Deep Feedforward Networks

Luis Fernando Lago Fernández

1. Presentación

Recursos

- **Neural Networks and Deep Learning**, M. Nielsen, online book, http://neuralnetworksanddeeplearning.com
- **Código** del libro de Nielsen disponible en github, https://github.com/mnielsen/neural-networks-and-deep-learning/archive/master.zip
- **Deep Learning**, I. Goodfellow, Y. Bengio, A. Courville, MIT Press, http://www.deeplearningbook.org/
- **Convolutional Neural Networks for Visual Recognition**, curso Stanford, http://cs231n.stanford.edu/
- **Tutorial de TensorFlow**, https://www.tensorflow.org/tutorials/
- **TensorFlow Playground**, http://playground.tensorflow.org

Deep Feedforward Networks

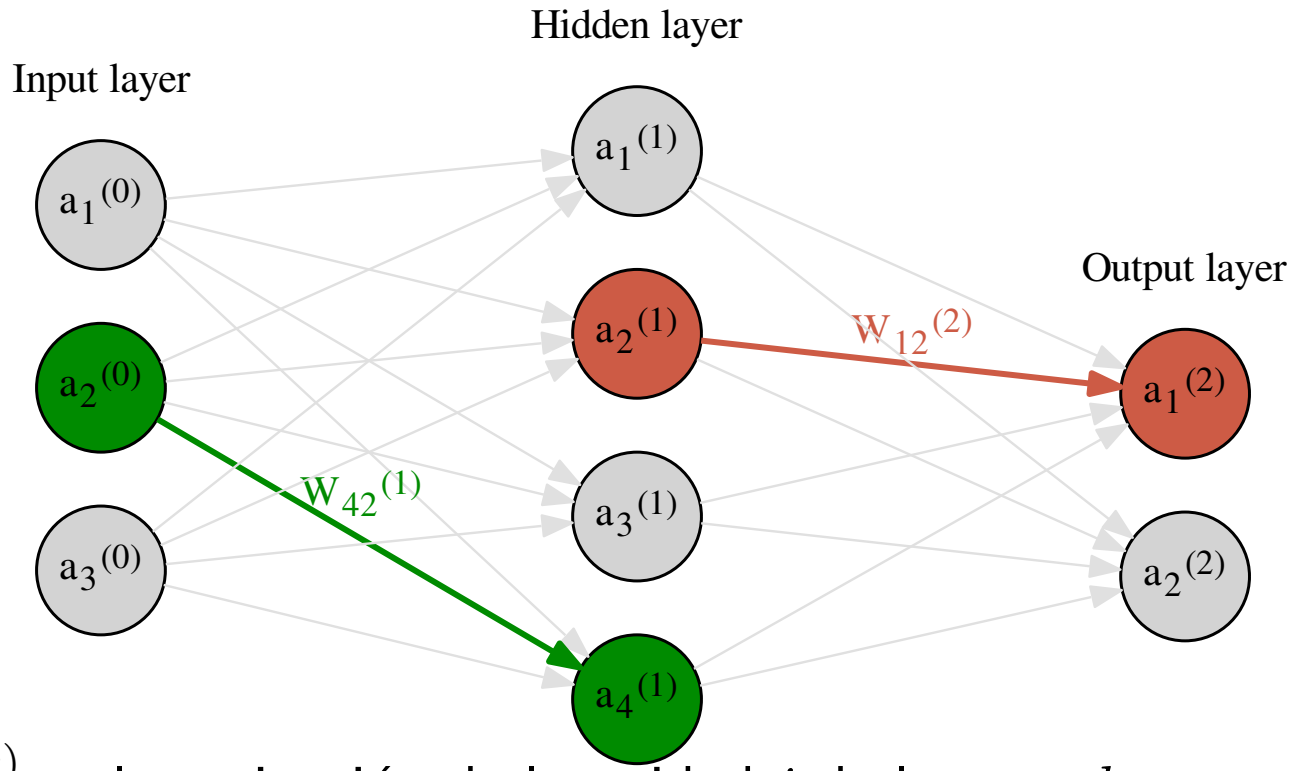
- Presentación
- Repaso de retropropagación
- Entrenamiento de una red neuronal: online, batch, mini-batch
- Función de coste: error cuadrático, cross-entropy, hinge loss
- Función de activación: sigmoide, tanh, ReLU, softmax
- Regularización
  - Regularización L2
  - Dropout
  - Otras técnicas

Deep Feedforward Networks

- Inicialización de los pesos
- Selección de los hiperparámetros
- Resumen de otras técnicas de optimización: métodos de segundo orden, momento
- Las redes neuronales pueden aproximar cualquier función
- El problema del vanishing gradient

2. Retropropagación

Estructura de una red neuronal



- $a_i^{(k)}$ es la activación de la unidad i de la capa k.
- $w_{ij}^{(k)}$ es el peso que conecta la unidad j de la capa $k-1$ con la unidad i de la capa k.

Propagación de la actividad

- La entrada a una unidad es una función lineal de las activaciones en la capa anterior:

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

  n_{k-1} es el número de unidades en la capa $k-1$, $b_i^{(k)}$ es el bias.
- Sobre esta entrada se aplica una función no lineal, la **función de activación** f:

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

- La activación de la capa de entrada $(k = 0)$ es simplemente la entrada a la red: $a_i^{(0)} = x_i$.

En forma matricial

Propagación de la actividad en una red neuronal feedforward

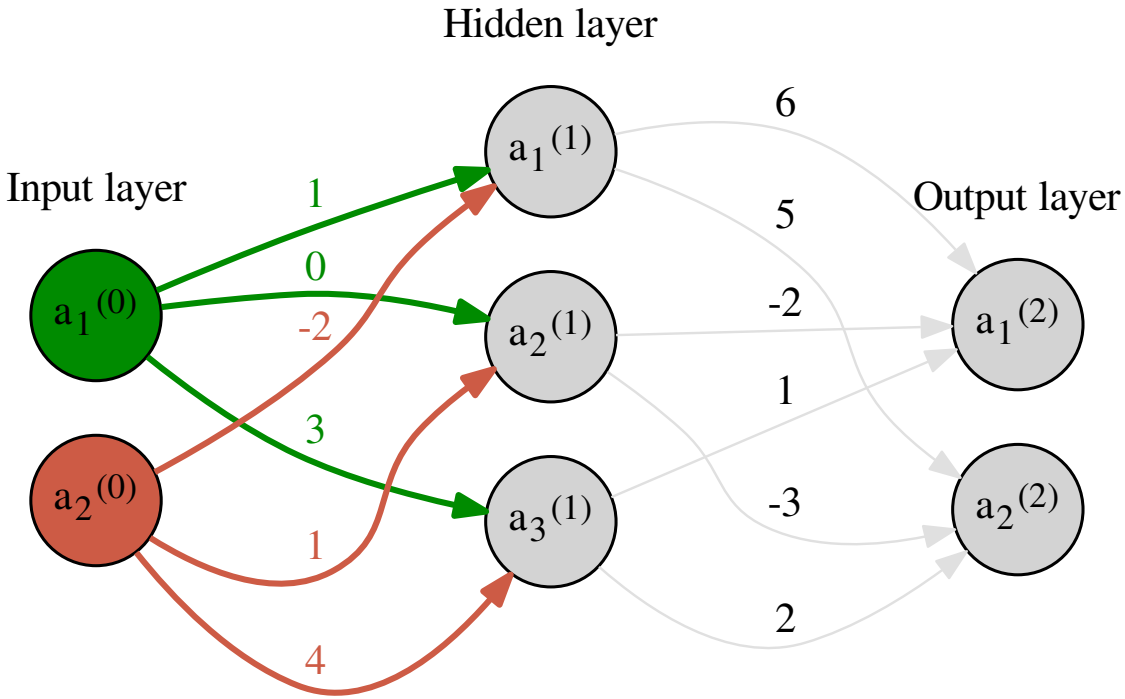
$$\mathbf{a}^{(0)} = \mathbf{x}$$

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

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

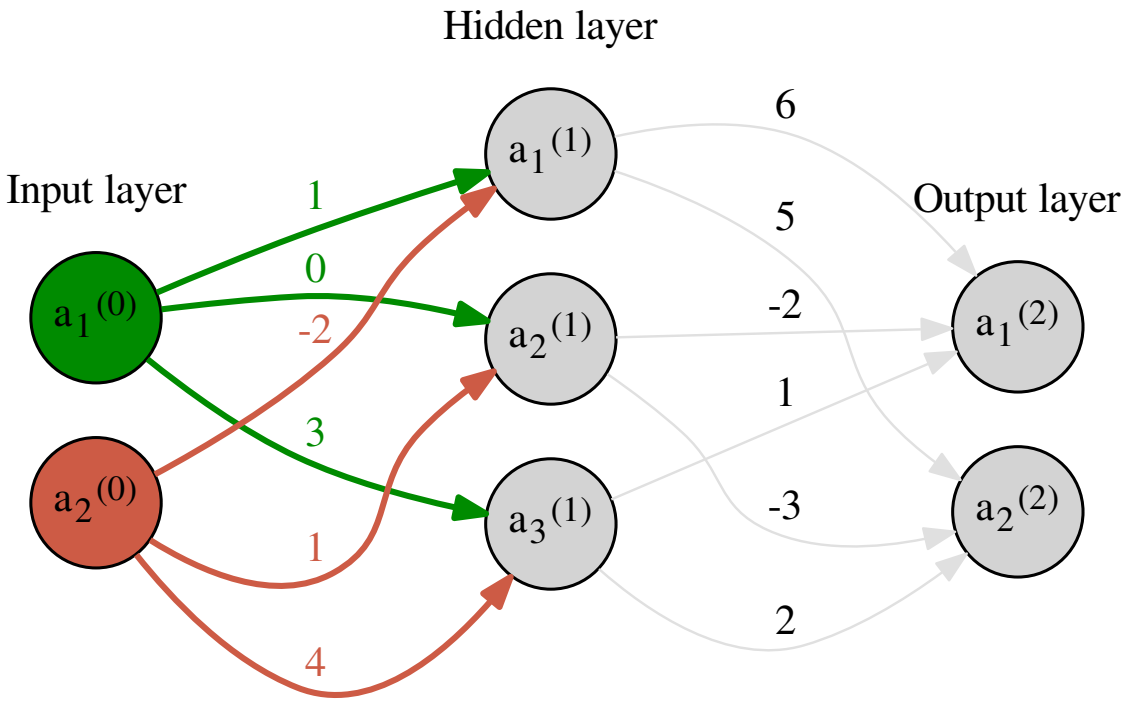
- $\mathbf{a}^{(k)}(n_k \times 1)$ es el **vector de activaciones** para la capa k.
- $\mathbf{W}^{(k)}(n_k \times n_{k-1})$ es la **matriz de pesos** para la capa k:
  - La fila i de $\mathbf{W}^{(k)}$ contiene los pesos que conectan cada unidad en la capa $k-1$ con la unidad i de la capa k.
- $\mathbf{b}^{(k)}(n_k \times 1)$ es el **vector de bias** para la capa k.
- $f(z)$ es la función de activación, que puede ser distinta para cada capa.

