Deep Learning

2/ Deep Neural Networks

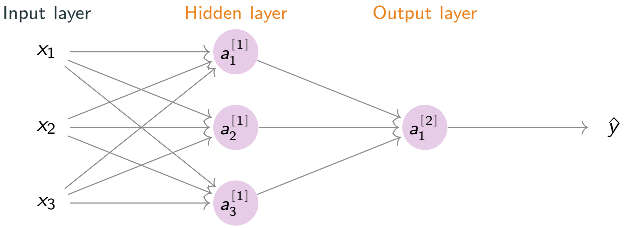
Francesca Galassi, MCF, Esir

francesca.galassi@irisa.fr
Lab Empenn Irisa-Inria

Activation Functions

Activation Functions

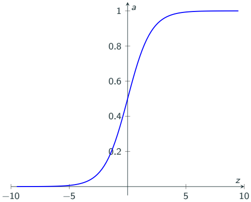
➟ Activation function : a non-linear function g that maps $\mathbb{R}$ to $\mathbb{R}$, e.g., the sigmoid function.



- Given input x :

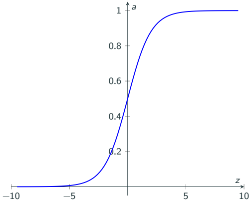
$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma\left(z^{[1]}\right) \leftarrow g\left(z^{[1]}\right)$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma\left(z^{[2]}\right) \leftarrow g\left(z^{[2]}\right)$$

- Sigmoid activation function : $\sigma(z) = \frac{1}{1+e^{-z}}$

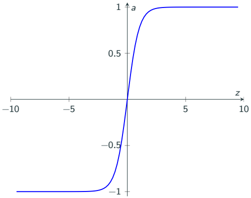


Activation Functions

Sigmoid : $a = \frac{1}{1+e^{-z}}$



Tanh : $a = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$



- Activation functions can vary across layers.

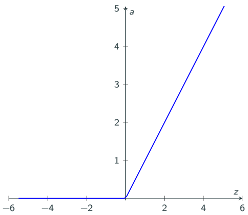
✓ **Tanh zero-centers** data → training is easier.

✓ **Sigmoid** at the **output layer** for **binary classification** | $y \in \{0, 1\}$, $0 \leqslant \hat{y} \leqslant 1$.

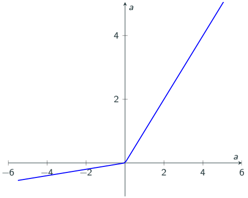
✘ Downside of both : if z is very small or large $\implies$ slope close to zero and gradient descent can slow down.

Activation Functions

ReLU : $a = \max(0, z)$



Leaky ReLU : $a = \max(0.01z, z)$



- ✓ Rectified Linear Unit **ReLU** : default choice.
- ✓ **Leaky ReLU** : if z negative, derivative is different than 0.
- ✓ For both : slope very different from zero, thus **training is faster**.

Activation Functions : why non-linear ?

- Why not use a linear function ?

  Suppose $g(z) = z$, i.e., identity :

$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = g^{[1]}\left(z^{[1]}\right) \leftarrow z^{[1]}$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}\left(z^{[2]}\right) \leftarrow z^{[2]}$$

  Solving :

$$\begin{aligned} a^{[2]} &= W^{[2]} \underbrace{\left(W^{[1]}x + b^{[1]}\right)}_{a^{[1]}} + b^{[2]} \\ &= \underbrace{\left(W^{[2]}W^{[1]}\right)}_{W'} x + \underbrace{\left(W^{[2]}b^{[1]} + b^{[2]}\right)}_{b'} \\ &= W'x + b' \end{aligned}$$

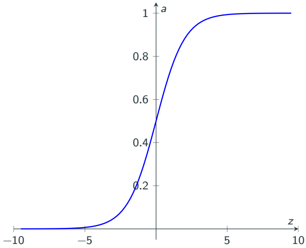
- Using a linear activation function in a neural network leads to successive linear transformations, essentially performing linear regression.

- **Remark** : linear function suitable for the output layer in regression tasks where the target variable is continuous.

Activation Functions : derivatives

➟ In **backpropagation** for neural networks, the derivative of the activation function is computed to calculate the gradients with respect to the loss function.

- Sigmoid : $g(z) = \frac{1}{1+e^{-z}}$



- Derivative :

$$g' = \frac{d}{dz}g(z) = \text{ slope of } g(z) \text{ at } z$$
$$= \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$$
$$= g(z)(1-g(z))$$

- Sanity check :

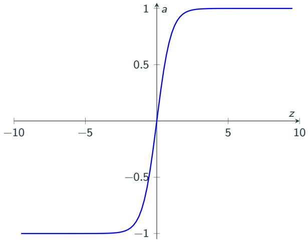
$z = 10$; $g(z) \approx 1$ $\quad g'(z) \approx 0$

$z = -10$; $g(z) \approx 0$ $\quad g'(z) \approx 0$

$z = 0$; $g(z) = \frac{1}{2}$ $\quad g'(z) = \frac{1}{4}$

Activation Functions : derivatives

- Tanh : $g(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$



- Derivative :

