Universidad Autónoma de Madrid

Escuela Politécnica Superior

Aprendizaje profundo

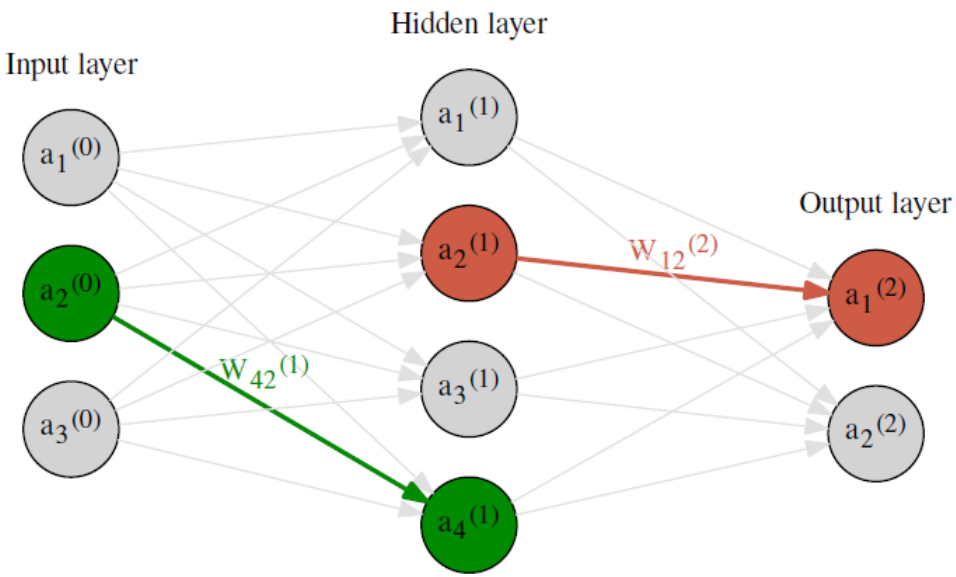
Cultura general sobre redes neuronales y aprendizaje profundo

- Terrence J. Sejnowski. The unreasonable effectiveness of deep learning in artificial intelligence. *PNAS, 20190737361*, 2020.

- Jürgen Schmidhuber comments on Sejnowski's paper: https://mailman.srv.cs.cmu.edu/pipermail/connectionists/2020-March/034010.html

- Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks* 61: 85-117, 2015.

Retropropagación en notación compacta



- $a_i^{(k)}$ is the activation function of unit i in the k-th layer
- $w_{ij}^{(k)}$ is the weight of the connection from unit j in layer k-1 to unit i in the k-th layer

Retropropagación en notación compacta

Feedforward propagation

- The input to a unit is a linear function of the activations of the neurons in the previous layer:

$$z_i^{(k)} = \sum_{j=1}^{n_{k-1}} w_{ij}^{(k)} a_j^{(k-1)} + b_i^{(k)}$$

n_{k-1} is the number of units in layer k-1, $b_i^{(k)}$ is the bias

- The activation function f is applied on the input:

$$a_i^{(k)} = f\left(z_i^{(k)}\right) = f\left(\sum_{j=1}^{n_{k-1}} w_{ij}^{(k)} a_j^{(k-1)} + b_i^{(k)}\right)$$

- The activation on the input layer (k=0) is simply the network input $a_i^{(0)} = x_i$

Retropropagación en notación compacta

Matrix notation for the feedforward propagation

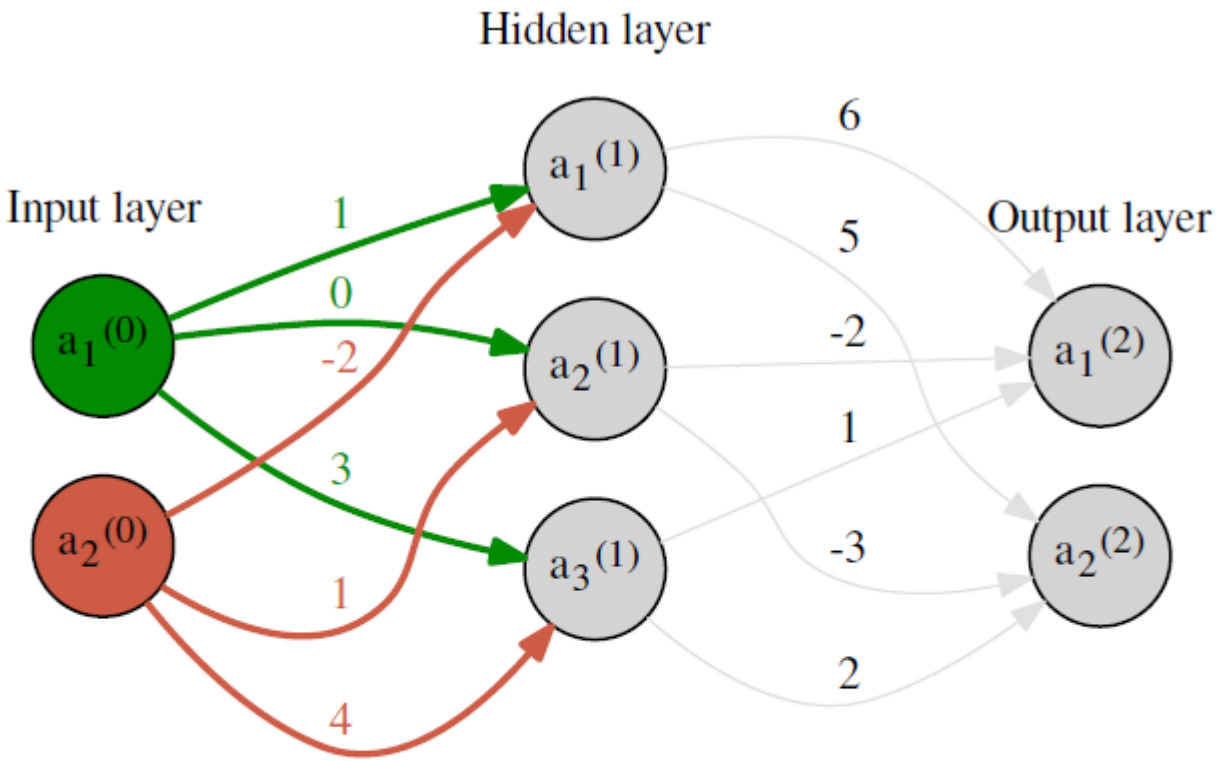
$$\mathbf{a}^{(0)} = \mathbf{x}$$

$$\mathbf{z}^{(k)} = \mathbf{W}^{(k)}\mathbf{a}^{(k-1)} + \mathbf{b}^{(k)}, \;\; k > 0$$

$$\mathbf{a}^{(k)} = f\left(\mathbf{z}^{(k)}\right) = f\left(\mathbf{W}^{(k)}\mathbf{a}^{(k-1)} + \mathbf{b}^{(k)}\right), \;\; k > 0$$

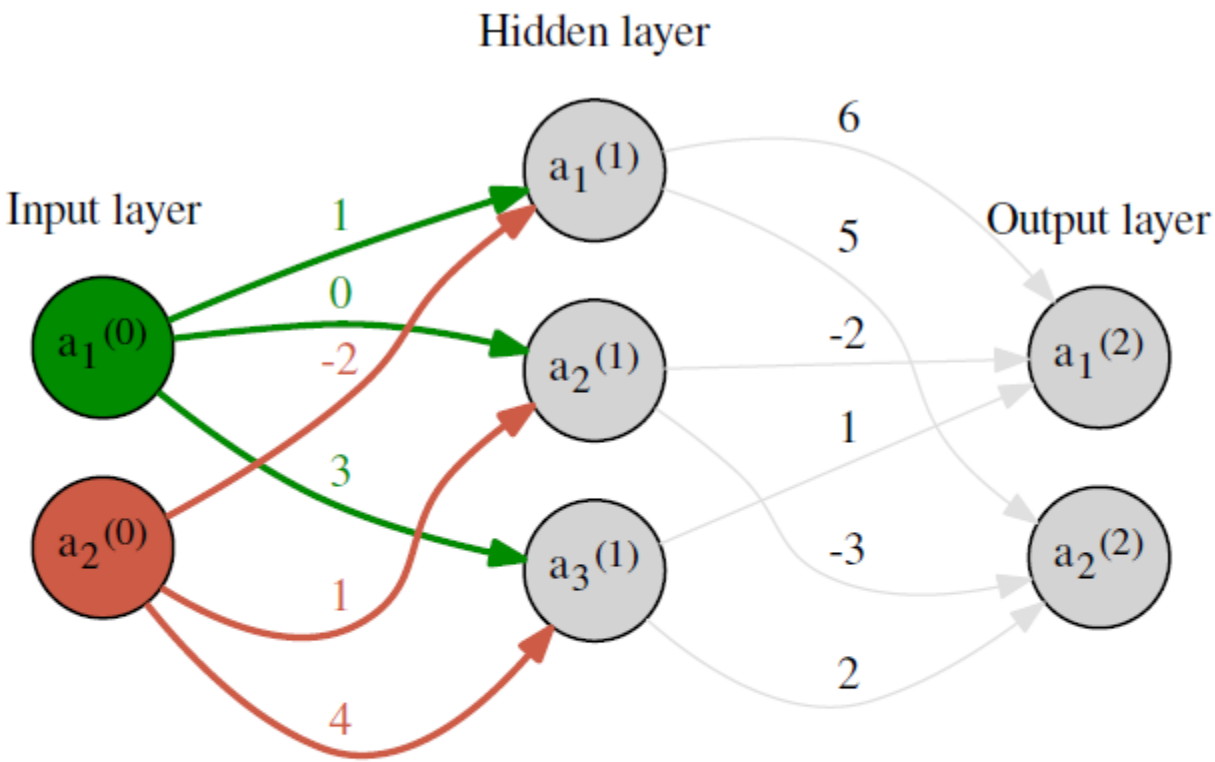
- $\mathbf{a}^{(k)}(n_k \times 1)$ is the **vector of activations** for layer k.
- $\mathbf{W}^{(k)}(n_k \times n_{k-1})$ is the weight matrix for layer k
  - Row i of $\mathbf{W}^{(k)}$ contains the weights that connect each unit in layer k-1 to unit i in layer k.
- $\mathbf{b}^{(k)}(n_k \times 1)$ is the **vector of bias** for layer k.
- $f(\mathbf{z})$ is the **activation function**, which can be different for each layer

Ejemplo



$$
\begin{aligned}
a_1^{(1)} &= f(1 \times a_1^{(0)} - 2 \times a_2^{(0)} + b_1^{(1)}) \\
a_2^{(1)} &= f(0 \times a_1^{(0)} + 1 \times a_2^{(0)} + b_2^{(1)}) \\
a_3^{(1)} &= f(3 \times a_1^{(0)} + 4 \times a_2^{(0)} + b_3^{(1)})
\end{aligned}
$$

Ejemplo



$$\begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} = f\left(\begin{bmatrix} 1 & -2 \\ 0 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} a_1^{(0)} \\ a_2^{(0)} \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix} \right)$$

Ejemplo

- Activations in the hidden layer:

$$\begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} = f\left(\begin{bmatrix} 1 & -2 \\ 0 & 1 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} a_1^{(0)} \\ a_2^{(0)} \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix} \right)$$