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

- Activación en la capa oculta:

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

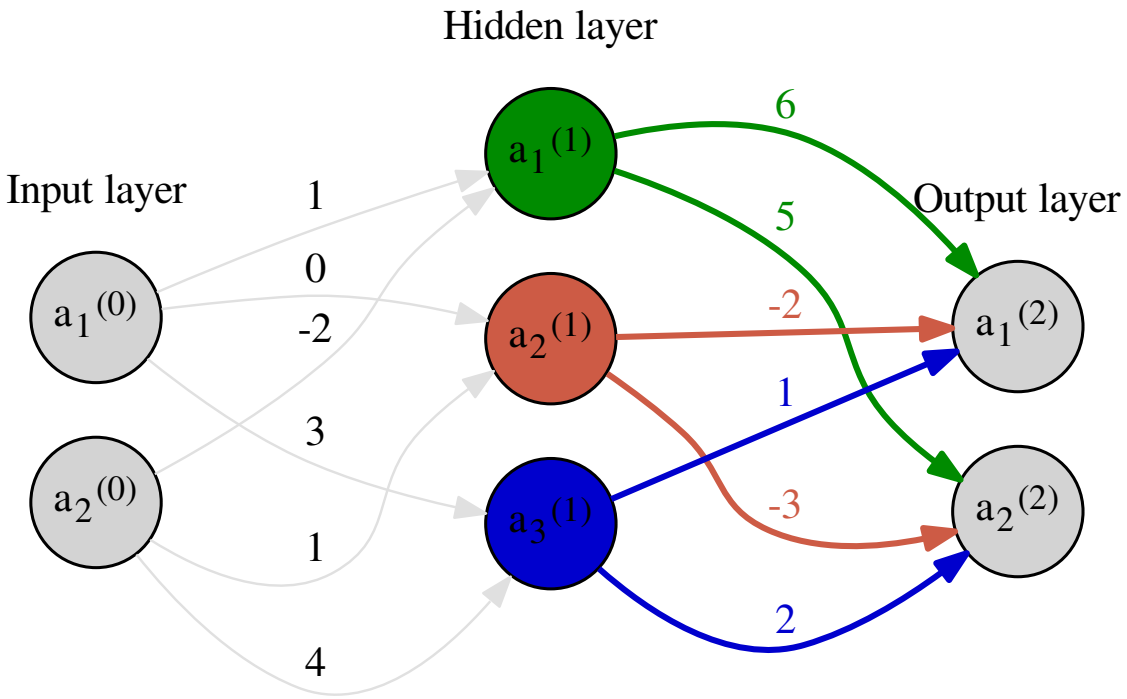
- O bien:

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

- Con:

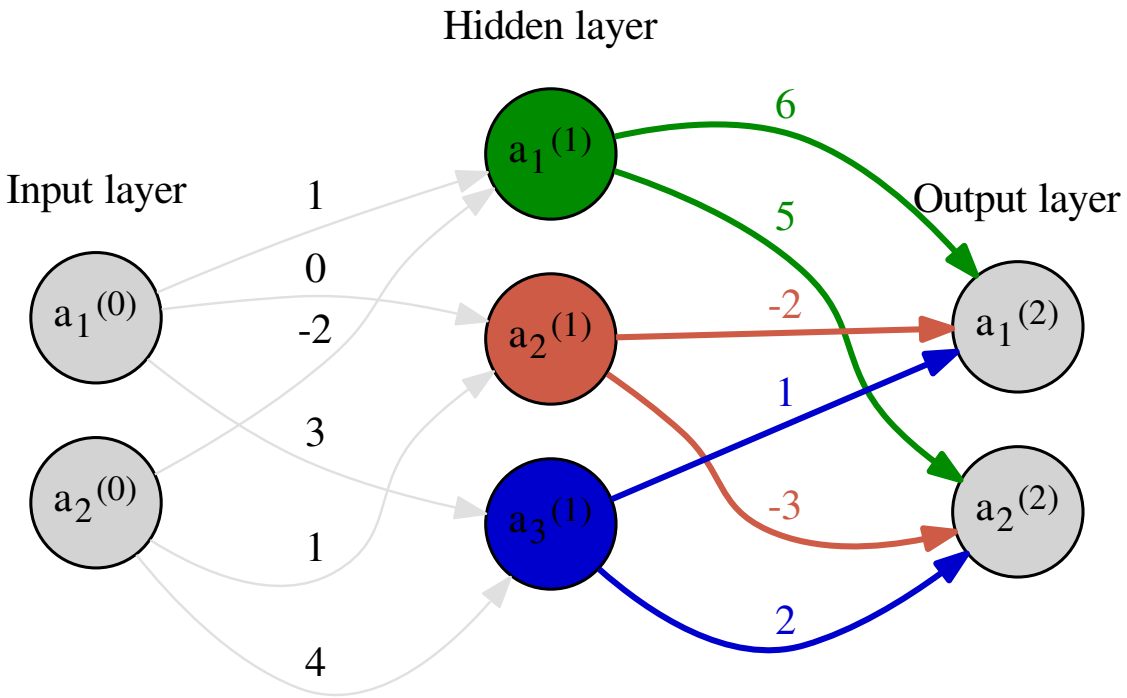
$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & -2 \\ 0 & 1 \\ 3 & 4 \end{bmatrix}$$

Ejemplo



$$
\begin{aligned}
a_1^{(2)} &= f(6 \times a_1^{(1)} - 2 \times a_2^{(1)} + 1 \times a_3^{(1)} + b_1^{(2)}) \\
a_2^{(2)} &= f(5 \times a_1^{(1)} - 3 \times a_2^{(1)} + 2 \times a_3^{(1)} + b_2^{(2)})
\end{aligned}
\tag{1}
$$

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

- Activación en la capa de salida:

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

- O bien:

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

- Con:

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

Código

Cálculo de la activación de la red

```
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
```

(red_simple_para_clase.ipynb)

Retropropagación del error

- Error (**función delta**) en la capa de salida (capa K):

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

  donde $C = C(\mathbf{a}^{(K)}, \mathbf{y})$ es la **función de coste**, que mide la discrepancia entre la salida de la red $\mathbf{a}^{(K)}$ y la esperada $\mathbf{y}$.

- Error (**función delta**) en la capa k ($k < K$):

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

Retropropagación del error

- Derivada de la función de coste respecto a los **bias**:

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

- Derivada de la función de coste respecto a los **pesos**:

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

Retropropagación del error

Descenso por gradiente:

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

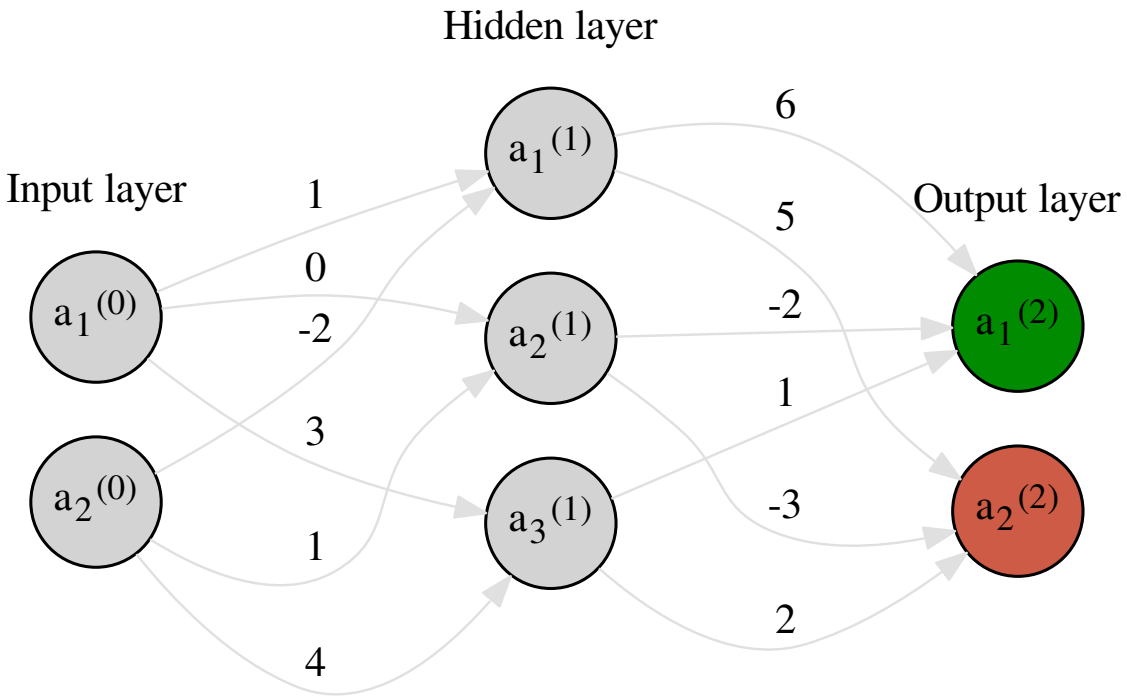
$$\boldsymbol{\delta}^{(k)} = ((\mathbf{W}^{(k+1)})^T \boldsymbol{\delta}^{(k+1)}) \odot f'(\mathbf{z}^{(k)}), \quad k < K$$

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

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

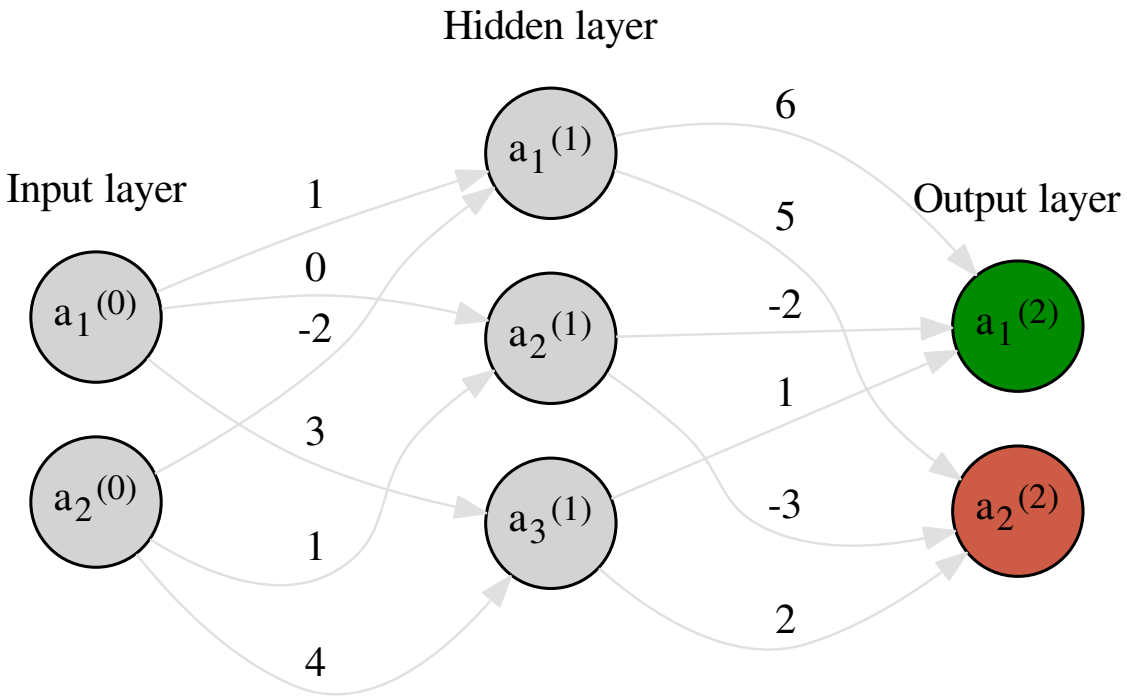
- La constante η es el **factor de aprendizaje**.

