matrix W can be thought of regular w which have been transposed and stacked. If the input has 3 features as column, w will have 3 rows, 4 columns before transpose.

$$z^{[1]} = \begin{bmatrix} \dots w_1^{[1]} \dots \\ \dots w_2^{[1]} \dots \\ \dots w_3^{[1]} \dots \\ \dots w_4^{[1]} \dots \end{bmatrix}_{(4\times3)} \times x_{(3\times1)} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}_{(4\times1)} = \begin{bmatrix} w_1^{[1]}x + b_1^{[1]} \\ w_2^{[1]}x + b_2^{[1]} \\ w_3^{[1]}x + b_3^{[1]} \\ w_4^{[1]}x + b_4^{[1]} \end{bmatrix}_{(4\times1)} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}_{(4\times1)}$$

$$a^{[1]} = \sigma \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}_{(4\times1)} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}_{(4\times1)}$$

$a^{[1]}$ becomes the input to the final layer, which is a single node and acts as a Logistic unit. It produces $a^{[2]} = \hat{y}$.

Summary of operations

- $z_1^{[1]} = W_1^{[1]}x + b_1^{[1]} \rightarrow a_1^{[1]} = \sigma(z_1^{[1]})$
- $z_2^{[1]} = W_2^{[1]}x + b_2^{[1]} \rightarrow a_2^{[1]} = \sigma(z_2^{[1]})$
- $z_3^{[1]} = W_3^{[1]}x + b_3^{[1]} \rightarrow a_3^{[1]} = \sigma(z_3^{[1]})$
- $z_4^{[1]} = W_4^{[1]}x + b_4^{[1]} \rightarrow a_4^{[1]} = \sigma(z_4^{[1]})$

$z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$ (4,1)←(4,3)(3,1) + (4,1)

$Z^{[1]} = \sigma(z^{[1]})$ (4,1)←(4,1)

$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$ (1,1)←(1,4)(4,1) + (1,1)

$Z^{[2]} = \sigma(z^{[2]})$ (1,1)←(1,1)

M-samples

- $Z_1^{[1]} = W_1^{[1]}X + b_1^{[1]} \rightarrow A_1^{[1]} = \sigma(Z_1^{[1]})$
- $Z_2^{[1]} = W_2^{[1]}X + b_2^{[1]} \rightarrow A_2^{[1]} = \sigma(Z_2^{[1]})$
- $Z_3^{[1]} = W_3^{[1]}X + b_3^{[1]} \rightarrow A_3^{[1]} = \sigma(Z_3^{[1]})$
- $Z_4^{[1]} = W_4^{[1]}X + b_4^{[1]} \rightarrow A_4^{[1]} = \sigma(Z_4^{[1]})$

$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]}$ (4,m)← $(4, n_x)(n_x, m) + (4, 1)*$

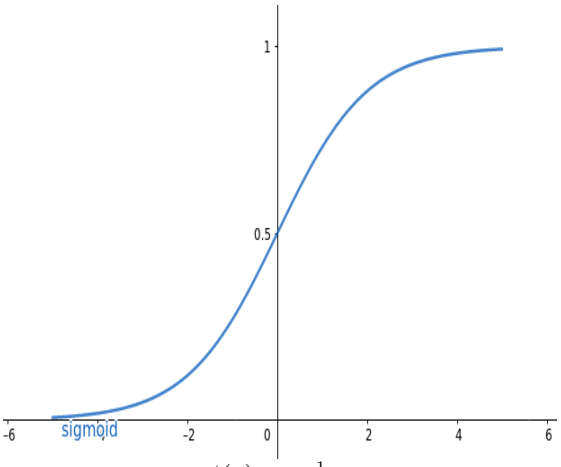
$A^{[1]} = \sigma(Z^{[1]})$ (4,m)← $(4, m)$

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$ (1,m)← $(1, 4)(4, m) + (1, 1)*$

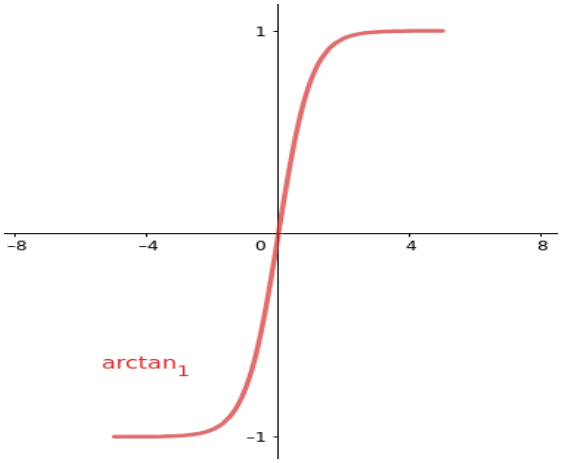
$A^{[2]} = \sigma(Z^{[2]})$ (1,m)← $(1, m)$