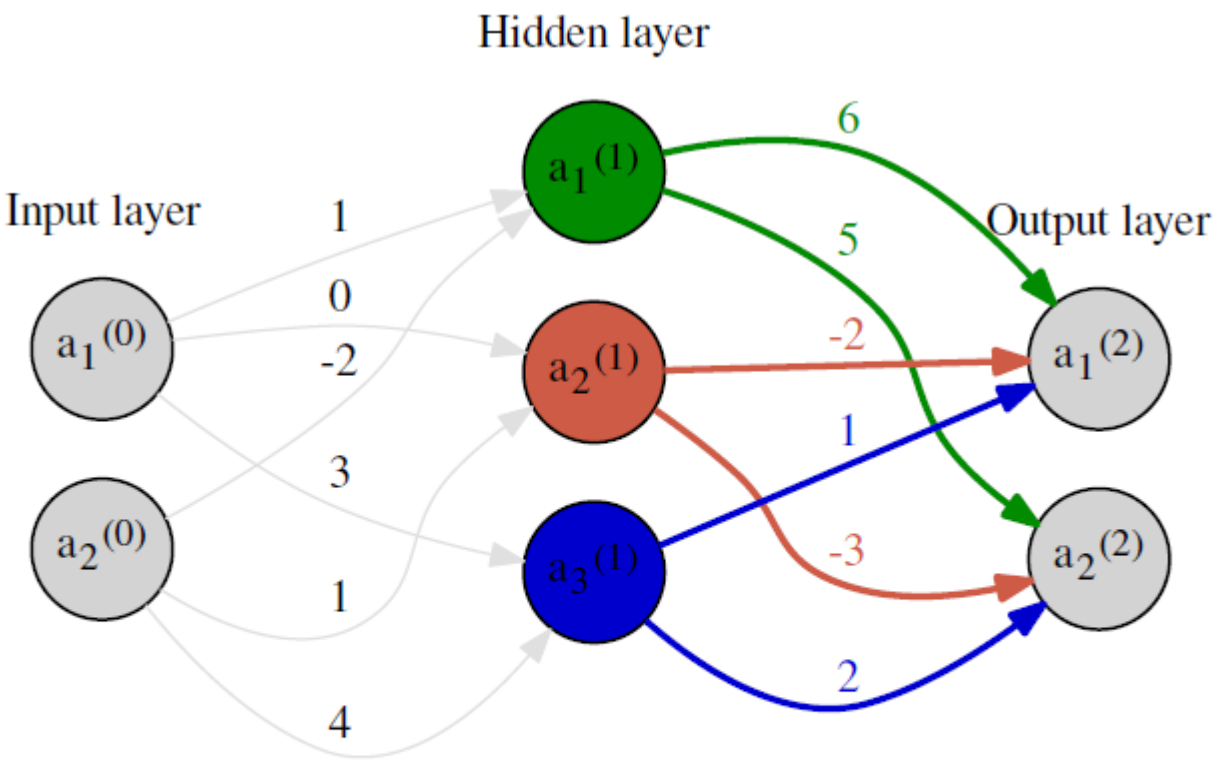
which is the same as

$$\mathbf{a}^{(1)} = f(\mathbf{W}^{(1)}\mathbf{a}^{(0)} + \mathbf{b}^{(1)})$$

with

$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & -2 \\ 0 & 1 \\ 3 & 4 \end{bmatrix}$$

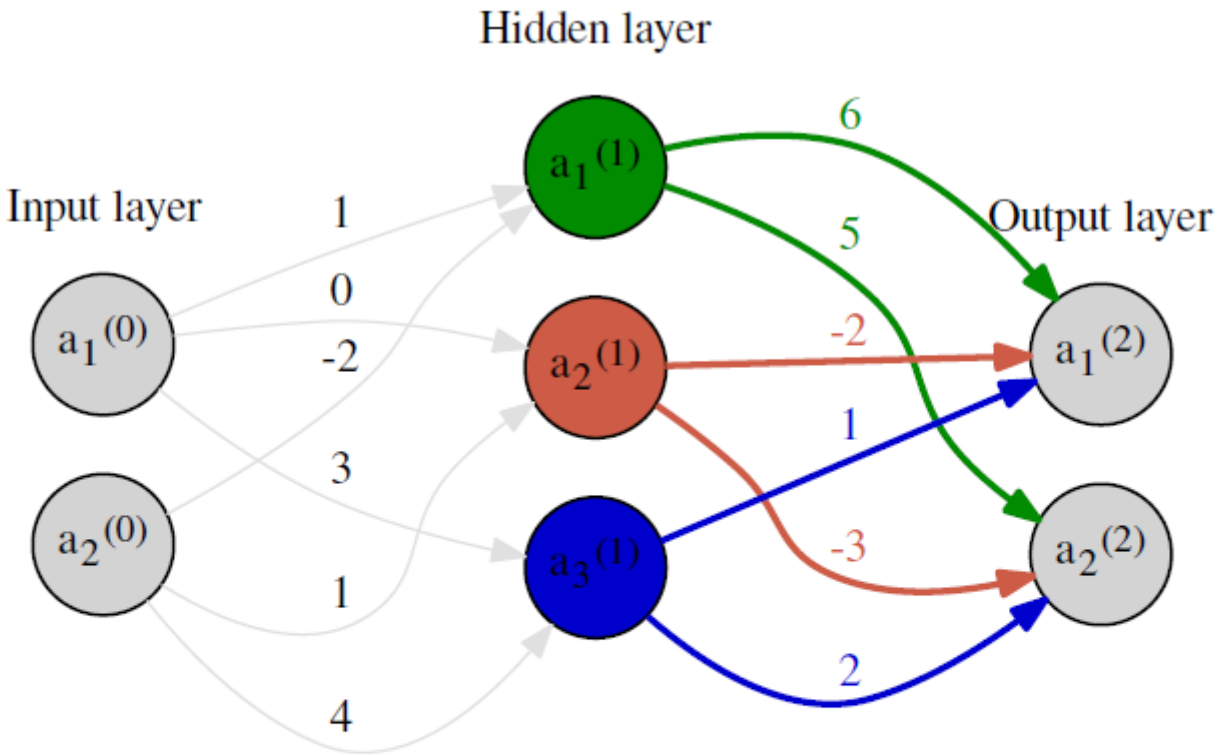
Ejemplo



$$a_1^{(2)} = f(6 \times a_1^{(1)} - 2 \times a_2^{(1)} + 1 \times a_3^{(1)} + b_1^{(2)})$$

$$a_2^{(2)} = f(5 \times a_1^{(1)} - 3 \times a_2^{(1)} + 2 \times a_3^{(1)} + b_2^{(2)})$$

Ejemplo



$$\begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} = f\left(\begin{bmatrix} 6 & -2 & 1 \\ 5 & -3 & 2 \end{bmatrix} \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_2^{(1)} \end{bmatrix} + \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix}\right)$$

Ejemplo

- Activations in the output layer:

$$\begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} = f\left(\begin{bmatrix} 6 & -2 & 1 \\ 5 & -3 & 2 \end{bmatrix} \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_2^{(1)} \end{bmatrix} + \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix}\right)$$

which is the same as

$$\mathbf{a}^{(2)} = f(\mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)})$$

with

$$\mathbf{W}^{(2)} = \begin{bmatrix} 6 & -2 & 1 \\ 5 & -3 & 2 \end{bmatrix}$$

Código ejemplo

Computing the activation of the network

```
def forward(self, x):
z1 = np.dot(self.W1, x) + self.b1
a1 = sigmoid(z1)
z2 = np.dot(self.W2, a1) + self.b2
y = z2
return z1, a1, z2, y
```

Retropropagación del error

- The error, **delta function, in the output layer** (K) is:

$$\delta_i^{(K)} = \frac{\partial C}{\partial z_i^{(K)}} = \frac{\partial C}{\partial a_i^{(K)}} f'\left(z_i^{(K)}\right)$$

where $C = C\left(\mathbf{a}^{(K)}, \mathbf{y}\right)$ is the **cost function**, which measures the discrepancy between the output of the network $\mathbf{a}^{(K)}$ and the expected output $\mathbf{y}$

- The error, **delta function, in layer** k ($k<K$) is:

$$\delta_i^{(k)} = \frac{\partial C}{\partial z_i^{(k)}} = \left(\sum_{j=1}^{n_{k+1}} w_{ji}^{(k+1)} \delta_j^{(k+1)}\right) f'\left(z_i^{(k)}\right)$$

Retropopagación del error

- The derivative of the cost function with respect to the **bias**:

$$\frac{\partial C}{\partial b_i^{(k)}} = \delta_i^{(k)}$$

- The derivative of the cost function with respect to the **weights**:

$$\frac{\partial C}{\partial w_{ij}^{(k)}} = a_j^{(k-1)} \delta_i^{(k)}$$

Retropropagación del error

Gradient descent:

$$\boldsymbol{\delta}^{(K)} = \nabla_{\mathbf{a}^{(K)}} C \odot f'(\mathbf{z}^{(K)})$$

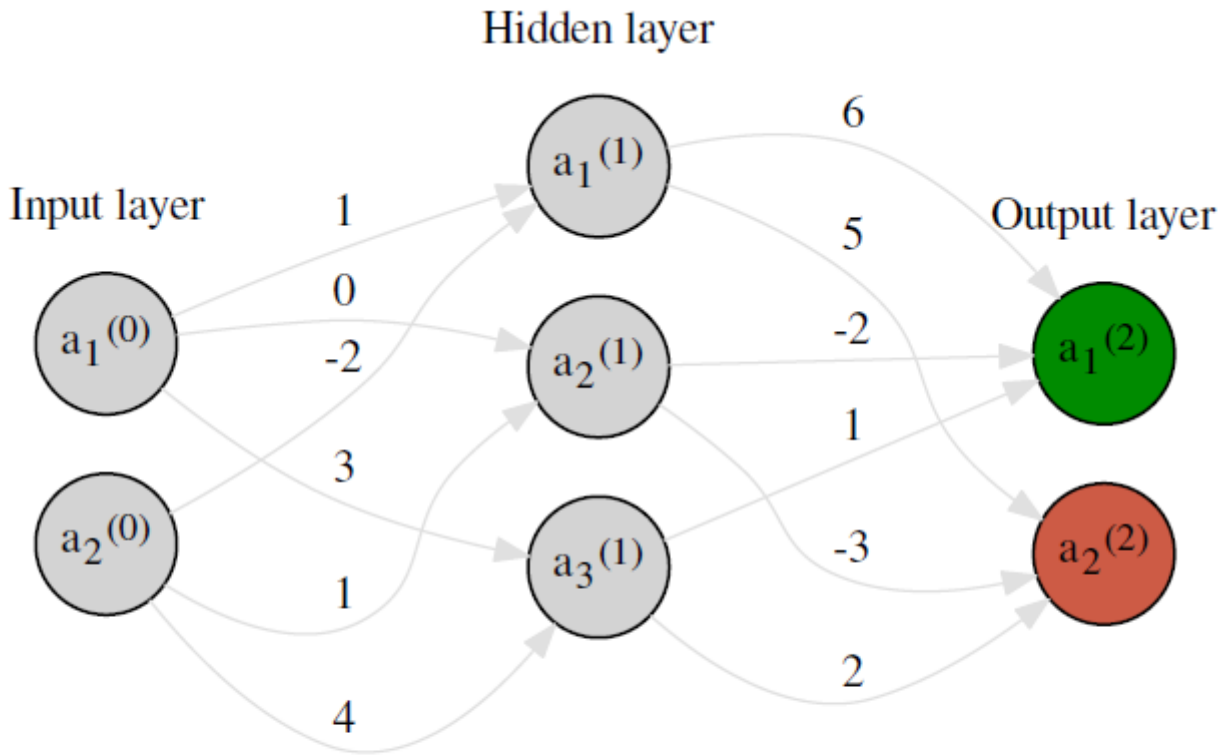
$$\boldsymbol{\delta}^{(k)} = ((\mathbf{W}^{(k+1)})^T \boldsymbol{\delta}^{(k+1)}) \odot f'(\mathbf{z}^{(k)}), \quad k < K$$

$$\mathbf{b}^{(k)} \leftarrow \mathbf{b}^{(k)} - \eta \boldsymbol{\delta}^{(k)}$$

$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} - \eta \boldsymbol{\delta}^{(k)} (\mathbf{a}^{(k-1)})^T$$

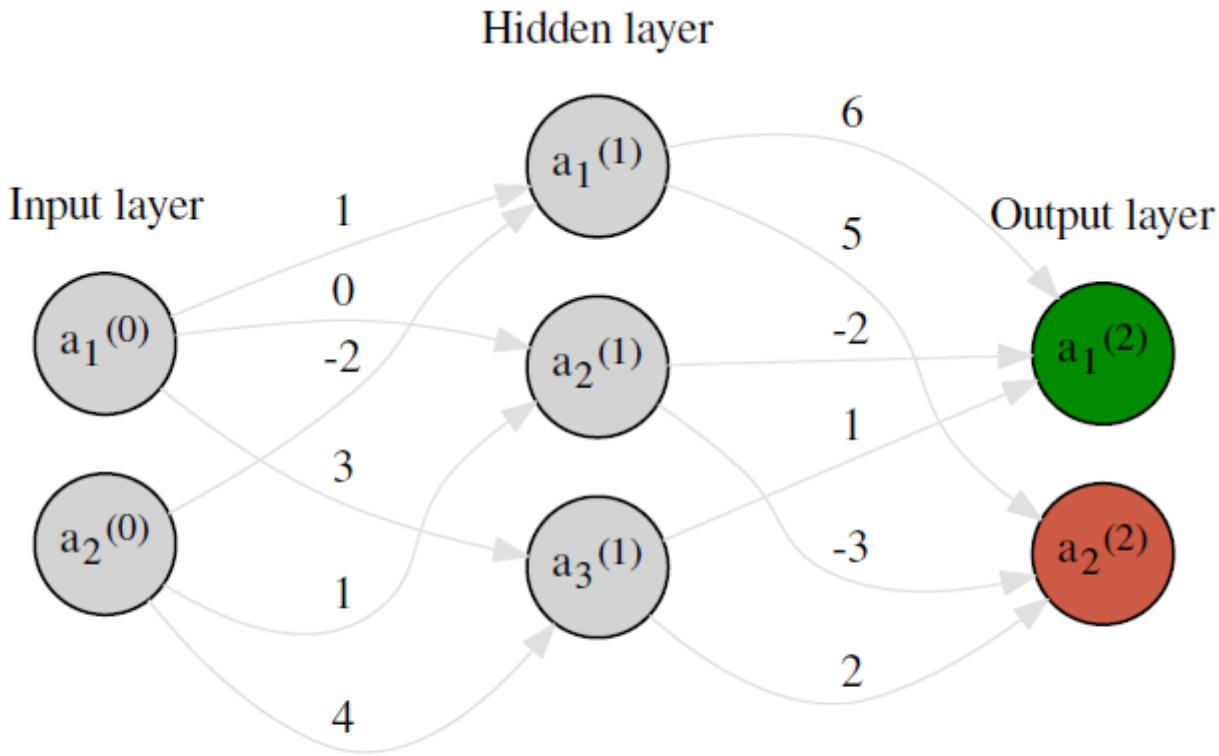
- Constant η is the **learning rate**.