Ejemplo



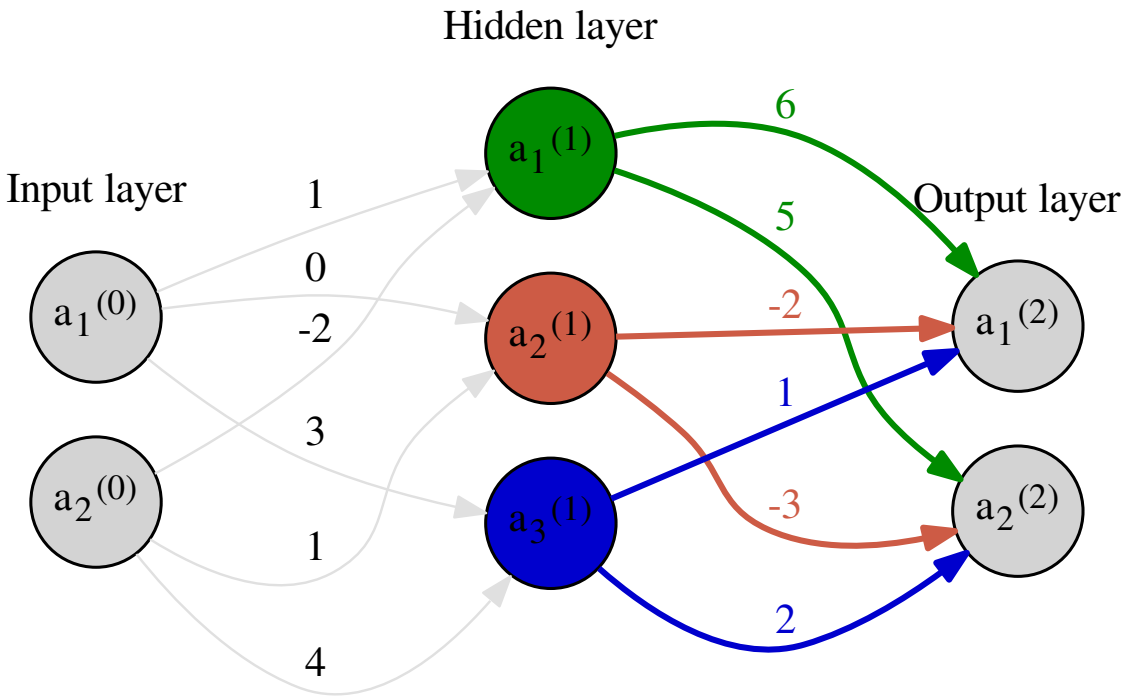
$$\boldsymbol{\delta}^{(2)} = \begin{bmatrix} \delta_1^{(2)} \\ \delta_2^{(2)} \end{bmatrix} = \begin{bmatrix} (\partial C/\partial a_1^{(2)}) f'(z_1^{(2)}) \\ (\partial C/\partial a_2^{(2)}) f'(z_2^{(2)}) \end{bmatrix}$$

Ejemplo



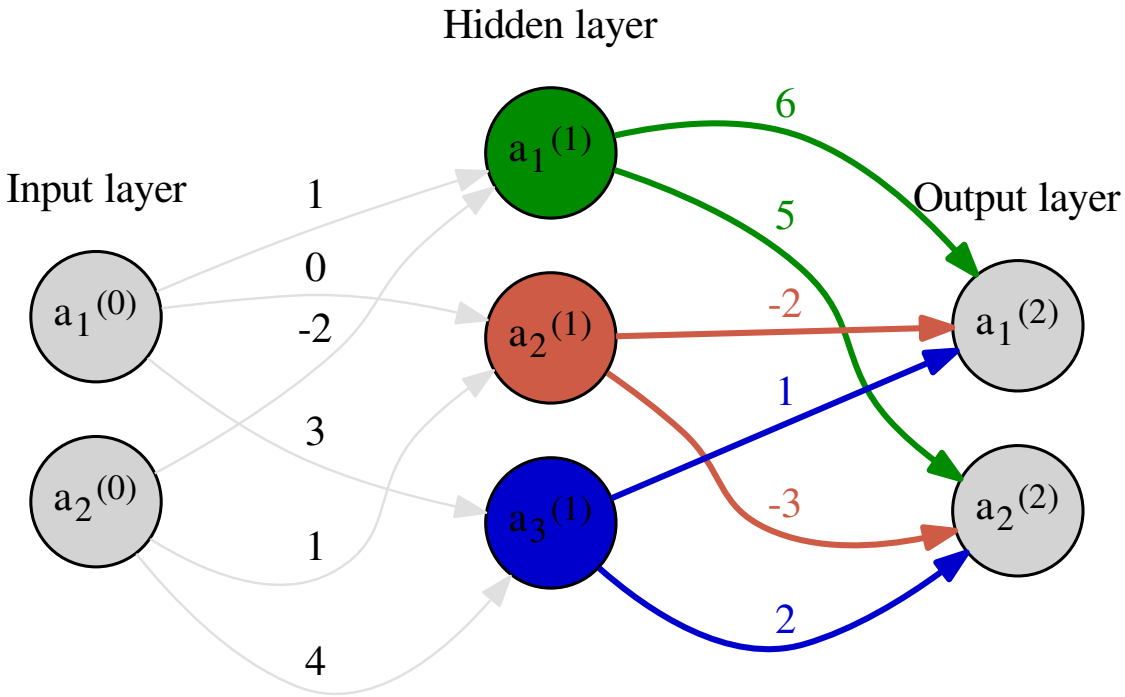
$$\mathbf{b}^{(2)} \leftarrow \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix} - \eta \begin{bmatrix} \delta_1^{(2)} \\ \delta_2^{(2)} \end{bmatrix}$$

Ejemplo



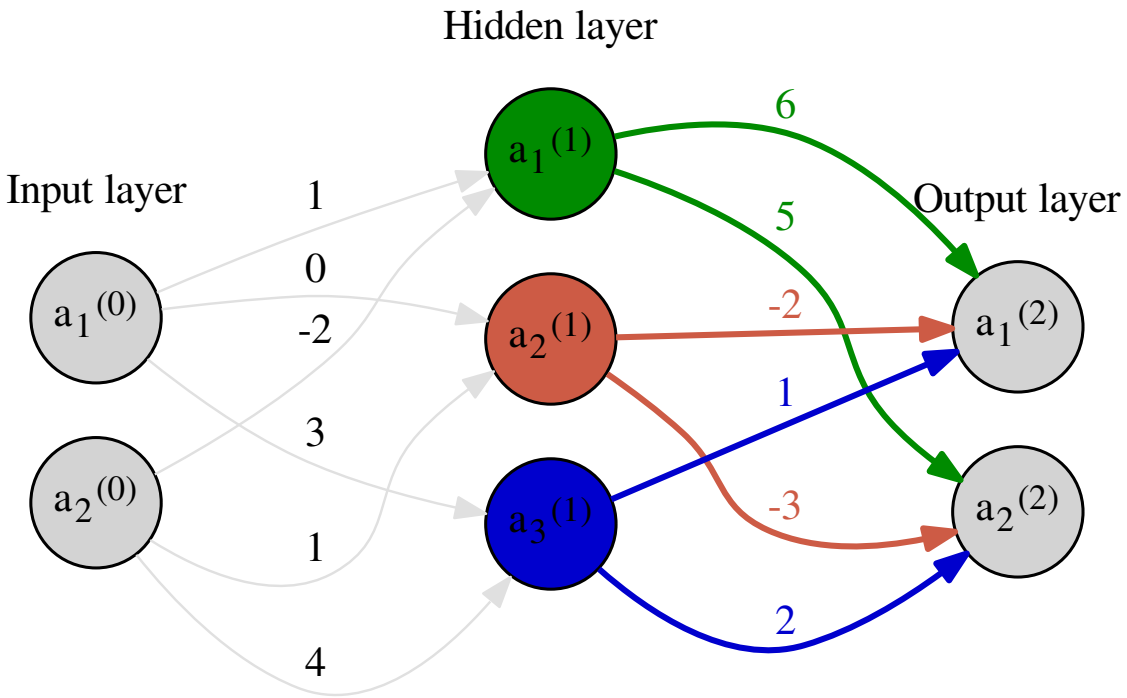
$$\mathbf{W}^{(2)} \leftarrow \begin{bmatrix} 6 & -2 & 1 \\ 5 & -3 & 2 \end{bmatrix} - \eta \begin{bmatrix} \delta_1^{(2)} \\ \delta_2^{(2)} \end{bmatrix} \begin{bmatrix} a_1^{(1)} a_2^{(1)} a_3^{(1)} \end{bmatrix}$$

