Fundamentos de las Redes Neuronales

Algoritmo de Retropropagación parte II

Bloques de construcción

Clase 6

Dra. Wendy Aguilar

Modelos Generativos Profundos

UN ENFOQUE DESDE LA CREATIVIDAD COMPUTACIONAL

Perceptrones Multicapa y Retropropagación

Actualizar los pesos $w_{1,1}^{[1]}, w_{2,1}^{[1]}, w_{1,2}^{[1]}$ y $w_{2,2}^{[1]}$

Enviarlos por email a weam@turing.iimas.unam.mx

Perceptrones Multicapa y Retropropagación

Ejemplo

Propagación
Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E?

$$\frac{dE}{dw_{1,1}^{[1]}}$$

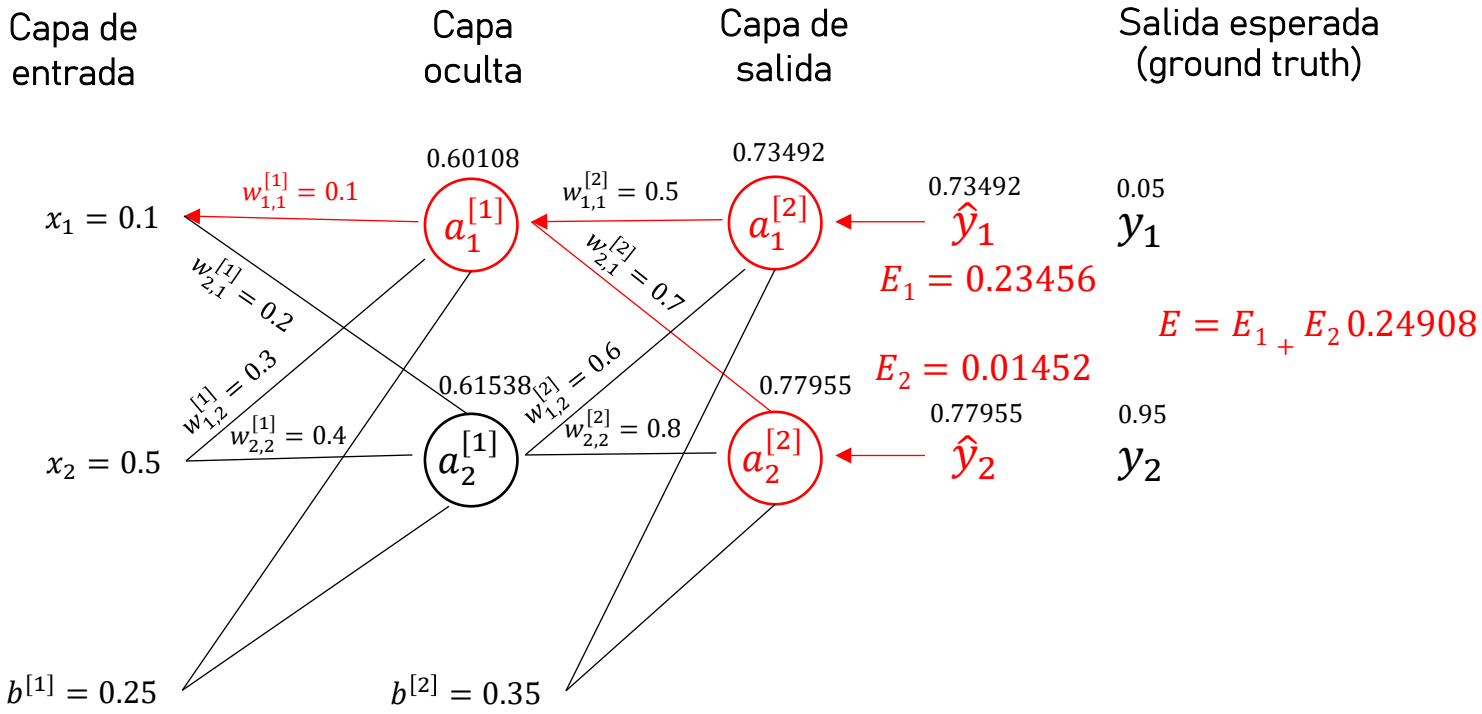
En este caso el peso $w_{1,1}^{[1]}$ afecta tanto a $\hat{y}_1$como a $\hat{y}_2$.

Calculamos por separado:

$$E_1 = \frac{1}{2}(y_1 - \hat{y}_1)^2 = 0.23456$$

$$E_2 = \frac{1}{2}(y_2 - \hat{y}_2)^2 = 0.01452$$

$$\frac{dE}{dw_{1,1}^{[1]}} = \frac{dE_1}{dw_{1,1}^{[1]}} + \frac{dE_2}{dw_{1,1}^{[1]}}$$



Nota: Recuerden que los pesos se actualizan hasta el final del cálculo de todas las derivadas.

Perceptrones Multicapa y Retropropagación

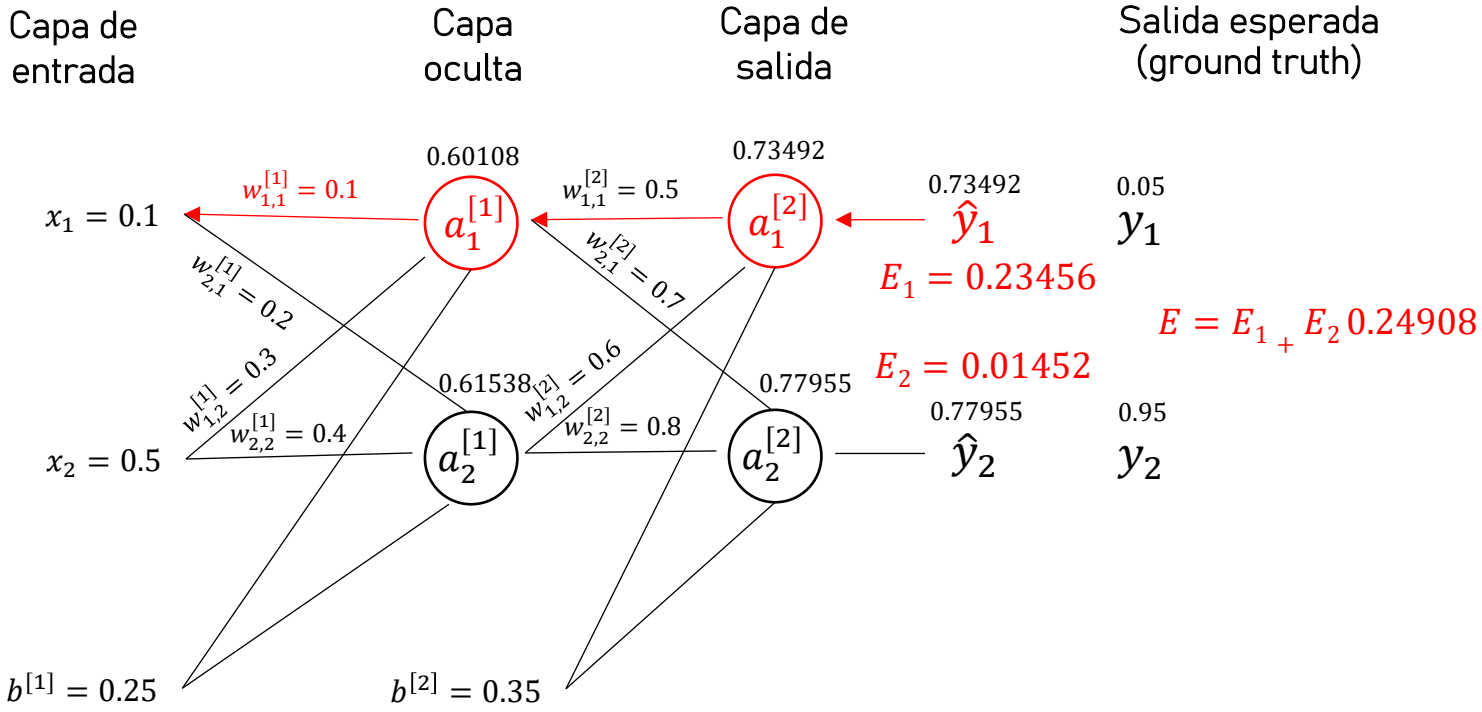
Ejemplo

Propagación

Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_1?

$$\frac{dE1}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_1}\frac{d\hat{y}_1}{dz_1^{[2]}}\frac{dz_1^{[2]}}{da_1^{[1]}}\frac{da_1^{[1]}}{dz_1^{[1]}}\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$



Perceptrones Multicapa y Retropropagación

Ejemplo

Propagación

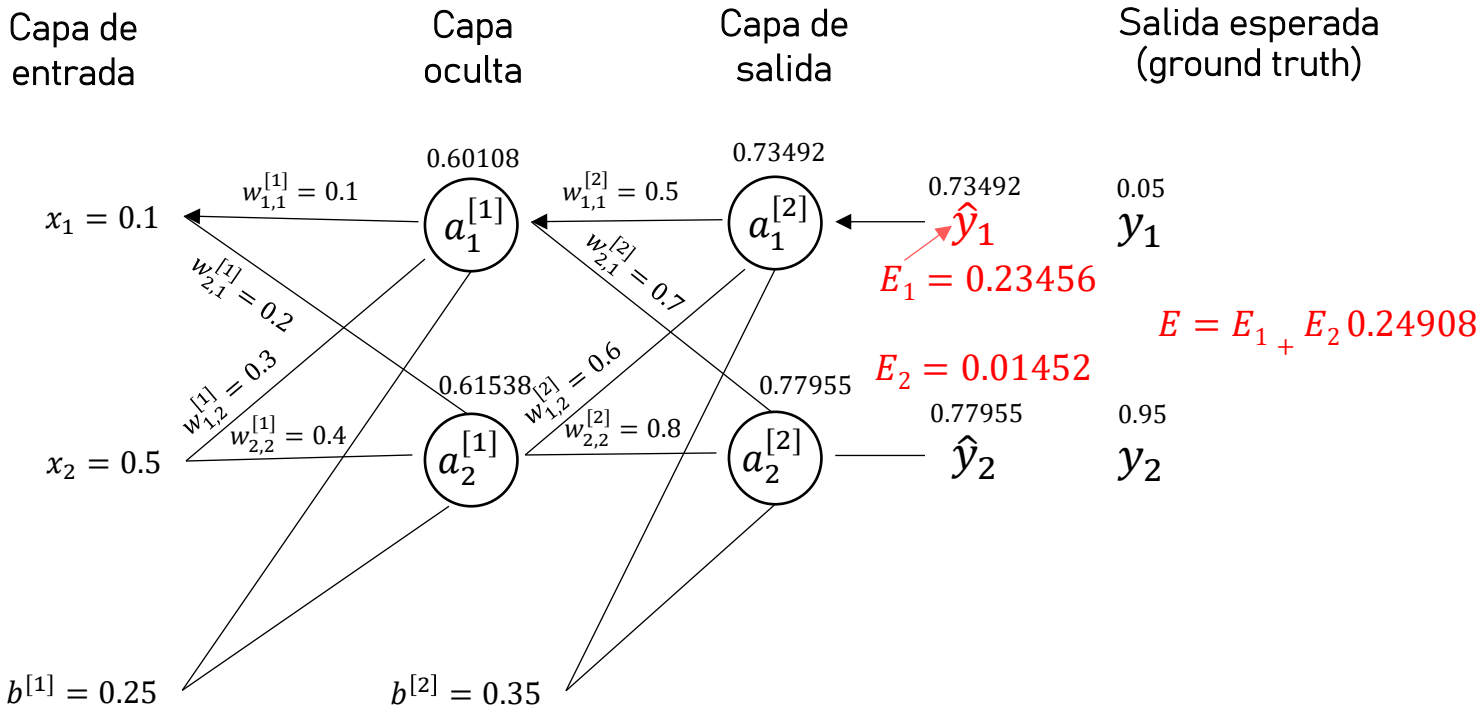
Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_1?