Ejemplo



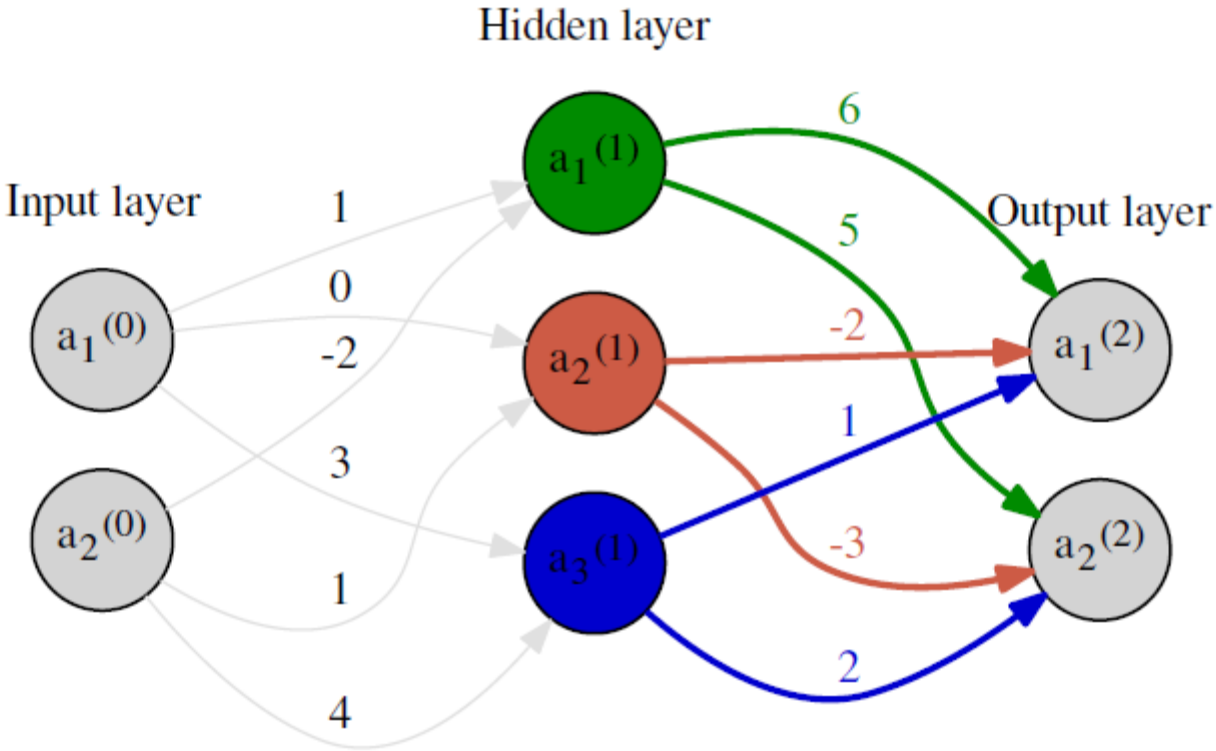
$$\boldsymbol{\delta}^{(2)} = \begin{bmatrix} \delta_1^{(2)} \\ \delta_2^{(2)} \end{bmatrix} = \begin{bmatrix} (\partial C/\partial a_1^{(2)}) f'(z_1^{(2)}) \\ (\partial C/\partial a_2^{(2)}) f'(z_2^{(2)}) \end{bmatrix}$$

Ejemplo



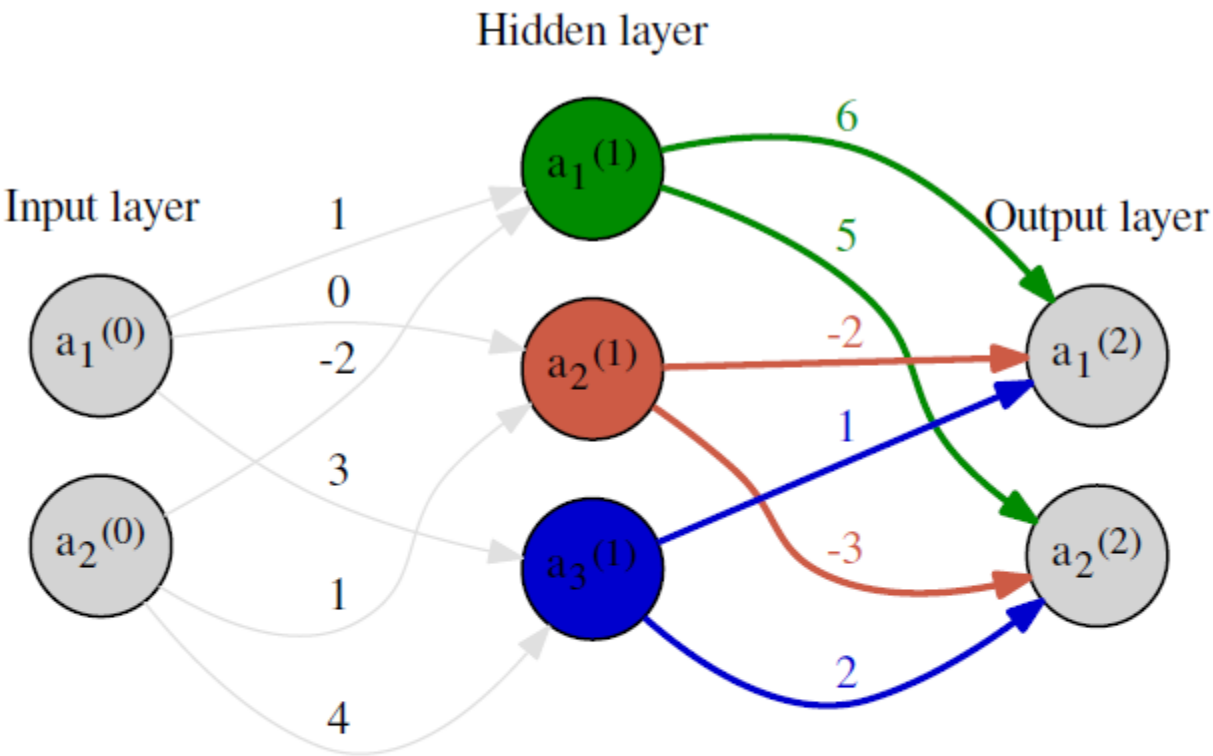
$$\mathbf{b}^{(2)} \leftarrow \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix} - \eta \begin{bmatrix} \delta_1^{(2)} \\ \delta_2^{(2)} \end{bmatrix}$$

Ejemplo



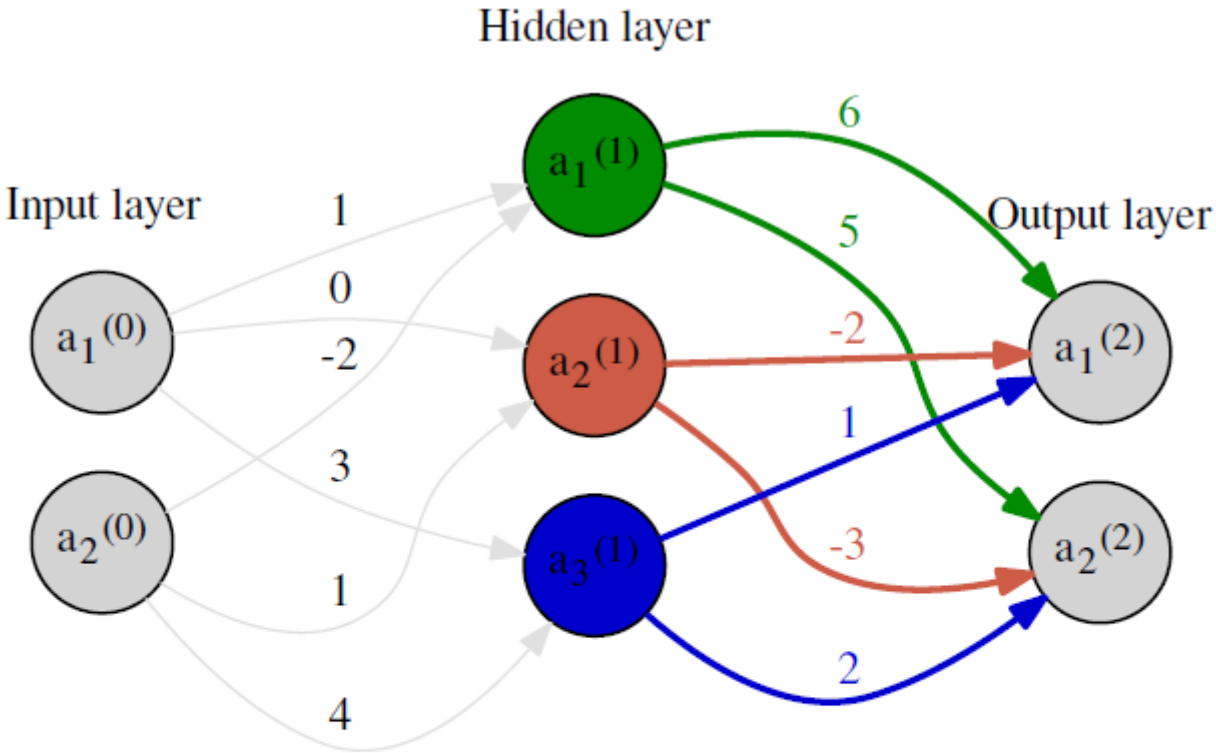
$$\mathbf{W}^{(2)} \leftarrow \begin{bmatrix} 6 & -2 & 1 \\ 5 & -3 & 2 \end{bmatrix} - \eta \begin{bmatrix} \delta_1^{(2)} \\ \delta_2^{(2)} \end{bmatrix} \begin{bmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{bmatrix}$$

Ejemplo



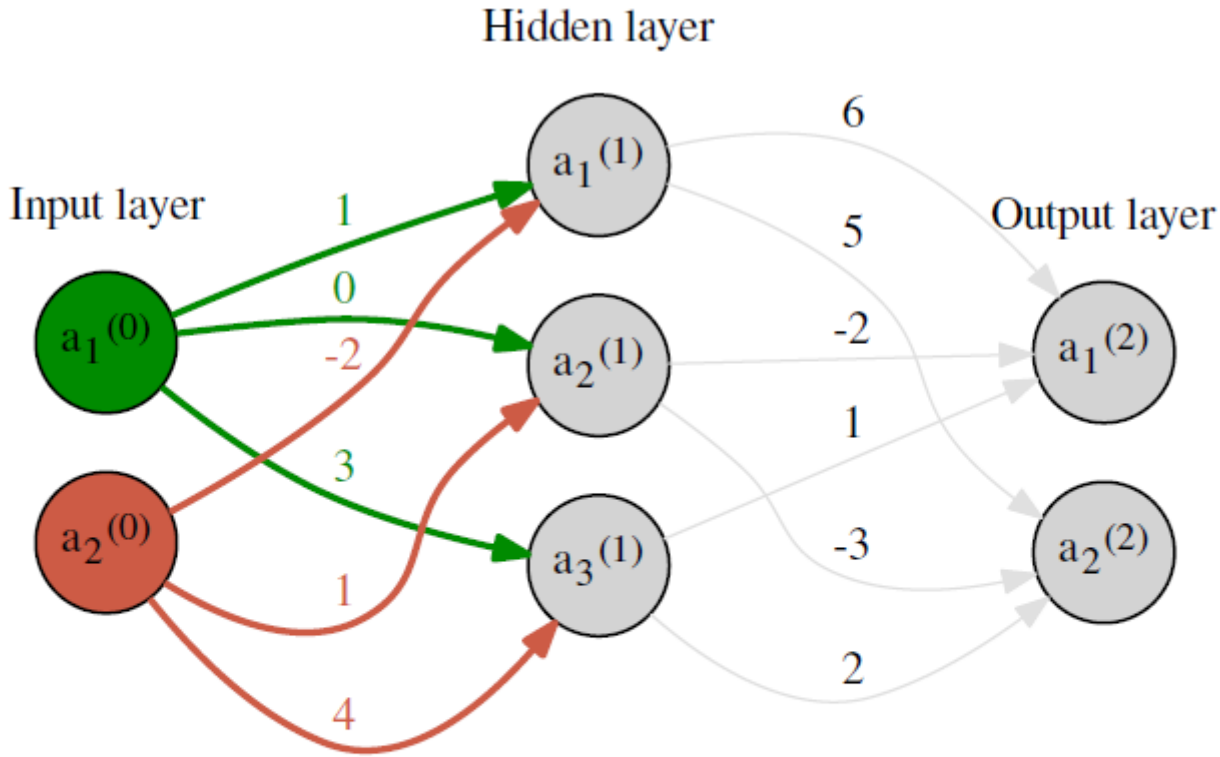
$$\boldsymbol{\delta}^{(1)} = \begin{bmatrix} \delta_1^{(1)} \\ \delta_2^{(1)} \\ \delta_3^{(1)} \end{bmatrix} = \left(\begin{bmatrix} 6 & 5 \\ -2 & -3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} \delta_1^{(2)} \\ \delta_2^{(2)} \end{bmatrix} \right) \odot \begin{bmatrix} f'(z_1^{(1)}) \\ f'(z_2^{(1)}) \\ f'(z_3^{(1)}) \end{bmatrix}$$