Ejemplo



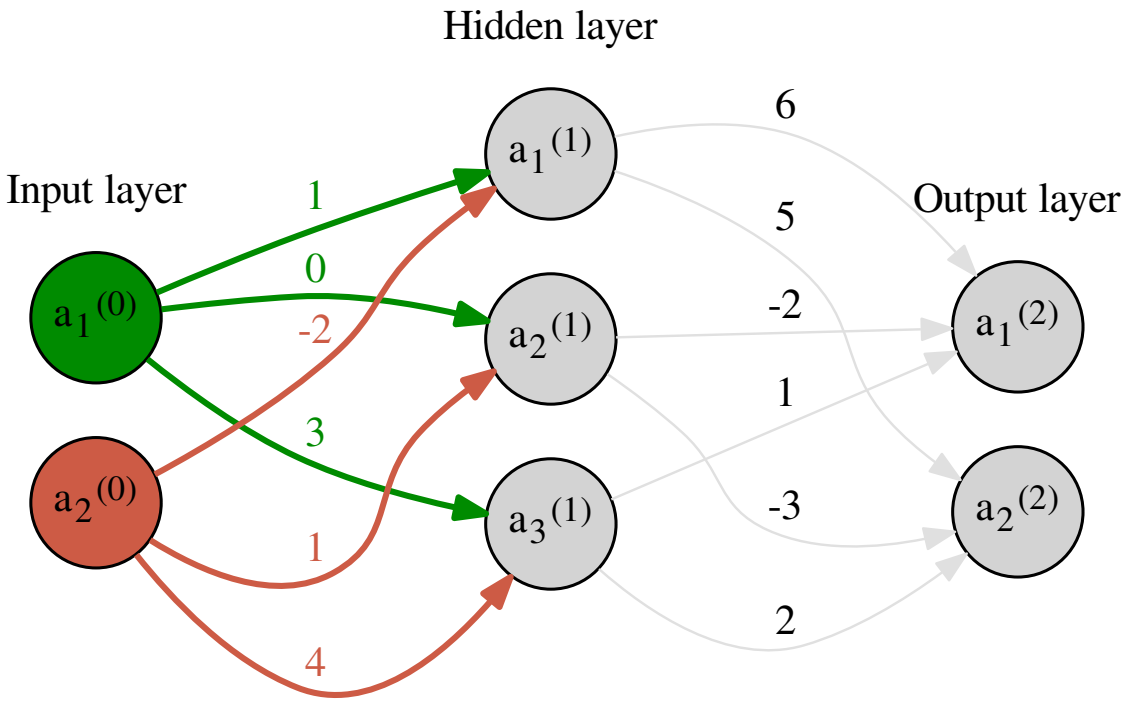
$$\boldsymbol{\delta}^{(1)} = \begin{bmatrix} \delta_1^{(1)} \\ \delta_2^{(1)} \\ \delta_3^{(1)} \end{bmatrix} = \left(\begin{bmatrix} 6 & 5 \\ -2 & -3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} \delta_1^{(2)} \\ \delta_2^{(2)} \end{bmatrix} \right) \odot \begin{bmatrix} f'(z_1^{(1)}) \\ f'(z_2^{(1)}) \\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

Ejemplo



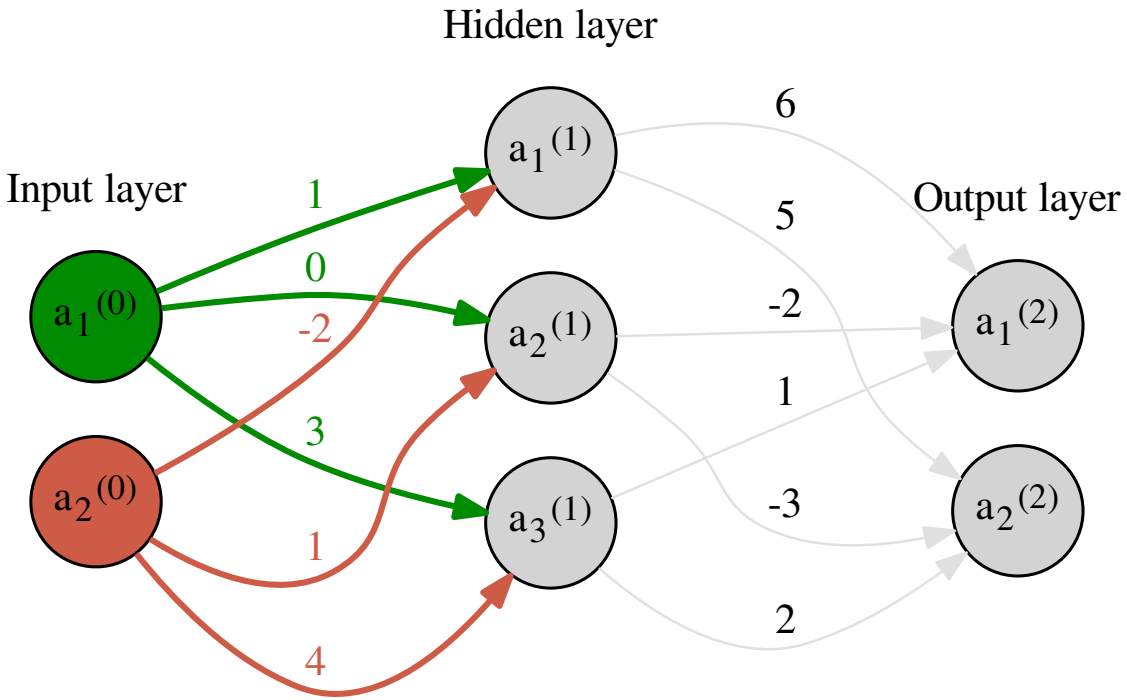
$$\mathbf{b}^{(1)} \leftarrow \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix} - \eta \begin{bmatrix} \delta_1^{(1)} \\ \delta_2^{(1)} \\ \delta_3^{(1)} \end{bmatrix}$$

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

Cálculo de los gradientes

```
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
```

(red_simple_para_clase.ipynb)

Código

Actualización de los pesos

```
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
```

(red_simple_para_clase.ipynb)

Ejercicio

- Estudiar el código red_simple_para_clase.ipynb
- Completar el código del método forward
- Completar el código del método backward
- Probarlo con algún problema de regresión

3. Entrenamiento de la red

Gradiente de la función de coste

- Notación: $\mathbf{x} = \mathbf{a}^{(0)}$ es la entrada a la red, $\mathbf{y} = \mathbf{y}(\mathbf{x}) = \mathbf{a}^{(K)}$ es la salida de la red, $\mathbf{t}$ es la salida esperada.
- La función de coste $C(\mathbf{y}, \mathbf{t})$ mide la discrepancia entre $\mathbf{y}$ y $\mathbf{t}$.
- Idealmente nos gustaría minimizar el valor esperado de la función de coste sobre todas las observaciones posibles $(\mathbf{x}, \mathbf{y})$:

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

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

- En la práctica debemos estimar el gradiente ∇J a partir de un conjunto finito de datos (**conjunto de entrenamiento**):

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

Descenso por gradiente (*batch*)

- Así, en cada paso del algoritmo de descenso por gradiente podemos actualizar los pesos y los bias de acuerdo a:

Descenso por gradiente batch (cada paso):

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

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

- $\mathbf{a}_{\mathbf{x}_i}^{(k)}$ es la activación de la red en la capa k para el ejemplo $\mathbf{x}_i$.
- $\boldsymbol{\delta}_{\mathbf{x}_i}^{(k)}$ es el error en la capa k para el ejemplo $\mathbf{x}_i$.
- n es el número de ejemplos de entrenamiento.

Descenso por *gradiente estocástico*

- Si el número n de ejemplos de entrenamiento es muy grande, el cálculo del gradiente puede llevar mucho tiempo
- En estos casos se estima el gradiente usando un subconjunto aleatorio de ejemplos (**mini-batch**) de tamaño $m < n$

Descenso por *gradiente estocástico* (cada paso):

1. Elegir mini-batch aleatorio de tamaño m: $\mathcal{B} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m\}$
2. Actualizar pesos y bias de acuerdo a:

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

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

Entrenamiento *online*

- El caso límite en que $m = 1$ se denomina entrenamiento **online**.
- Tiene sentido cuando los ejemplos de entrenamiento van llegando sobre la marcha.
- Sin embargo introduce más ruido, pues el gradiente se estima a partir de un único ejemplo cada vez.

Ejercicios

Tomando como punto de partida el notebook keras.ipynb, prueba a realizar distintas ejecuciones modificando el factor de aprendizaje y el tamaño del batch

- ¿Qué ocurre con factores de aprendizaje muy altos?
- ¿Qué ocurre con factores de aprendizaje muy bajos?
- ¿Qué ocurre con batches muy grandes?
- ¿Qué ocurre con batches muy pequeños?

4. Función de coste

Error cuadrático

- **Error cuadrático**:

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

- El gradiente respecto a la activación en la capa de salida es:

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

- Y por tanto el error en la última capa se puede expresar como:

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

Cross entropy

- Función de coste **cross-entropy**:

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

  La suma es sobre las unidades de salida (componentes de $\mathbf{y}$).

- Gradiente respecto a la activación de la capa de salida:

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

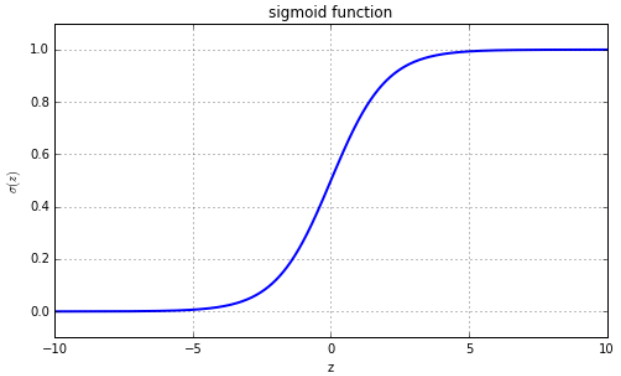
- Error en la capa de salida (se omite el superíndice K en $z_j^{(K)}$ para simplificar la notación):

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

Función de activación sigmoide

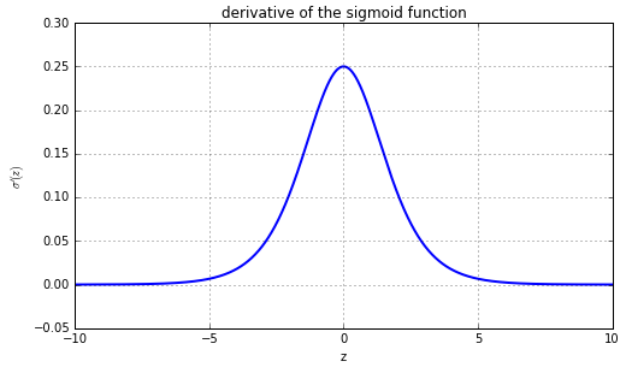
- Si la función de activación en la capa de salida es una sigmoide:

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



- Su derivada es:

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



- La derivada se **satura**.

Función de activación sigmoide

Y las deltas quedan así:

Error cuadrático:

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

Cross-entropy:

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

Ejercicio

- Leer las primeras páginas del capítulo 3 del libro de M. Nielsen, http://neuralnetworksanddeeplearning.com/chap3.html
- Probar las dos animaciones para la función de coste cuadrática
- Probar las dos animaciones para la función de coste cross-entropy
- Interpretar las observaciones en base a los resultados anteriores

Cross-entropy

La función de coste **cross-entropy** es la mejor elección cuando las unidades de salida son **sigmoides**

Máxima verosimilitud

- La salida de la red se interpreta como una probabilidad.
- Se utiliza como función de coste:

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

- Sumando a todos los ejemplos de entrenamiento:

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

- Minimizar J es equivalente a minimizar la distancia KL entre la distribución empírica de los datos y la obtenida por el modelo.

