2025-07-09T10-01-13
Tags: DLS

DLS C1 week 3

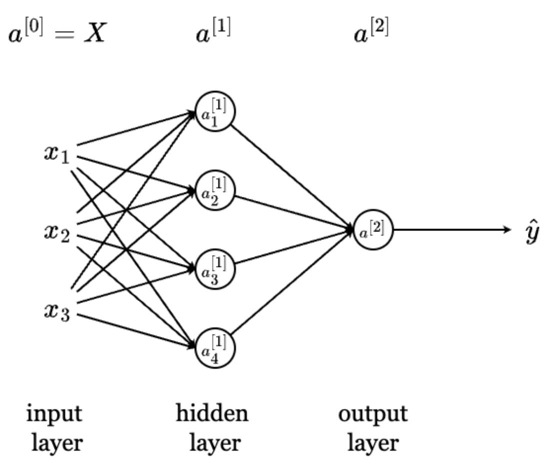
Bracket notations

Superscript square brackets	• Used to refer quantities associated with a layer. $z^{[1]} = W^{[1]} + b^{[1]}$	Superscript round brackets	• Used to refer to individual training examples. $z^{(1)} = w^{\intercal} x^{(1)} + b$

In a neural network, you do the linear and activation function multiple times before computing the loss.

Neural network representation

A two layer neural network:



It is a two layer neural network because the input layer is not counted per convention.

The notation $a^{[l]}$ means activations.

$a^{[1]}$ above is a four dimensional vector:

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_2^{[1]} \\ a_2^{[1]} \end{bmatrix}$$

$a^{[2]}$ is a real number.

The prediction is $\hat{y} = a^{[2]} = \sigma(z)$.

The hidden layer has associated parameters $w^{[1]}$ and $b^{[1]}$:

- $w^{[1]}$ has a shape (4, 3)
- $b^{[1]}$ has a shape (4, 1)
- $z^{[1]}$ has a sahep (4, 1)

General shape conventions:

- $w^{[1]}$: (nodes in current layer, features from previous layer)
- $b^{[1]}$: (nodes in current layer, 1)

Computing the output of a neural network

Given:

$$a_i^{[l]}$$

- l is the layer.
- i is the node in layer.

The first node ($i = 1$) of the first layer ($l = 1$) contains the following computations:

1. $z_1^{[1]} = w_1^{[1]\intercal} x + b_1^{[1]}$
2. $a_1^{[1]} = \sigma(z_1^{[1]})$

Hidden layer node computations:

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

Vectorizing across multiple examples

Given:

$$a^{[l](i)}$$

- l is the layer.
- i is the example index.

Matrix representations

Input X:

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}, \quad X \in \mathbb{R}^{n_x \times m}$$

Pre-activation linear output Z:

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \dots & z^{[1](m)} \\ | & | & & | \end{bmatrix}, \quad Z^{[1]} \in \mathbb{R}^{n \times m}$$

Activation output A:

$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \dots & a^{[1](m)} \\ | & | & & | \end{bmatrix}, \quad A^{[1]} \in \mathbb{R}^{n \times m}$$

where n is the number of units in a hidden layer.

Forward propagation

Forward propagation for an example of a 2 layer NN:

$$x \longrightarrow a^{[2]} = \hat{y}$$

Forward propagation for m examples of a 2 layer NN:

$$\begin{aligned} x^{(1)} &\longrightarrow a^{[2](1)} = \hat{y}^{(1)} \\ x^{(2)} &\longrightarrow a^{[2](2)} = \hat{y}^{(2)} \\ \vdots &\longrightarrow \ \vdots \ = \ \vdots \\ x^{(m)} &\longrightarrow a^{[2](m)} = \hat{y}^{(m)} \end{aligned}$$

Forward propagation implementations

For-loop implementation of forward propagation of a 2 layer NN:

$$\begin{aligned} &\text{For } i = 1 \text{ to } m\text{:} \\ &\qquad z^{[1](i)} = w^{[1]} x^{(i)} + b^{[1]} \\ &\qquad a^{[1](i)} = \sigma(z^{[1](i)}) \\ &\qquad z^{[2](i)} = w^{[2]} a^{[1](i)} + b^{[2]} \\ &\qquad a^{[2](i)} = \sigma(z^{[2](i)}) \end{aligned}$$

Vectorized forward propagation of a 2 layer NN:

$$
\begin{aligned}
Z^{[1]} &= w^{[1]}X + b^{[1]} \\
A^{[1]} &= \sigma(Z^{[1]}) \\
Z^{[2]} &= w^{[2]}A^{[1]} + b^{[2]} \\
A^{[2]} &= \sigma(Z^{[2]})
\end{aligned}
$$

Activation functions

Given another activation function $g(z)$

$$
\begin{aligned}
Z^{[1]} &= w^{[1]}X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]}) \\
Z^{[2]} &= w^{[2]}A^{[1]} + b^{[2]} \\
A^{[2]} &= g^{[2]}(Z^{[2]})
\end{aligned}
$$

The activation function also needs a superscript squares brackets because different layers may need different activation functions.

sigmoid

A sigmoid σ is an activation function:

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

Almost never used, except in the output layer for binary classification.
Generally not recommended because $\tanh$ is strictly better.