Ejemplo



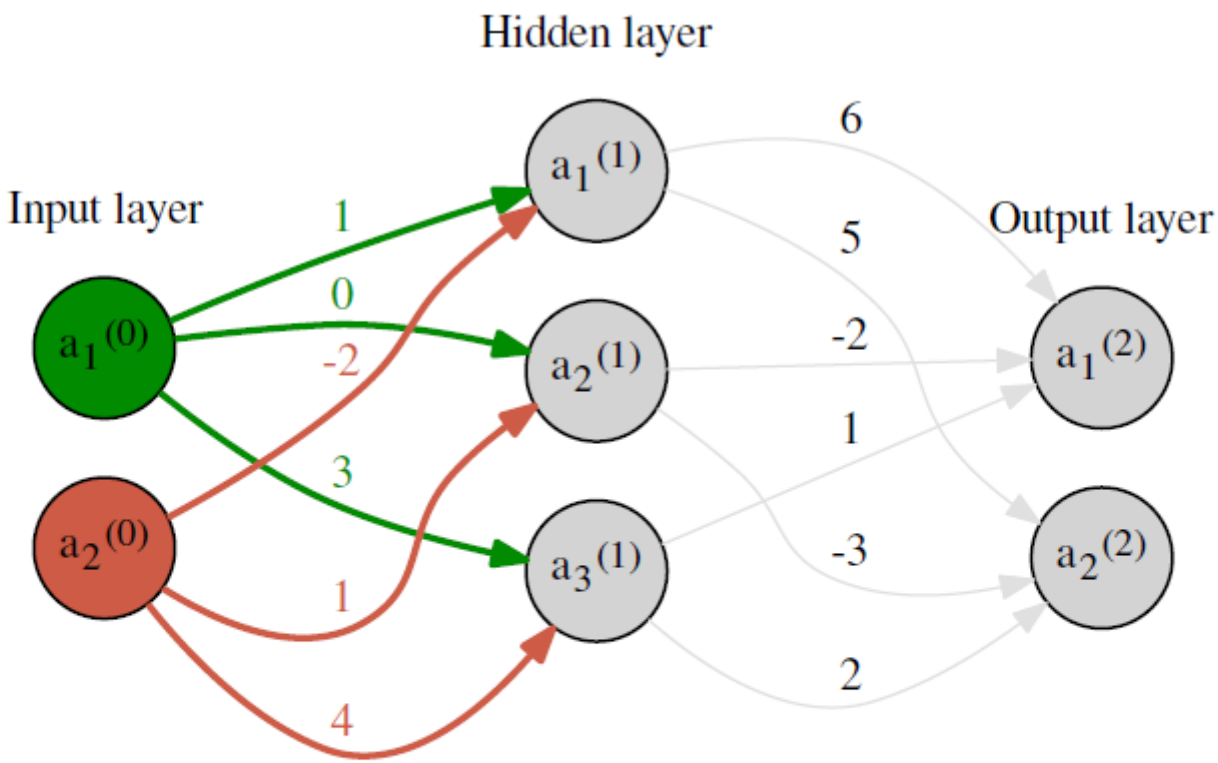
$$\mathbf{b}^{(1)} \leftarrow \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix} - \eta \begin{bmatrix} \delta_1^{(1)} \\ \delta_2^{(1)} \\ \delta_3^{(1)} \end{bmatrix}$$

Ejemplo



$$\mathbf{W}^{(1)} \leftarrow \begin{bmatrix} 1 & -2 \\ 0 & 1 \\ 3 & 4 \end{bmatrix} - \eta \begin{bmatrix} \delta_1^{(1)} \\ \delta_2^{(1)} \\ \delta_3^{(1)} \end{bmatrix} \begin{bmatrix} a_1^{(0)} & a_2^{(0)} \end{bmatrix}$$

Ejemplo



$$\delta^{(0)} = \begin{bmatrix} \delta_1^{(0)} \\ \delta_2^{(0)} \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 & 3 \\ -2 & 1 & 4 \end{bmatrix} \begin{bmatrix} \delta_1^{(1)} \\ \delta_2^{(1)} \\ \delta_3^{(1)} \end{bmatrix} \right) \odot \begin{bmatrix} f'(z_1^{(0)}) \\ f'(z_2^{(0)}) \end{bmatrix}$$

Código de ejemplo

Computing the gradients

```
def backward(self, x, t):
z1, a1, z2, a2 = self.forward(x)
da2 = a2 - t
dz2 = da2
db2 = dz2
dW2 = np.dot(dz2, a1.T)
da1 = np.dot(self.W2.T, dz2)
dz1 = dsigmoid(z1)*da1
db1 = dz1
dW1 = np.dot(dz1, x.T)
return dW1, db1, dW2, db2
```

Código de Ejemplo

Weight update

```
def gradient_step(self, x, t, eta):
dW1, db1, dW2, db2 = self.backward(x, t)

self.W1 -= eta*dW1
self.b1 -= eta*db1
self.W2 -= eta*dW2
self.b2 -= eta*db2
```

Entrenamiento

Cost function gradient:

- In this notation $\mathbf{x} = \mathbf{a}^{(0)}$ is the network input, $\hat{\mathbf{y}} = \hat{\mathbf{y}}(\mathbf{x}) = \mathbf{a}^{(K)}$ is the network output, $\mathbf{y}$ is the expected output.

- The cost function $C\left(\hat{\mathbf{y}}, \mathbf{y}\right)$ measures the discrepancy between the output of the network and the expected output.

- Ideally we would like to minimize the expected value of the cost function over all possible observations:

$$J = \mathbf{E}_{(\mathbf{x},\mathbf{t})}[C(\mathbf{y}(\mathbf{x}), \mathbf{t})]$$

$$\nabla J = \mathbf{E}_{(\mathbf{x},\mathbf{t})}[\nabla C(\mathbf{y}(\mathbf{x}), \mathbf{t})]$$

- In practice, we estimate the gradient ∇J from a finite set of data (training set):

$$\nabla J \approx \frac{1}{n} \sum_{i=1}^{n} \nabla C(\mathbf{y}(x_i), \mathbf{t}_i)$$

Descenso por gradient (batch)

Cost function gradient:

- Thus, in each step of the gradient descent algorithm we can update the weights and the bias according to:

$$\mathbf{b}^{(k)} \leftarrow \mathbf{b}^{(k)} - \frac{\eta}{n} \sum_{i=1}^{n} \boldsymbol{\delta}_{\mathbf{x}_i}^{(k)}$$

$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} - \frac{\eta}{n} \sum_{i=1}^{n} \boldsymbol{\delta}_{\mathbf{x}_i}^{(k)} (\mathbf{a}_{\mathbf{x}_i}^{(k-1)})^T$$

- $\mathbf{a}_{\mathbf{x}_i}^{(k)}$ is the network activation in layer k for the $\mathbf{x}_i$ example.
- $\boldsymbol{\delta}_{\mathbf{x}_i}^{(k)}$ is the error in layer k for the $\mathbf{x}_i$ example.
- n is the number of examples of the training set.

Descenso por gradient estocástico

- If the number of examples n is very large, computing the gradient can take a lot of time.

- In these cases, the gradient is estimated using a random subset of the examples (mini-batch) of size $m<n$

Stochastic gradient descent (each step):

1. Choose a random mini-batch of size m: $\mathcal{B} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m\}$
2. Update weights and bias according to:

$$\mathbf{b}^{(k)} \leftarrow \mathbf{b}^{(k)} - \frac{\eta}{m} \sum_{\mathbf{x} \in \mathcal{B}} \boldsymbol{\delta}_{\mathbf{x}}^{(k)}$$

$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} - \frac{\eta}{m} \sum_{\mathbf{x} \in \mathcal{B}} \boldsymbol{\delta}_{\mathbf{x}}^{(k)} (\mathbf{a}_{\mathbf{x}}^{(k-1)})^T$$

Entrenamiento online

- The limit case in which $m=1$ is called **online** training.
- Online training makes sense when the training examples keep coming.
- However it introduces additional noise since the gradient is estimated from one example at a time.

Funciones de coste (cost functions)

Squared error:

- The squared error is calculated as:

$$C(\mathbf{y}(\mathbf{x}), \mathbf{t}) = \frac{1}{2}||\mathbf{y}(\mathbf{x}) - \mathbf{t}||^2$$

- The gradient with respect to the activation in the output layer is:

$$\nabla_{\mathbf{y}} C(\mathbf{y}(\mathbf{x}), \mathbf{t}) = \mathbf{y}(\mathbf{x}) - \mathbf{t}$$

- Thus, the error in the last layer can be expressed as:

$$\boldsymbol{\delta}^{(K)} = (f(\mathbf{z}^{(K)}) - \mathbf{t}) \odot f'(\mathbf{z}^{(K)})$$

Cost functions

Cross entropy:

- The cross-entropy cost function is calculated as:

$$C(\mathbf{y}(\mathbf{x}), \mathbf{t}) = -\sum_{j=1}^{n_K} [t_j \log y_j + (1 - t_j) \log (1 - y_j)]$$

The sum is over the output units (components of $\mathbf{y}$)

- The gradient with respect to the activation in the output layer is:

$$(\nabla_{\mathbf{y}} C(\mathbf{y}(\mathbf{x}), \mathbf{t}))_j = \frac{y_j - t_j}{y_j(1 - y_j)}$$

- The error in the last layer:

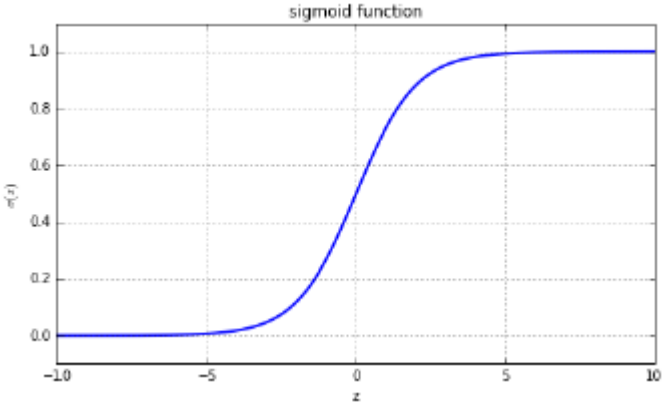
$$\delta_j^{(K)} = \frac{f(z_j) - t_j}{f(z_j)(1 - f(z_j))} f'(z_j)$$

superindex K in z_j^K is not specified to simplify the notation

Recordando la función de activación sigmoidal

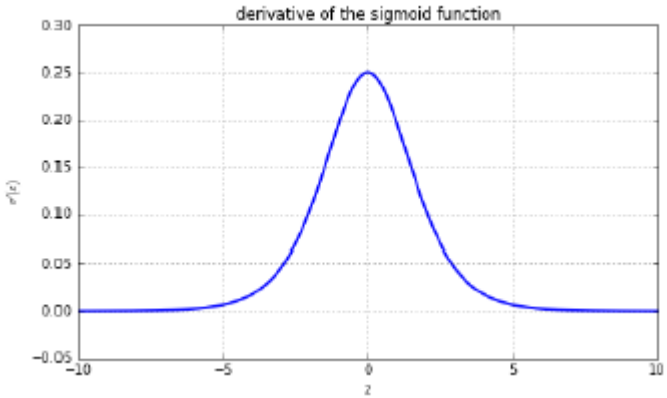
- If the activation function in the output layer is a sigmoid:

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



its derivative is:

$$f'(z_j) = \sigma'(z_j) = \sigma(z_j)(1 - \sigma(z_j))$$



such derivative saturates

Deltas con la función de activación sigmoidal

Squared error:

$$\boldsymbol{\delta}^{(K)} = (\sigma(\mathbf{z}^{(K)}) - \mathbf{t}) \odot \sigma'(\mathbf{z}^{(K)})$$

Cross entropy:

$$\boldsymbol{\delta}^{(K)} = (\sigma(\mathbf{z}^{(K)}) - \mathbf{t})$$

best choice when output units use sigmoid functions

Cost functions

Maximum likelihood (I):

- The output of the network is interpreted as a probability with the following cost function

$$C(\mathbf{y}|(\mathbf{x}), \mathbf{t}) = -\log p(\mathbf{t}|\mathbf{y}(\mathbf{x}))$$

- Adding over all examples of the training set:

$$J = -\frac{1}{n}\sum_{i=1}^{n} \log p(\mathbf{t}_i|\mathbf{y}(\mathbf{x}_i))$$

- Minimizing J is equivalent to minimizing the Kullback-Leibler divergence between the empirical data distribution and that obtained with the model.

