Course 1 - Neural Networks and Deep Learning

Mohammad Khalaji

August 11, 2020

1 Week 1

Sequence Data: Audio signals and natural language sentences are considered sequence data because they have a ”temporal” component. For these kinds of inputs, we often use Recurrent Neural Networks (RNNs).

Hybrid Neural Networks: Consider a situation where we have an image alongside with radar information and we want to predict the position of other cars. In this example, we need a CNN for processing the image, in addition to a standard NN in order to process the radar info. As a result, we have to make a custom hybrid neural network.

Neural Networks' take off reasons

- **Data:** more data is available now, and the performance of neural networks is highly dependent on the availability of data.
- **Computation:** Hardware advancement etc. The process of deep learning is extremely iterative, so we need to do the computations fast enough in order to work effectively.
- **Algorithms:** Algorithmic innovations. Example: switching from sigmoid, which slows down the learning process due to it's gradient's being small, to ReLU. Switching to ReLU made the gradient descent algorithm much more efficient.

2 Week 2

Notation

- number of features: n_x
- (x, y)
- $x \in \mathbb{R}^{n_x}$
- m training samples: $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$
- $X = \begin{bmatrix} | & & | \\ x^{(1)} & \ldots & x^{(m)} \\ | & & | \end{bmatrix}$, EACH ROW IS A FEATURE. $X_{n_x \times m}$. This way, implementation will be much easier.
- $Y = \left[y^{(1)}, \ldots, y^{(m)}\right]$, $Y_{1 \times m}$

2.1 Logistic Regression

Given X, want $\hat{y} = P(y = 1|X)$, knowing that $x_i \in \mathbb{R}^{n_x}$.

Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

Output: $\sigma(w^T x + b)$, given that $\sigma(z) = \frac{1}{1+e^{-z}}$

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

- if $y = 1$, $Loss = -log(\hat{y})$, so we want $\log(\hat{y})$ and therefore $\hat{y}$ as big as possible (one) to minimize the loss.
- if $y = 0$, $Loss = -log(1-\hat{y})$, so we want $\log(1-\hat{y})$ and therefore $1-\hat{y}$ as big as possible, which means $\hat{y}$ as small as possible (zero)), to minimize the loss.
- Cost function:

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

2.2 Gradient Descent

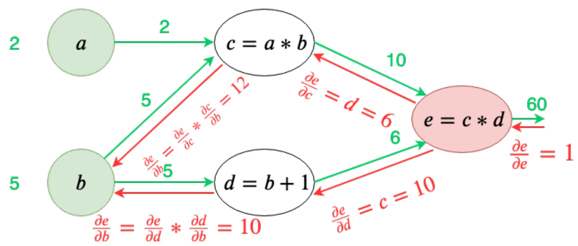
We want to find w, b such that $J(w, b)$ is minimized.

- Repeat:

  $w = w - \alpha \frac{\partial J(w,b)}{\partial w}$

  $b = b - \alpha \frac{\partial J(w,b)}{\partial b}$

2.3 Partial Derivatives



2.4 Gradient Descent for Logistic Regression

Feed Forward:

$$z = w^T x + b$$

$$\hat{y} = a = \sigma(z)$$

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

$$z = w_1 x_1 + w_2 x_2 + b$$

$$a = \sigma(z), L(a, y) = -\Big(ylog(a) + (1-y)log(1-a)\Big)$$

Notation:

$$dx = \frac{dL}{dx}$$

Now backpropagate:

$$da = \frac{dL}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$dz = \frac{dL}{dz} = \frac{dL}{da}\frac{da}{dz} = \Big[-\frac{y}{a} + \frac{1-y}{1-a}\Big]\Big[a(1-a)\Big] = a - y$$

$$dw_1 = \frac{dL}{dw_1} = \frac{dL}{dz}\frac{dz}{dw_1} = x_1 dz$$

$$dw_2 = \frac{dL}{dw_2} = \frac{dL}{dz}\frac{dz}{dw_2} = x_2 dz$$

$$db = \frac{dL}{db} = \frac{dL}{dz}\frac{dz}{db} = dz$$

2.5 Gradient Descent on m Training Samples

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

$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

What we need for each sample in each iteration:

$$dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$$

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_1} L(a^{(i)}, y^{(i)}) = \frac{1}{m}\sum_{i=1}^{m} dw_1^{(i)}$$

2.6 Vectorization

Vectorization: getting rid of explicit for loops and calculating $dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$ for each sample (and accumulating them into dw_1, dw_2, db over the course of the loop)