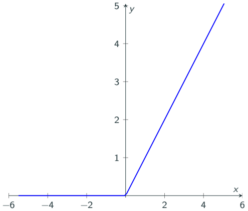
$$g' = \frac{d}{dz} g(z) = \text{ slope of } g(z) \text{ at } z$$
$$= 1 - (\tanh(z))^2$$

- Sanity check :

$$z = 10 \quad ; \ \tanh(z) \approx 1 \qquad g'(z) \approx 0$$
$$z = -10 \quad ; \ \tanh(z) \approx -1 \qquad g'(z) \approx 0$$
$$z = 0 \quad ; \ \tanh(z) = 0 \qquad g'(z) = 1$$

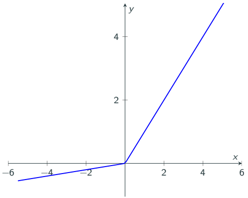
Derivatives of Activation Functions

- ReLU : $g(z) = \max(0, z)$



$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \textit{undefined} & \text{if } z = 0 \end{cases}$$

- Leaky ReLU : $g(z) = \max(0.01z, z)$



$$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \textit{undefined} & \text{if } z = 0 \end{cases}$$

Gradient Descent for NN

Gradient descent

- **Parameters** (2-layer NN) :$W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$
- **Cost function** : $J\left(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}\right) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$
- **Gradient descent algortihm** :

  i. Initialize parameters

  ii. Repeat until convergence :

  a. Compute predictions : $\hat{y}^{(i)}$

  b. Compute derivatives : $\frac{dJ}{dW^{[1]}}$, $\frac{dJ}{db^{[1]}}$, $\frac{dJ}{dW^{[2]}}$, $\frac{dJ}{db^{[2]}}$

  c. Update parameters :

$$W^{[1]} := W^{[1]} - \alpha \frac{dJ}{dW^{[1]}}$$
$$b^{[1]} := b^{[1]} - \alpha \frac{dJ}{db^{[1]}}$$
$$W^{[2]} := W^{[2]} - \alpha \frac{dJ}{dW^{[2]}}$$
$$b^{[2]} := b^{[2]} - \alpha \frac{dJ}{db^{[2]}}$$

, α is the learning rate

  iii. Return parameters.

Gradient Descent : Computing Derivatives

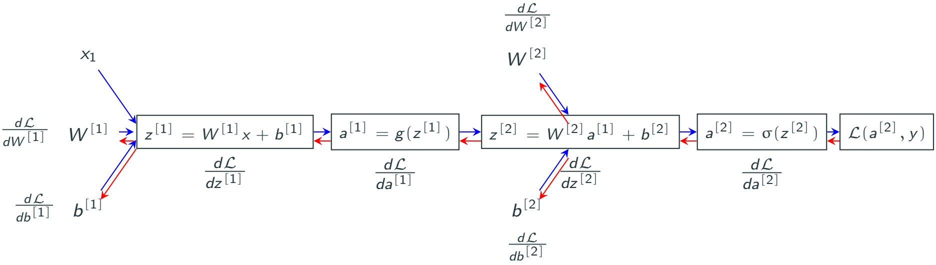
➡ **Forward Propagation** (left-to-right) for computing the output, e.g., for binary classification :

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g^{[1]}\left(Z^{[1]}\right)$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}\left(Z^{[2]}\right) = \sigma\left(Z^{[2]}\right)$$

➡ **Back-Propagation** (right-to-left) for deriving derivatives to apply gradient descent.

Gradient Descent : Computing Derivatives

➟ **Computation Graph** : Deriving the equations to update the parameters for a 2-layer NN.



➟ One backward pass :

$$\frac{d\mathcal{L}}{da^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

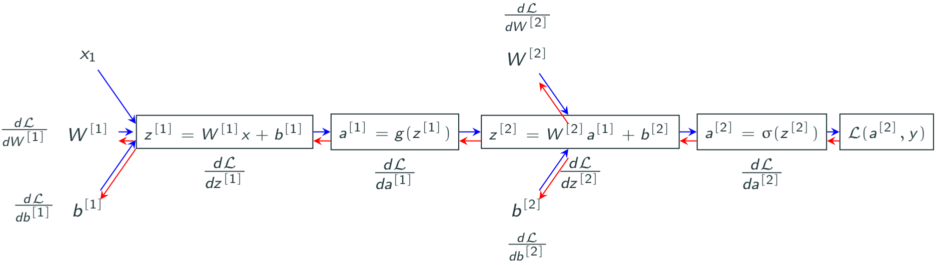
➟ Applying the chain rule, at layer $\ell = 2$:

$$\frac{d\mathcal{L}}{dz^{[2]}} = \frac{d\mathcal{L}}{da^{[2]}} \cdot \frac{da^{[2]}}{dz^{[2]}} = \left(-\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}\right) \cdot a^{[2]}(1-a^{[2]}) = a^{[2]} - y$$

$$\frac{d\mathcal{L}}{dW^{[2]}} = \frac{d\mathcal{L}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{dW^{[2]}} = (a^{[2]} - y) \cdot a^{[1]T} \quad ; \quad \frac{d\mathcal{L}}{db^{[2]}} = a^{[2]} - y$$

Computation graph for NN

➟ Computation graph for one training example and a 2-layer NN :