*broadcast variable

Activation functions

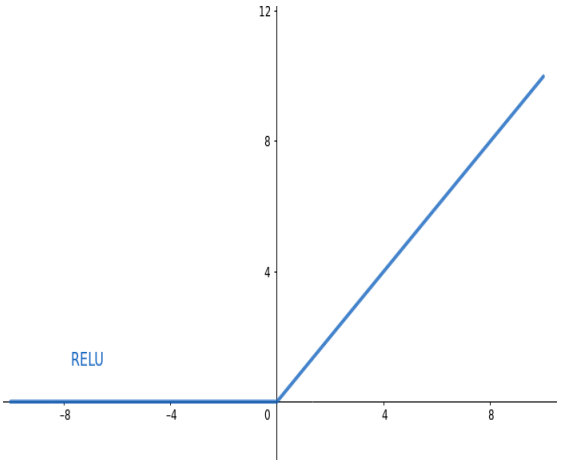


$\phi(x) = \frac{1}{1+e^{-x}}$

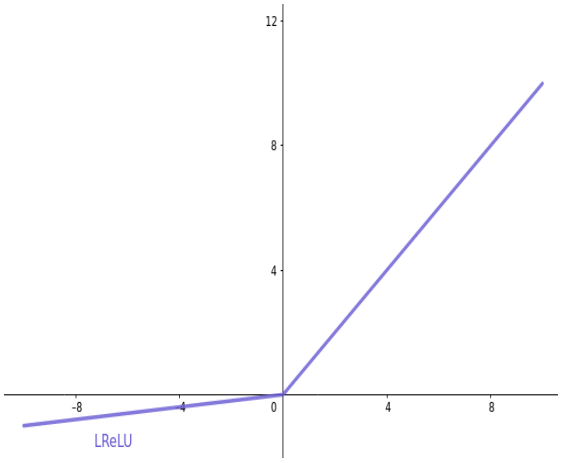


$\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Sigmoid activation is only useful in binary classification. It has a non-zero mean at 0.5. It suffers from vanishing gradients in the saturated regions. Hyperbolic tangent better known as Tanh activation is usually always better than Sigmoid, except in the final node where the output needs to be binary. $Tanh(x)$ lies between ± 1. It can be seen as a shifted version of Sigmoid activation. Similar to sigmoid, it also suffers from vanishing gradients in saturated regions.



$\phi(x) = max(0, x)$



$\phi(x) = max(0.01x, x)$

ReLU is usually the default choice nowadays. The derivative is 1 if $x > 0$ and 0 otherwise. The probability of $x = 0.00\ldots$ is extremely low and hence ReLU is considered safe in all use-cases. Leaky ReLU overcomes the deficiency of ReLU where a large proportion of linear terms before activation could be zero or negative. The scaling factor (of 0.01) is arbitrary.

Derivatives of Activation functions

Sigmoid

Function: $g(x) = \frac{1}{1+e^{-x}}$
Derivative: $g'(x) = g(x)[1 - g(x)]$

$$g'(x) = \frac{-1}{(1+e^{-x})^2} \times (-e^{-x})$$
$$= \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} = \frac{1}{(1+e^{-x})}\left[1 - \frac{1}{1+e^{-x}}\right] = g(z)[1-g(z)]$$

Tanh

Function: $g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1-e^{-2x}}{1+e^{-2x}}$
Derivative: $g'(x) = 1 - g^2(x)$

$$g'(x) = \frac{2e^{-2x}}{(1+e^{-2x})^2} - \left[\frac{2e^{-4x}}{(1+e^{-2x})^2} - \frac{2e^{-2x}}{(1+e^{-2x})}\right]$$
$$= \frac{2e^{-2x} - 2e^{-4x} + 2e^{-2x}(1+e^{-2x})}{(1+e^{-2x})^2} = \frac{4e^{-2x}}{(1+e^{-2x})^2}$$
$$= \frac{1 + e^{-4x} + 2e^{-2x} - 1 - e^{-4x} + 2e^{-2x}}{(1+e^{-2x})^2} = \frac{(1+e^{-2x})^2 - (1-e^{-2x})^2}{(1+e^{-2x})^2}$$
$$= 1 - \left(\frac{1-e^{-2x}}{1+e^{-2x}}\right)^2 = 1 - g^2(x)$$

ReLU

Function: $g(x) = max(0, x)$
Derivative:

$$g'(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$$

Leaky ReLU

Function: $g(x) = max(0.01x, x)$
Derivative:

$$g'(x) = \begin{cases} 0.01 & x < 0 \\ 1 & x > 0 \end{cases}$$

In both ReLU and Leaky ReLU, the function is not defined and non-differentiable at $x = 0$.