Máxima verosimilitud

- Cuando la salida de la red es una única unidad sigmoide, máxima verosimilitud es equivalente a cross-entropy.
- Suponemos dos clases, $t \in \{0, 1\}$, e interpretamos la salida de la red como la probabilidad de clase 1:

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

- Entonces:

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

- Que se puede escribir como:

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

Hinge loss (SVM)

- La capa de salida es lineal (sin función de activación):

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

- Función de coste **hinge loss**:

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

- El vector de clase $\mathbf{t}$ se supone de la forma:

$$t_j = \delta_{jo}$$

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

Binary cross-entropy:

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

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

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

- ¿Qué aspecto tiene el gradiente de la función de coste hinge loss?
- ¿Cómo se comportará hinge loss como función de coste de una red neuronal? ¿Habrá problemas de saturación?

5. Función de activación

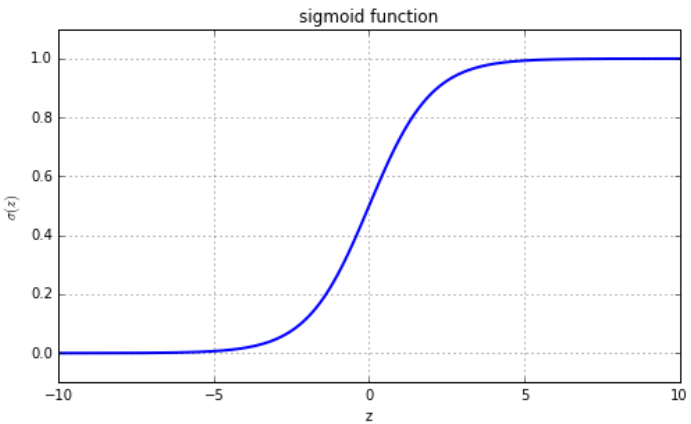
Capa de salida

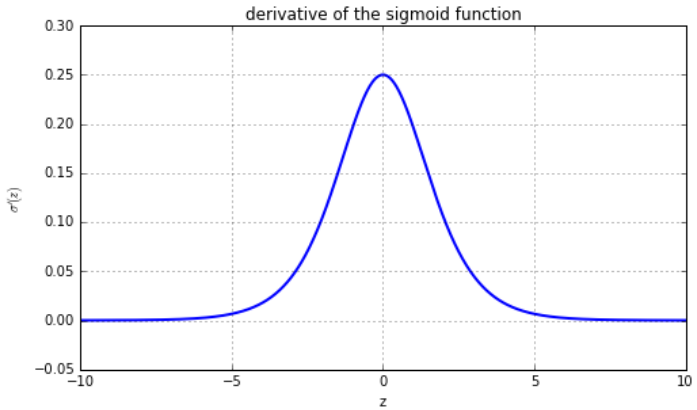
- La **función de activación** en la última capa la elegimos en función del tipo de salida esperado para la red.
- Además debemos tener en cuenta la **función de coste** que vamos a utilizar.
- En general supondremos que se usa **máxima verosimilitud**.

Salida binaria

- Dos clases, $t \in \{0, 1\}$, y una única unidad de salida.
- Usamos una función de activación **sigmoide**:

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





- Como vimos, en este caso máxima verosimilitud produce la función de coste **cross-entropy**.

Problema multiclase

- Número arbitrario n de clases, usamos n unidades de salida, y la función de activación **softmax**.
- Sea $\mathbf{h}$ la salida de la última capa oculta, y $\mathbf{b}$ y $\mathbf{W}$ el vector de bias y la matriz de pesos para la capa de salida.
- La entrada $\mathbf{z}$ a la última capa es:

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

- Y la salida de la red viene dada por:

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

- Nótese que las salidas de todas las unidades de la última capa están "acopladas".

Función softmax

- Las salidas de la red son todas positivas y suman 1.
- Las interpretamos como probabilidades:

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

- Aplicando máxima verosimilitud, obtenemos una función de coste que genera como delta para la capa de salida:

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

  donde el vector $\mathbf{t}$ tiene un 1 en la posición correspondiente a la clase del punto.
- De este modo **softmax** es una generalización de la **sigmoide**.

Salida gausiana (regresión lineal)

- Usamos una unidad de salida **lineal** cuando queremos generar una salida que represente la media de una distribución gausiana condicionada a la entrada:

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

- La función de activación es la identidad:

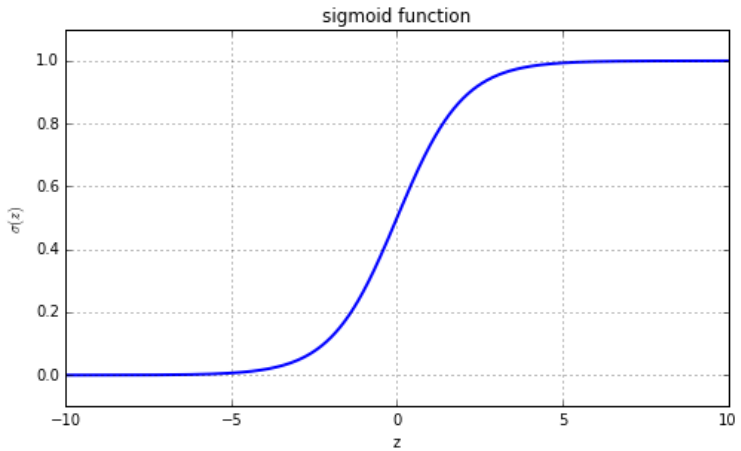
$$\mathbf{y} = \mathbf{W}\mathbf{h} + \mathbf{b}$$

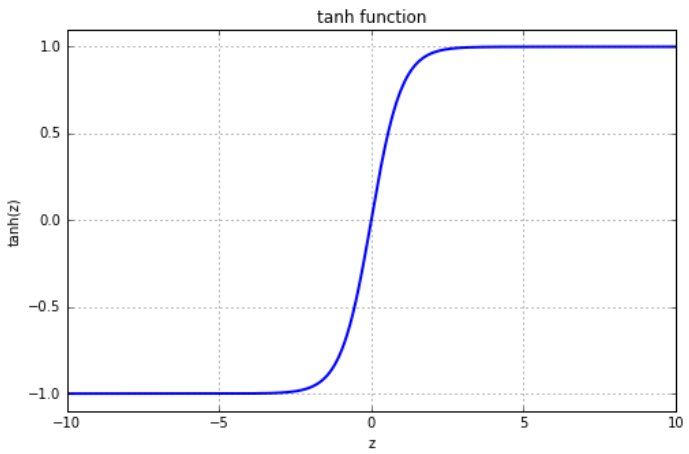
- En este caso máxima verosimilitud produce como función de coste el **error cuadrático**.
- Las unidades lineales no se saturan

Activación en las capas ocultas

- Funciones **sigmoide** y **tanh**:

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



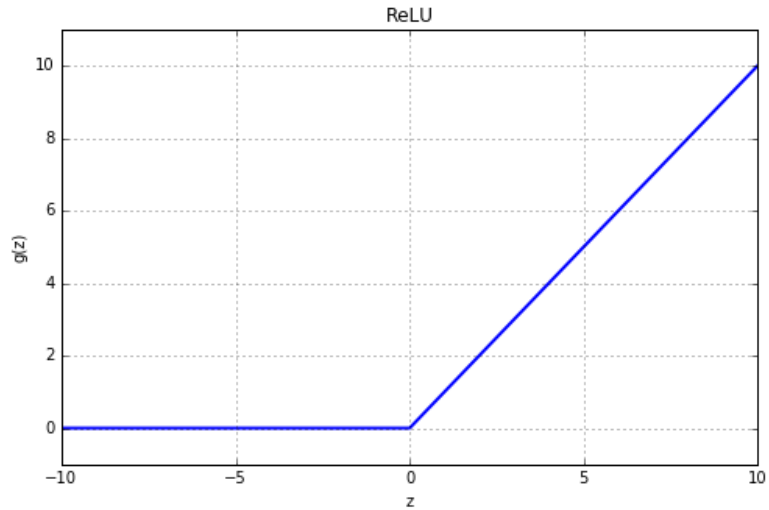


- Desaconsejadas porque se saturan, anulando el gradiente y dificultando el aprendizaje.

Rectificador lineal (ReLU)

- Las unidades de tipo **ReLU** (rectifier linear unit) son las más utilizadas en la actualidad:

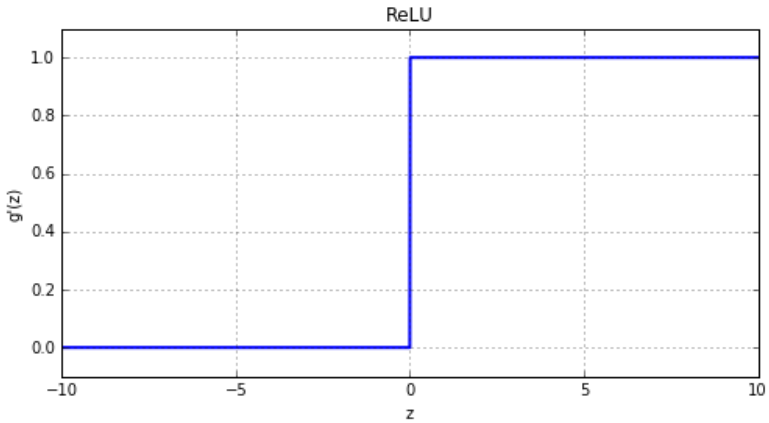
$$g(z) = \max(0, z)$$



Rectificador lineal (ReLU)

- ReLU no es diferenciable en $z = 0$.
- Desde el punto de vista práctico, en $z = 0$ se suele tomar la derivada por la izquierda, es decir:

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



- La derivada se anula cuando la neurona está inactiva. Sólo los ejemplos que activan la neurona permiten modificar los pesos.