➟ Applying the chain rule, at layer $\ell = 1$:

$$\underset{(n^{[1]},1)}{\frac{d\mathcal{L}}{da^{[1]}}} = \underset{(n^{[1]},n^{[2]})}{W^{[2]T}} \cdot \underset{(n^{[2]},1)}{(a^{[2]} - y)}$$

$$\underset{(n^{[1]},1)}{\frac{d\mathcal{L}}{dz^{[1]}}} = \frac{d\mathcal{L}}{da^{[1]}} \cdot \frac{da^{[1]}}{dz^{[1]}} = \underset{(n^{[1]},n^{[2]})}{W^{[2]T}} \cdot \underset{(n^{[2]},1)}{(a^{[2]} - y)} * \underset{(n^{[1]},1)}{g'(z^{[1]})}$$

$$\underset{(n^{[1]},n^{[0]})}{\frac{d\mathcal{L}}{dW^{[1]}}} = \frac{d\mathcal{L}}{dz^{[1]}} \cdot \frac{dz^{[1]}}{dW^{[1]}} = \frac{d\mathcal{L}}{dz^{[1]}} \cdot x^T \quad ; \quad \frac{d\mathcal{L}}{db^{[1]}} = \frac{d\mathcal{L}}{dz^{[1]}}$$

Computation graph for NN

- Main equations :

$$dz^{[2]} = a^{[2]} - y$$
$$dW^{[2]} = dz^{[2]} {a^{[1]}}^T$$
$$db^{[2]} = dz^{[2]}$$
$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}\left(z^{[1]}\right)$$
$$dW^{[1]} = dz^{[1]} x^T$$
$$db^{[1]} = dz^{[1]}$$

- Vectorizing across m training examples :

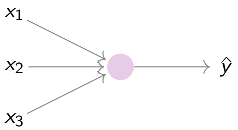
$$dZ^{[2]} = A^{[2]} - Y$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} {A^{[1]}}^T$$
$$db^{[2]} = \frac{1}{m} \text{np.sum}\left(dZ^{[2]},\ \text{axis} = 1,\ \text{keepdims} = \text{ True }\right)$$
$$\underset{(n^{[1]},m)}{dZ^{[1]}} = \underset{(n^{[1]},m)}{W^{[2]T} dZ^{[2]}} * \underset{(n^{[1]},m)}{g^{[1]'}(Z^{[1]})}$$
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} \text{np.sum}\left(dZ^{[1]},\ \text{axis} = 1,\ \text{keepdims} = \text{True }\right)$$

Note : $\frac{d\mathcal{L}}{dvar} = dvar$

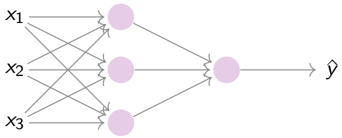
Deep Neural Network

Deep Neural Network

- Depth in neural networks refers to the number of hidden layers in the network.
- Logistic Regression : 1-layer NN, i.e., shallow NN :

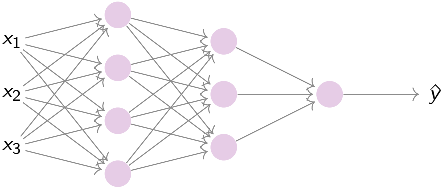


- 2-layer NN (1 hidden layer) :

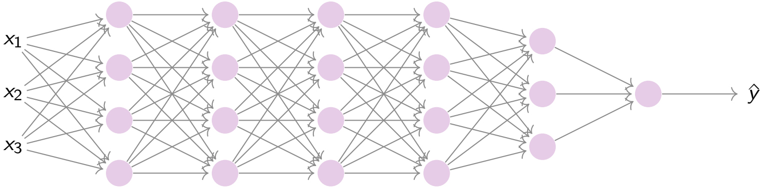


Deep Neural Network

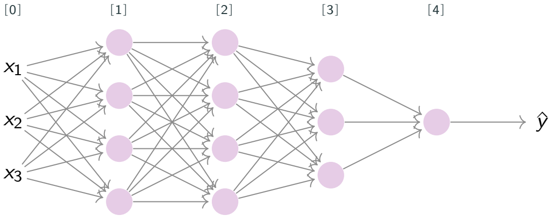
➟ 3-layer NN (2 hidden layers) :



➟ 6-layer NN (5 hidden layers), i.e., deep NN :

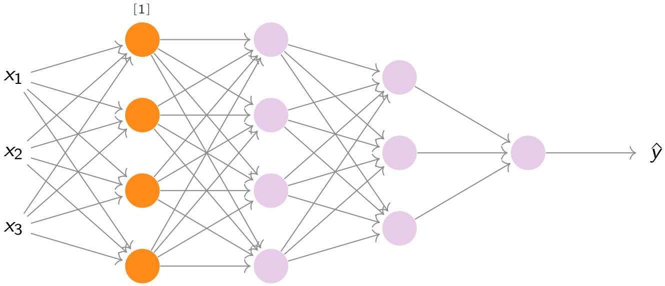


Deep Neural Network : Notation



- $L = 4$ number of layers (3 hidden layers)
- $n^{[\ell]}$ denotes the number of units in layer ℓ
  - $n^{[1]} = 4$; $n^{[2]} = 4$; $n^{[3]} = 3$; $n^{[4]} = n^{[L]} = 1$
  - $n^{[0]} = n = 3$
- $a^{[\ell]}$ denotes the activations in layer ℓ
  - $a^{[\ell]} = g^{[\ell]}(z^{[\ell]})$
  - $a^{[0]} = x$, $a^{[L]} = \hat{y}$
- $W^{[\ell]}$, $b^{[\ell]}$ parameters for computing $z^{[\ell]}$