Need for non-linear activation

Let us say we use a linear function, such as the identity function for activation.

$$Z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = g_1(Z^{[1]}) = Z^{[1]}$$
$$Z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$= W^{[2]}[W^{[1]}x + b^{[1]}] + b^{[2]}$$
$$= \left[W^{[2]}W^{[1]}\right]x + [W^{[2]}b^{[1]} + b^{[2]}]$$
$$= W'x + b'$$

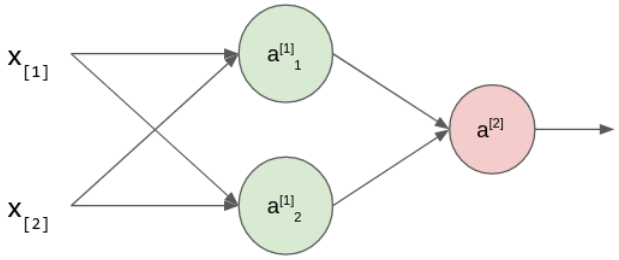
Having an identity function makes it no better than having a single node logistic regression, as the nodes can be combined to get a composite $\{W', b'\}$. Composition of 2 linear functions always results in a linear function. Linear activation is only useful in regression where $Y, \hat{Y} \in \mathbb{R}$.

Random initialization

If we initialize both weights as zero, or similar values, both activations will have the same output. They will contribute equally and symmetrically. When we perform gradient descent, both will have updated equally (because of similar contribution) resulting in same values again.

$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \begin{aligned} a_1^{[1]} &= a_2^{[1]} \\ dz_1^{[1]} &= dz_2^{[1]} \end{aligned}$$

By induction it is possible to show that this process will repeat over and over again because of the symmetry. This does not serve any useful purpose. We randomly initialize to perform symmetry breaking. Also, we initialize the weights with small values such that they do not saturate too quickly during the fitting process. In Sigmoid/Tanh, the saturation regions have very slow gradients ($\sigma'(x) \approx 0$). Only typically large values should reach such saturation regions. It should be noted that $\mathbf{b}$ does not symmetry problems, and can be conveniently set to zero often.



Random initialization enables breaking symmetry. If the values are identical, the values propagated forward-backward will be identical *ad infinitum*.

```
# Initialization of weights:
# Assume NN of 2 input, 2 hidden node, 1 output node
W1 = np.random.randn((2,2)) * 0.01
b1 = np.zeros(2,1)
W2 = np.random.randn((1,2)) * 0.01
b1 = 0
```

Gradient Descent

- Parameters: $\underset{(n_{[1]},n_{[0]})}{W^{[1]}}$, $\underset{(n_{[1]},1)}{b^{[1]}}$ $\underset{(n_{[2]},n_{[1]})}{W^{[2]}}$, $\underset{(n_{[2]},1)}{b^{[2]}}$
- Cost Function: $J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{L}(a^{[2](i)}, y^{(i)})$

Pseudo code

Algorithm: Gradient Descent via 1-hidden layer

while $J! = J_{min}$ **do**
  Compute $\hat{y} \, \forall \, i$;
  Compute $dW^{[1]}, dW^{[2]}, db^{[1]}, db^{[2]}$;
  $W^{[1]} = W^{[1]} - \alpha dW^{[1]}$;
  $b^{[1]} = b^{[1]} - \alpha db^{[1]}$;
  $W^{[2]} = W^{[2]} - \alpha dW^{[2]}$;
  $b^{[2]} = b^{[2]} - \alpha db^{[1]}$;
end

Summary

Forward Propagation

$Z^{[1]} = W^{[1]} A^{[0]} + b^{[1]}$
$A^{[1]} = \sigma(Z^{[1]})$
$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$
$A^{[2]} = \sigma(Z^{[2]})$