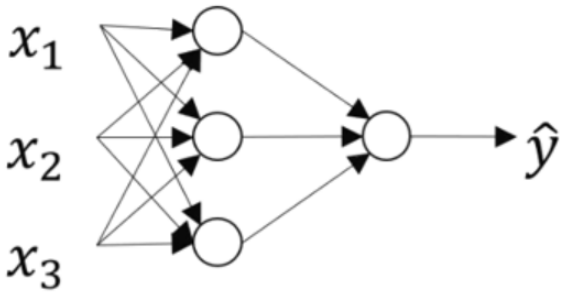
$$Z = \begin{bmatrix} | & & | \\ z^{(1)} & \dots & z^{(m)} \\ | & & | \end{bmatrix} = w^T X + \begin{bmatrix} b & \dots & b \end{bmatrix}_{1\times m} = \begin{bmatrix} w^T x^{(1)} + b & \dots & w^T x^{(m)} + b \end{bmatrix}$$

The numpy command to calculate Z: $Z = np.dot(w.T, X) + b$

$$A = \begin{bmatrix} | & & | \\ a^{(1)} & \dots & a^{(m)} \\ | & & | \end{bmatrix}$$

3 Week 3

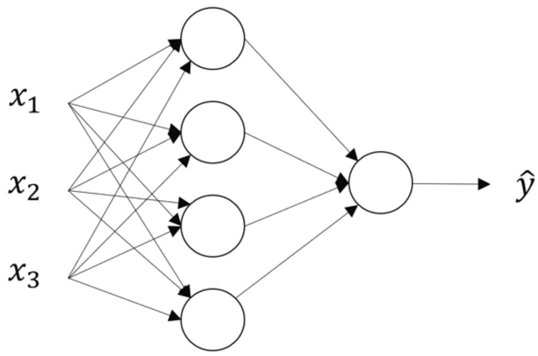
3.1 Overview



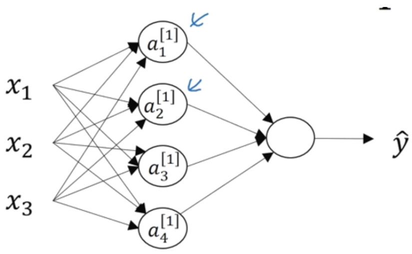
Steps:

1. $z^{[1]} = W^{[1]}x + b^{[1]}$
2. $a^{[1]} = \sigma(z^{[1]})$
3. $z^{[2]} = W^{[2]}a^{[2]} + b^{[2]}$
4. $a^{[2]} = \sigma(z^{[2]})$
5. $\hat{y} = a^{[2]}$
6. $L(a^{[2]}, y)$

3.2 Neural Network Representations



- Two layers, no. of layers = no. of hidden layers + 1 output layer
- $$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$
- $$w^{[1]}_{4\times 3}, b^{[1]}_{4\times 1}$$
- $$w^{[2]}_{1\times 4}, b^{[2]}_{1\times 1}$$



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

$$z^{[1]} = W^{[1]}x + b^{[1]} = \begin{bmatrix} -w_1^{[1]T}- \\ -w_2^{[1]T}- \\ -w_3^{[1]T}- \\ -w_4^{[1]T}- \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T}x + b_1^{[1]} \\ w_2^{[1]T}x + b_2^{[1]} \\ w_3^{[1]T}x + b_3^{[1]} \\ w_4^{[1]T}x + b_4^{[1]} \end{bmatrix}$$

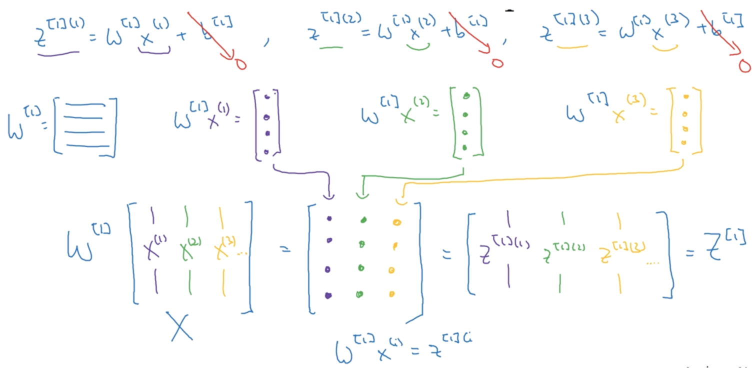
Note that $w_i^{[1]T} = i$'th row of the $W^{[1]}$ matrix. Given input vector x:

$$z^{[1]}_{4\times 1} = W^{[1]}_{4\times 3} x_{3\times 1} + b^{[1]}_{4\times 1}$$

$$a^{[1]}_{4\times 1} = \sigma(z^{[1]})$$
$$z^{[2]}_{1\times 1} = W^{[2]}_{1\times 4} a^{[1]}_{4\times 1} + b^{[2]}_{1\times 1}$$
$$a^{[2]}_{1\times 1} = \sigma(z^{[2]})$$