Deep Neural Network : Forward Propagation

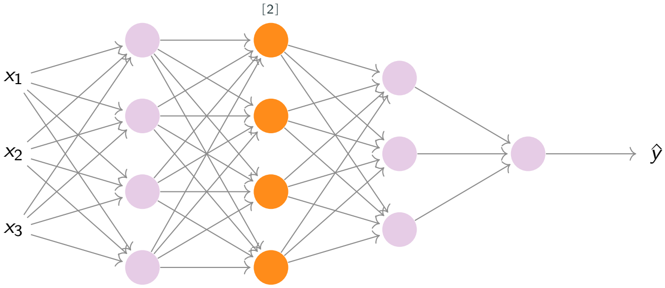


- For a single training example x, at layer $\ell = 1$:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = g^{[1]}\left(z^{[1]}\right)$$

Deep Neural Network : Forward Propagation

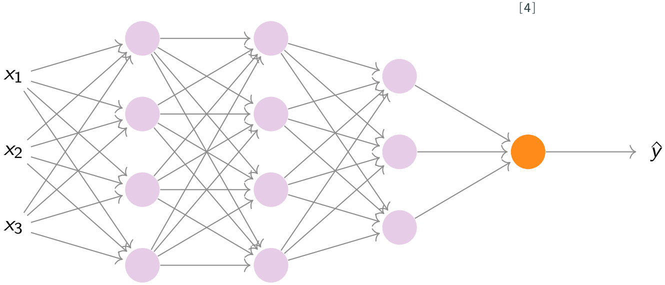


- For a single training example x, at layer $\ell = 2$:

$$z^{[2]} = W^{[2]} \; a^{[1]} \; + b^{[2]}$$

$$a^{[2]} = g^{[2]}\left(z^{[2]}\right)$$

Deep Neural Network : Forward Propagation



- For a single training example x, at layer $\ell = 4$:

$$z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$$

$$a^{[4]} = g^{[4]}\left(z^{[4]}\right)$$

- General **forward propagation** equations :

$$\boxed{\begin{aligned} z^{[\ell]} &= W^{[\ell]} a^{[\ell-1]} + b^{[\ell]} \\ a^{[\ell]} &= g^{[\ell]}\left(z^{[\ell]}\right) \end{aligned}}$$

Note : $a^{[0]} = x$

Recall : Vectorized Version Over Training Examples

- Stacking training examples in columns :

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix}$$

- Computing linear combinations :

$$Z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = W^{[1]}X + b^{[1]}$$

- Broadcasting for bias addition :

$$\tilde{b}^{[1]} = \begin{bmatrix} | & | & | \\ b^{[1]} & b^{[1]} & b^{[1]} \\ | & | & | \end{bmatrix}$$

Deep Neural Network : forward propagation

➡ The **vectorized version** computes **for all** m **training examples simultaneously** using matrix operations, leading to significant speedup.

- **Vectorized** version :

$$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]} \quad , X = A^0$$
$$A^{[1]} = g^{[1]}\left(Z^{[1]}\right)$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}\left(Z^{[2]}\right)$$
$$\dots$$
$$\hat{Y} = g\left(Z^{[4]}\right) = A^{[4]}$$

➡ Vectorized version for an L-layer Neural Network :

for $\ell = 1$ to L **do**

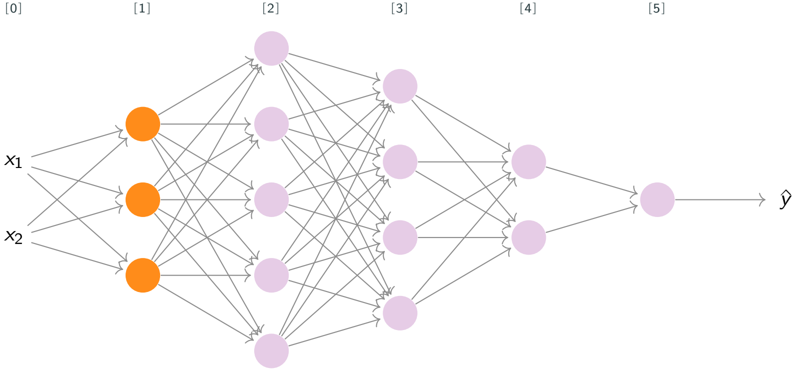
$\quad Z^{[\ell]} \leftarrow W^{[\ell]}A^{[\ell-1]} + b^{[\ell]}$