Backpropagation

1-sample

$dz^{[2]} = a^{[2]} - y$

$dW^{[2]} = dz^{[2]} \, a^{[1]T}$

$db^{[2]} = dz^{[2]}$

$dz^{[1]} = W^{[2]T} dz^{[2]} \times g'(z^{[1]})$

$dW^{[1]} = dz^{[1]} \, x^T$

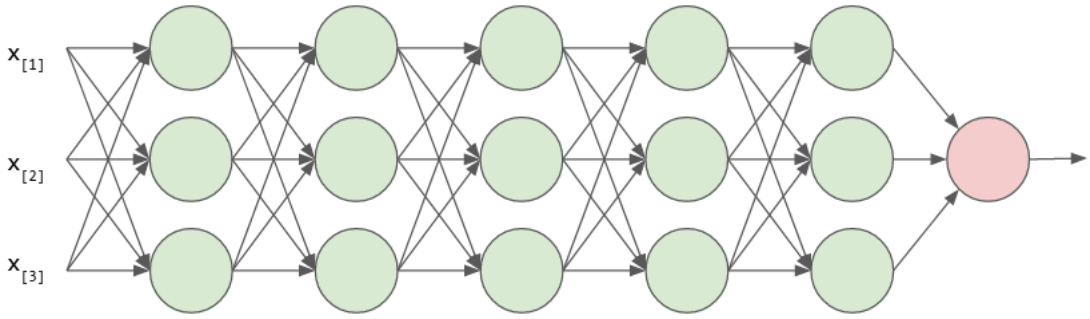
$db^{[1]} = dz^{[1]}$

m-sample/code

```
dZ2  = A2 - Y
dW2 = 1/m * (dZ2 A1.T)
db2 = 1/m * np.sum(dz2, axis=1, keepdims=True)
dZ1  = (W2.T dZ2) * g'(Z1)
dW1 = 1/m * (dZ1 X.T)
db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)
```

Deep Neural Networks

Logistic Regression and 1-hidden layer networks are *shallow networks*. We get better performance by adding more hidden layers. Although the definitions are nuanced and variable, any network with



Deep Neural network of 6-layers (5-hidden) layers.

5 or more layers is considered a Deep network. There are functions that a deep network can learn which a shallow one cannot. Although we can't be always certain how many layers do we require exactly in our model, we can start by the simplest ones and work our way up. The number of layers in a network is also considered a hyperparameter. Henceforth, we denote number of layers as **L**. $n^{[}m]$ will denote the number of units in the m^{th} layer of the network. Things to remember:

- The 0^{th} layer is input. $n^{[0]}$ is number of input features.
- $X = a^{[0]}, \hat{y} = a^{[L]}$
- A node outputs a real number $\rightarrow$ Z, A $\in \mathbb{R}$ with dimensions $(n^{[l]}, 1)$.
- Vectorized: Z & A dimensions $(n^{[l]}, m)$
- Dimensions of $W^{[l]}$, $dW^{[l]} \rightarrow (n^{[l]}, n^{[l-1]})$
- Dimensions of $b^{[l]}$, $db^{[l]} \rightarrow (n^{[l]}, 1)$

The general rule for computing forward pass in deep network is as follows:

$$Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g_l(Z^{[l]})$$

Requirement for Deep representations

In a neural network, the initial layers capture low-level information such as lines and edges. Subsequent layers keep grouping information into learned features. By composition learning, the network can detect complex features and shapes in object recognition/classification. This paradigm is supposed to mimic the neural cognition where the brain detects simple features and groups them towards identification.

Circuit theory and deep learning

From circuit theory, we know that we can compute some functions with small L-layer network which will require exponentially more hidden units. For example, consider $x_1 \oplus x_2 \oplus \cdots \oplus x_n$; With a L-layer network, we require a depth of the order of $log_2(N) \approx O(logN)$. With 1-hidden layer network, the same computation requires 2^{n-1} units to compute XOR $\approx O(2^n)$ [We require several units in a single layer to exhaustively compute all combinations. Although not a canonical example, it is sufficient to demonstrate utility.]

Building blocks

Forward propagation

- Input: $a^{[l-1]}$
- Output: $a^{[l]}, Cache(z^{[l]})$

$$Z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g_l(Z^{[l]})$$

Back-propagation

- Input: $da^{[l]}, Cache(z^{[l]})$
- Output: $da^{[l-1]}, dw^{[l]}, db^{[l]})$

For 1-sample,

$$dz^{[l]} = da^{[l]} * g_l'(z^{[l]}) = W^{[l+1]T}dz^{[l+1]} * g_l'(z^{[l]})$$

$$dW^{[l]} = dz^{[l]}a^{[l-1]T}$$

$$db^{[l]} = dz^{[l]}$$

$$da^{[l-1]} = W^{[l]T}dz^{[l]}$$

For m-samples,

$$dZ^{[l]} = dA^{[l]} * g_l'(Z^{[l]})$$

$$dW^{[l]} = 1/m * dZ^{[l]}A^{[l-1]T}$$

$$db^{[l]} = 1/m * np.sum(dZ^{[l]}, axis = 1, keepdims = True)$$

$$dA^{[l-1]} = W^{[l]T}\, dZ^{[l]}$$

$$dZ^{[l-1]} = W^{[l]T}\, dz^{[l]} * g_{l-1}'(Z^{[l-1]})$$

Hyperparameters

Parameters are the weights and biases in the nodes. Quantities which determine the learning, modification or update of these parameters are called hyperparameters. Examples are learning rate α, iterations, batch size, hidden layers L, choice of activation function etc.