The derivative of the sigmoid function:

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z))$$

hyperbolic tangent

The $\tanh(z)$ activation function is a scaled and shifted version of the sigmoid function. The range becomes $(-1, 1)$ instead of $(0, 1)$.

$$a = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

The derivative of the $\tanh(z)$ function:

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

It is almost always strictly superior to the sigmoid function because of its better performance in hidden layers.

The one exception is for the output layer. Because $\hat{y}$ should be between 0 and 1 to match the binary label $y \in 0, 1$, not between -1 and 1.

ReLU

ReLU (rectified linear unit) outputs zero for negative values:

$$a = \mathrm{ReLU}(z) = \max(0, z)$$

The derivative of the ReLU function:

$$\frac{d}{dz}\mathrm{ReLU}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

It is increasingly the default choice for the activation function. If unsure, start with ReLU.

There is a variant called Leaky ReLU:

$$\mathrm{LeakyReLU}(z) = \max(0.01z, z)$$

The derivative of the Leaky ReLU:

$$\frac{d}{dz}\mathrm{LeakyReLU}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0.01 & \text{if } z \leq 0 \end{cases}$$

Leaky ReLU helps avoid dead neurons by allowing a small gradient for $z < 0$.

The slope (e.g., 0.01) can even be made a learnable parameter. But it's hardly done.

If Leaky ReLU performs well—stick with it.

Why non-linear activation functions?

Without a non-linear activation, the network only performs a linear transformation $a = z$.

The composition of two linear functions is itself a linear function. Therefore, no matter how many layers a network with only linear activations has—it is still equivalent to a single-layer model.

When to use linear activation functions?

A linear activation function is sometimes used in the output layer for regression tasks.

Even in regression, hidden layers should still use non-linear activations.

In some rare cases related to data compression, linear activations may appear in hidden layers.

Gradient descent for neural networks

Parameters:

- $w^{[1]}$ with a shape of $(n^{[1]}, n^{[0]})$
- $b^{[1]}$ with a shape of $(n^{[1]}, 1)$
- $w^{[2]}$ with a shape of $(n^{[2]}, n^{[1]})$
- $b^{[1]}$ with a shape of $(n^{[2]}, 1)$

Dimensions per layer:

- $n_x = n^{[0]}$
- $n^{[1]}$
- $n^{[2]} = 1$

Cost function:

$$J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y), \quad \text{where } \hat{y} = a^{[2]}$$

After initializing parameters, do gradient descent:

$$\begin{aligned}
&\textbf{Repeat until convergence:} \\
&\textbf{Forward pass:} \quad \hat{y}^{(i)} \text{ for } i = 1, \ldots, m \\
&\textbf{Backward pass:} \\
&dw^{[1]} = \frac{\partial J}{\partial w^{[1]}}, \quad db^{[1]} = \frac{\partial J}{\partial b^{[1]}} \\
&dw^{[2]} = \frac{\partial J}{\partial w^{[2]}}, \quad db^{[2]} = \frac{\partial J}{\partial b^{[2]}} \\
&\textbf{Update parameters:} \\
&w^{[1]} := w^{[1]} - \alpha \cdot dw^{[1]}, \quad b^{[1]} := b^{[1]} - \alpha \cdot db^{[1]} \\
&w^{[2]} := w^{[2]} - \alpha \cdot dw^{[2]}, \quad b^{[2]} := b^{[2]} - \alpha \cdot db^{[2]}
\end{aligned}$$

Forward propagation:

$$\begin{aligned}
Z^{[1]} &= w^{[1]} X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]}) \\
Z^{[2]} &= w^{[2]} A^{[1]} + b^{[2]} \\
A^{[2]} &= g^{[2]}(Z^{[2]})
\end{aligned}$$

Assuming it's binary classification, use sigmoid for the output layer:

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

Back propagation:

$$\begin{aligned}
dz^{[2]} &= A^{[2]} - Y \\
dw^{2]} &= \tfrac{1}{m} \cdot dz^{[2]} A^{[1]\intercal} \\
db^{[2]} &= \tfrac{1}{m} \text{np.sum}(dz^{[2]}, \text{axis=1, keepdims=True}) \\
dz^{[1]} &= w^{[2]\intercal} \cdot dz^{[2]} * g^{[1]'}(z^{[1]}) \\
dw^{[1]} &= \tfrac{1}{m} \cdot dz^{[1]} \cdot X^\intercal \\
db^{[1]} &= \tfrac{1}{m} \text{np.sum}(dz^{[1]}, \text{axis=1, keepdims=True})
\end{aligned}$$