$\quad A^{[\ell]} \leftarrow g^{[\ell]}\left(Z^{[\ell]}\right)$

end for

$\hat{Y} \leftarrow g\left(Z^{[\ell]}\right) = A^{[\ell]}$

Forward Propagation : Matrix Dimensions

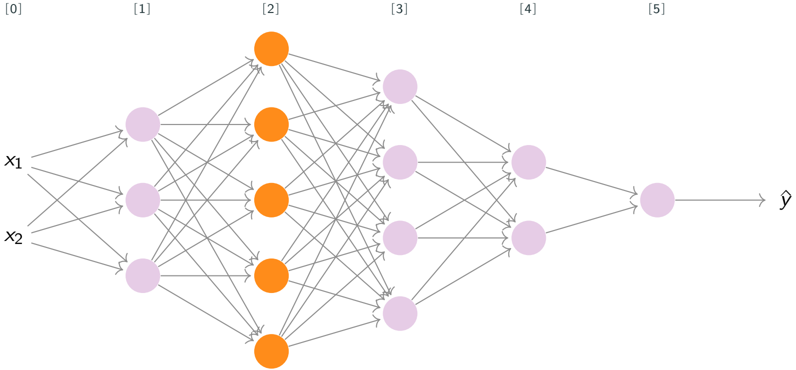


$n^{[0]} = n = 2$; $n^{[1]} = 3$; $n^{[2]} = 5$; $n^{[3]} = 4$; $n^{[4]} = 2$; $n^{[5]} = n^{[L]} = 1$

- For one training example, matrix dimensions at layer 1 :

$$\underset{\substack{(3,1)\\(n^{[1]},1)}}{z^{[1]}} = \underset{\substack{(3,2)\\(n^{[1]},n^{[0]})}}{W^{[1]}} \cdot \underset{\substack{(2,1)\\(n^{[0]},1)}}{x} + \underset{\substack{(3,1)\\(n^{[1]},1)}}{b^{[1]}} ;$$

Forward Propagation : Matrix Dimensions

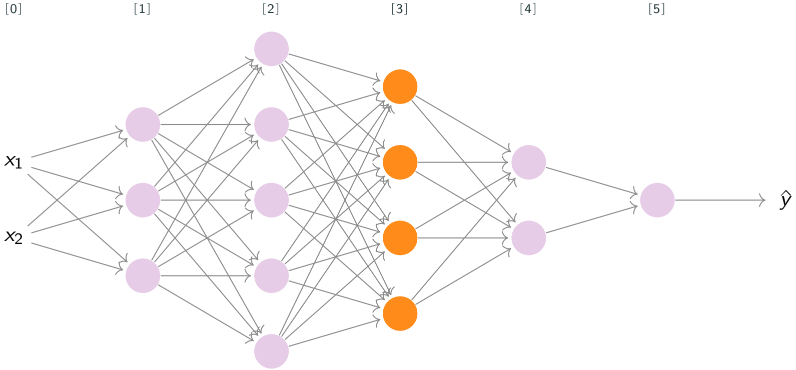


$n^{[0]} = n_x = 2$; $n^{[1]} = 3$; $n^{[2]} = 5$; $n^{[3]} = 4$; $n^{[4]} = 2$; $n^{[5]} = n^{[L]} = 1$

- At layer 2 :

$$\underset{\substack{(5,1)\\(n^{[2]},1)}}{z^{[2]}} = \underset{\substack{(5,3)\\(n^{[2]},n^{[1]})}}{W^{[2]}} \cdot \underset{\substack{(3,1)\\(n^{[1]},1)}}{a^{[1]}} + \underset{\substack{(5,1)\\(n^{[2]},1)}}{b^{[2]}}$$

Forward Propagation : Matrix Dimensions

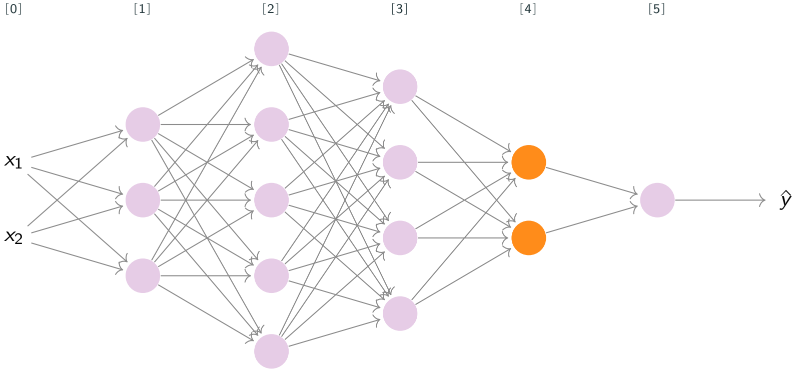


$n^{[0]} = n_x = 2\,;\; n^{[1]} = 3\,;\; \boxed{n^{[2]} = 5}\,;\; \boxed{n^{[3]} = 4}\,;\; n^{[4]} = 2\,;\; n^{[5]} = n^{[L]} = 1$

- At layer 3 :

$$\underset{\substack{(4,1)\\(n^{[3]},1)}}{z^{[3]}} = \underset{\substack{(4,5)\\(n^{[3]},n^{[2]})}}{W^{[3]}} \cdot \underset{\substack{(5,1)\\(n^{[2]},1)}}{a^{[2]}} + \underset{\substack{(4,1)\\(n^{[3]},1)}}{b^{[3]}}$$