$$\frac{dE1}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_1}\frac{d\hat{y}_1}{dz_1^{[2]}}\frac{dz_1^{[2]}}{da_1^{[1]}}\frac{da_1^{[1]}}{dz_1^{[1]}}\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$

$$E_1 = \frac{1}{2}(y_1 - \hat{y}_1)^2$$

$$\frac{dE_1}{d\hat{y}_1} = \hat{y}_1 - y_1 = 0.73492 - 0.05 = 0.68492$$



Perceptrones Multicapa y Retropropagación

Ejemplo

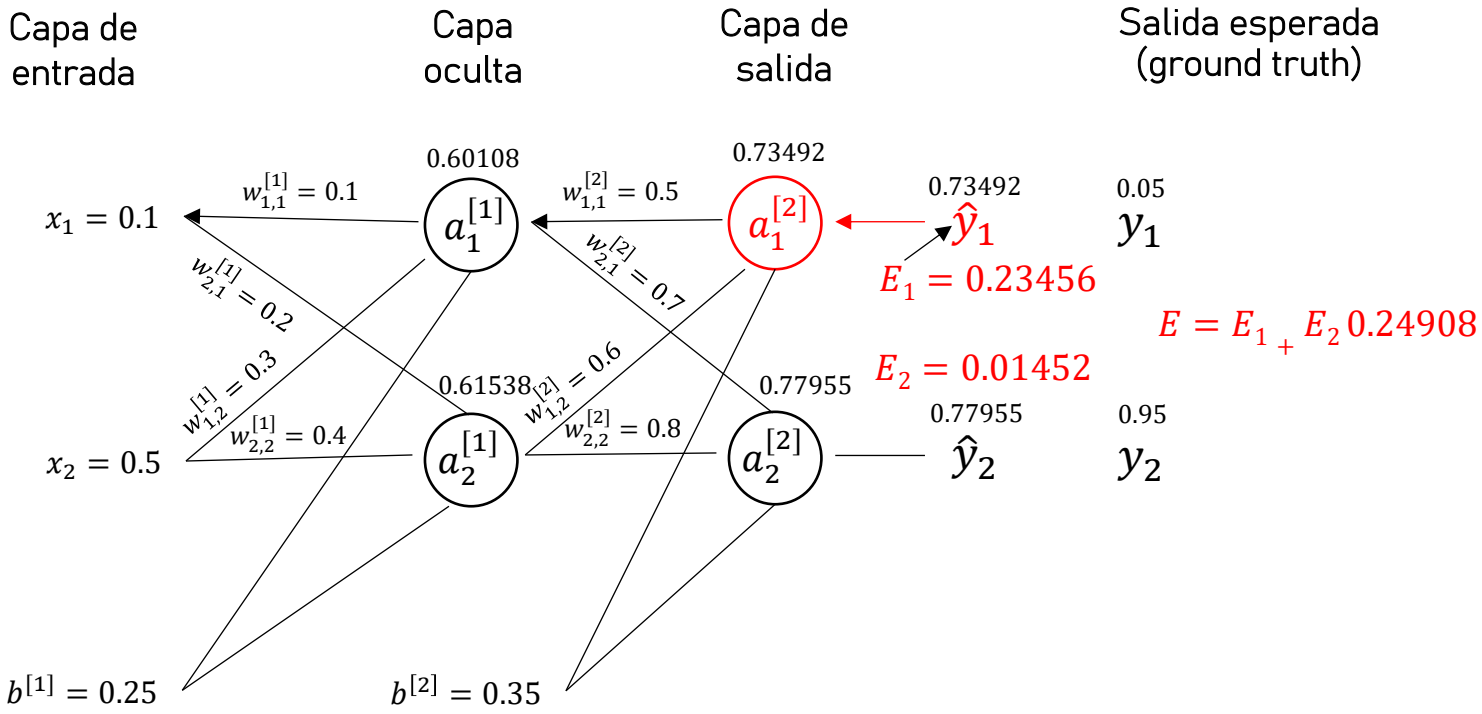
Propagación
Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_1?

$$\frac{dE1}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_1} \frac{d\hat{y}_1}{dz_1^{[2]}} \frac{dz_1^{[2]}}{da_1^{[1]}} \frac{da_1^{[1]}}{dz_1^{[1]}} \frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$

Ya lo habíamos calculado.

$$0.19480$$



Perceptrones Multicapa y Retropropagación

Ejemplo

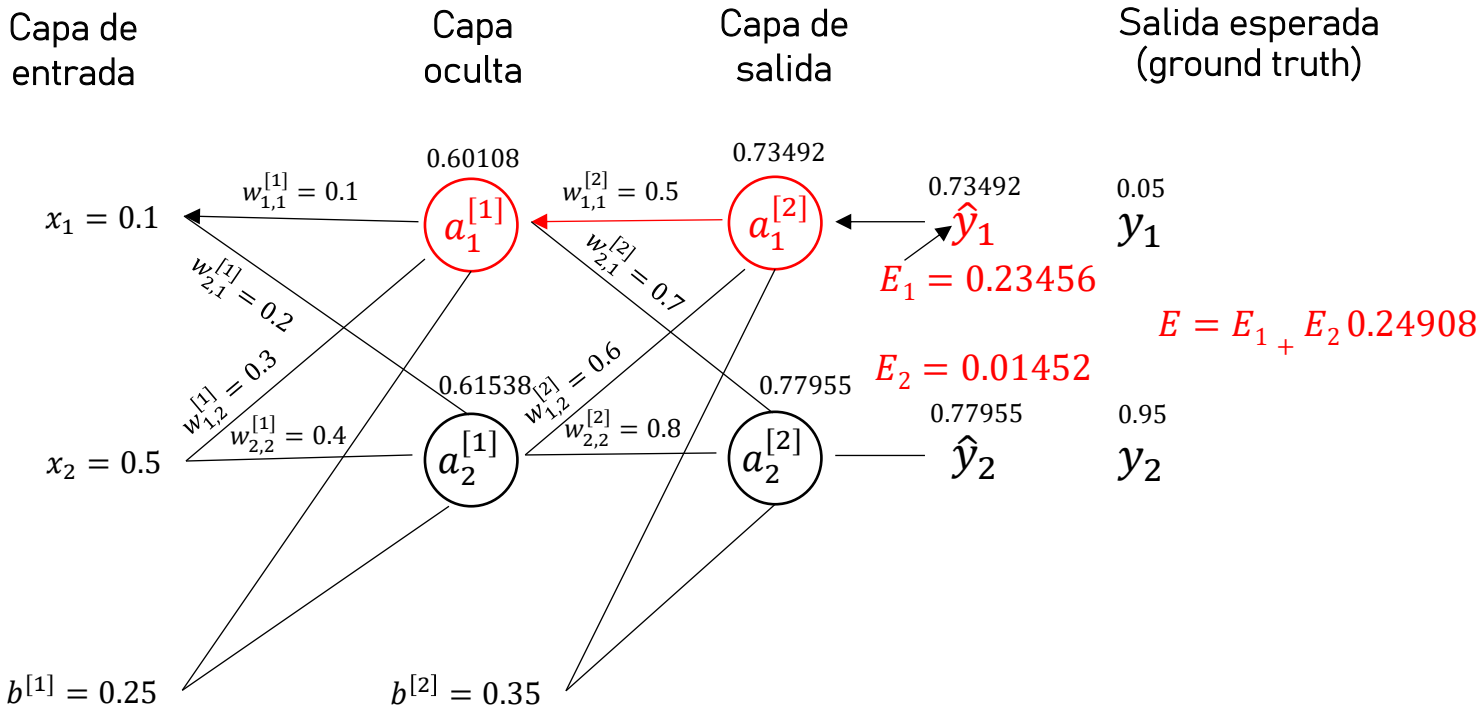
Propagación
Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_1?

$$\frac{dE1}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_1}\frac{d\hat{y}_1}{dz_1^{[2]}}\frac{dz_1^{[2]}}{da_1^{[1]}}\frac{da_1^{[1]}}{dz_1^{[1]}}\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$

$$z_1^{[2]} = a_1^{[1]}w_{1,1}^{[2]} + a_2^{[1]}w_{1,2}^{[2]} + b^{[2]}$$

$$\frac{dz_1^{[2]}}{da_1^{[1]}} = w_{1,1}^{[2]} = 0.5$$



Perceptrones Multicapa y Retropropagación

Ejemplo

Propagación

Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_1?

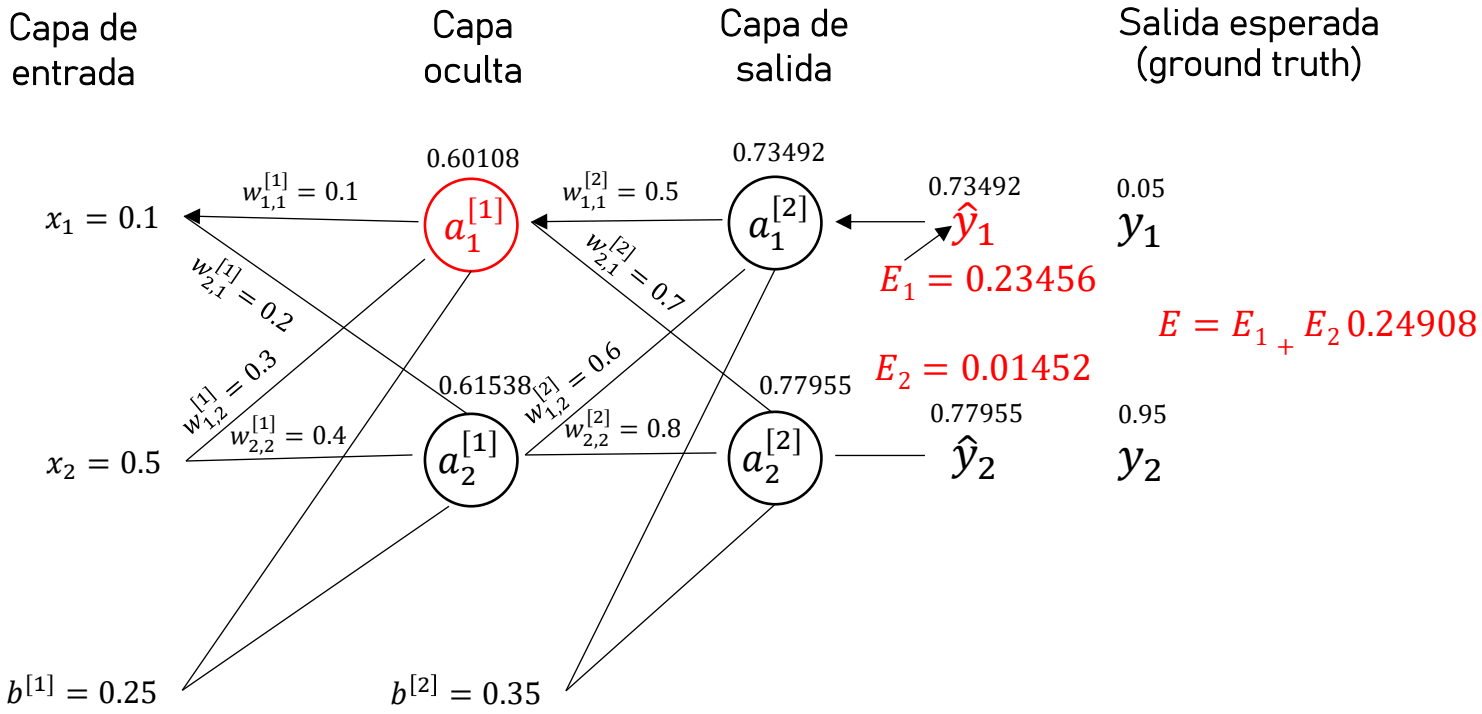
$$\frac{dE1}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_1}\frac{d\hat{y}_1}{dz_1^{[2]}}\frac{dz_1^{[2]}}{da_1^{[1]}}\boxed{\frac{da_1^{[1]}}{dz_1^{[1]}}}\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$

$$a_1^{[1]} = \boldsymbol{\sigma}(dz_1^{[1]})$$

$$\frac{da_1^{[1]}}{dz_1^{[1]}} = \boldsymbol{\sigma}(z_1^{[1]})(1 - z_1^{[1]})$$

$$= 0.60108(1 - 0.60108)$$

$$= \boxed{0.23978}$$



Perceptrones Multicapa y Retropropagación

Ejemplo

Propagación

Hacia atrás

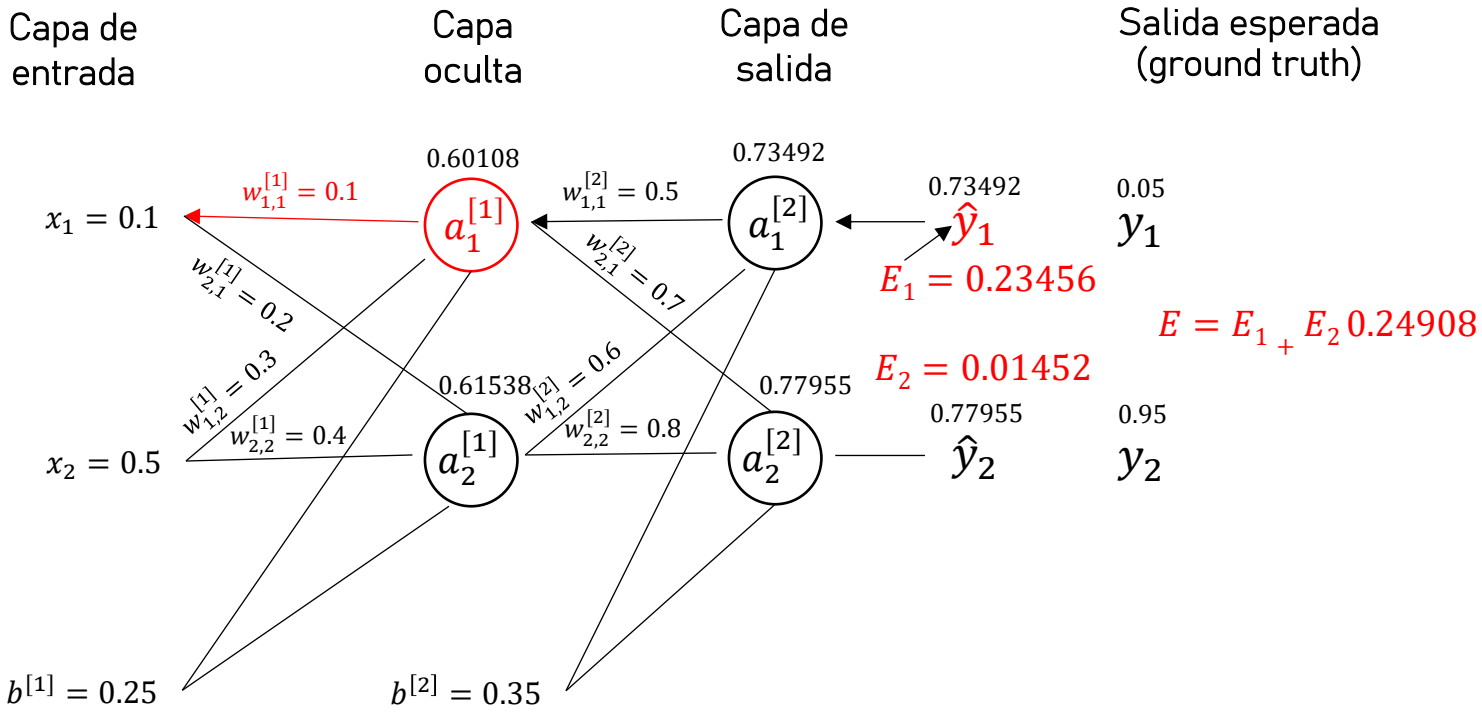
¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_1?