Otras funciones de activación

- **Leaky ReLU**

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

- **Maxout**

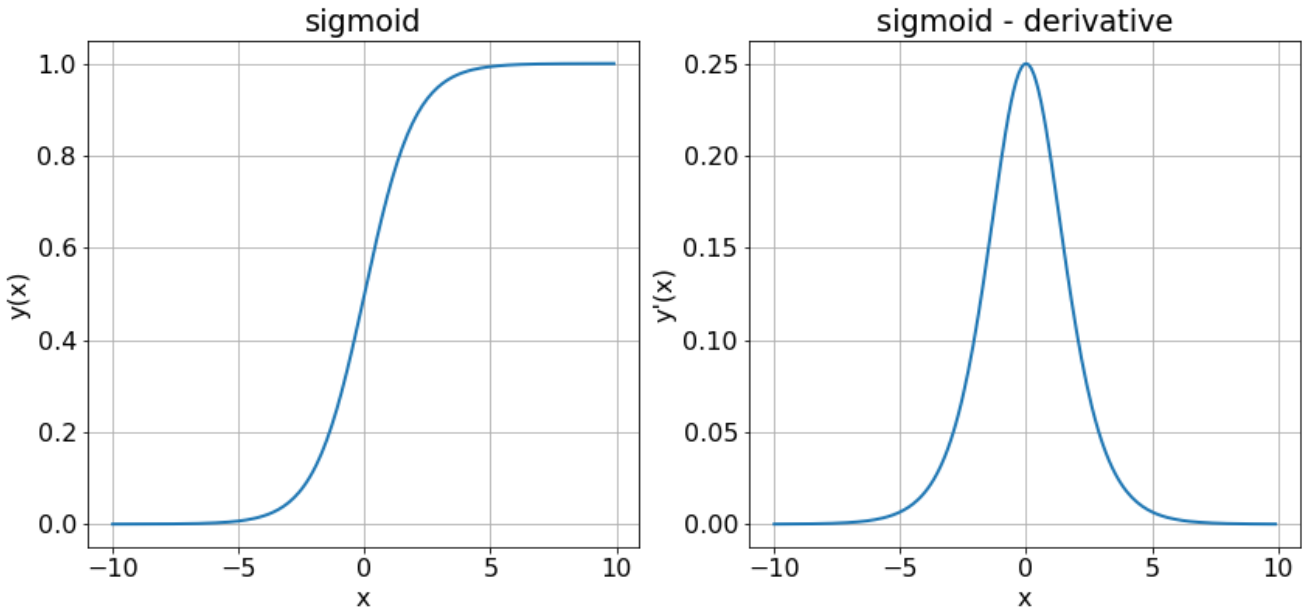
$$f(\mathbf{x}) = \max\left(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1, \mathbf{W}_2\mathbf{x} + \mathbf{b}_2\right)$$

Algunas funciones de activación disponibles en Keras

sigmoid:

```
tf.keras.activations.sigmoid(x)
```

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

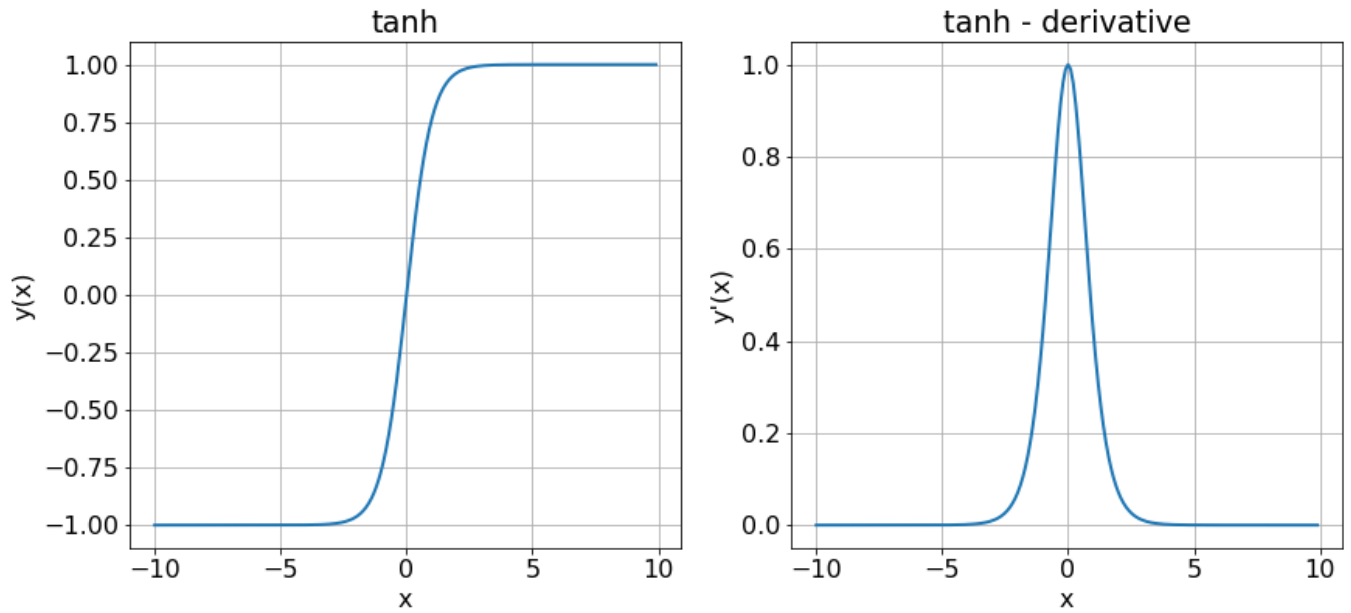


Algunas funciones de activación disponibles en Keras

tanh:

```
tf.keras.activations.tanh(x)
```

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

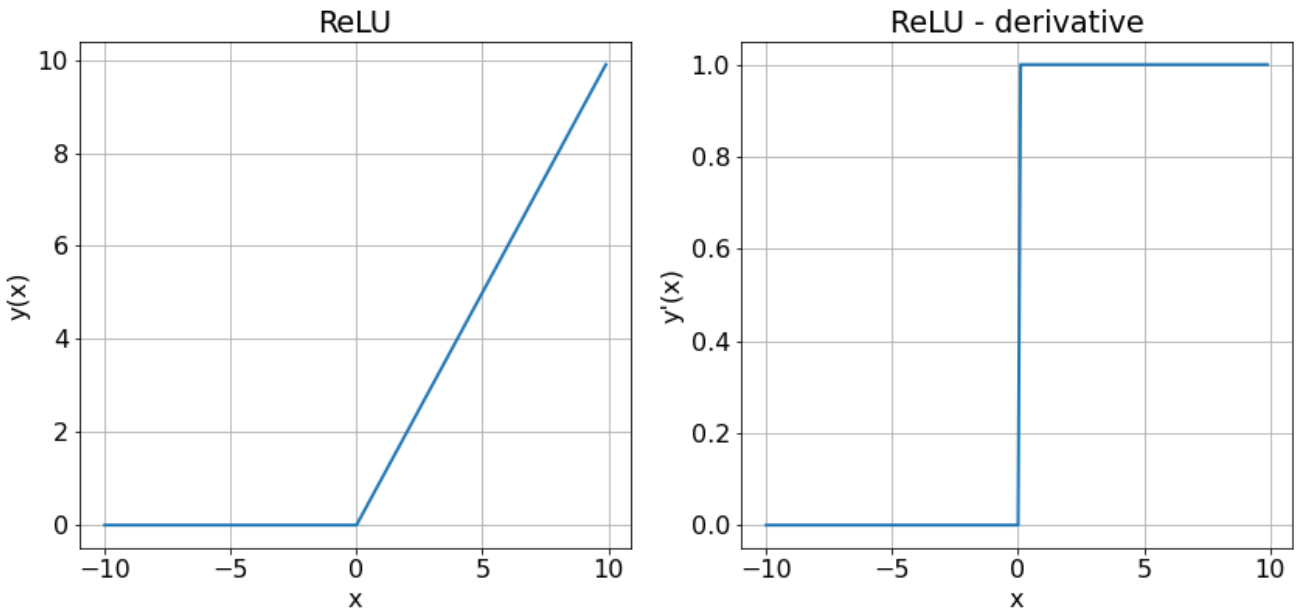


Algunas funciones de activación disponibles en Keras

ReLU:

```
tf.keras.activations.relu(x, alpha=0.0, max_value=None, threshold=0)
```