Cost functions

Maximum likelihood (II):

- When the network output is a single sigmoid unit, maximum likelihood is equivalent to cross-entropy.
- Let us consider two classes $t \in \{0,1\}$, and we interpret the network output as the probability of class 1:

$$\begin{aligned} p(t=1|y(\mathbf{x})) &= y(\mathbf{x}) \\ p(t=0|y(\mathbf{x})) &= 1 - y(\mathbf{x}) \end{aligned}$$

then

$$C(y(\mathbf{x}), t) = -\log p(t|y(\mathbf{x})) = \begin{cases} -\log y, & \text{if } t = 1 \\ -\log(1-y), & \text{if } t = 0 \end{cases}$$

which can be written as:

$$C(y(\mathbf{x}), t) = -[t \log y + (1-t) \log(1-y)]$$

Cost functions

Hinge loss (widely used in SVM):

- The output is linear (no activation function):

$$\mathbf{y}(\mathbf{x}) = \mathbf{z}^{(K)}$$

- Hinge loss cost function:

$$C(\mathbf{y}(\mathbf{x}), \mathbf{t}) = C(\mathbf{z}^{(K)}, \mathbf{t}) = \sum_{j \neq o} \text{máx}\,(0, z_j^{(K)} - z_o^{(K)} + \Delta)$$

- The class **t** vector is assumed to be in the form:

$$t_j = \delta_{jo}$$

Funciones de coste en Keras

https://www.tensorflow.org/api_docs/python/tf/keras/losses

Mean squared error:

```
tf.keras.losses.MeanSquaredError(
reduction=losses_utils.ReductionV2.AUTO,
name='mean_squared_error'
)
```

Funciones de coste en Keras

https://www.tensorflow.org/api_docs/python/tf/keras/losses

Categorical cross-entropy:

```
tf.keras.losses.CategoricalCrossentropy(
from_logits=False, label_smoothing=0,
reduction=losses_utils.ReductionV2.AUTO,
name='categorical_crossentropy'
)
```

```
tf.keras.losses.SparseCategoricalCrossentropy(
from_logits=False,
reduction=losses_utils.ReductionV2.AUTO,
name='sparse_categorical_crossentropy'
)
```

Funciones de coste en Keras

https://www.tensorflow.org/api_docs/python/tf/keras/losses

Hinge loss:

```
tf.keras.losses.Hinge(
reduction=losses_utils.ReductionV2.AUTO,
name='hinge'
)
```

```
tf.keras.losses.CategoricalHinge(
reduction=losses_utils.ReductionV2.AUTO,
name='categorical_hinge'
)
```

Ejercicio

- What is the gradient of the cost function in hinge loss?
- How hinge loss will perform as a cost function for a feedforward neural network? Will there be saturation problems?

Funciones de activación

Output layer

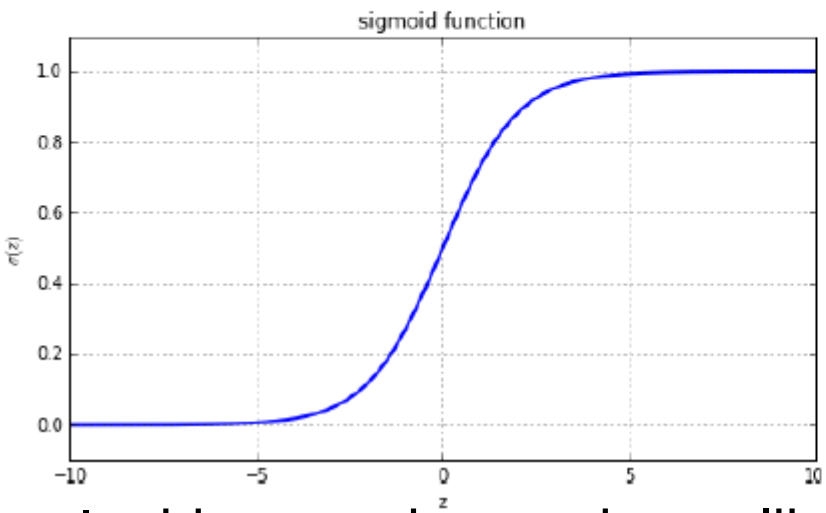
- The **activation function** in the output layer is chosen with regard to the expected output/encoding of the network.
- It is important to take into account the **cost function** to be used for the training.
- In general, we will assume here that **maximum likelihood** is used.

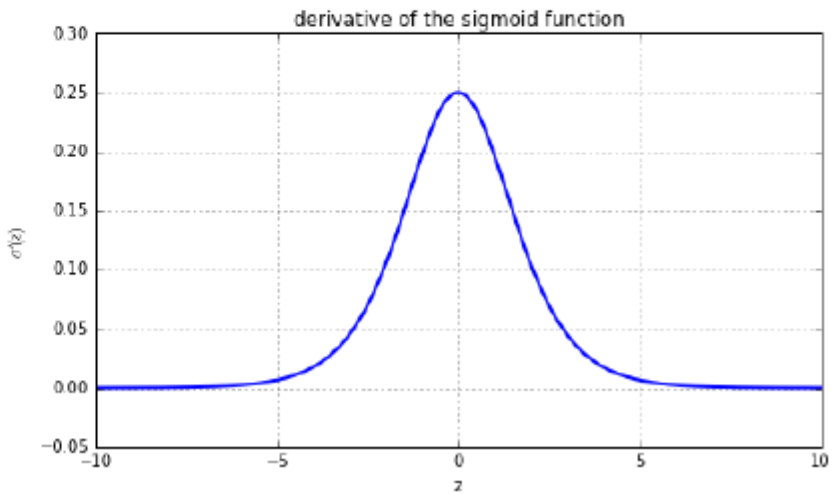
Activation functions

Binary output

- Two classes, $t \in \{0,1\}$, and a single output neuron.
- We use the sigmoid activation function:

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





- In this case the maximum likelihood produces the cross-entropy cost function.

Activation functions

Multiclass problem (softmax function):

- For an arbitrary number n of classes, we use n output units, and the **softmax** activation function.

- Let $\mathbf{h}$ be the output of the last hidden layer, and $\mathbf{b}$ and $\mathbf{W}$ the bias and the weight matrix for the output layer, respectively.

- The input $\mathbf{z}$ to the output layer is: $\mathbf{z} = \mathbf{W}\mathbf{h} + \mathbf{b}$

- The output of the network is given by:

$$y_j = softmax(z_j) = \frac{e^{z_j}}{\sum_{l=1}^{n} e^{z_l}}$$

- Note that the output of all units in the output layer are “coupled”.

Activation functions

Multiclass problem (softmax function):

- The outputs of the network are all positive and add up to 1.

- We can interpret them as probabilities:

$$y_j(\mathbf{x}) = p(t = j|\mathbf{x})$$

- Applying maximum likelihood, we obtain a cost function that generates as delta for the output layer:

$$\boldsymbol{\delta}^{(K)} = (\sigma(\mathbf{z}^{(K)}) - \mathbf{t})$$

where vector $\mathbf{t}$ has a 1 in the position corresponding to the class.

- In this way, **softmax is a generalization of the sigmoid**.

Activation functions

Gaussian output (linear regression):

- We use an **output linear unit** when we wish to generate an output that represents the mean of a Gaussian distribution conditioned to the input:

$$p(\mathbf{t}|\mathbf{x}) = N(\mathbf{t}; \mathbf{y}, \mathbf{I})$$

- The activation is the identity function:

$$\mathbf{y} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

- In this case maximum likelihood produces as a function the **squared error** for the output layer:

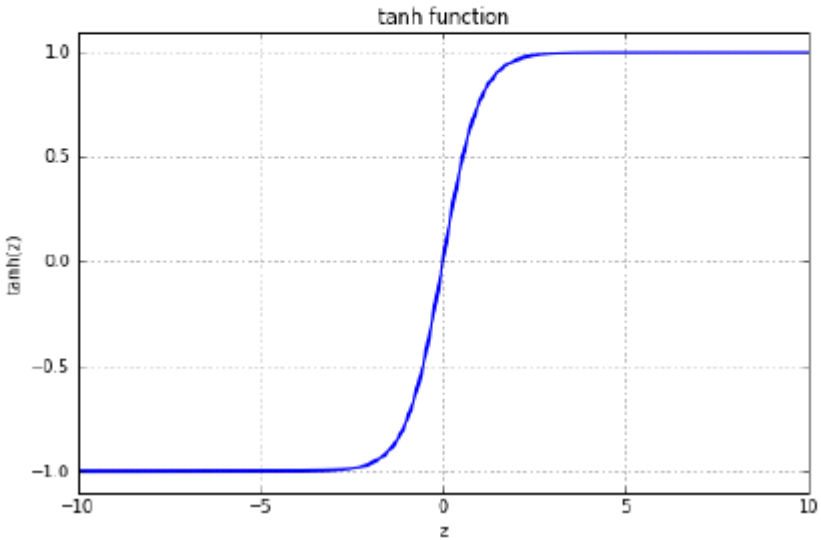
- Linear units do not saturate.

Activation functions

Activations in the hidden layers

- Sigmoid and $\tanh$ functions were frequently used for hidden layers.

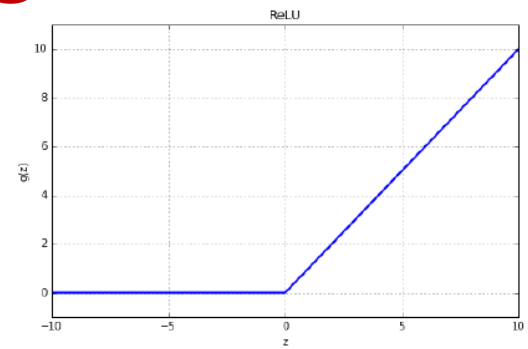
$$\tanh(z) = \frac{1 - e^{-2z}}{1 + e^{-2z}} = 2\sigma(2z) - 1$$



- They are discouraged because saturate, canceling out the gradient and making the learning process difficult.

Activation functions

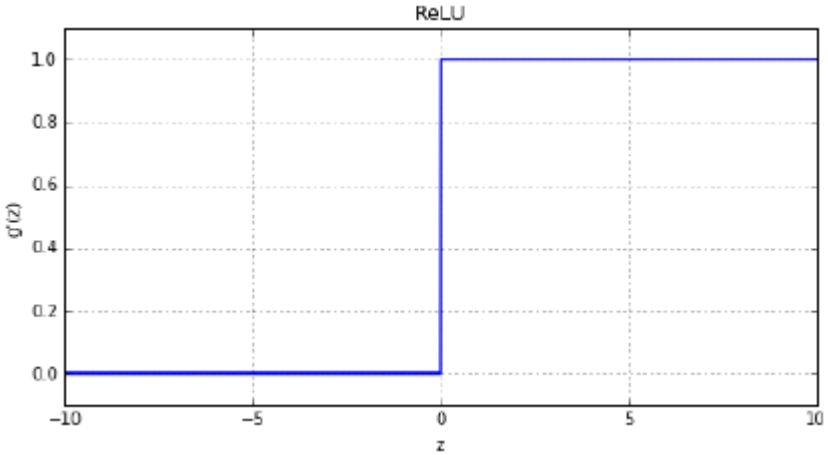
$$g(z) = \text{máx}\,(0, z)$$



ReLU activation function

- **ReLU** is not differentiable in z=0.
- In practice, for z=0 it is common to take the derivative from the left, i.e.:

$$g'(z) = \begin{cases} 0, & \text{if } z \leq 0 \\ 1, & \text{if } z > 0 \end{cases}$$



- The derivative is discarded when the unit is inactive. Only the examples that activate the neuron are allowed to modified the weights.