$$\frac{dE1}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_1}\frac{d\hat{y}_1}{dz_1^{[2]}}\frac{dz_1^{[2]}}{da_1^{[1]}}\frac{da_1^{[1]}}{dz_1^{[1]}}\boxed{\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}}$$

$$z_1^{[1]} = w_{1,1}^{[1]}x_1 + w_{1,2}^{[1]}x_2 + b^{[1]}$$

$$\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}} = x_1$$

$$= \boxed{0.1}$$



Perceptrones Multicapa y Retropropagación

Ejemplo

Propagación

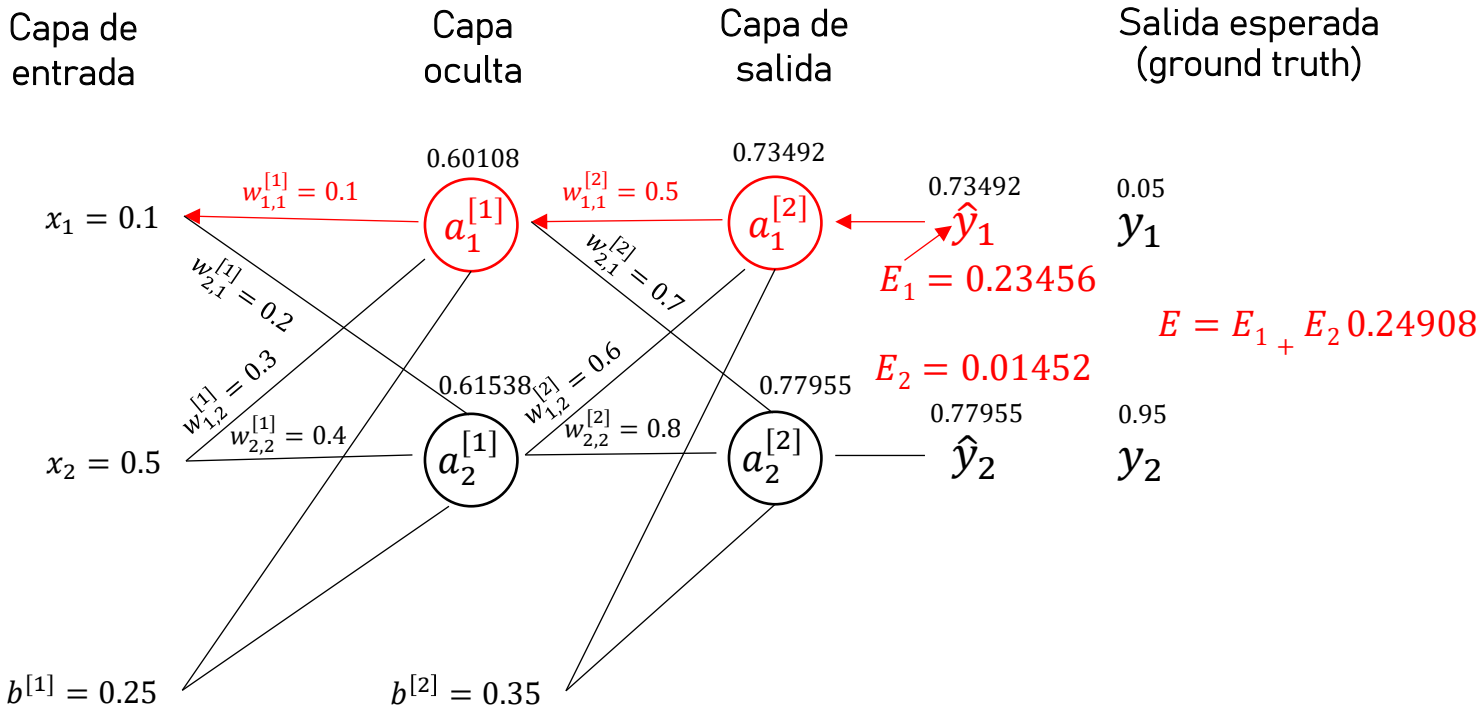
Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_1?

$$\frac{dE1}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_1}\frac{d\hat{y}_1}{dz_1^{[2]}}\frac{dz_1^{[2]}}{da_1^{[1]}}\frac{da_1^{[1]}}{dz_1^{[1]}}\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$

$$= 0.68492 * 0.19480 * 0.5 * 0.23978 * 0.1$$

$$= 0.00159$$



Perceptrones Multicapa y Retropropagación

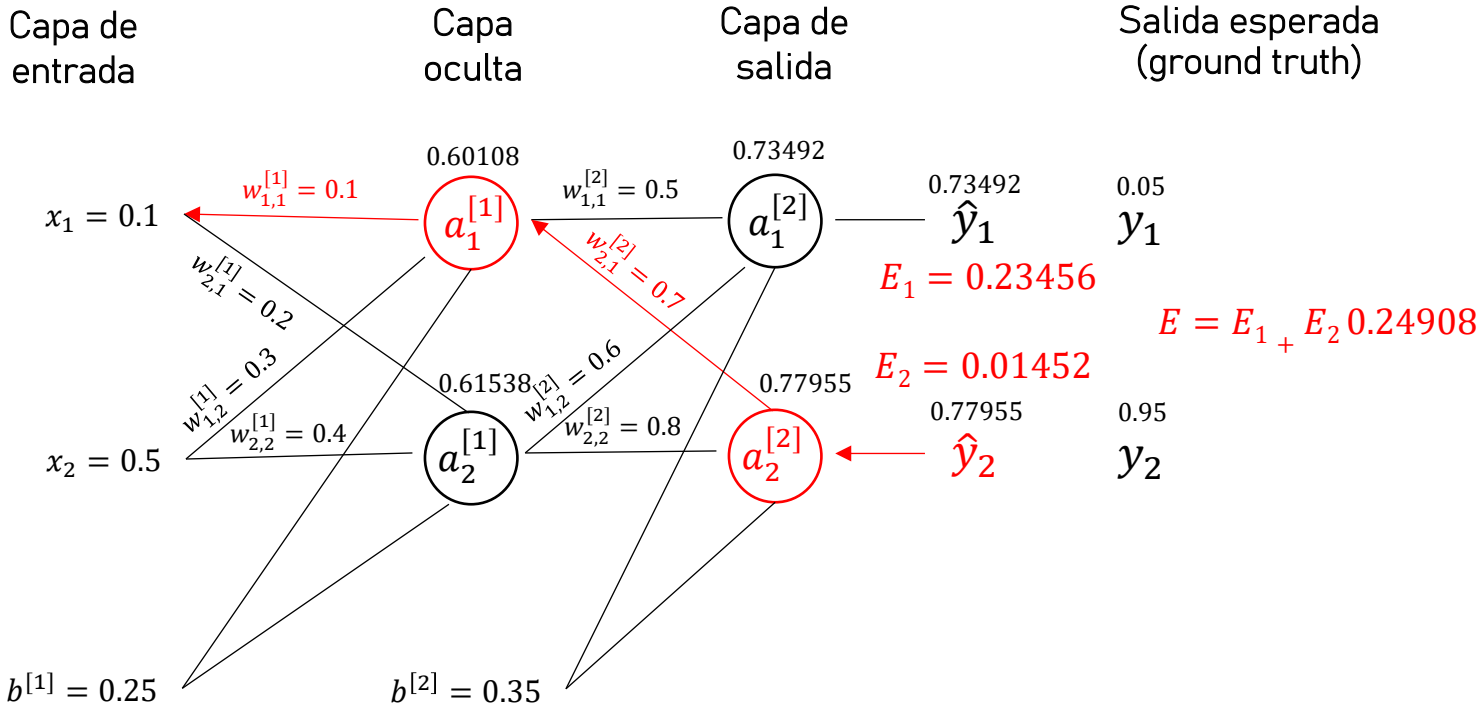
Ejemplo

Propagación

Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_2?

$$\frac{dE2}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_2}\frac{d\hat{y}_2}{dz_2^{[2]}}\frac{dz_2^{[2]}}{da_1^{[1]}}\frac{da_1^{[1]}}{dz_1^{[1]}}\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$



Perceptrones Multicapa y Retropropagación

Ejemplo

Propagación

Hacia atrás

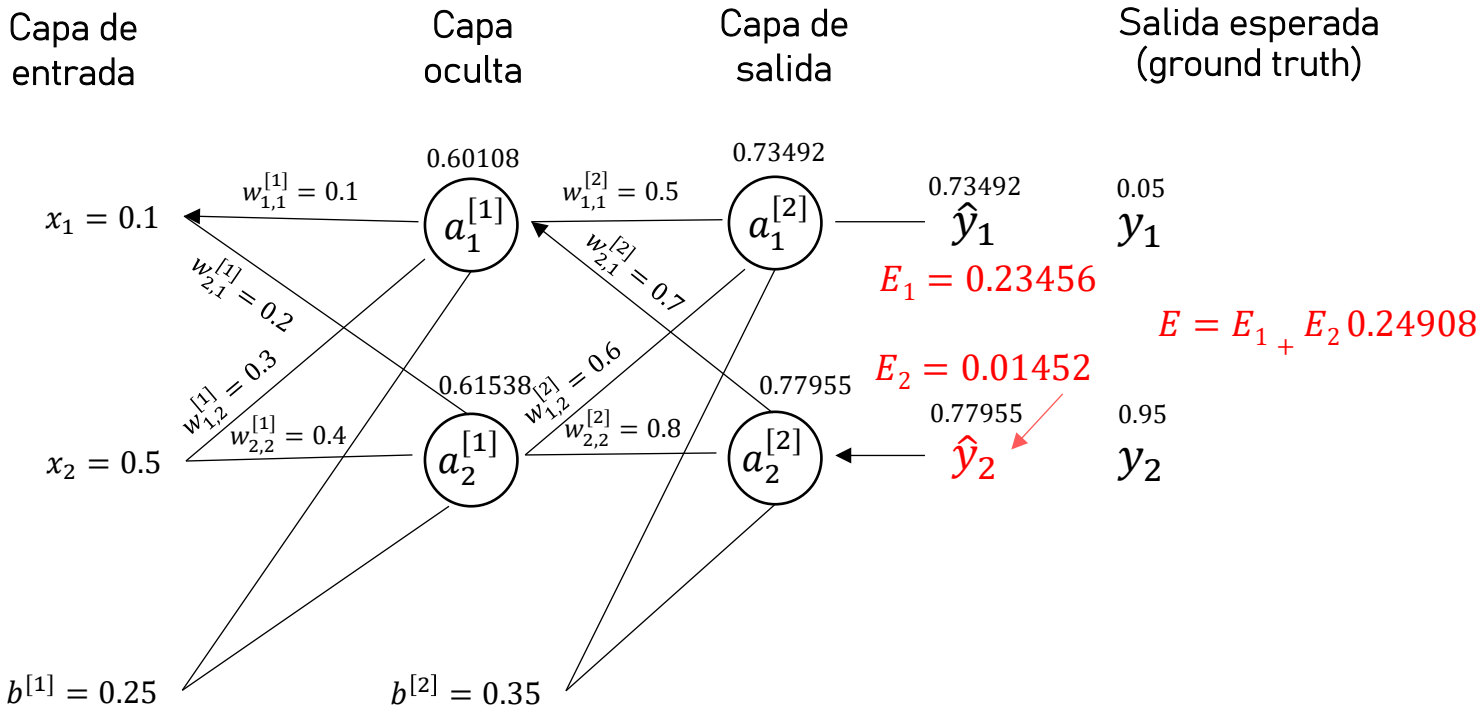
¿Cuánto contribuye $\boldsymbol{w_{1,1}^{[1]}} = 0.1$ en $\boldsymbol{E_2}$?