Forward Propagation : Matrix Dimensions

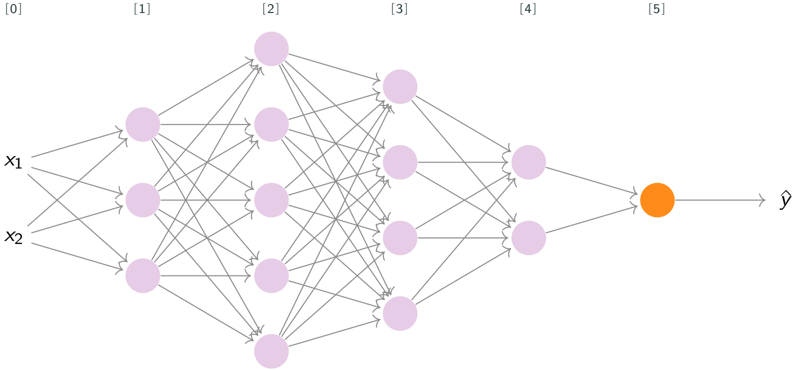


$n^{[0]} = n_x = 2\,;\ n^{[1]} = 3\,;\ n^{[2]} = 5\,;\ n^{[3]} = 4\ ;\ n^{[4]} = 2\ ;\ n^{[5]} = n^{[L]} = 1$

- At layer 4 :

$$\underset{\substack{(2,1)\\(n^{[4]},1)}}{z^{[4]}} = \underset{\substack{(2,4)\\(n^{[4]},n^{[3]})}}{W^{[4]}} \cdot \underset{\substack{(4,1)\\(n^{[4]},1)}}{a^{[3]}} + \underset{\substack{(2,1)\\(n^{[4]},1)}}{b^{[4]}}$$

Forward Propagation : Matrix Dimensions

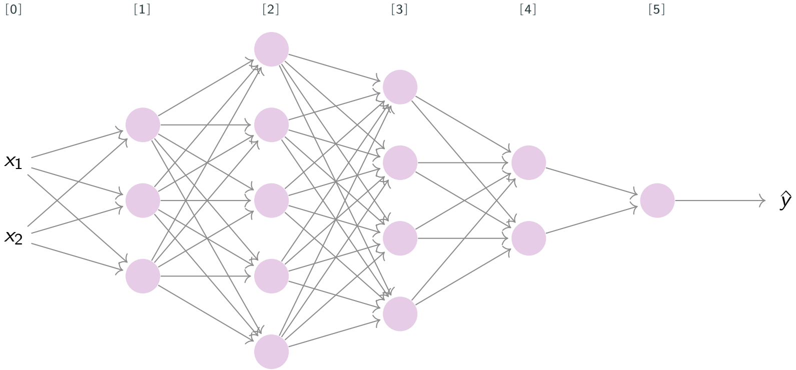


$n^{[0]} = n = 2\,;\; n^{[1]} = 3\,;\; n^{[2]} = 5\,;\; n^{[3]} = 4\,;\; n^{[4]} = 2\,;\; n^{[5]} = n^{[L]} = 1$

- At layer 5 :

$$\underset{\substack{(1,1)\\(n^{[5]},1)}}{z^{[5]}} = \underset{\substack{(1,2)\\(n^{[5]},n^{[4]})}}{W^{[5]}} \cdot \underset{\substack{(2,1)\\(n^{[4]},1)}}{a^{[4]}} + \underset{\substack{(1,1)\\(n^{[5]},1)}}{b^{[5]}}$$

Forward Propagation : Matrix Dimensions



- Matrix dimensions at layer ℓ :

  $W^{[\ell]} : \left(n^{[\ell]}, n^{[\ell-1]}\right)$

  $b^{[\ell]} : \left(n^{[\ell]}, 1\right)$

  $z^{[\ell]} : \left(n^{[\ell]}, 1\right) \leftarrow a^{[\ell]}$ has the same dim

☞ Same dimensions when doing backpropagation :

$dW^{[\ell]} : \left(n^{[\ell]}, n^{[\ell-1]}\right)$

$db^{[\ell]} : \left(n^{[\ell]}, 1\right)$

Forward and Backward Propagation

➟ **Vectorizing** across m training examples :

$$\underset{(n^{[1]},m)}{Z^{[1]}} = \underset{(n^{[1]},n^{[0]})}{W^{[1]}} \cdot \underset{(n^{[0]},m)}{X} + \underset{\underset{(n^{[1]},m)}{\downarrow}}{\underset{(n^{[1]},1)}{b^{[1]}}}$$

$$Z^{[1]} = \underset{(\textit{number of units},\ \textit{number of training examples } m)}{\begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \dots & z^{[1](m)} \\ | & | & & | \end{bmatrix}}$$

➟ Matrix dimensions at a layer ℓ :

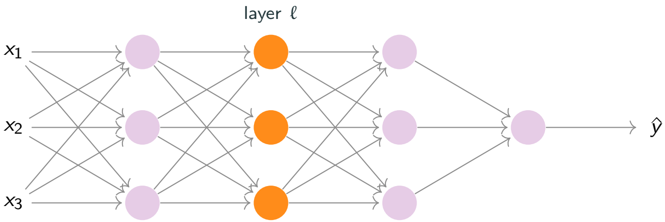
$$Z^{[\ell]}, A^{[\ell]} : \left(n^{[\ell]}, m\right)$$

☞ Same dimensions when doing backprop :

$$dZ^{[\ell]}, dA^{[\ell]} : \left(n^{[\ell]}, m\right)$$

Forward and Backward Functions

Forward and backward functions



➟ Forward implementation at layer $[\ell]$:

- Input : $a^{[\ell-1]}$
- Output : $a^{[\ell]}$

➟ Equations :

$$z^{[\ell]} = W^{[\ell]} \cdot a^{[\ell-1]} + b^{[\ell]}$$

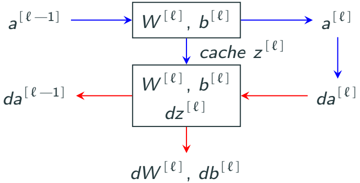
$$a^{[\ell]} = g^{[\ell]}\left(z^{[\ell]}\right)$$

☞ Cache $z^{[\ell]}$, $W^{[\ell]}$, $b^{[\ell]}$

➟ Backward implementation :

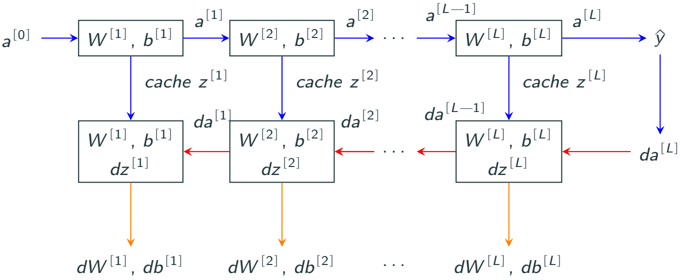
- Input : $da^{[\ell]}$ and the cache $z^{[\ell]}$
- Output : $da^{[\ell-1]}$, $dW^{[\ell]}$, $db^{[\ell]}$

➟ Blocks at layer ℓ :



Forward and Backward Functions

➟ Implementation blocks :



➟ After one iteration of forward and backward propagation, the parameters are updated :

$$W^{[\ell]} = W^{[\ell]} - \alpha \cdot dW^{[\ell]}$$

$$b^{[\ell]} := b^{[\ell]} - \alpha \cdot db^{[\ell]}$$

Forward and Backward Propagation

➟ Backward implementation :

- Equations :

$$dz^{[\ell]} = da^{[\ell]} * g^{[\ell]\prime}\left(z^{[\ell]}\right)$$
$$dW^{[\ell]} = dz^{[\ell]} \cdot a^{[\ell-1]T}$$
$$db^{[\ell]} = dz^{[\ell]}$$
$$da^{[\ell-1]} = W^{[\ell]T} \cdot dz^{[\ell]}$$

➟ Vectorized implementation :

$$\boxed{\begin{aligned} dZ^{[\ell]} &= dA^{[\ell]} * g^{[\ell]\prime}\left(Z^{[\ell]}\right) \\ dW^{[\ell]} &= \frac{1}{m} dZ^{[\ell]} \cdot A^{[\ell-1]T} \\ db^{[\ell]} &= \frac{1}{m}\ \text{np.sum}\ \left(dZ^{[\ell]},\ \text{axis} = 1,\ \text{keepdims} = \text{True}\right) \\ dA^{[\ell-1]} &= W^{[\ell]T} \cdot dZ^{[\ell]} \end{aligned}}$$

Optimization problem

Parameters vs Hyperparameters

➟ Parameters : $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]}, \ldots$

➟ Hyperparameters include :

- Learning rate α
- Number of iterations
- Number of hidden layers L
- Number of hidden units $n^{[1]}, n^{[2]}, \ldots$
- Batch size, regularizations,...

➟ Hyperparameters control the ultimate parameters W and b

➟ Finding the best values is an empirical process that often involves trying out many different values

Normalizing Input Features

➡ Normalizing inputs : Helps optimize the cost function by ensuring input features are on similar scales.

1. Subtracting the mean :

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$$

$$x := x - \mu$$

2. Normalizing the variance :

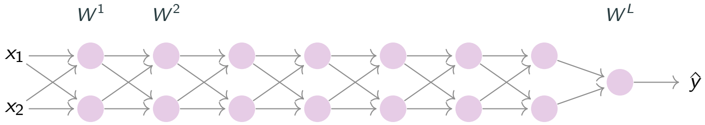
$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})^2$$

$$x := x / \sigma$$

3. Use the same transformation to normalize the test data.

Vanishing / Exploding Gradients

➡ Vanishing or exploding gradients : Derivatives in deep neural networks can become extremely small or large, hindering training.



$g(z) = z \quad , b^{[\ell]} = 0$

$$\hat{y} = W^{[L]} W^{[L-1]} W^{[L-2]} \cdots W^{[3]} W^{[2]} \underbrace{W^{[1]} x}_{z^{[1]} = W^{[1]} x}$$

- Increase exponentially

$$W^{[\ell]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \quad , \quad \hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x \quad \sim \quad 1.5^{L-1} x$$

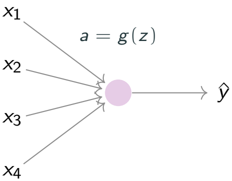
- Decrease exponentially

$$W^{[\ell]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \quad , \quad \hat{y} = W^{[L]} \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{L-1} x \quad \sim \quad 0.5^{L-1} x$$

- Similar behaviour for the derivates

Weight Initialization

- A partial solution to vanishing and exploding gradients is carefully choosing the random initialization for neural networks.