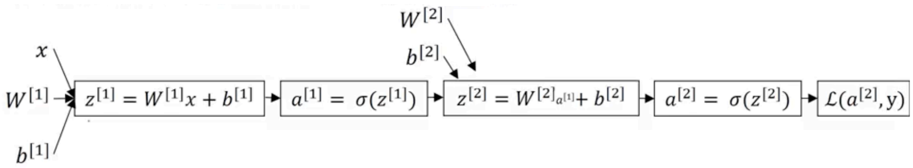
where

- $Y = [y^{(1)} y^{(2)}, \ldots, y^{(m)}]$
- $*$ means element-wise product
- $db^{[2]}$ will have a shape of $(n^{[2]}, 1)$ instead of $(n^{[2]},)$
- $w^{[2]\intercal} \cdot dz^{[2]}$ and $g^{[1]'}(z^{[1]})$ have a shape of $(n^{[1]}, m)$
- $db^{[1]}$ will have a shape of $(n^{[1]}, 1)$ instead of $(n^{[1]},)$

When the keepdims is True, it prevents outputting a rank one array:

```
np.sum(..., ..., keepdims=True)
```

Backpropagation intuition



Backpropagation relies on chain rule and consistent matrix dimensions:

$$
\begin{aligned}
\frac{d\mathcal{L}}{dz^{[2]}} &= \frac{d\mathcal{L}}{da^{[2]}} \cdot \frac{da^{[2]}}{dz^{[2]}} = a^{[2]} - y \\
\frac{\partial\mathcal{L}}{\partial W^{[2]}} &= \frac{\partial\mathcal{L}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial W^{[2]}} = dz^{[2]} \cdot a^{[1]\intercal} \\
\frac{\partial\mathcal{L}}{\partial b^{[2]}} &= dz^{[2]} \\
\frac{d\mathcal{L}}{dz^{[1]}} &= \frac{\partial\mathcal{L}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} = (W^{[2]})^\intercal dz^{[2]} * g^{[1]'}(z^{[1]}) \\
\frac{\partial\mathcal{L}}{\partial W^{[1]}} &= dz^{[1]} \cdot X^\intercal \\
\frac{\partial\mathcal{L}}{\partial b^{[1]}} &= dz^{[1]}
\end{aligned}
$$

Order of backpropagation for a 2 layer NN:

1. $da^{[2]}$
2. $dz^{[2]}$
3. $dw^{[2]}$
4. $db^{[2]}$
5. $da^{[1]}$
6. $dz^{[1]}$
7. $dw^{[1]}$
8. $db^{[1]}$

Computing $dz^{[2]}, dw^{[2]}, db^{[2]}$:

$$
\begin{aligned}
dz^{[2]} &= a^{[2]} - y \\
dw^{[2]} &= dz^{[2]} a^{[1]\intercal} \\
db^{[2]} &= dz^{[2]}
\end{aligned}
$$

Computing $dz^{[1]}, dw^{[1]}, db^{[1]}$:

$$
\begin{aligned}
dz^{[1]} &= w^{[2]\intercal} \cdot dz^{[2]} * g^{[1]'}(z^{[1]}) \\
dw^{[1]} &= \tfrac{1}{m} \cdot dz^{[1]} \cdot X^\intercal \\
db^{[1]} &= dz^{[1]}
\end{aligned}
$$

Shapes of variables:

- $w^{[2]} \rightarrow (n^{[2]}, n^{[1]})$
- $z^{[2]}, dz^{[2]} \rightarrow (n^{[2]}, 1)$
- $z^{[1]}, dz^{[1]} \rightarrow (n^{[1]}, 1)$

Vectorized implementation of gradient descent:

$$
\begin{aligned}
& dZ^{[2]} = A^{[2]} - Y \\
& dW^{[2]} = \frac{1}{m} \cdot dZ^{[2]} \cdot A^{[1]\intercal} \\
& db^{[2]} = \frac{1}{m} \text{np.sum}(dZ^{[2]}, \text{axis=1}, \text{keepdims=True}) \\
& dZ^{[1]} = W^{[2]\intercal} \cdot dZ^{[2]} * g^{[1]}(Z^{[1]})
\end{aligned}
$$

Random initialization

Initializing weights to zero ($w = 0$):

- hidden units receive the same input.
- hidden units compute the same output.
- gradients (like $dz^{[1]}$, $dw^{[1]}$) becomes identical due to symmetry.
- the parameters gets updated to the same values.

Due to this, neurons don't learn different features no matter the number of training steps.

Initializing biases to zero ($b = 0$) is fine.

Random initialization solves it by breaking symmetry:

```
W1 = np.random.randn(n1, n0) * 0.01
W2 = np.random.randn(n2, n1) * 0.01
b1 = np.zeros((n1, 1))
b2 = np.zeros((n2, 1))
```

We multiply the weights by a small constant i.e., 0.01 to prevent saturation caused by large z values in our activation function (sigmoid, tanh).

For deeper networks, He or Xavier initialization are better.