$$\frac{dE_2}{dw_{1,1}^{[1]}} = \boxed{\frac{dE_1}{d\hat{y}_2}} \frac{d\hat{y}_2}{dz_2^{[2]}} \frac{dz_2^{[2]}}{da_1^{[1]}} \frac{da_1^{[1]}}{dz_1^{[1]}} \frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$

$$2 = \frac{1}{2}(y_2 - \hat{y}_2)^2$$

$$\frac{dE_2}{d\hat{y}_2} = \hat{y}_2 - y_2 = 0.77955 - 0.95$$

$$= \boxed{-0.17044}$$



Perceptrones Multicapa y Retropropagación

Ejemplo

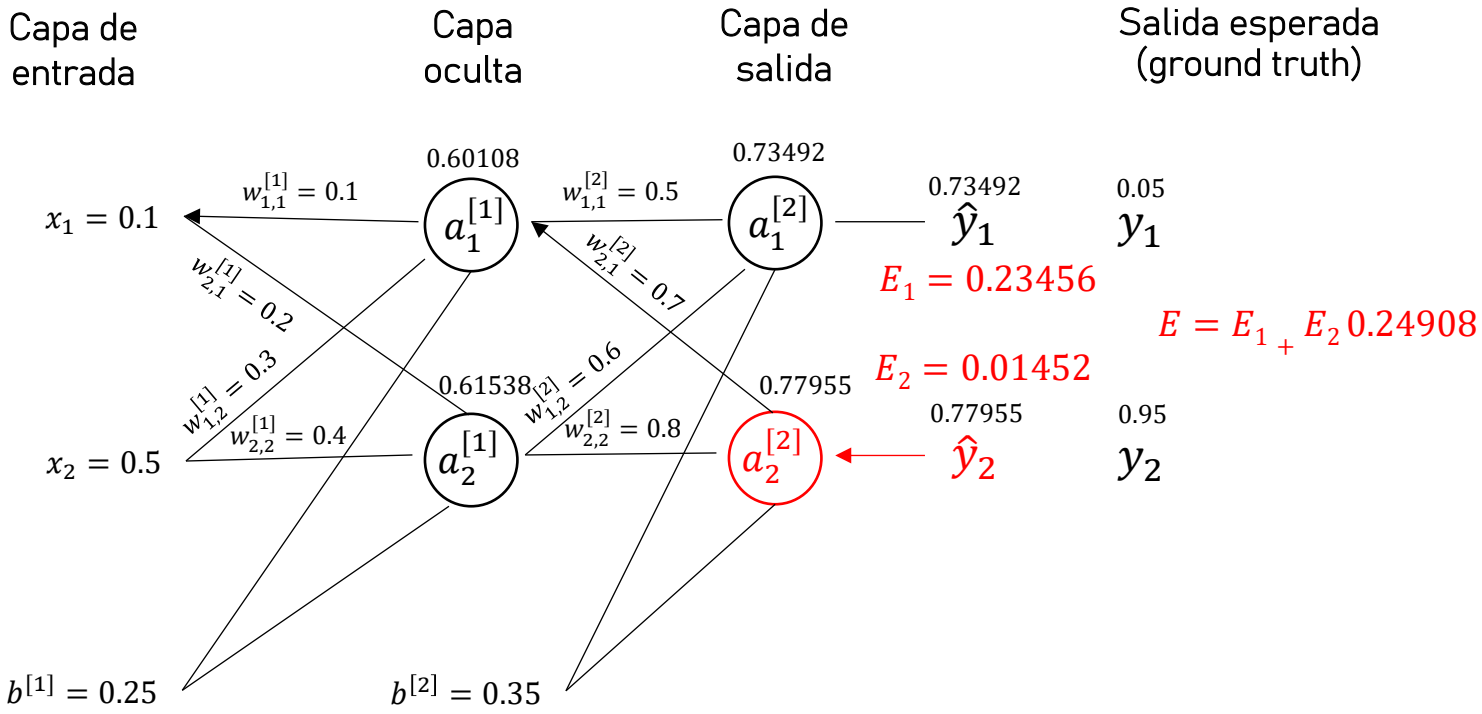
Propagación
Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_2?

$$\frac{dE_2}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_2} \frac{d\hat{y}_2}{dz_2^{[2]}} \frac{dz_2^{[2]}}{da_1^{[1]}} \frac{da_1^{[1]}}{dz_1^{[1]}} \frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$

Ya lo habíamos calculado

0.17184



Perceptrones Multicapa y Retropropagación

Ejemplo

Propagación

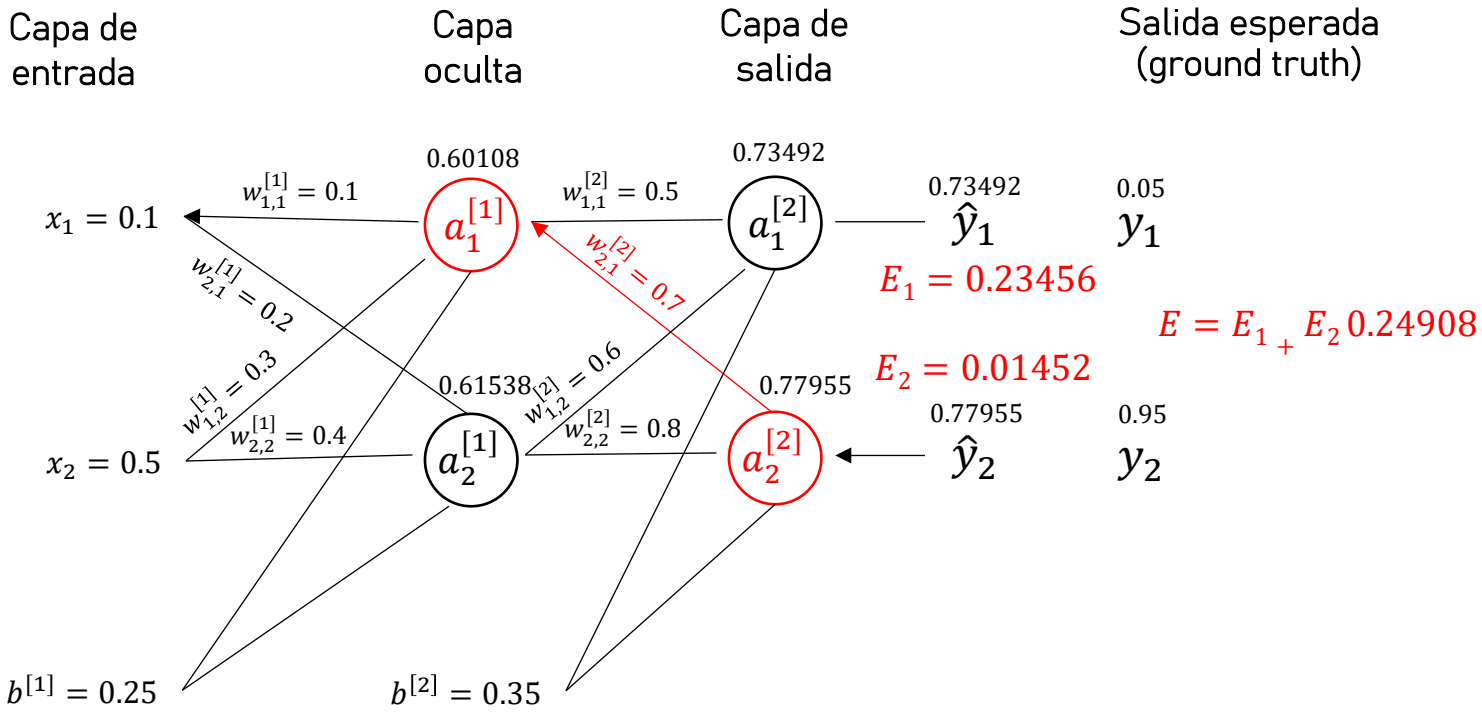
Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_2?

$$\frac{dE_2}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_2}\frac{d\hat{y}_2}{dz_2^{[2]}}\frac{dz_2^{[2]}}{da_1^{[1]}}\frac{da_1^{[1]}}{dz_1^{[1]}}\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$

$$z_2^{[2]} = w_{2,1}^{[2]} * a_1^{[1]} + w_{2,2}^{[2]} * a_2^{[1]} + b^{[2]}$$

$$\frac{dz_2^{[2]}}{da_1^{[1]}} = w_{2,1}^{[2]} = 0.7$$



Perceptrones Multicapa y Retropropagación

Ejemplo

Propagación

Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_2?

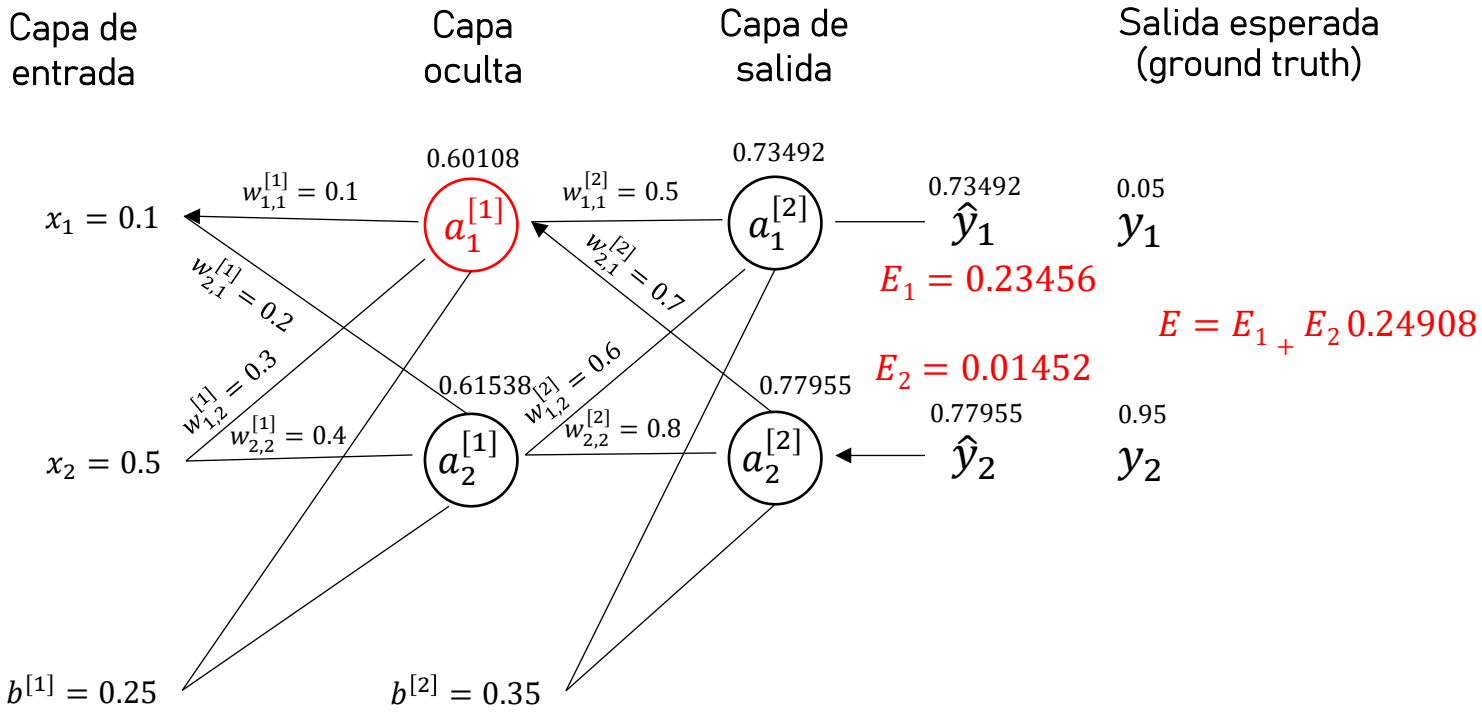
$$\frac{dE_2}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_2}\frac{d\hat{y}_2}{dz_2^{[2]}}\frac{dz_2^{[2]}}{da_1^{[1]}}\boxed{\frac{da_1^{[1]}}{dz_1^{[1]}}}\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$

$$a_1^{[1]} = \boldsymbol{\sigma}(z_1^{[1]})$$

$$\frac{da_1^{[1]}}{dz_1^{[1]}} = \boldsymbol{\sigma}(z_1^{[1]})(1 - \boldsymbol{\sigma}(z_1^{[1]}))$$

$$= 0.60108(1 - 0.60108)$$

$$= \boxed{0.23978}$$



Perceptrones Multicapa y Retropropagación

Ejemplo

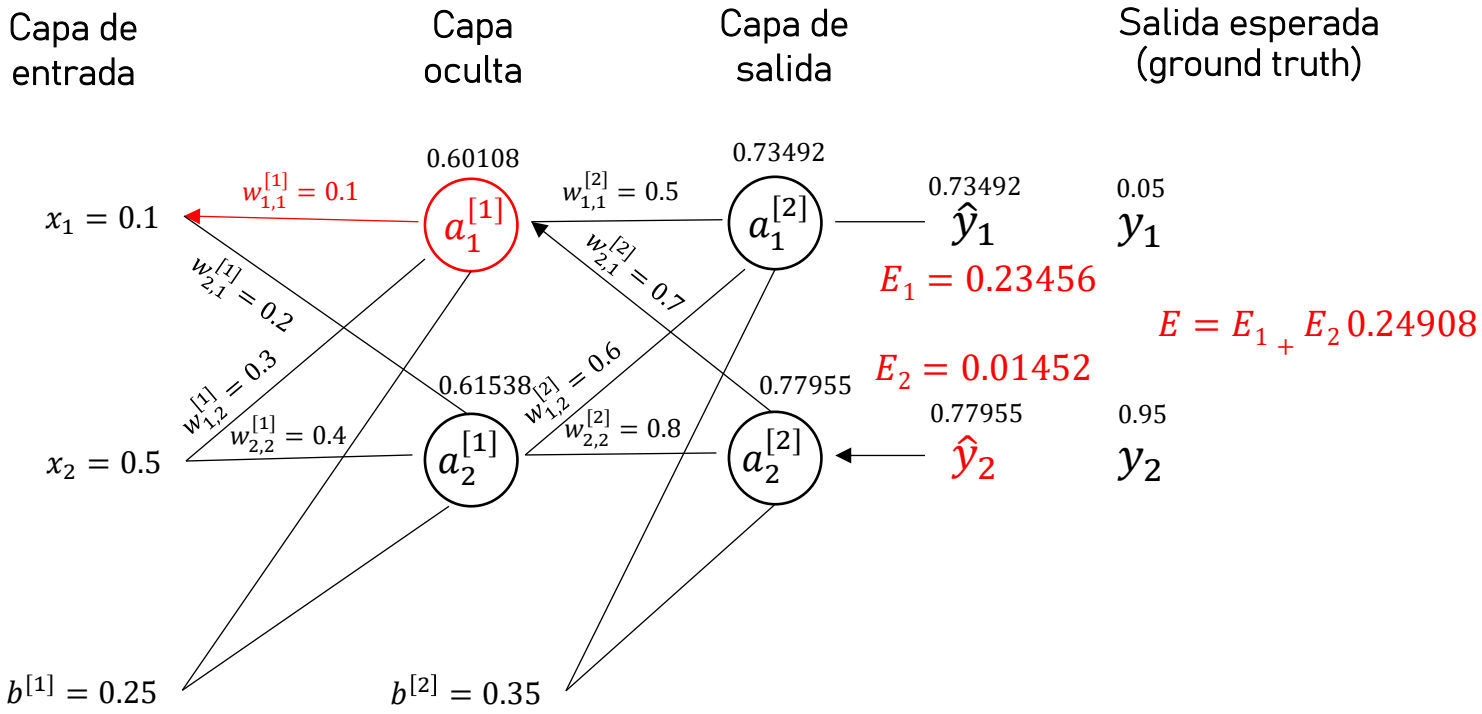
Propagación
Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_2?