$$f(x) = max(0, x)$$

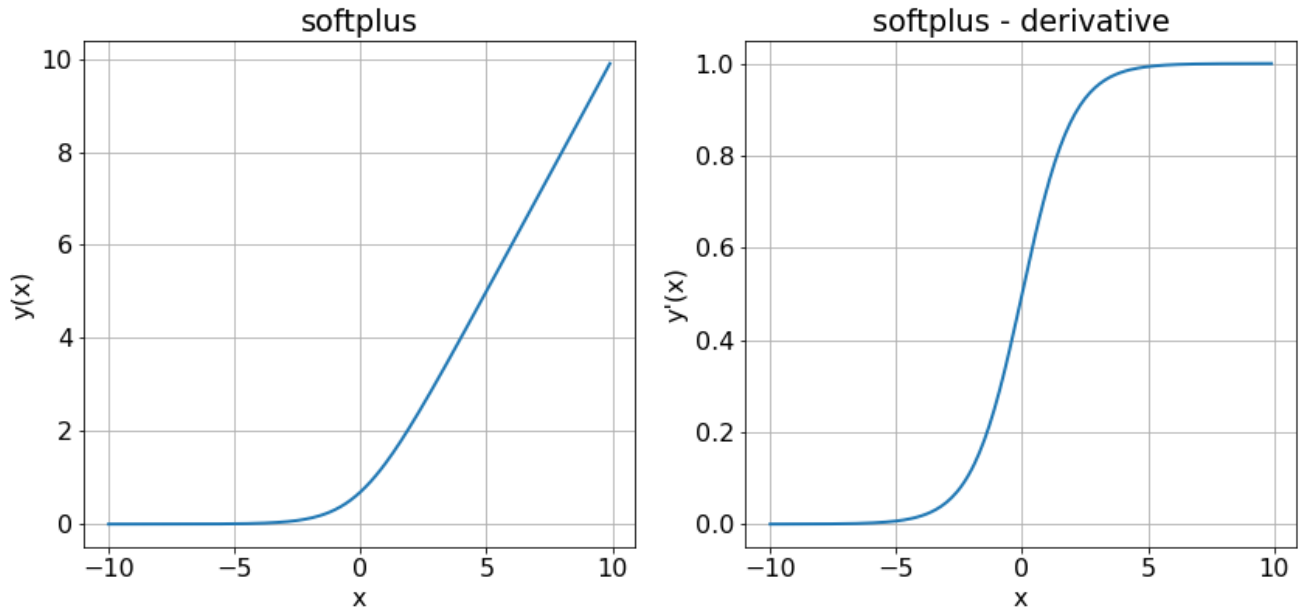


Algunas funciones de activación disponibles en Keras

softplus:

```
tf.keras.activations.softplus(x)
```

$$f(x) = \log(\exp(x) + 1)$$

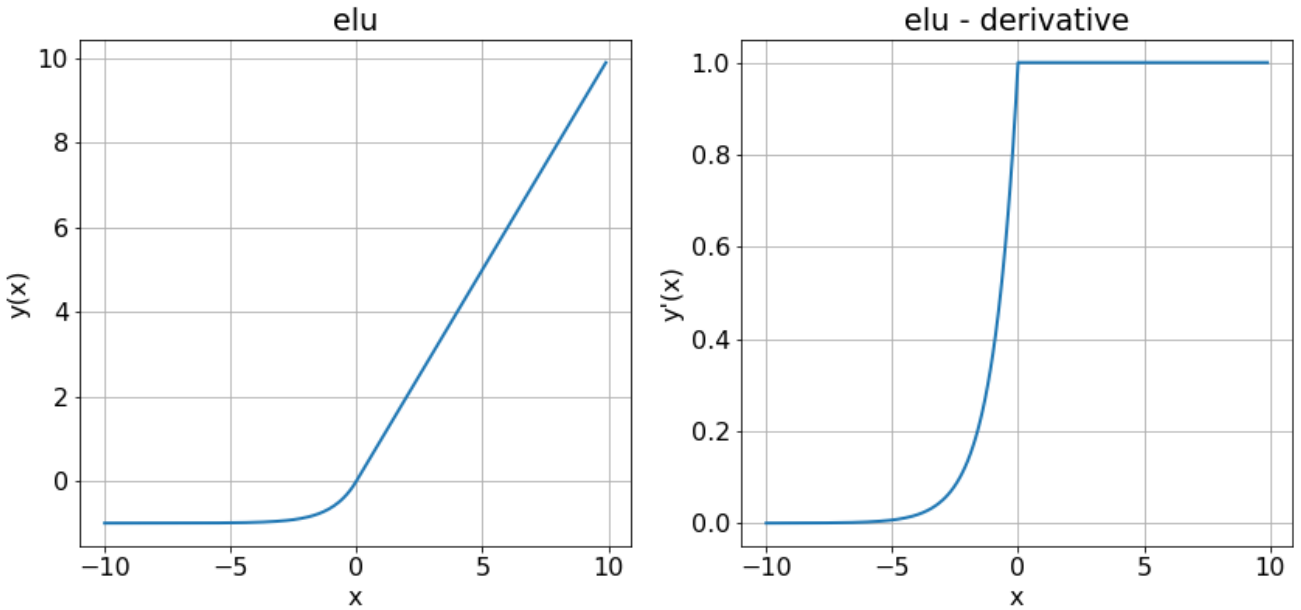


Algunas funciones de activación disponibles en Keras

ELU:

```
tf.keras.activations.elu(x, alpha=1.0)
```

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(\exp(x) - 1), & \text{if } x < 0 \end{cases}$$

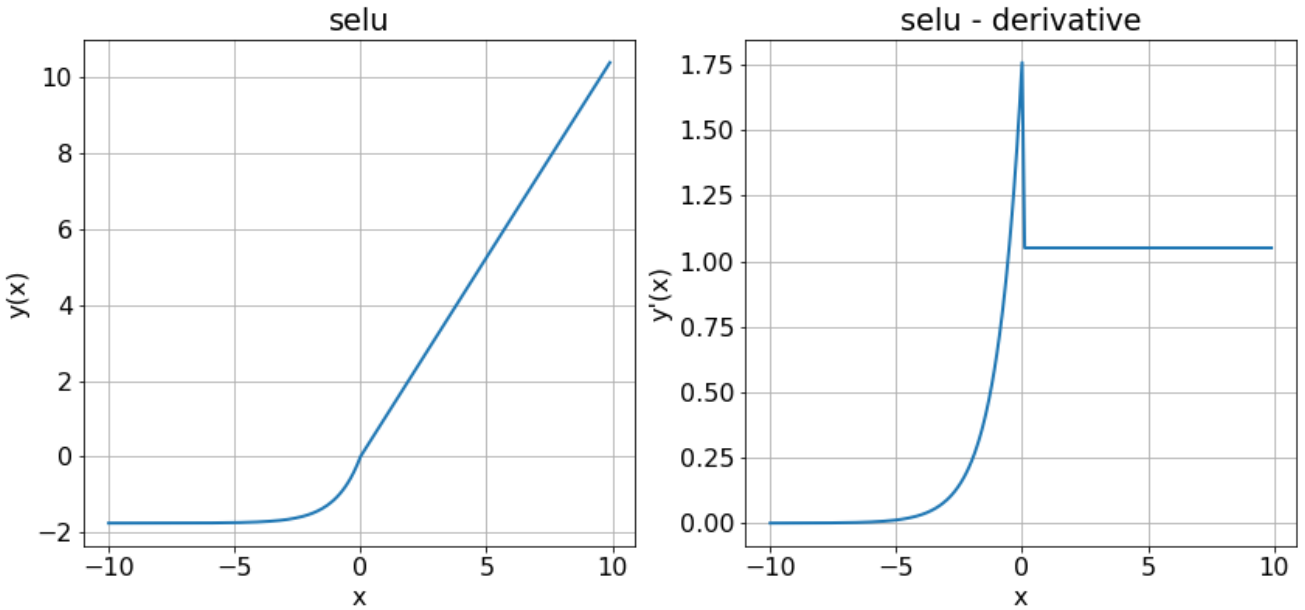


Algunas funciones de activación disponibles en Keras

SELU:

```
tf.keras.activations.selu(x)
```

$$f(x) = \begin{cases} sx, & \text{if } x > 0 \\ s\alpha(\exp(x) - 1), & \text{if } x < 0 \end{cases}$$



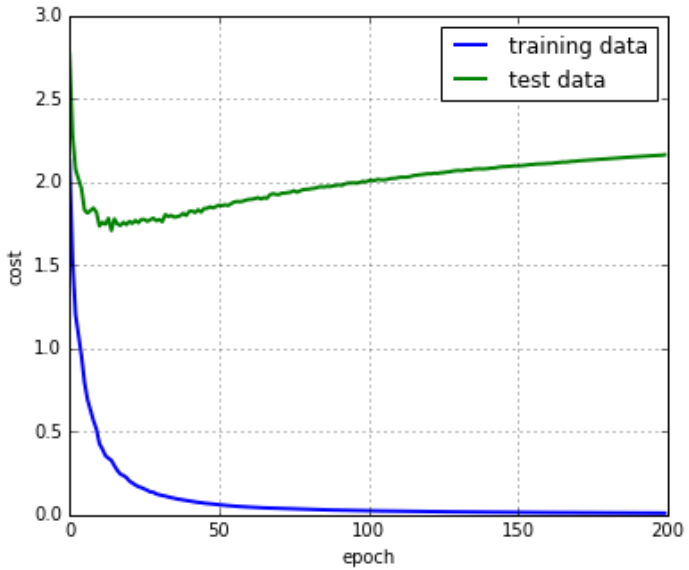
Ejercicio

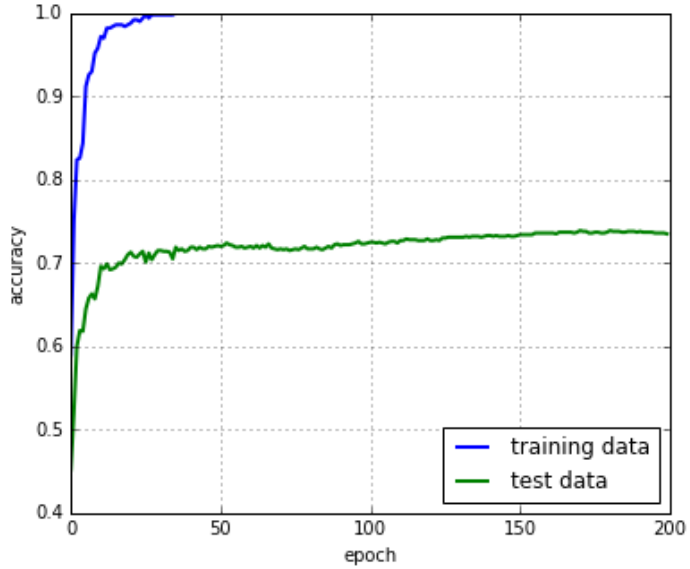
- Probar, en el notebook keras.ipynb, la función de coste ReLU

6. Regularización

Overfitting

- ¿Cómo funciona el modelo con un conjunto de **test**?





- **Overfitting**: A partir de un determinado número de épocas el modelo se adapta demasiado a los datos de entrenamiento y empieza a generalizar mal.

Overfitting

Cómo evitarlo:

1. Usar **más datos**.
2. Usar un conjunto de validación, **early stopping**.
3. Aplicar **regularización**.

Regularización

- En un sentido amplio, cualquier modificación hecha a un algoritmo de aprendizaje cuyo objetivo es **reducir el error de generalización** y no el de entrenamiento.
- En un sentido estricto, introducción de **términos adicionales** en la función de coste que **penalizan la complejidad** del modelo favoreciendo su capacidad de generalización.

Regularización L^2

- Se modifica la función de coste de la siguiente manera:

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

  Donde:
  - J es la función de coste no regularizada (error).
  - n es el número de ejemplos de entrenamiento.
  - La suma es sobre todos los pesos de la red.
  - $\lambda > 0$ es el **parámetro de regularización**.
- La regularización L^2 también se conoce como **weight decay**

Regularización L^2

Retropropagación con regularización L^2

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

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

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

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

- Las deltas se calculan como antes.
- Los bias se actualizan como antes.
- Los pesos se reescalan por un factor $1 - 2\eta\lambda$ antes de hacer el descenso por gradiente (**weight decay**).

Regularización L^1

- Se modifica la función de coste de la siguiente manera:

$$J_{L^1} = J + \lambda \sum_w |w|$$

- También provoca un decaimiento de los pesos, pero de manera constante que no depende de la magnitud de los mismos.

Retropropagación con regularización L^1

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

Regularización L^1 y L^2 en Keras

Al definir una capa podemos indicar:

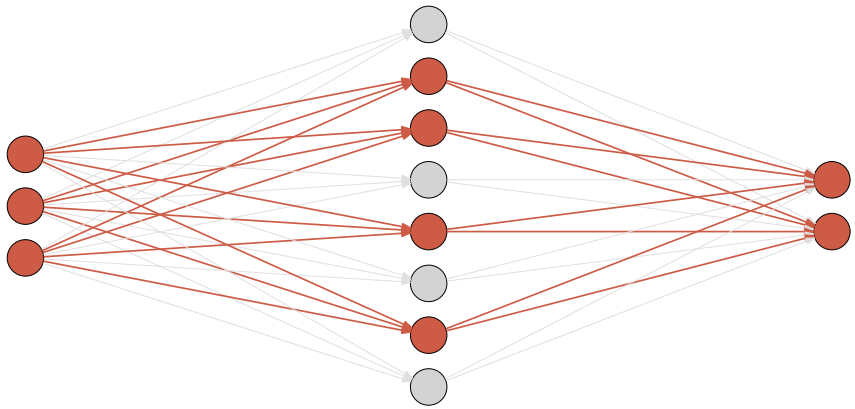
- `kernel_regularizer`: Regularizador para los pesos
- `bias_regularizer`: Regularizador para los bias
- `activity_regularizer`: Regularizador para las activaciones

Ejemplo: regularización L^2 en una capa densa

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

Dropout

- Con cada mini-batch, se eliminan al azar algunas de las unidades ocultas (típicamente la mitad).