3.3 Vectorization over the Full X Matrix

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



Where:

$$Z^{[1]} = \begin{bmatrix} | & & | \\ z^{[1](1)} & \dots & z^{[1](m)} \\ | & & | \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} | & & | \\ a^{[1](1)} & \dots & a^{[1](m)} \\ | & & | \end{bmatrix}$$

3.4 Why do we need non-linear activation functions?

Consider a identity $a = z$ activation function, which implies that $A = Z$. Given the input vector x:

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

Let's rewrite $\hat{y} = a^{[2]}$ in terms of x:

$$a^{[2]} = W^{[2]}a^{[1]} + b^{[2]} = W^{[2]}\Big(W^{[1]}x + b^{[1]}\Big) + b^{[2]}$$
$$= W^{[2]}W^{[1]}x + W^{[2]}b^{[1]} + b^{[2]}$$
$$W^{'}x + b^{'}$$

As a result, a neural network with a linear activation function is just outputting a linear combination of the input. No matter how many layer it has, it is as if they are non-existent. A linear layer is essentially useless. Using $g(z) = z$ as an activation function is allowed if you're solving a regression problem with a neural network + some other rare cases.

3.5 Derivatives of Activation Functions

Sigmoid:

$$a = g(z) = \frac{1}{1 + e^{-z}}$$
$$\frac{d}{dz}g(z) = g(z)\Big(1 - g(z)\Big) = a(1 - a)$$

Hyperbolic Tangent:

$$a = g(z) = tanh(z)$$
$$\frac{d}{dz}g(z) = 1 - tanh^2(z) = 1 - a^2$$

ReLU and Leaky ReLU: too obvious.

3.6 Gradient Descent for Neural Networks

- Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$
- Parameters' Shapes: $(n^{[1]} \times n^{[0]}), (n^{[1]} \times 1), (n^{[2]} \times n^{[1]}), (n^{[2]} \times 1)$
- Cost Function: $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$

Gradient Descent:

- Repeat

$$
\begin{aligned}
&\text{Compute } \hat{y}^{(i)} \text{ for } i = 1 \dots m \\
&dW^{[1]} = \frac{\partial J}{\partial W^{[1]}},\ db^{[1]} = \frac{\partial J}{\partial b^{[1]}},\ dW^{[2]} = \frac{\partial J}{\partial W^{[2]}},\ db^{[2]} = \frac{\partial J}{\partial b^{[2]}} \\
&W^{[1]} = W^{[1]} - \alpha dW^{[1]} \\
&b^{[1]} = b^{[1]} - \alpha db^{[1]} \\
&W^{[2]} = W^{[2]} - \alpha dW^{[2]} \\
&b^{[2]} = b^{[2]} - \alpha db^{[2]}
\end{aligned}
$$

Summary of gradient descent

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]} a^{[1]^T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$
$dW^{[1]} = dz^{[1]} x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Andrew Ng

4 Week 4

4.1 Notation

- L: Number of Layers (# hidden + one output layer)
- $n^{[l]}$: Number of Neurons in Layer l
- $g^{[l]}$: The Activation Function in Layer l
- $w^{[l]}$: The Weights for $z^{[l]}$
- $b^{[l]}$: The Biases for $z^{[l]}$

4.2 Forward Propagation

The non-vectorized version is not included here. However, this is how the vectorized version works given $X_{n_x \times m}$ as the input matrix ($L = 4$):