$$\frac{dE_2}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_2}\frac{d\hat{y}_2}{dz_2^{[2]}}\frac{dz_2^{[2]}}{da_1^{[1]}}\frac{da_1^{[1]}}{dz_1^{[1]}}\boxed{\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}}$$

Ya lo habíamos calculado

$$0.23978$$



Perceptrones Multicapa y Retropropagación

Ejemplo

Propagación

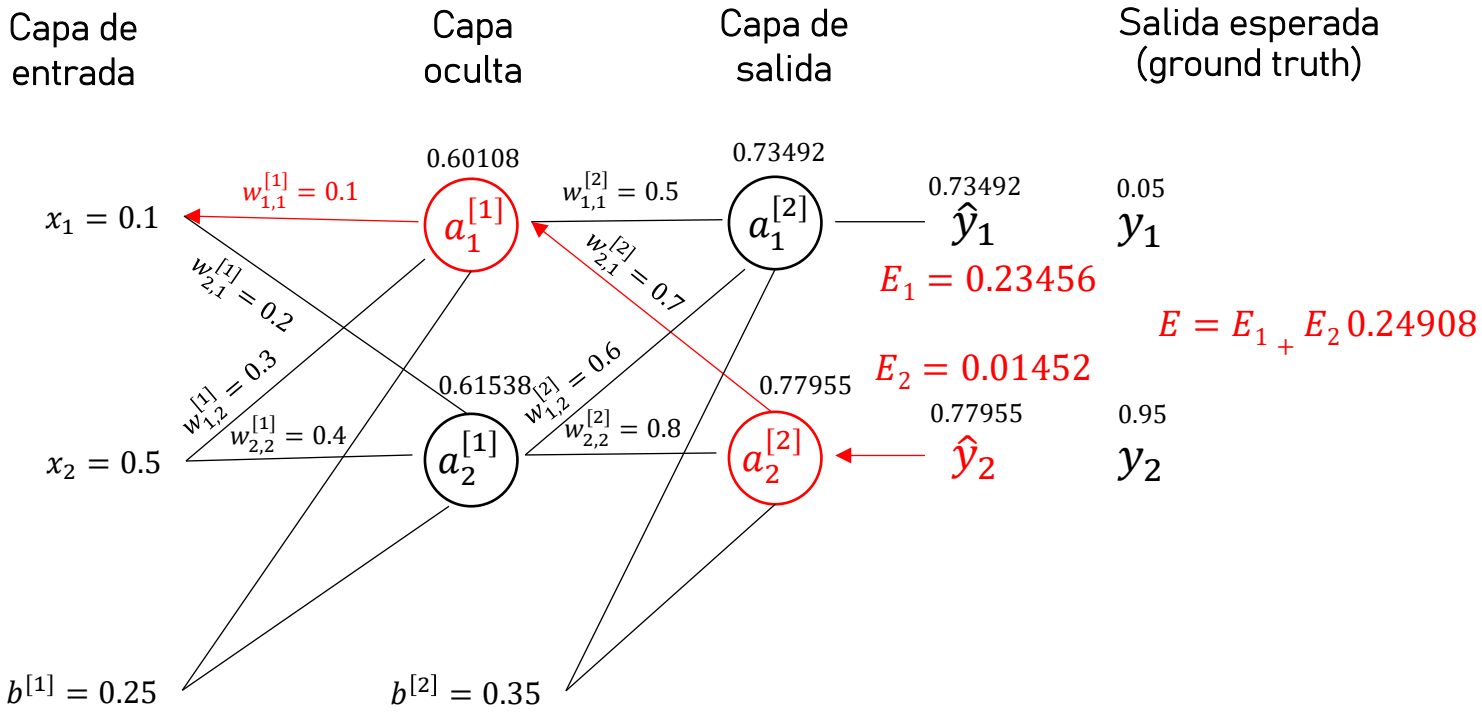
Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E_2?

$$\frac{dE_2}{dw_{1,1}^{[1]}} = \frac{dE_1}{d\hat{y}_2}\frac{d\hat{y}_2}{dz_2^{[2]}}\frac{dz_2^{[2]}}{da_1^{[1]}}\frac{da_1^{[1]}}{dz_1^{[1]}}\frac{dz_1^{[1]}}{dw_{1,1}^{[1]}}$$

$$= -0.17044 * 0.17184 * 0.7 * 0.23978 * 0.1$$

$$= -0.00049$$



Perceptrones Multicapa y Retropropagación

Ejemplo

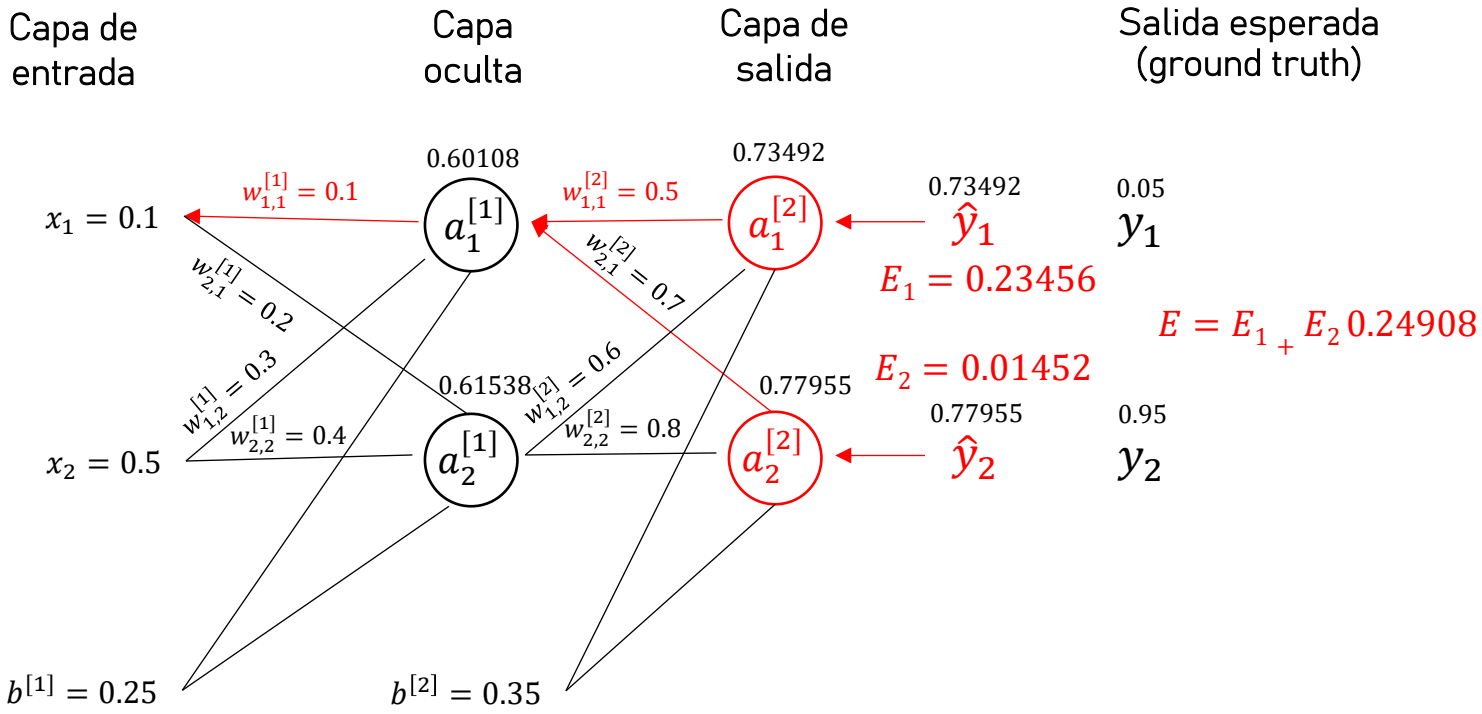
Propagación
Hacia atrás

¿Cuánto contribuye $w_{1,1}^{[1]} = 0.1$ en E?

$$\frac{dE}{dw_{1,1}^{[1]}} = \frac{dE_1}{dw_{1,1}^{[1]}} + \frac{dE_2}{dw_{1,1}^{[1]}}$$

$$= 0.00159 + (-0.00049)$$

$$= 0.0011$$



Perceptrones Multicapa y Retropropagación

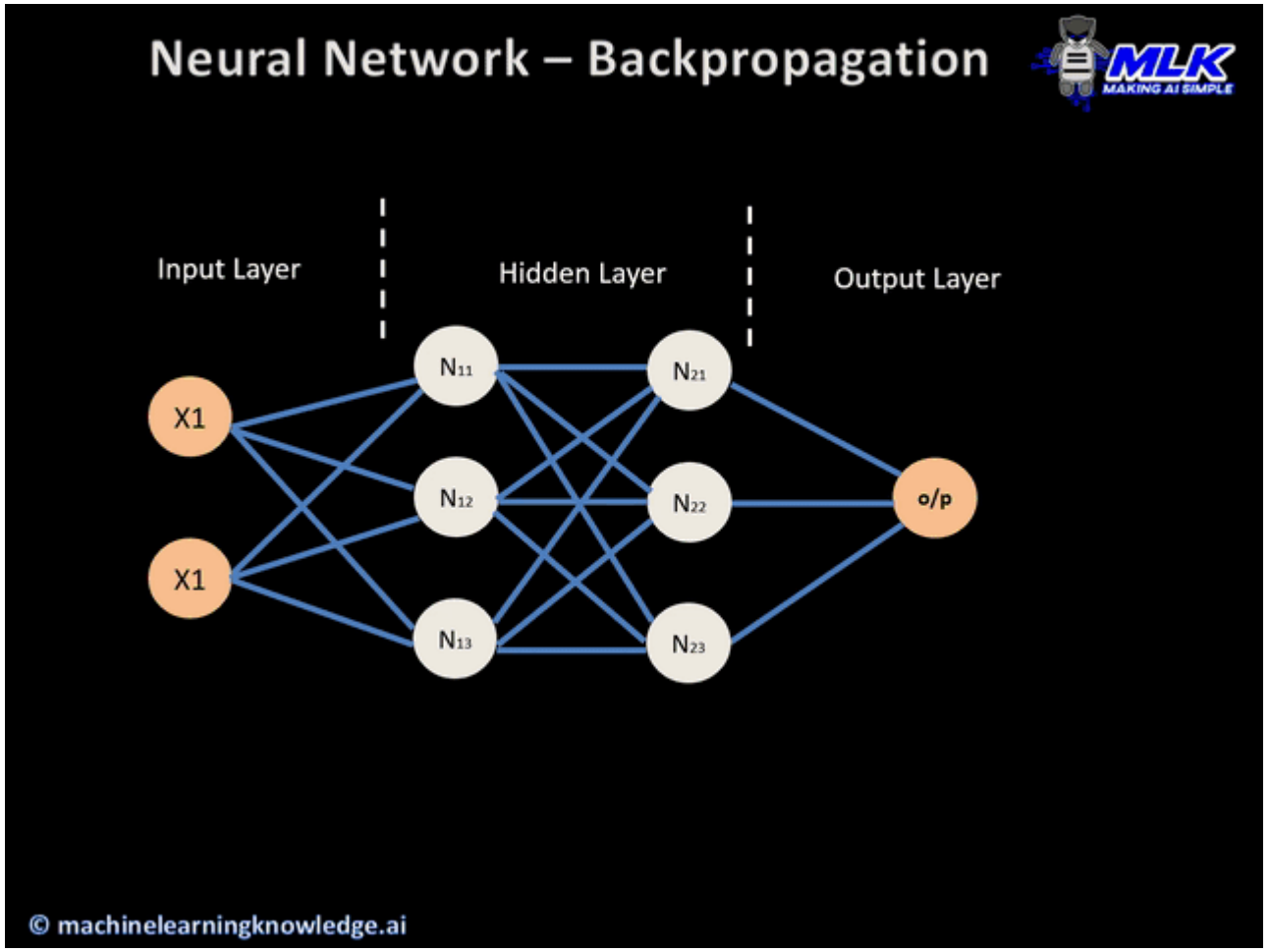
Se realiza lo mismo para $w_{2,1}^{[1]}, w_{1,2}^{[1]}, w_{2,2}^{[1]}$, $\mathrm{b}^1, \mathrm{b}^2$

- Al finalizar de calcular los nuevos valores para todos los pesos y sesgos, los viejos valores se sustituyen por los nuevos.
- Se vuelve a realizar la propagación hacia adelante y se calcula el nuevo error total de la red.
- Se vuelve a propagar el error hacia atrás , y se vuelven a actualizar los pesos y sesgos.
- Continuar hasta que el valor de la pérdida se minimice.