$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

large $n \longrightarrow$ smaller w_i

$$\mathrm{Var}(w_i) = \frac{1}{n}$$

- Small random values are scaled by a factor determined by the number of units n in the previous layer.
  - For ReLU :
    $W^{[\ell]} =$ np.random.randn(shape) $*$ np.sqrt $\left(\frac{2}{n^{[\ell-1]}}\right)$ He initialization
  - For tanh :
    $W^{[\ell]} =$ np.random.randn(shape) $*$ np.sqrt $\left(\frac{1}{n^{[\ell-1]}}\right)$ Xavier initialization

Mini-batch gradient descent

- Vectorization allows you to efficiently compute on m examples (+ stability, + generalization)

$$\underset{(n_x,m)}{X} = [x^{(1)} x^{(2)} x^{(3)} \dots x^{(m)}]$$

$$\underset{(1,m)}{Y} = [y^{(1)} y^{(2)} y^{(3)} \dots y^{(m)}]$$

- We process the entire training sets before taking one step of gradient descent

- If m is very large, e.g., $m = 5,000,000 \longrightarrow$ split the training set into mini-batches of size 1000 :

$$\underset{(n,m)}{X} = [\underbrace{x^{(1)} x^{(2)} x^{(3)} \dots x^{(1000)}}_{\underset{(n,1000)}{X^{\{1\}}}} \mid \underbrace{x^{(1001)} \dots x^{(2000)}}_{X^{\{2\}}} \mid \dots \mid \underbrace{\dots x^{(m)}}_{X^{\{5000\}}}]$$

$$\underset{(1,m)}{Y} = [\underbrace{y^{(1)} y^{(2)} y^{(3)} \dots y^{(1000)}}_{\underset{(1,1000)}{Y^{\{1\}}}} \mid \underbrace{y^{(1001)} \dots y^{(2000)}}_{Y^{\{2\}}} \mid \dots \mid \underbrace{\dots y^{(m)}}_{Y^{\{5000\}}}]$$

$\{t\}$: the $t - th$ mini batch

Mini-batch gradient descent

➟ Update step performed on mini-batches (batch size is a hyperparameter)

for $t = 1$ to 5000 **do**

$$\left.\begin{aligned} Z^{[1]} &= W^{[1]}X^{\{t\}} + b^{[1]} \\ A^{[1]} &= g^{[1]}\left(Z^{[1]}\right) \\ &\vdots \\ A^{[L]} &= g^{[L]}\left(Z^{[L]}\right) \end{aligned}\right\}\text{Vectorized implementation (1000 examples)}$$

$$J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{1000}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \text{ regularisation term}$$

Backpropagate to compute gradients w.r.t. $J^{\{t\}}$

$W^{[\ell]} := W^{[\ell]} - \alpha dW^{[\ell]}, \quad b^{[\ell]} := b^{[\ell]} - \alpha db^{\ell]}$

end for

➟ 1 **epoch** : a single pass through the training set.

Mini-batch gradient descent

➟ Batch gradient descent : Batch size = m

- ☞ Cost function decreases consistently
- Takes too long per iteration
- Suitable for small training sets (<2000 examples)

➟ Stochastic gradient descent : Batch size = 1

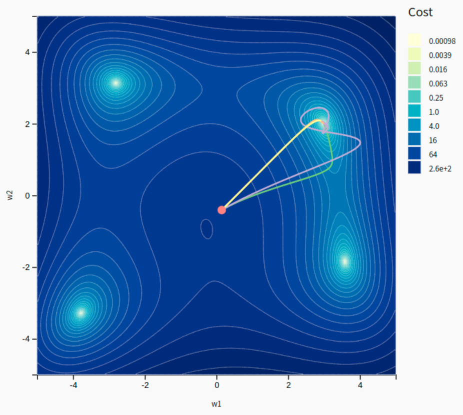
- Cost function oscillates, resulting in noisy gradients
- Wanders around the minimum
- Inefficient
- Requires smaller learning rates

➟ Mini-batch gradient descent : Batch size between 1 and m

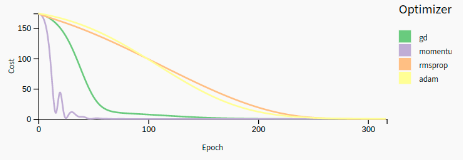
- ☞ Faster than batch gradient descent
- ☞ More stable direction towards the minimum than stochastic gradient descent
- May oscillate in a small region and not always converge precisely
- Reducing the learning rate gradually can help
- Typical batch sizes : 64, 128, 256, 512 ; Ensure mini-batch fits in memory

Optimizers : Overview

➟ Analyzing Cost for Different Optimizers Across Successive Epochs ($\alpha = 0.001$).



Visualizing the Cost Landscape



Cost Evolution Over Epochs

Optimizers : Overview

- Gradient Descent (GD)
  - ✓ Simple and easy to implement.
  - ✗ Sensitive to learning rate.
  - ✗ Can be slow in convergence.
- Momentum
  - ✓ Accelerates convergence with a *momentum* term.
  - ✓ Helps overcome flat gradients.
  - ✗ May overshoot minima.
- RMSprop
  - ✓ Adapts learning rates for each parameter.
  - ✓ Effective with different feature scales.
  - ✗ Requires more memory.
- Adam
  - ✓ Fast convergence with adaptive learning rates.
  - ✓ Combines benefits of Momentum and RMSprop.
  - ✗ May require hyperparameter tuning.