Activation functions

Other activation functions

- **Leaky ReLU:**

$$f(z) = \mathbb{1}(z < 0)\alpha z + \mathbb{1}(z \geq 0)z$$

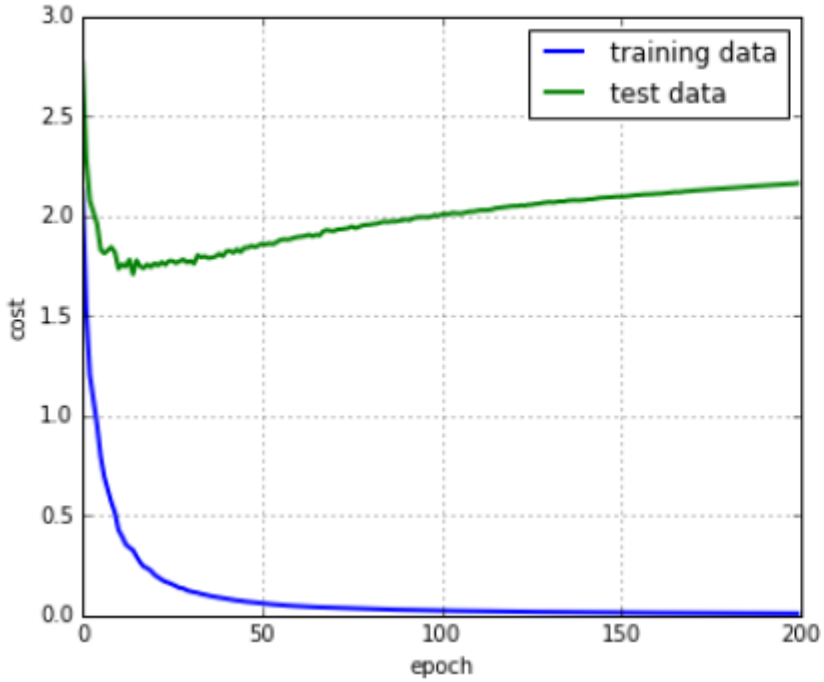
- **Maxout:**

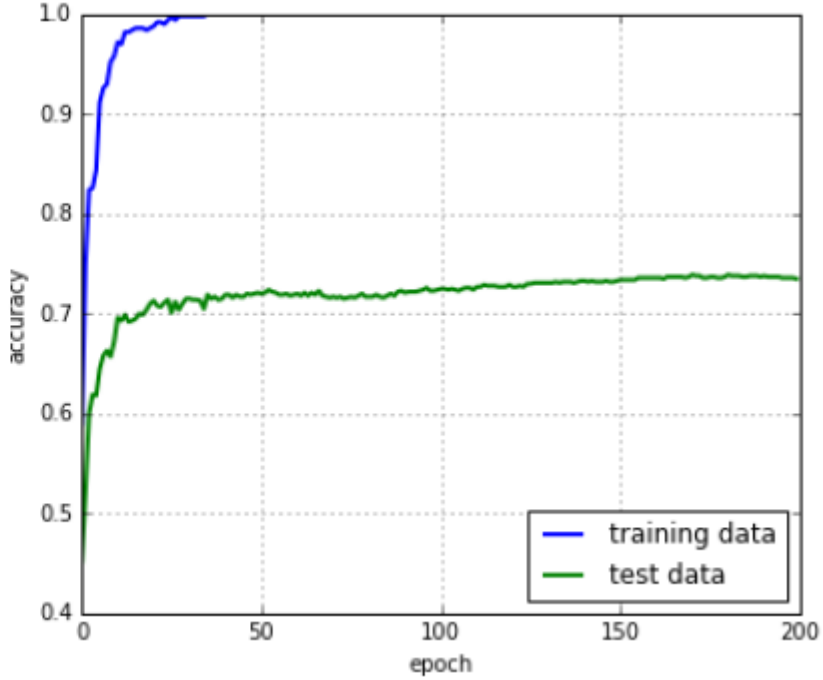
$$f(\mathbf{x}) = \text{máx}\left(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1, \mathbf{W}_2\mathbf{x} + \mathbf{b}_2\right)$$

Regularización

The concept of overfitting

- How does the model work with a test set?





- **Overfitting**: from a given number of epochs on, the model fits too much to the training set and starts generalizing worst.

Regularización

How to avoid overfitting?

- Using more data
- Using a validation set, early stopping.
- Apply **regularization**.

Regularization

- In a broad sense, regularization is any modification in the learning algorithm whose goal is to **reduce the generalization error** and not the training error.

- In a strict sense, regularization is the **adding of terms** in the cost function **which penalize the complexity** of the model favoring its generalization capability.

Regularización

L^2 regularization

- The cost function is modified as follows:

$$J_{L^2} = J + \lambda \sum_w w^2$$

where:

  - J is the non-regularized cost function (error).
  - Summation is over all weights in the network.
  - $\lambda > 0$ is the **regularization parameter**.

- L^2 regularization is also known as **weight decay**

Regularización

Backpropagation with L^2 regularization

$$\boldsymbol{\delta}^{(K)} = \nabla_{\mathbf{a}^{(K)}} C \odot f'(\mathbf{z}^{(K)})$$

$$\boldsymbol{\delta}^{(k)} = ((\mathbf{W}^{(k+1)})^T \boldsymbol{\delta}^{(k+1)}) \odot f'(\mathbf{z}^{(k)}), \quad k < K$$

$$\mathbf{b}^{(k)} \leftarrow \mathbf{b}^{(k)} - \eta \boldsymbol{\delta}^{(k)}$$

$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)}(1 - 2\eta\lambda) - \eta \boldsymbol{\delta}^{(k)} (\mathbf{a}^{(k-1)})^T$$

- Deltas and bias are computed as usual.
- Weights are rescaled by a factor $1\text{-}2\eta\lambda$ before the gradient descent (weight decay).

Regularización

L^1 regularization

- The cost function is modified as follows: $J_{L^1} = J + \lambda \sum_w |w|$

- It also results in a **weight decay**, but in a constant way that does not depend on their magnitude.

Backpropagation with L^1 regularization

$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} - \eta\lambda \operatorname{sgn}(\mathbf{W}^{(k)}) - \eta\boldsymbol{\delta}^{(k)}(\mathbf{a}^{(k-1)})^T$$

Regularización

L^1 and L^1 regularization in Keras

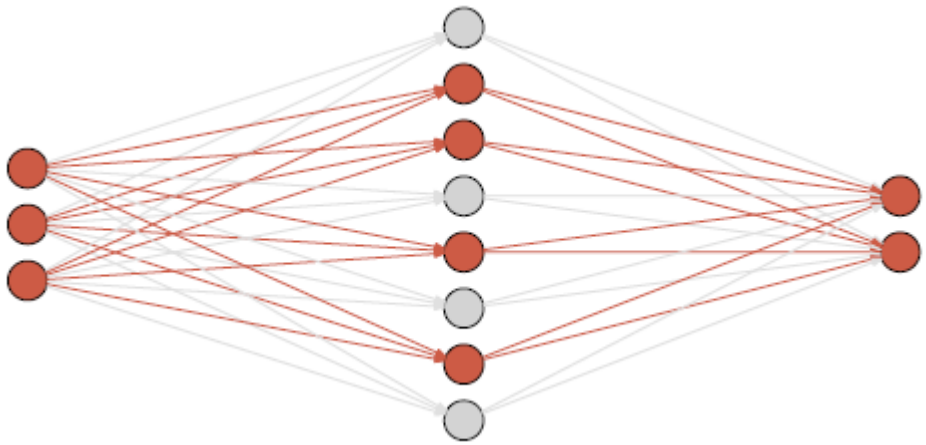
- When defining a layer, we can indicate:
  - `kernel_regularizer`: regularizer for the weights
  - `bias_regularizer`: regularizer for the bias
  - `activity_regularizer`: regularizer for the activations

Example: L^2 regularization in a dense layer

```
layer = keras.layers.Dense(
units=32,
kernel_regularizer=keras.regularizers.l2(0.01)
)
```

Dropout

- With each mini-batch, we randomly remove some of the hidden units (typically half of them).



- Both feedforward activity propagation and error propagation are computed using only the remaining units.

- Thus, with each mini-batch only some weights of the network are trained.

Dropout

- Finally**, for classification we use the whole network**, multiplying the weights coming out from a given unit times the probability of using that unit during training (typically ½).

- With this procedure we are building an **ensemble of networks** and **averaging their opinions** in each of them.

- By dropping out, we force the network to be **robust against the loss of units**.

Further reading:

Srivastava et al. 2014. Dropout: A Simple Way to Prevent Neural Networks from overfitting. *Journal of Machine Learning Research* 15 (2014):1929-1958.

Dropout

Vanilla Dropout, training

```
p = 0.5 # probability of keeping a unit active. higher =
less dropout
def train_step(X):
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = np.random.rand(*H1.shape) < p # first dropout mask
H1 *= U1 # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = np.random.rand(*H2.shape) < p # second dropout mask
H2 *= U2 # drop!
out = np.dot(W3, H2) + b3
# backward pass: compute gradients... (not shown)
# perform parameter update... (not shown)
```

From http://cs231n.github.io/neural-networks-2/

Dropout

Vanilla Dropout, prediction

```
def predict(X):
# ensembled forward pass
H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale
the activations
H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale
the activations
out = np.dot(W3, H2) + b3
```

From http://cs231n.github.io/neural-networks-2/

Dropout

Inverted Dropout, training

Drop and scale at train time, the forward pass at test time does not change

```
p = 0.5 # probability of keeping a unit active. higher = less
dropout
def train_step(X):
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask.
Notice /p!
H1 *= U1 # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask.
Notice /p!
H2 *= U2 # drop!
out = np.dot(W3, H2) + b3
# backward pass: compute gradients... (not shown)
# perform parameter update... (not shown)
```

From http://cs231n.github.io/neural-networks-2/

Dropout

Inverted Dropout, prediction

Nothing new here

```
def predict(X):
# ensembled forward pass
H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale
the activations
H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale
the activations
out = np.dot(W3, H2) + b3
```

From http://cs231n.github.io/neural-networks-2/

Dropout en Keras

Implemented as an additional layer:

```
tf.keras.layers.Dropout(
rate, noise_shape=None, seed=None, **kwargs
)
```

Example: model with dropout in the hidden layer

```
model = keras.Sequential()
model.add(keras.layers.Flatten(input_shape=(28, 28)))
model.add(keras.layers.Dense(32, activation="relu"))
model.add(tf.keras.layers.Dropout(0.25))
model.add(keras.layers.Dense(10, activation="softmax")
```

Exercise: test adding a dropout layer just after the hidden layer

Inicialización de pesos

- Weights should not be initialized to 0.
- A first option is to initialize the weights randomly, with low values centered around 0:

```
# Being W a DxH matrix:
W = 0.01* np.random.randn(D,H)
```

- Problem: How do we choose the scale (variance) of the distribution?

Inicialización de pesos

- A first goal is to avoid that the neurons are saturated at the beginning of the training.

- In the case of a **sigmoid neuron** with n incoming weights $w_1, w_2, \ldots, w_n$ and bias b, we can initialize as:

$$w_i \sim N(0, 1/\sqrt{n})$$
$$b \sim N(0, 1)$$

  - It is also common to initialize the bias to 0.

Inicialización de pesos

Xavier initialization

- Weights are initialized as:

$$w_i \sim N(0, \sqrt{\frac{2}{n_{in} + n_{out}}})$$

- n_{in} is the number of neurons in the previous layer
- n_{out} is the number of neurons in the next layer.

Glorot, Xavier; Bengio, Yoshua. Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010. p. 249-256.

Inicialización de pesos

Initialization for ReLU units

- The current recommendation to initialize weights in a ReLU neuron is:

$$w_i \sim N(0, \sqrt{\frac{2}{n}})$$

  - Bias are initialized to a small positive value (typically 0.1 or 0.01) to bias the units towards the positive part and achieve that they are initially active.

He et al. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. https://arxiv.org/abs/1502.01852.

Inicialización de pesos

Keras initializers

https://keras.io/api/layers/initializers/

Batch Normalization

- This technique simplifies the task of initializing the weights.
- The idea is to **normalize the activity of each unit** (before the nonlinearity).
- It can be applied because the **normalization is differentiable**.
- With batch-normalization the network is less sensible to a bad weight initialization

Further reading:

S. Ioffe, C. Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. https://arxiv.org/abs/1502.03167

Batch Normalization

Keras batch normalization layer

https://keras.io/api/layers/normalization_layers/batch_normalization/

Otras técnicas de optimización

Second order methods: the Newton method

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \mathbf{H}^{-1} \nabla J$$

- $\mathbf{H}$ is the Hessian matrix:

$$\mathbf{H}_{ij} = \frac{\partial^2 J}{\partial w_i w_j}$$

- The convergence is faster than plain gradient descent.
- In practice, it is a little tricky to use for large networks, as inverting $\mathbf{H}$ is computationally expensive.