Perceptrones Multicapa y Retropropagación

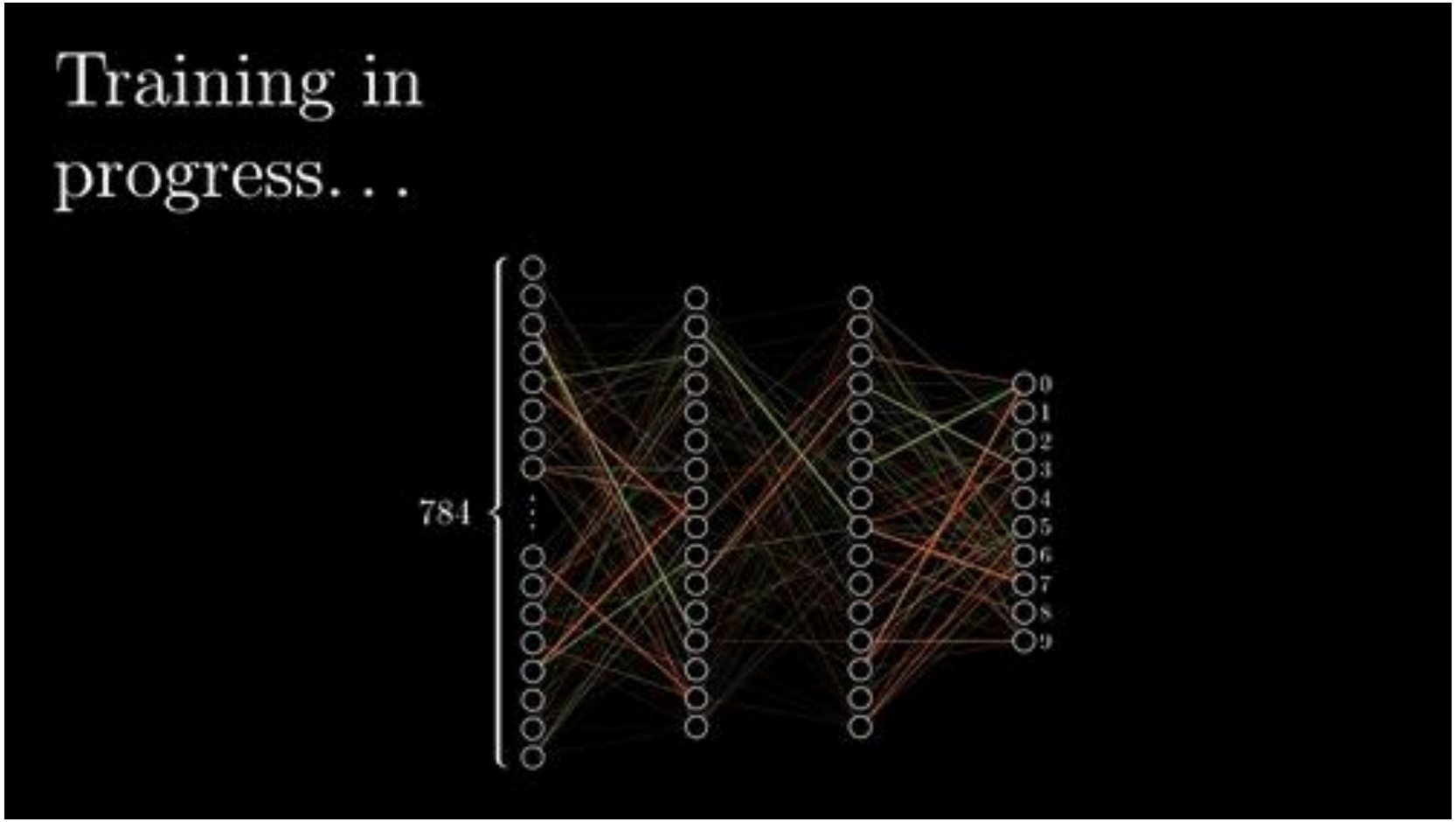
Se realiza lo mismo para $w_{2,1}^{[1]}, w_{1,2}^{[1]}, w_{2,2}^{[1]}$, $\mathrm{b}^1, \mathrm{b}^2$

- Al finalizar de calcular los nuevos valores para todos los pesos y sesgos, los viejos valores se sustituyen por los nuevos.
- Se vuelve a realizar la propagación hacia adelante y se calcula el nuevo error total de la red.
- Se vuelve a propagar el error hacia atrás , y se vuelven a actualizar los pesos y sesgos.
- Continuar hasta que el valor de la pérdida se minimice.

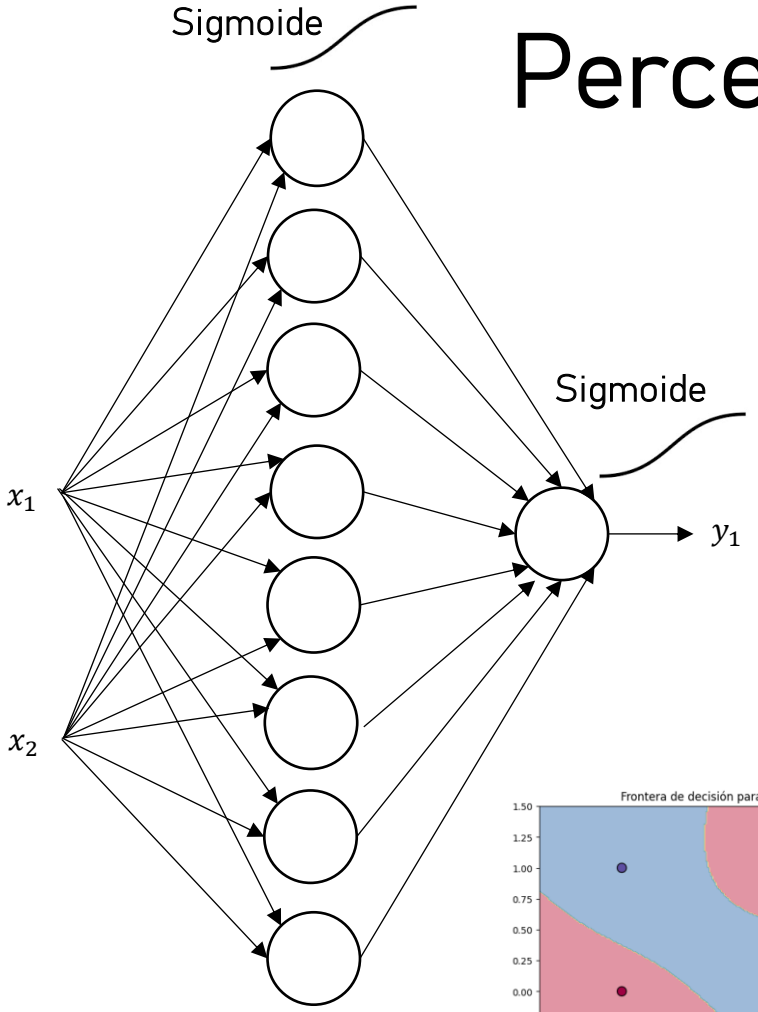


Perceptrones Multicapa y Retropropagación

Esto mismo se hace para todos los ejemplos

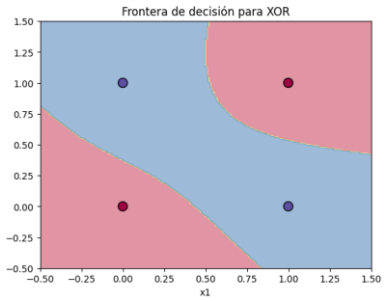


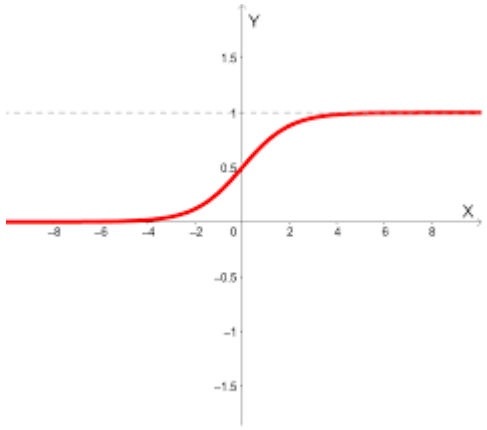
Perceptrones Multicapa y XOR



learning_rate=4.5

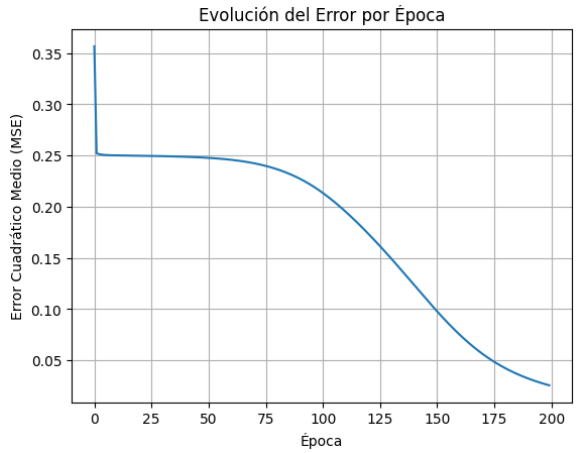
Entradas		Salida
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0



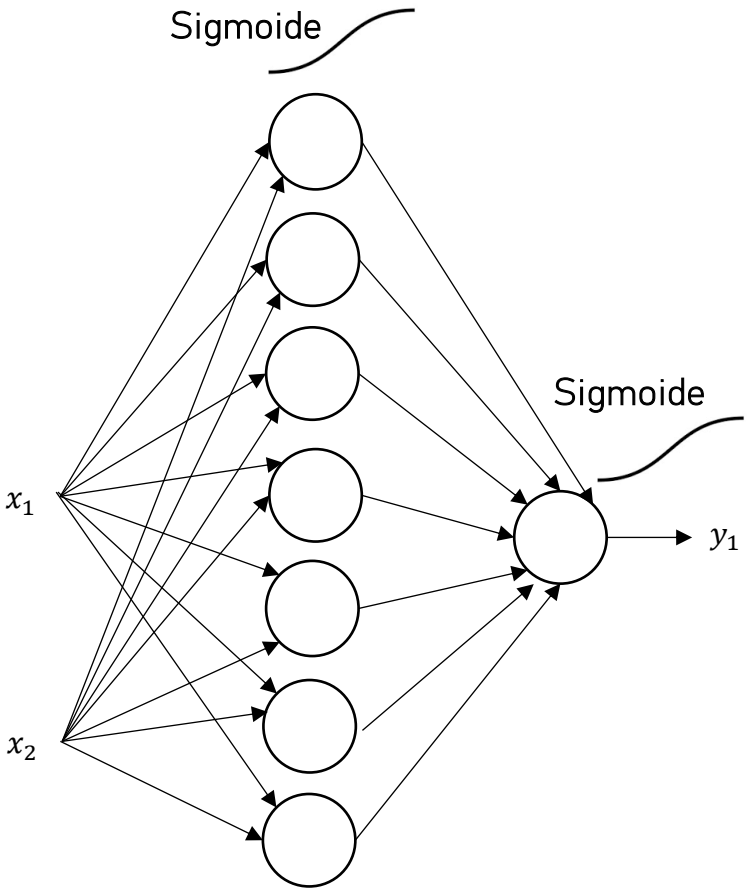


backprop_TF_XOR.ipynb


```
Entrada: [0. 0.], Predicción continua: 0.1810, Predicción binaria: 0, Predicción verdadera: [0.]
Entrada: [0. 1.], Predicción continua: 0.7685, Predicción binaria: 1, Predicción verdadera: [1.]
Entrada: [1. 0.], Predicción continua: 0.7230, Predicción binaria: 1, Predicción verdadera: [1.]
Entrada: [1. 1.], Predicción continua: 0.3167, Predicción binaria: 0, Predicción verdadera: [0.]
```



Perceptrones Multicapa y XOR

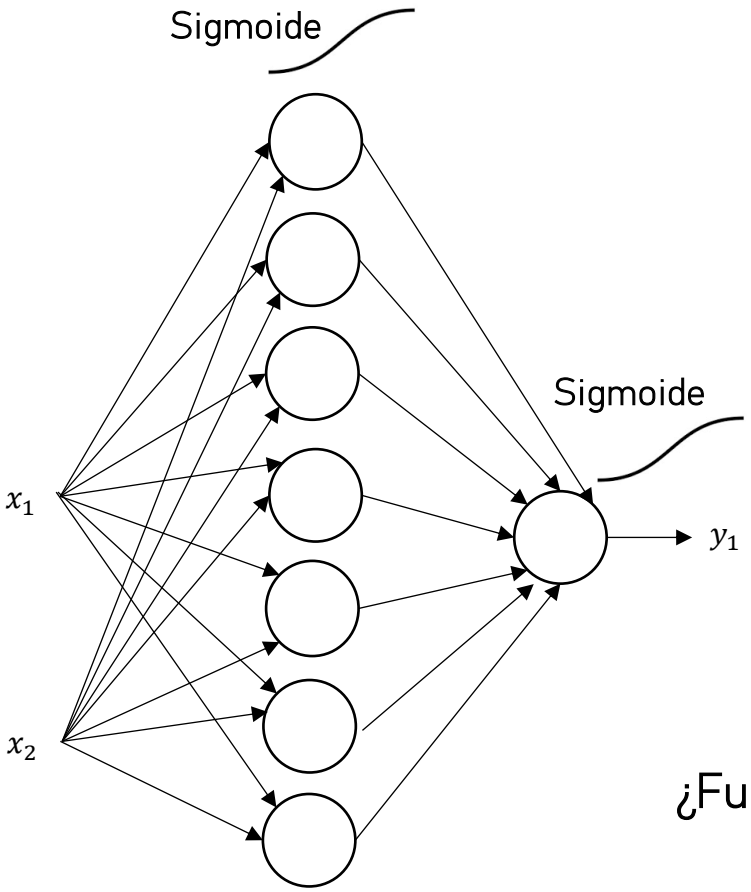


Hiperparámetros

Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

Perceptrones Multicapa y XOR



Hiperparámetros

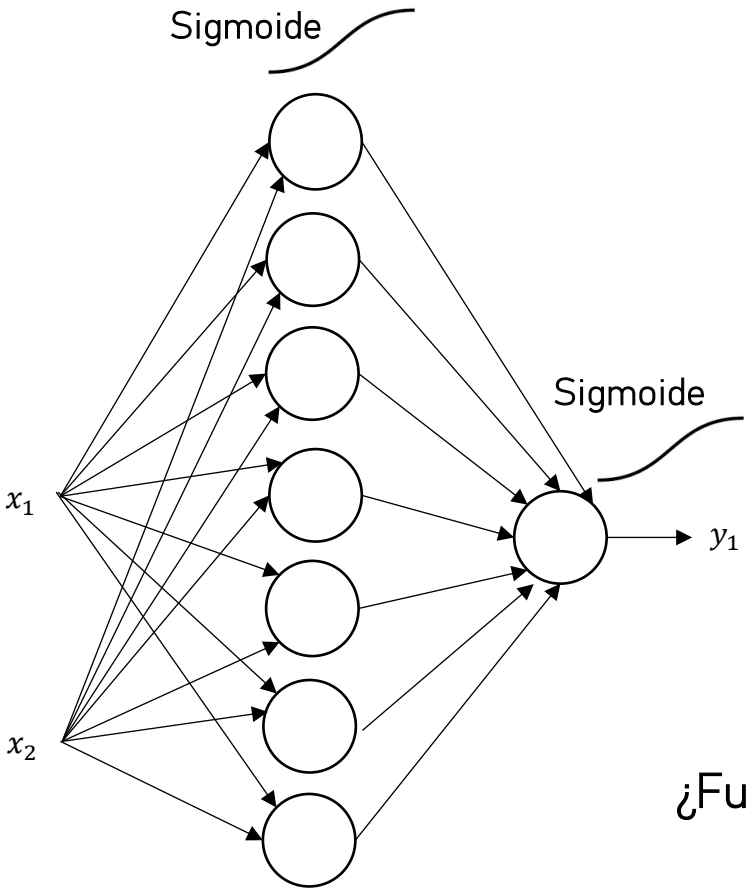
Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

¿Funcionará para una red con 1 capa oculta y 7 neuronas?