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(Z^{[2]})$$

$$Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]}$$

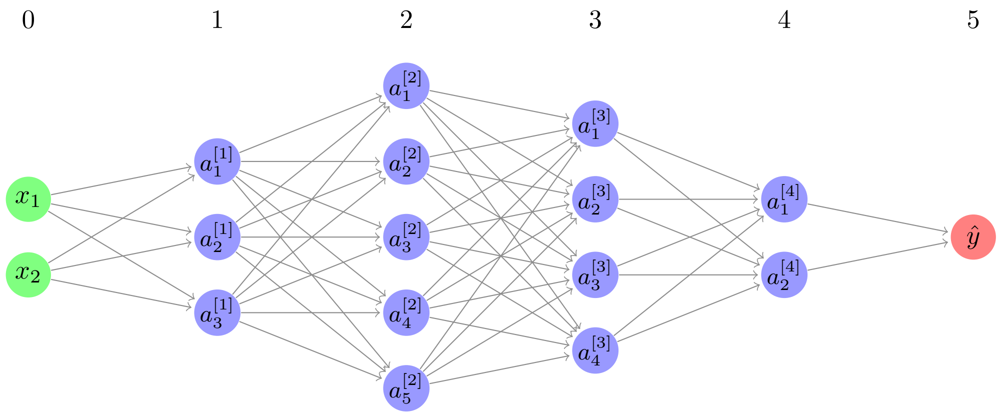
$$A^{[3]} = g^{[3]}(Z^{[3]})$$

$$Z^{[4]} = W^{[4]}A^{[3]} + b^{[4]}$$

$$\hat{Y} = A^{[4]} = g^{[4]}(Z^{[4]})$$

4.3 Getting Matrix Dimensions Right

Consider a neural network in which $n_x = 2$, $L = 5$, $n^{[1]} = 3$, $n^{[2]} = 5$, $n^{[3]} = 4$, $n^{[4]} = 2$, and $n^{[5]} = 1$. Only the vectorized version is included here.



$$Z^{[1]} = W^{[1]}X + b^{[1]} \rightarrow X_{n_x \times m}, W^{[1]}_{n^{[1]} \times n_x}, b^{[1]}_{n^{[1]} \times 1}$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \rightarrow A^{[1]}_{n^{[1]} \times m}, W^{[2]}_{n^{[2]} \times n^{[1]}}, b^{[2]}_{n^{[2]} \times 1}$$

As a general rule:

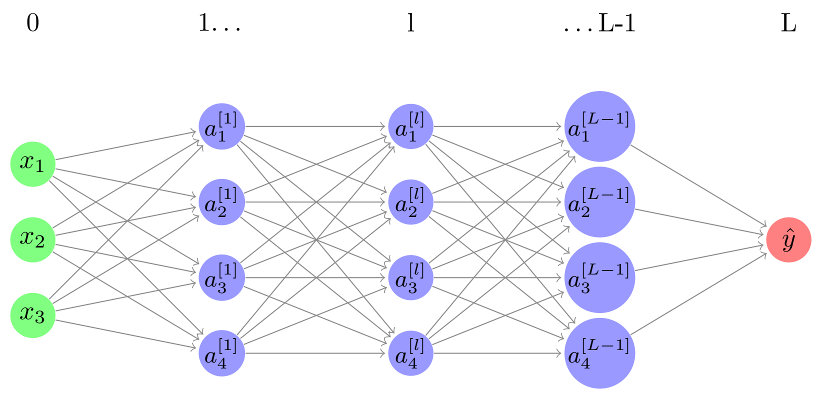
$$W^{[l]}.shape = n^{[l]} \times n^{[l-1]}$$

$$b^{[l]}.shape = n^{[l]} \times 1$$

$$Z^{[l]}.shape = A^{[l]}.shape = n^{[l]} \times m$$

4.4 Building Blocks of A Deep Neural Network

Consider this neural network as an example:



In layer l:

- Parameters: $W^{[l]}, b^{[l]}$
- Input: $a^{[l-1]}$
- Output: $z^{[l]}$
- Cache: $z^{[l]}, W^{[l]}, b^{[l]}$
- Backward Input: $da^{[l]}$
- Backward Output: $da^{[l-1]}, dW^{[l]}, db^{[l]}$

So, the backward step takes $da^{[l]}$ as input, and outputs $da^{[l-1]}, dW^{[l]}, db^{[l]}$. How?

Non-vectorized version:

$$dz^{[l]} = da^{[l]} * g^{[l]'(z^{[l]})}$$
$$dW^{[l]} = dz^{[l]}.a^{[l-1]}$$
$$db^{[l]} = dz^{[l]}$$
$$da^{[l-1]} = W^{[l]T}.dz^{[l]}$$

Vectorized version:

$$dZ^{[l]} = dA^{[l]} * g^{[l]'(Z^{[l]})}$$
$$dW^{[l]} = \frac{1}{m} dZ^{[l]}.A^{[l-1]T}$$
$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis = 1, keepDims = True)$$
$$dA^{[l-1]} = W^{[l]T}.dZ^{[l]}$$