Other optimization techniques

Gradient with momentum

- Gradient descent with **momentum**:

$$\begin{aligned} \mathbf{v} &\leftarrow \mu\mathbf{v} - \eta\nabla J \\ \mathbf{w} &\leftarrow \mathbf{w} + \mathbf{v} \end{aligned}$$

- The parameter μ is the **momentum coefficient**:

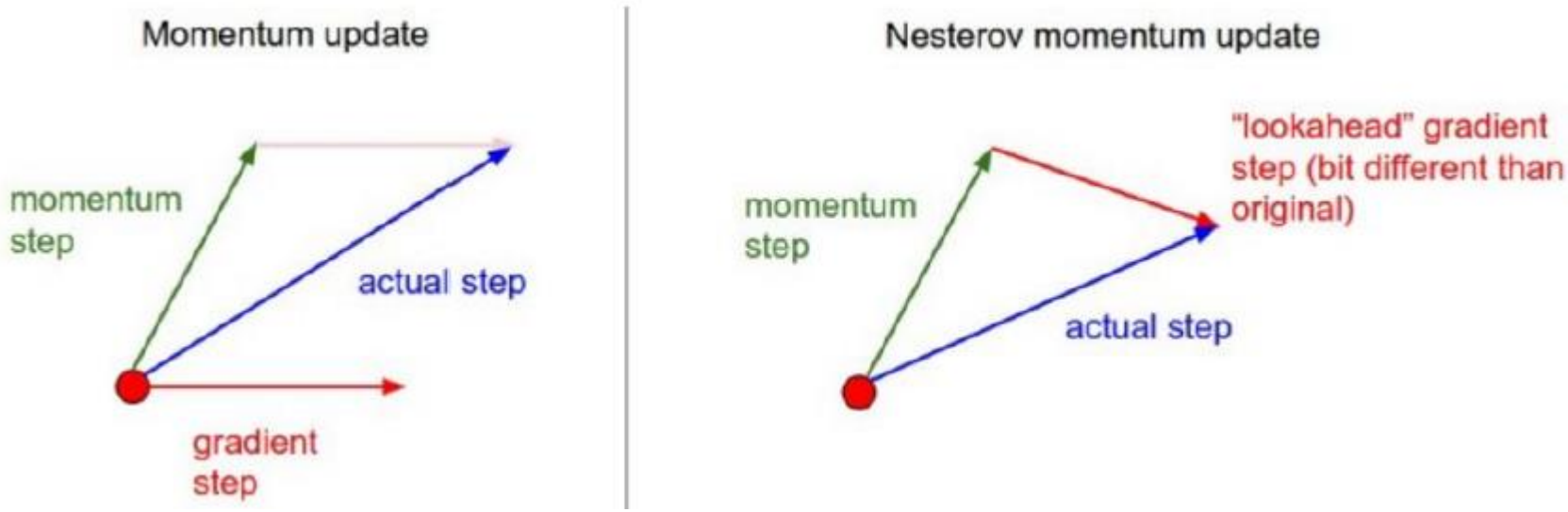
$$0 < \mu < 1$$

```
#Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

from http://cs231n.github.io/neural-networks-3/

Other optimization techniques

Nesterov momentum




```
x_ahead = x + mu * v
# evaluate dx_ahead (the gradient at x_ahead instead of at x)
v = mu * v - learning_rate * dx_ahead
x += v
```

from http://cs231n.github.io/neural-networks-3/

Other optimization techniques

Gradient with momentum in Keras

```
tf.keras.optimizers.SGD(
learning_rate=0.01, momentum=0.0, nesterov=False,
name="SGD", **kwargs
)
```

https://keras.io/api/optimizers/sgd/

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD

Other optimization techniques

AdaGrad

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

(From http://cs231n.github.io/neural-networks-3/)

Further reading:

J. Duchi, E. Hazan, Y. Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. http://jmlr.org/papers/v12/duchi11a.html

Other optimization techniques

AdaGrad in Keras

```
tf.keras.optimizers.Adagrad(
learning_rate=0.001, initial_accumulator_value=0.1,
epsilon=1e-07, name='Adagrad', **kwargs
)
```

https://keras.io/api/optimizers/adagrad/

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adagrad

Other optimization techniques

RMSProp

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps))
```

(From http://cs231n.github.io/neural-networks-3/)

Further reading:

Neural Networks for Machine Learning, Lecture 6a, G. Hinton (2012).

http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Other optimization techniques

RMSProp in Keras

```
tf.keras.optimizers.RMSprop(
learning_rate=0.001, rho=0.9, momentum=0.0, epsilon=1e-
07,
centered=False, name='RMSprop', **kwargs
)
```

https://keras.io/api/optimizers/rmsprop/

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/RMSprop

Other optimization techniques

Adam

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

(From http://cs231n.github.io/neural-networks-3/)

Further reading:

D.P. Kingma, J. Ba. 2014. Adam: A Method for Stochastic Optimization
https://arxiv.org/abs/1412.6980

Other optimization techniques

Adam in Keras

```
tf.keras.optimizers.Adam(
learning_rate=0.001, beta_1=0.9, beta_2=0.999,
epsilon=1e-07,
amsgrad=False, name='Adam', **kwargs
)
```

https://keras.io/api/optimizers/adam/

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

Other optimization techniques

Other optimizers

- Adadelta
- Adamax
- Nadam
- Ftrl

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers

Hyperparameter selection

- **Regularization constant, size of the mini-batch, network architecture** (number of hidden units, etc.):
  - Typically are chosen through some type of validation.
  - Automatic grid search techniques or random search techniques.

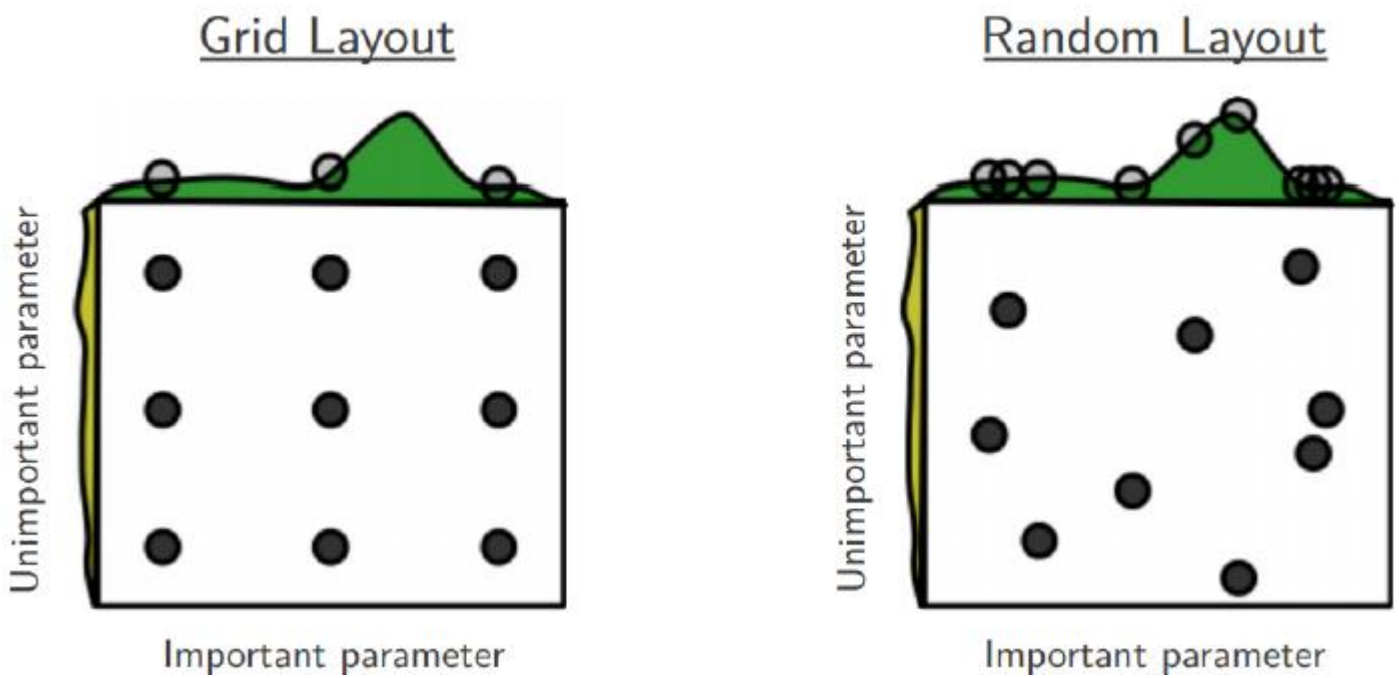
- Number of **training epochs**:
  - It can be adjusted by early stopping.

- **Learning rate**:
  - It can be adjusted by monitoring the cost on the training set.

Hyperparameter selection

Grid search vs. random search

- In general, random search is more efficient as a hyperparameter optimization method.

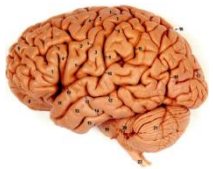


(From: Bergstra & Bengio: Random Search for Hyper-Parameter Optimization
http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf

Recomendaciones prácticas

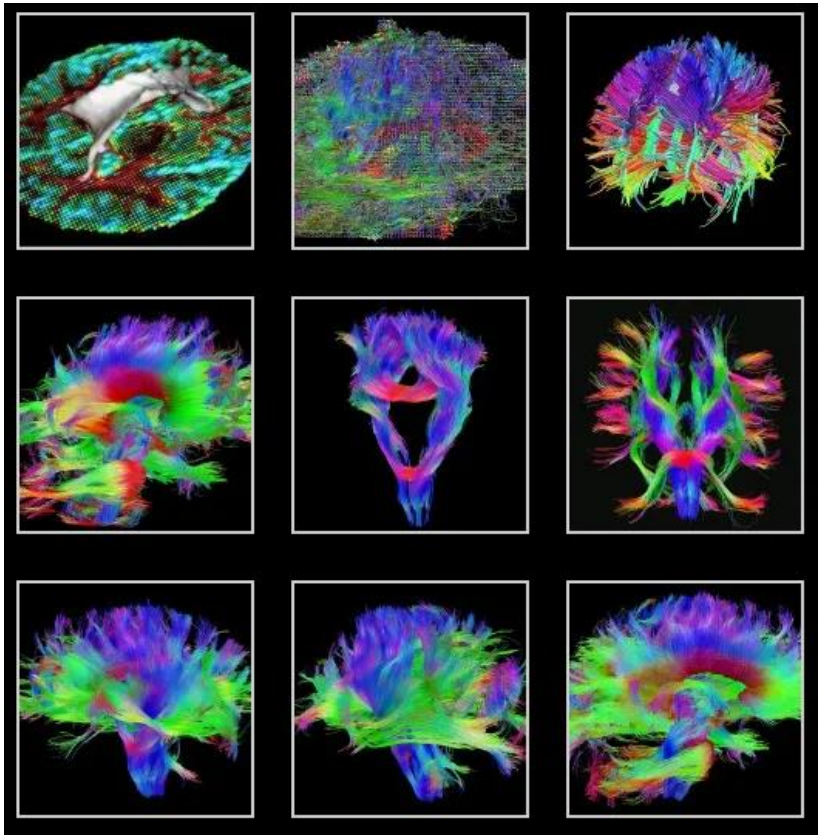
- Practical Recommendations for Gradient-Based Training of Deep Architectures, Y. Bengio (2012). https://arxiv.org/abs/1206.5533
- Neural Networks: Tricks of the Trade, G. Montavon, G. Orr, K.R. Müller (2012).
- Stochastic Gradient Descent Tricks, L. Bottou (2012). https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/tricks-2012.pdf