Backprop_TF_XOR.ipynb

Perceptrones Multicapa y XOR

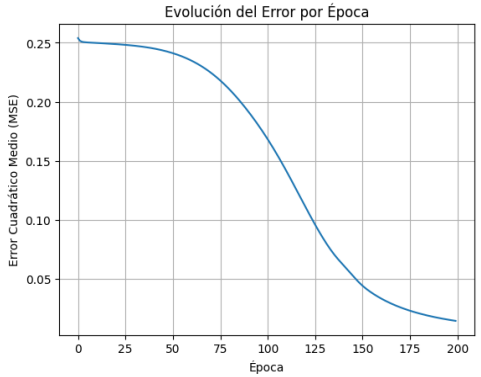


Hiperparámetros

Relacionados con la arquitectura de la red

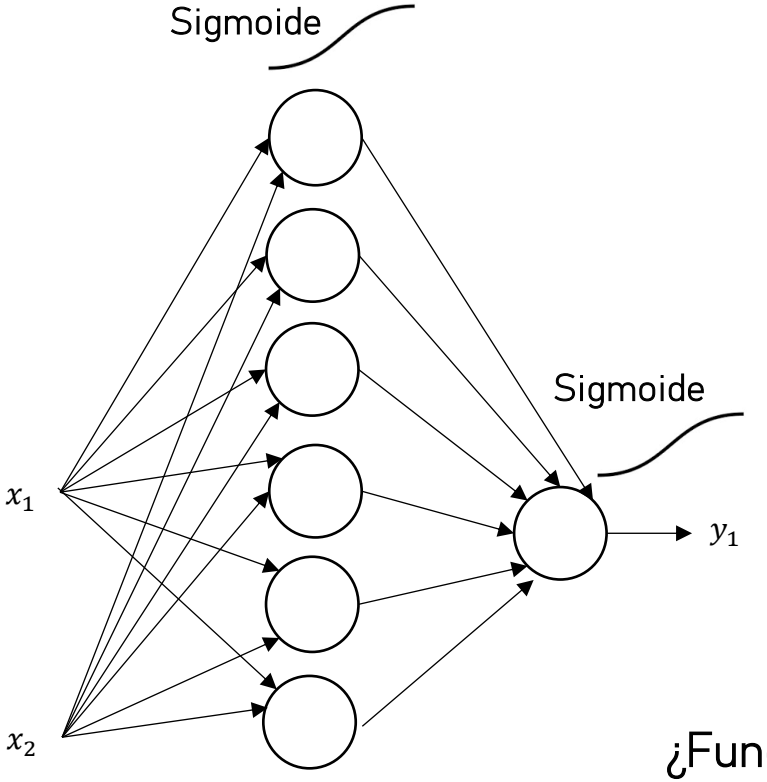
Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

¿Funcionará para una red con 1 capa oculta y 7 neuronas?



Backprop_TF_XOR.ipynb

Perceptrones Multicapa y XOR



Hiperparámetros

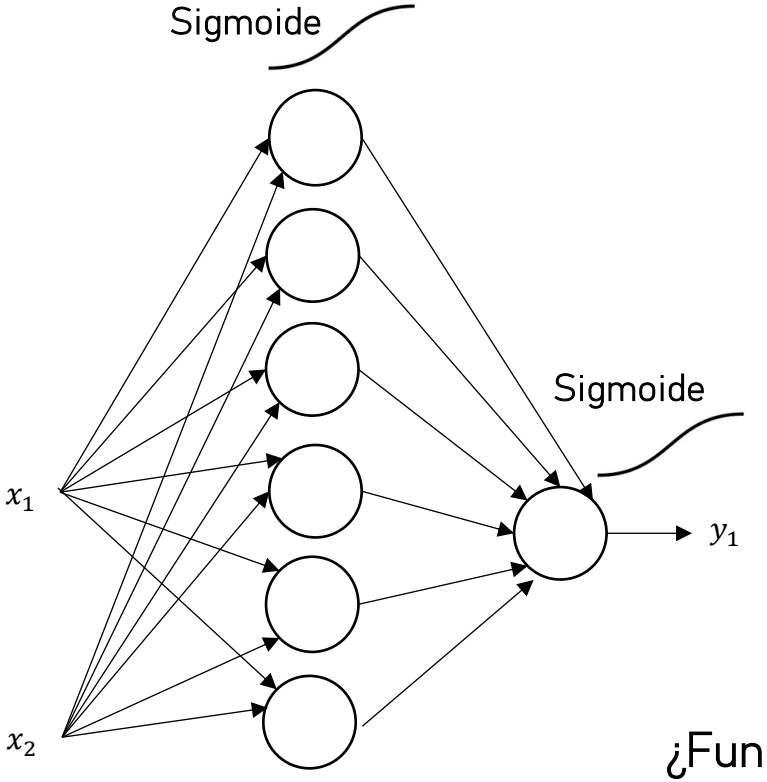
Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

¿Funcionará para una red con 1 capa oculta y 6 neuronas?

Backprop_TF_XOR.ipynb

Perceptrones Multicapa y XOR

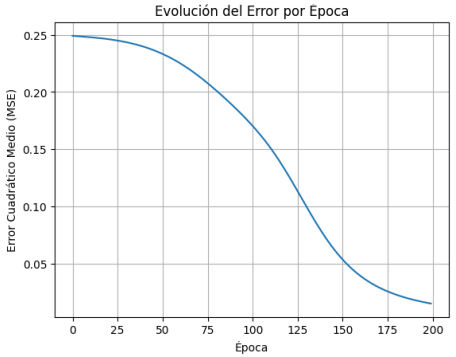


Hiperparámetros

Relacionados con la arquitectura de la red

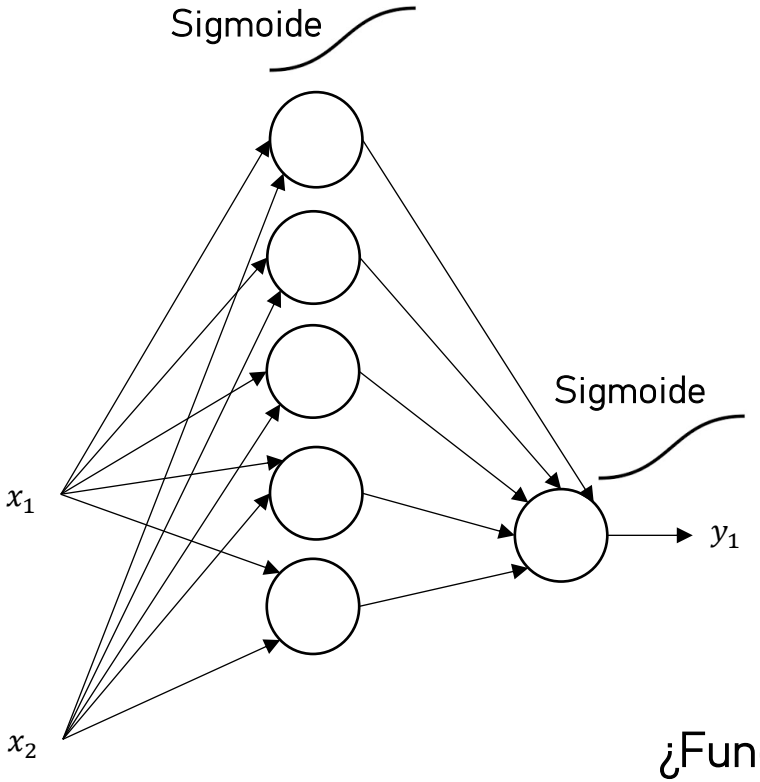
Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

¿Funcionará para una red con 1 capa oculta y 6 neuronas?



Backprop_TF_XOR.ipynb

Perceptrones Multicapa y XOR



Hiperparámetros

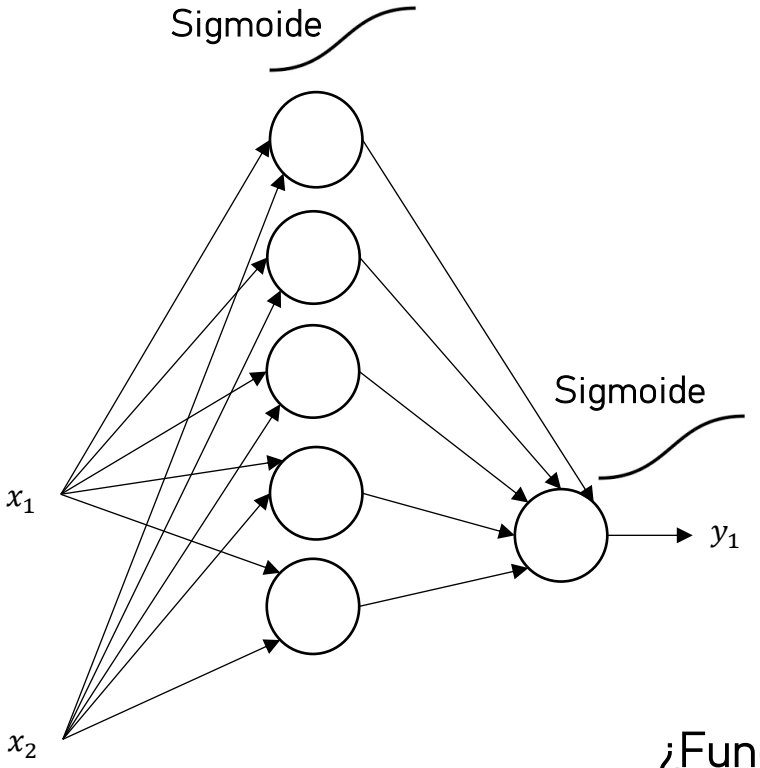
Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

¿Funcionará para una red con 1 capa oculta y 5 neuronas?

Backprop_TF_XOR.ipynb

Perceptrones Multicapa y XOR

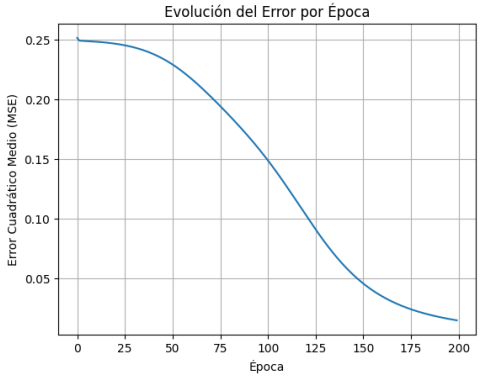


Hiperparámetros

Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

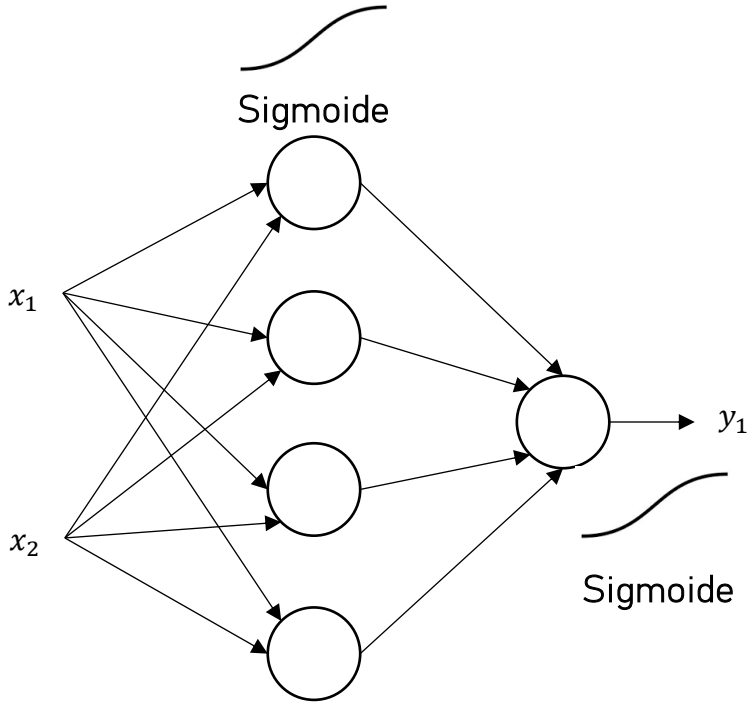
¿Funcionará para una red con 1 capa oculta y 5 neuronas?