- Tanto la propagación de la actividad (feedforward) como la retropropagación del error se hacen usando sólo las unidades que no se han eliminado.
- Por tanto, con cada mini-batch se aprenden sólo algunos de los pesos de la red.

Dropout

- Finalmente, para clasificar se utiliza la red completa, multiplicando los pesos que salen de una determinada unidad por la probabilidad de usar esa unidad durante el entrenamiento (típicamente $1/2$).
- De algún modo, estamos construyendo un **ensemble** de redes y promediando las opiniones de cada una de ellas.
- Al hacer dropout forzamos a la red a que sea **robusta** frente a la pérdida de unidades.

- *Dropout: A Simple Way to Prevent Neural Networks from Overfitting:* `http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf`

Dropout

Vanilla Dropout, entrenamiento

```
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)
```

(de: http://cs231n.github.io/neural-networks-2/)

Dropout

Vanilla Dropout, predicción

```
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
```

(de: http://cs231n.github.io/neural-networks-2/)

Inverted Dropout

Inverted Dropout, entrenamiento

```
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)
```

(de: http://cs231n.github.io/neural-networks-2/)

Inverted Dropout

Inverted Dropout, predicción

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

(de: http://cs231n.github.io/neural-networks-2/)

Dropout en Keras

Se implementa como una capa más:

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

Ejemplo: modelo con dropout en la capa oculta

```
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")
```

Ejercicio

- Probar, en el notebook keras.ipynb, la regularización L^1 y la regularización L^2.
- Probar a meter una capa de dropout justo después de la capa oculta.

7. Inicialización de los pesos

Inicialización de los pesos

¿Por qué no es una buena idea inicializar los pesos a 0?

- Mejor inicializar aleatoriamente, con valores pequeños centrados alrededor de 0:

```
# Si W es una matriz DxH:
W = 0.01* np.random.randn(D,H)
```

- Problema: ¿Cómo elegimos la escala (varianza) de la distribución?

Inicialización de los pesos

- El objetivo debe ser evitar que las neuronas estén saturadas al comenzar el entrenamiento
- Neurona **sigmoide** con n pesos entrantes, w_1, w_2, ..., w_n y bias b:
  - Inicializamos $w_i \sim N(0, 1/\sqrt{n})$
  - Inicializamos $b \sim N(0, 1)$
  - También es frecuente inicializar los bías a 0

Inicialización Xavier

- *Understanding the difficulty of training deep feedforward neural networks:*

  http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

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

- n_{in} es el número de unidades en la capa anterior
- n_{out} es el número de unidades en la capa siguiente

Inicialización para unidades ReLU

- La recomendación actual para inicializar los pesos de una neurona ReLU es:

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

- *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification:* https://arxiv.org/abs/1502.01852
- Los bías b se inicializan a un valor positivo pequeño (típicamente $0{,}1$ o $0{,}01$) para sesgar las unidades hacia la parte positiva y conseguir que inicialmente estén activas

Keras Initializers

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

Batch Normalization

- *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift:*
  https://arxiv.org/abs/1502.03167
- Técnica reciente (2015) que simplifica la tarea de inicializar los pesos
- La idea es normalizar la activación de cada unidad (antes de la no linealidad)
- Es posible porque la normalización es diferenciable
- Con batch normalization la red no es tan sensible a una mala inicialización de los pesos

Keras Batch Normalization Layer

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

Ejercicio

- Probar, en el notebook keras.ipynb, diferentes formas de inicializar los pesos de la red.
- Probar a utilizar batch normalization.

8. Otras técnicas de optimización

Métodos de segundo orden

- Método de **Newton**:

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

- $\mathbf{H}$ es la matriz hessiana:

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

- Convergencia más rápida que con descenso por gradiente.
- Difícil de usar en la práctica para una red grande (muy costoso invertir $\mathbf{H}$).

Gradiente con momento

- Descenso por gradiente con **momento**:

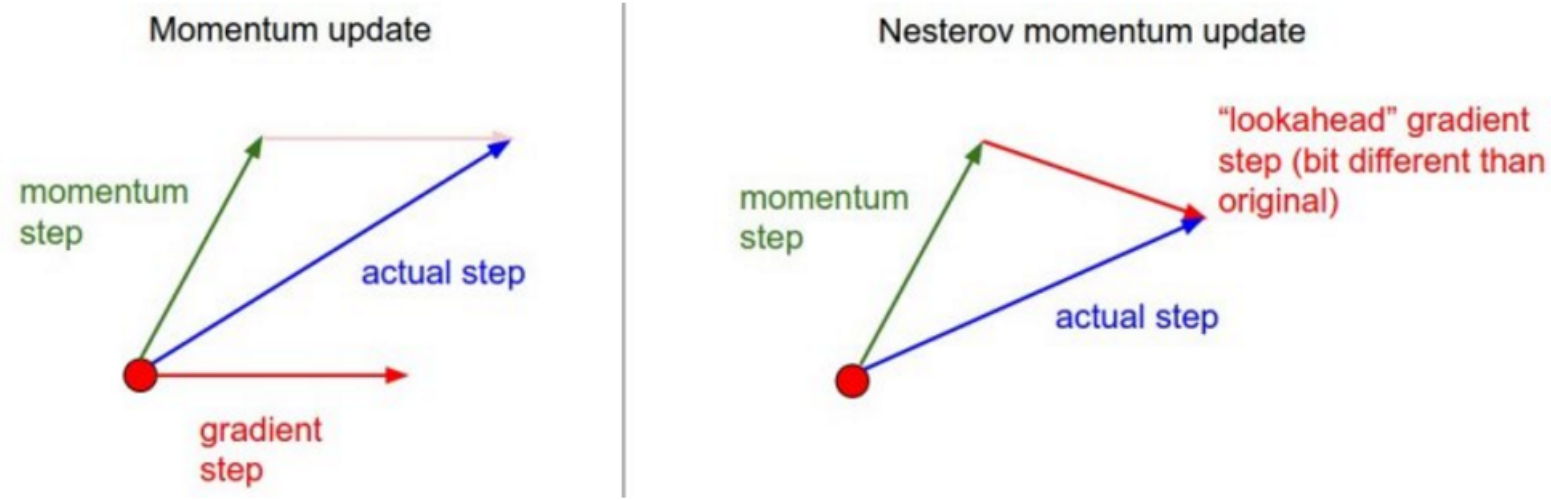
$$\begin{aligned} \mathbf{v} &\leftarrow \mu\mathbf{v} - \eta\nabla J \\ \mathbf{w} &\leftarrow \mathbf{w} + \mathbf{v} \end{aligned}$$

- El parámetro μ es el **coeficiente de momento**, $0 < \mu < 1$.

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

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

Momento de Nesterov




```
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
```

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

Gradiente con momento - 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

AdaGrad

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

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

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

AdaGrad - 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

RMSProp

- *Neural Networks for Machine Learning, Lecture 6a*, G. Hinton (2012). http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

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

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

RMSProp - 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

Adam

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

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

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

Adam - 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

Otros optimizadores disponibles

- Adadelta
- Adamax
- Nadam
- Ftrl

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

Ejercicio

- Con el código keras.ipynb, probar los siguientes métodos de optimización:
  - Descenso por gradiente con momento
  - Momento de Nesterov
  - AdaGrad
  - RMSProp
  - Adam

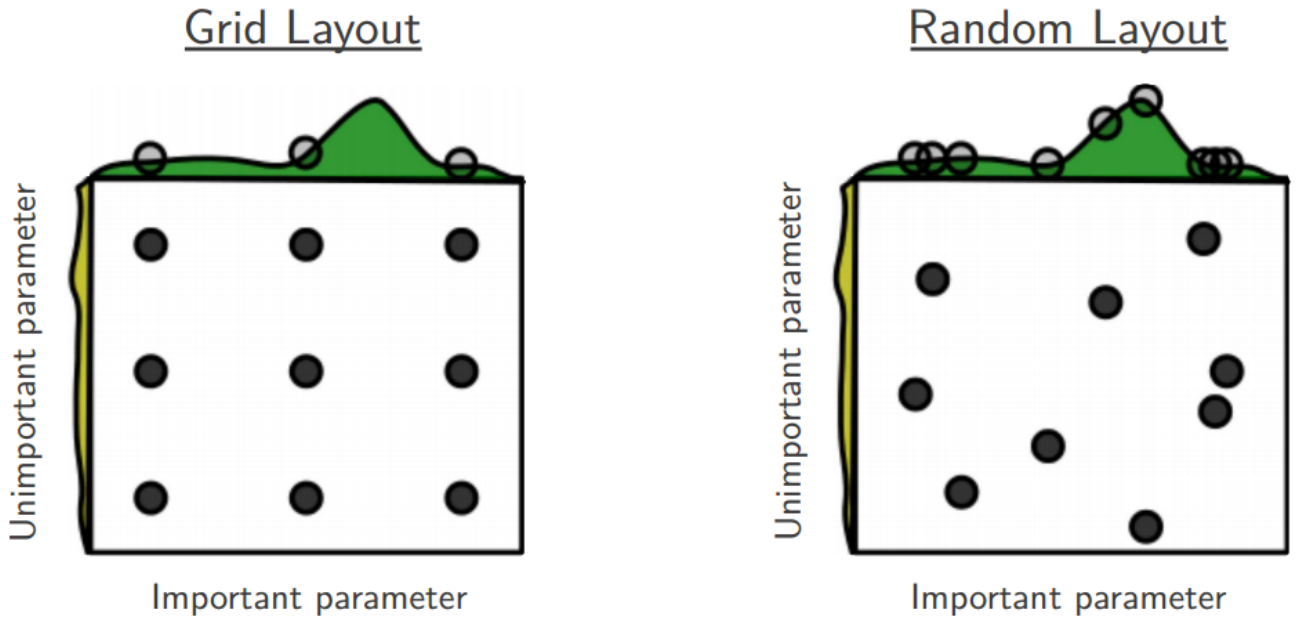
9. Selección de los hiperparámetros

Selección de los hiperparámetros

- Constante de regularización, tamaño del mini-batch, arquitectura de la red (número de capas, número de unidades ocultas, etc.):
  - Típicamente se eligen mediante algún tipo de **validación**.
  - Técnicas automáticas de búsqueda en rejilla (**grid search**), búsqueda aleatoria (**random search**).
- Número de épocas de entrenamiento:
  - Puede ajustarse mediante **early stopping**.
- Factor de aprendizaje:
  - Puede ajustarse monitorizando el coste sobre el conjunto de entrenamiento.

Grid search vs random search

- En general, la búsqueda aleatoria es más eficiente como método de optimización de los hiperparámetros



(de: *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