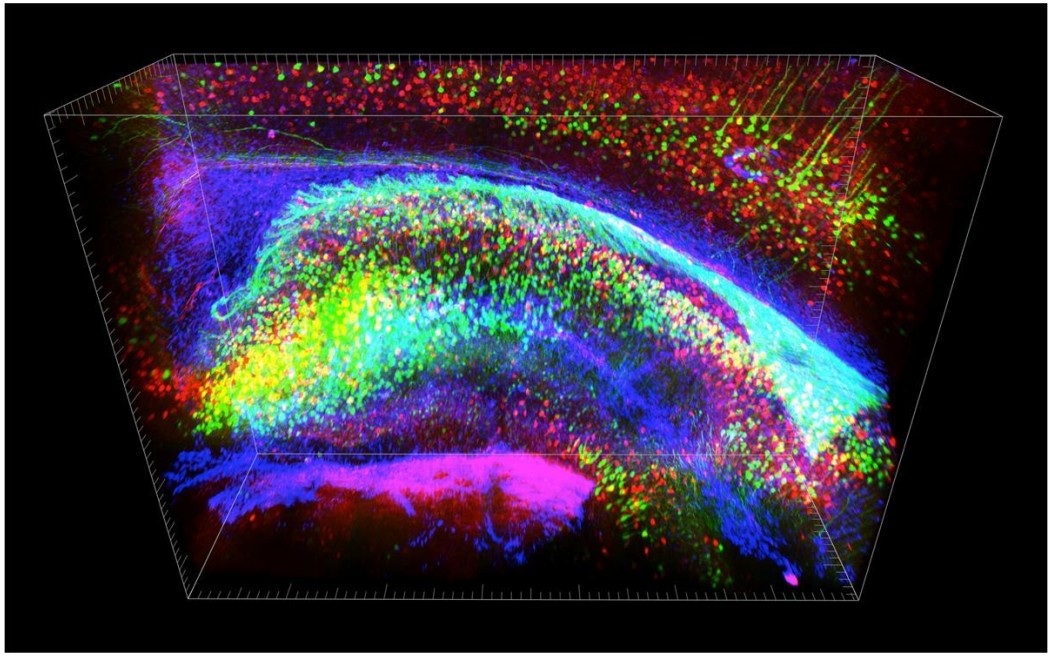
El cerebro funciona de forma masivamente paralela

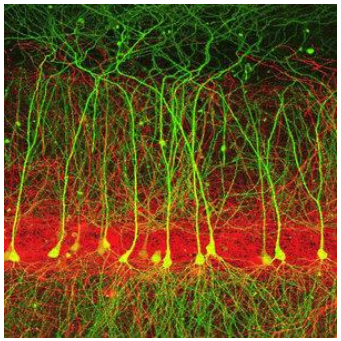
Information processor	Number of processing elements	Element size	Energy consumption	Processing speed	Type of computation	Fault Tolerance	Able to learn	Intelligent, conscious
	10^{14} synapses	10^{-6} m	30 W	100 Hz	Parallel, distributed	Yes	Yes	Often
	10^{8} transistors	10^{-6} m	30W (CPU)	10^{9} Hz	In series, centralized	NO	A little bit	No (yet)

El cerebro funciona de forma masivamente paralela

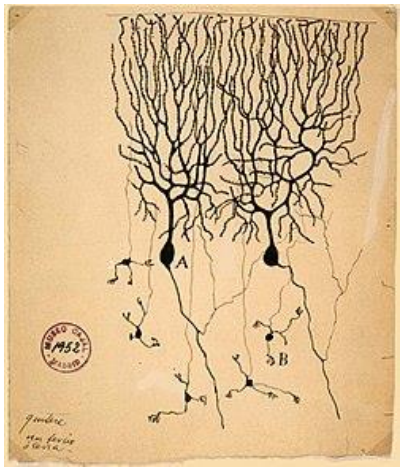
Diffusion MRI of the human brain. Source Human Connectome Project)

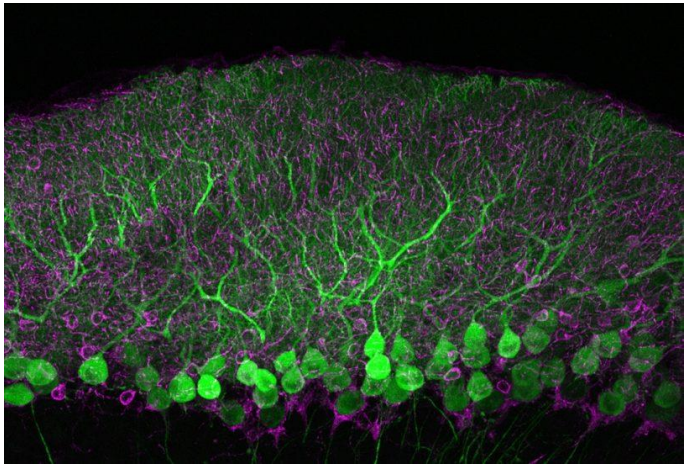






https://doi.org/10.1016/j.neuron.2016.01.029





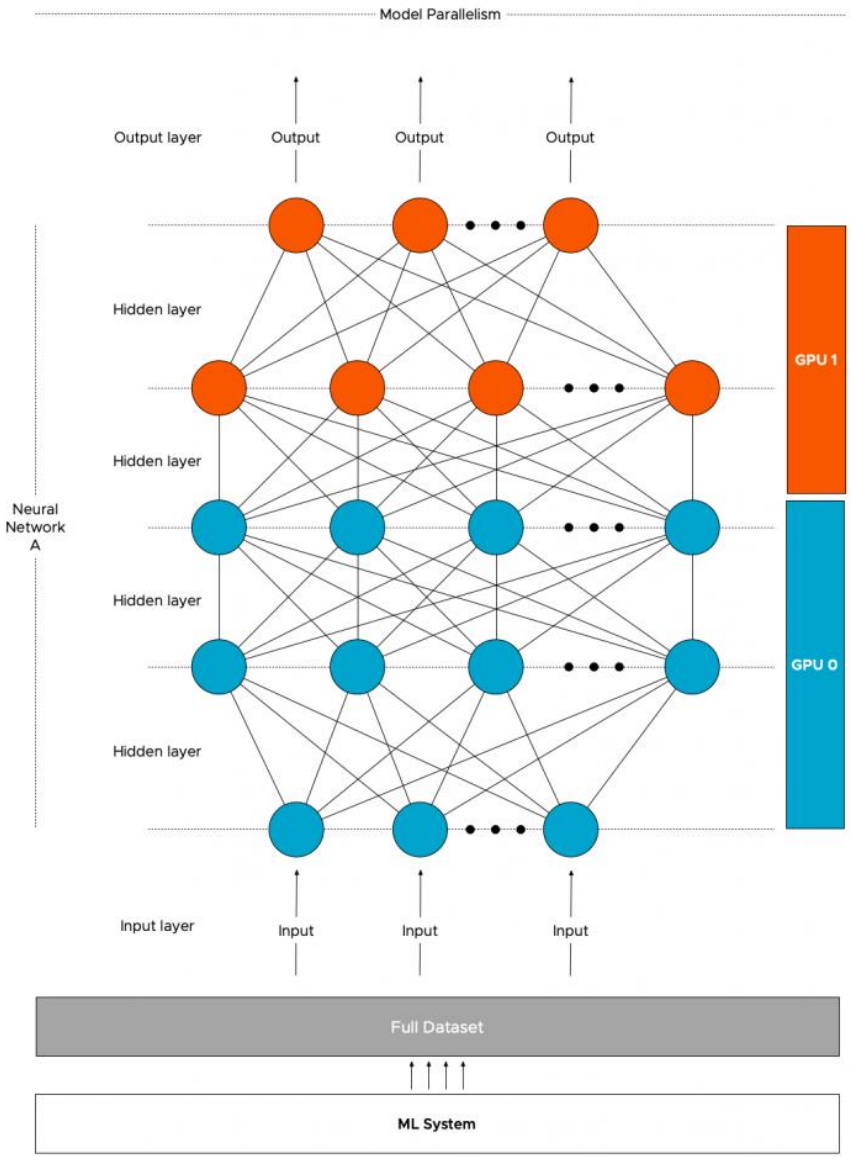
https://doi.org/10.7554/eLife.36246.001

Las redes neuronales se pueden paralelizar fácilmente

- Because of their **modular nature**, neural networks in general, an in particular deep neural networks can be parallelized both using multicore hardware architectures and GPUs.

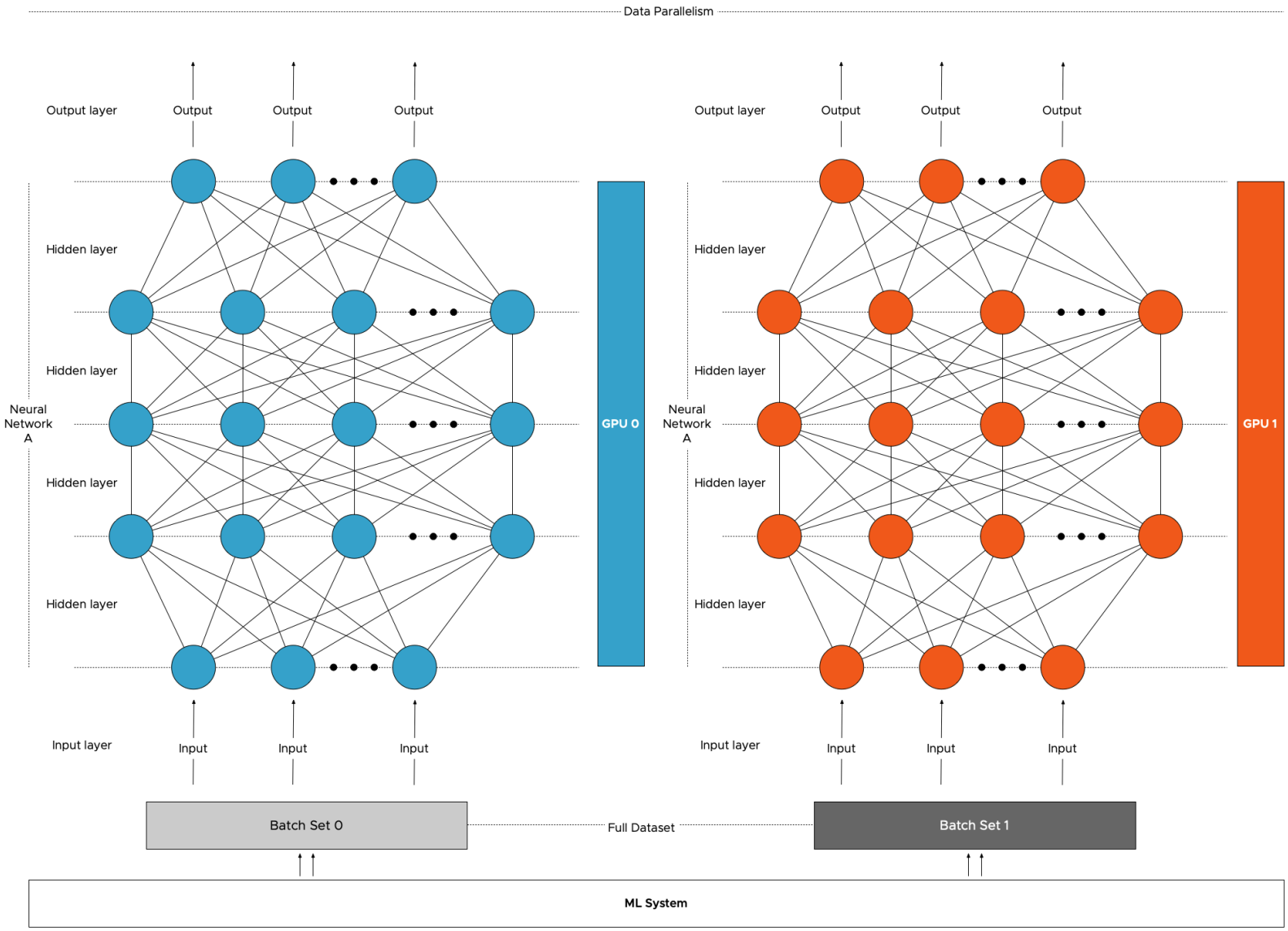
- Parallelization strategies may include both network **model implementation** and **data parallelization** strategies.

Paralelización del modelo en DNNs



https://frankdenneman.nl/2020/02/19/multi-gpu-and-distributed-deep-learning/

Paralelización de datos en DNNs



https://frankdenneman.nl/2020/02/19/multi-gpu-and-distributed-deep-learning/

Paralelización en Keras

https://keras.io/guides/distributed_training/

Uso de GPUs enTensorflow y Keras

- How to enable GPUs for the notebook:

1. Navigate to Edit→Notebook Settings
2. Select GPU from the Hardware Accelerator drop-down

Check that GPU is working and run a simple timing test:

https://colab.research.google.com/notebooks/gpu.ipynb#scrollTo=oM_8ELnJq_wd

Check difference between GPU an TPU (Tensor Processing Units, Google's custom-developed application-specific integrated circuits –ASICs – used to accelerate machine learning workloads).

Uso de GPUs enTensorflow y Keras

- How to **enable GPUs for the notebook**:

1. Navigate to Edit→Notebook Settings
2. Select GPU from the Hardware Accelerator drop-down

Check that GPU is working and run a simple timing test:

https://colab.research.google.com/notebooks/gpu.ipynb#scrollTo=oM_8ELnJq_wd

Uso de GPUs enTensorflow y Keras

- Check difference between **GPU** and **TPU** performance (Tensor Processing Units, Google's custom-developed application-specific integrated circuits – ASICs – used to accelerate machine learning workloads). Adapt this examples to your own network-

https://colab.research.google.com/notebooks/gpu.ipynb

https://colab.research.google.com/notebooks/tpu.ipynb