Backprop_TF_XOR.ipynb

jupyter

Perceptrones Multicapa y XOR



Hiperparámetros

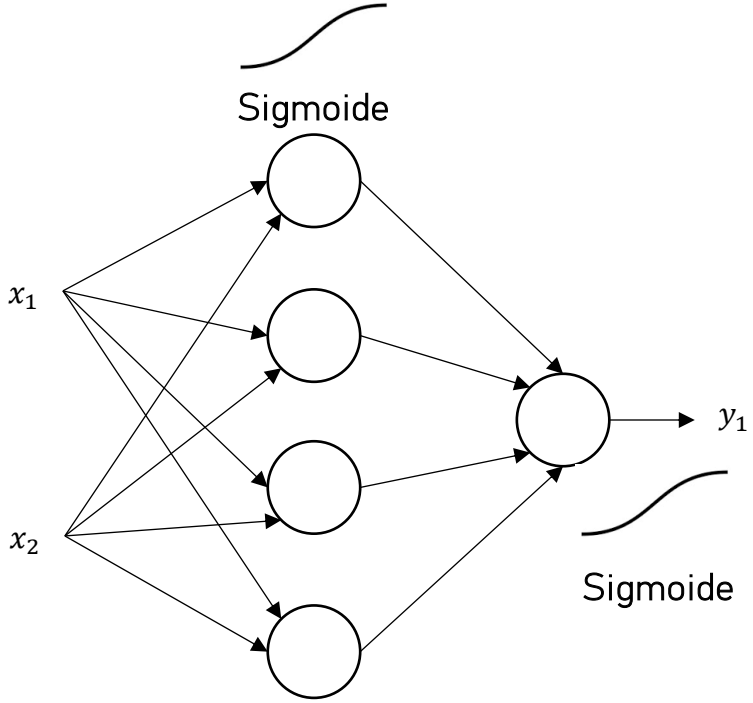
Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

¿Funcionará para una red con 1 capa oculta y 4 neuronas?

Backprop_TF_XOR.ipynb

Perceptrones Multicapa y XOR

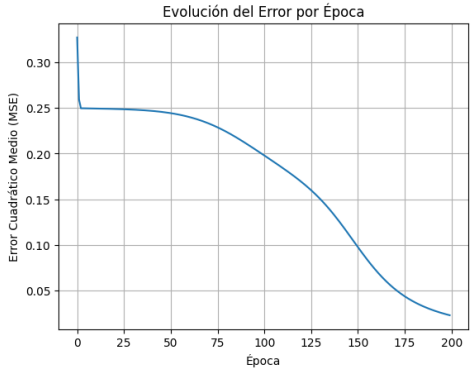


Hiperparámetros

Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

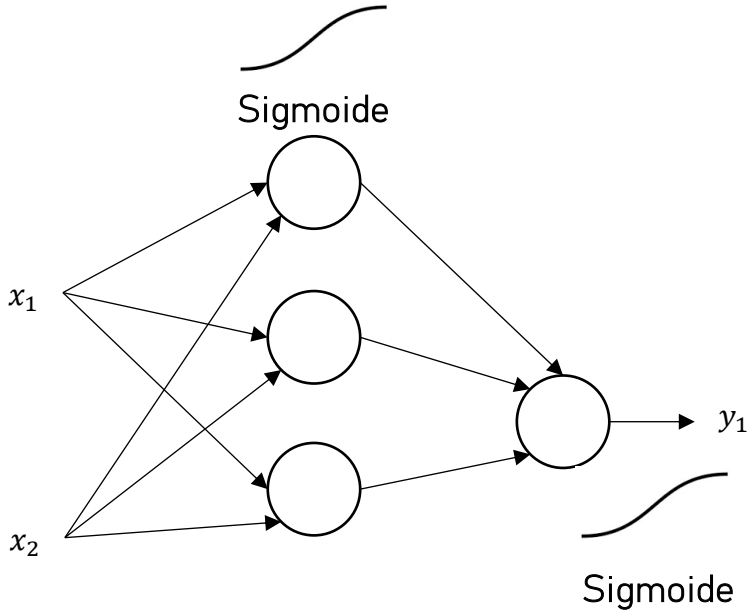
¿Funcionará para una red con 1 capa oculta y 4 neuronas?



Backprop_TF_XOR.ipynb

jupyter

Perceptrones Multicapa y XOR



Hiperparámetros

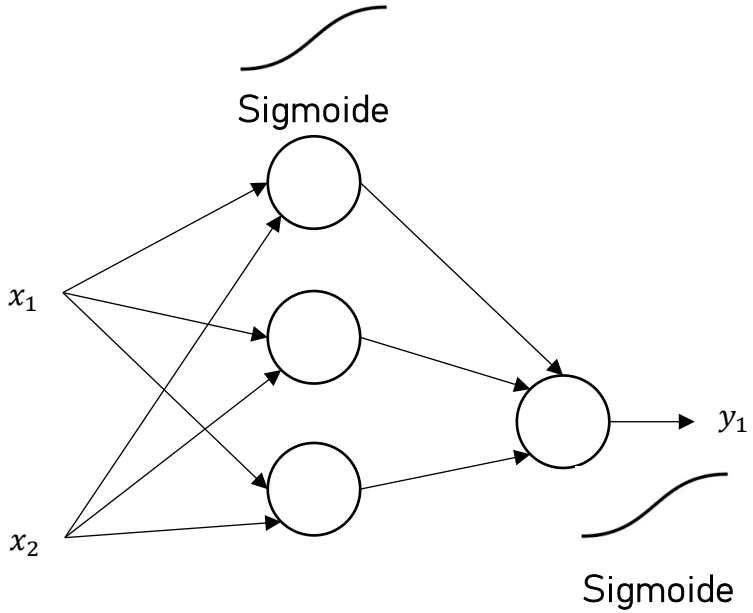
Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

¿Funcionará para una red con 1 capa oculta y 3 neuronas?

Backprop_TF_XOR.ipynb

Perceptrones Multicapa y XOR

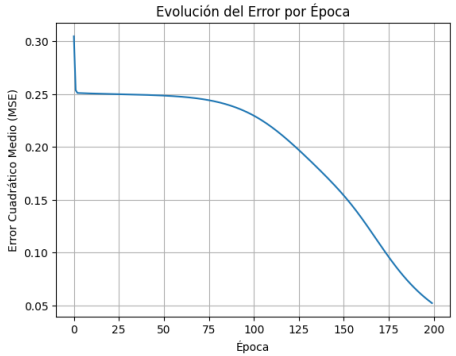


Hiperparámetros

Relacionados con la arquitectura de la red

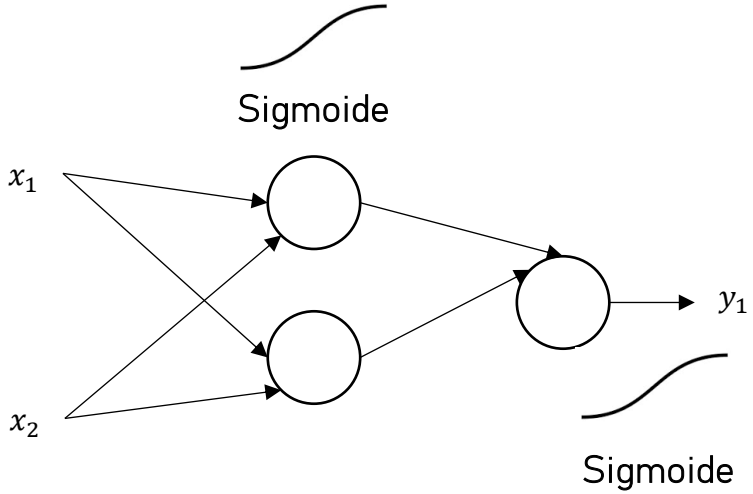
Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

¿Funcionará para una red con 1 capa oculta y 3 neuronas?



Backprop_TF_XOR.ipynb

Perceptrones Multicapa y XOR



Hiperparámetros

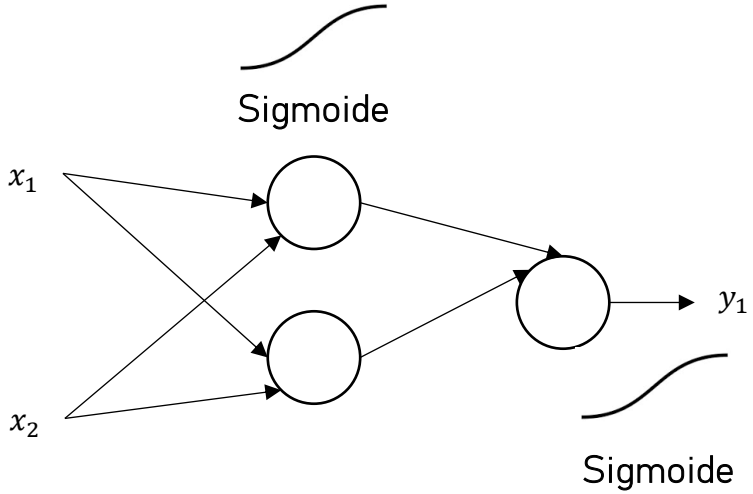
Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

¿Funcionará para una red con 1 capa oculta y 2 neuronas?

Backprop_TF_XOR.ipynb

Perceptrones Multicapa y XOR

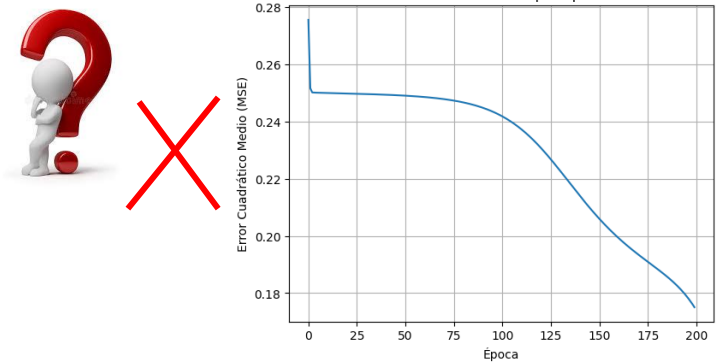


Hiperparámetros

Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

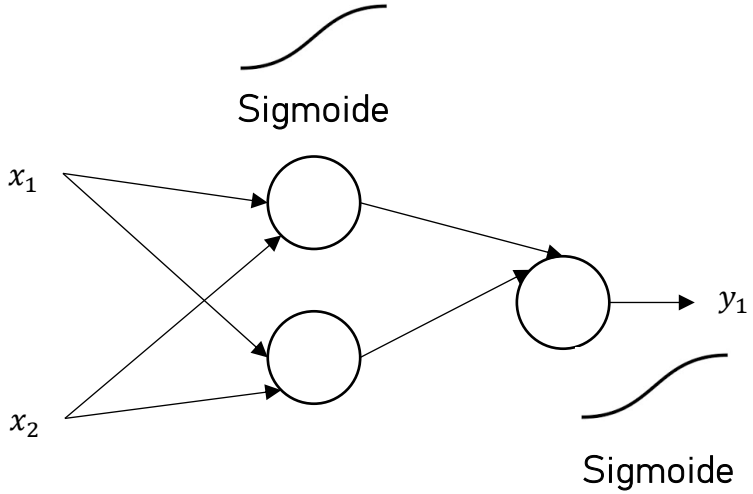
¿Funcionará para una red con 1 capa oculta y 2 neuronas?



Backprop_TF_XOR.ipynb

jupyter

Perceptrones Multicapa y XOR



Hiperparámetros

Relacionados con el entrenamiento

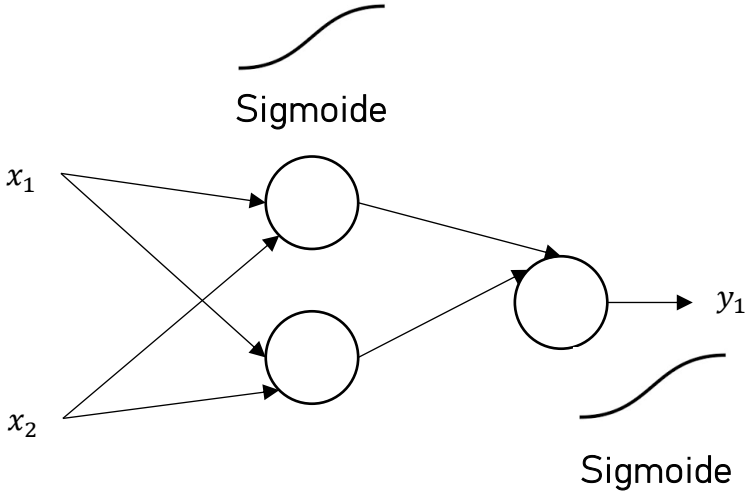
Función de pérdida	`binary_crossentropy`, `categorical_crossentropy`, `mean_squared_error`	Escoge según la tarea: clasificación binaria, multiclase o regresión.
Optimizador	`SGD`, `Adam`, `RMSProp`	Adam suele ser la opción más estable y rápida para la mayoría de problemas.
Tasa de aprendizaje (learning rate)	0.001–0.5	Depende del optimizador y tamaño de red; learning rate demasiado alto puede hacer que no converja.
Épocas (epochs)	50–1000	Depende del dataset; usar early stopping ayuda a evitar sobreentrenamiento.
Tamaño de batch (batch size)	4–256	Pequeños → más ruido, exploración; grandes → aprendizaje más estable, pero mayor memoria.

¿Funcionará para una red con 1 capa oculta y 2 neuronas y 400 épocas?

Backprop_TF_XOR.ipynb

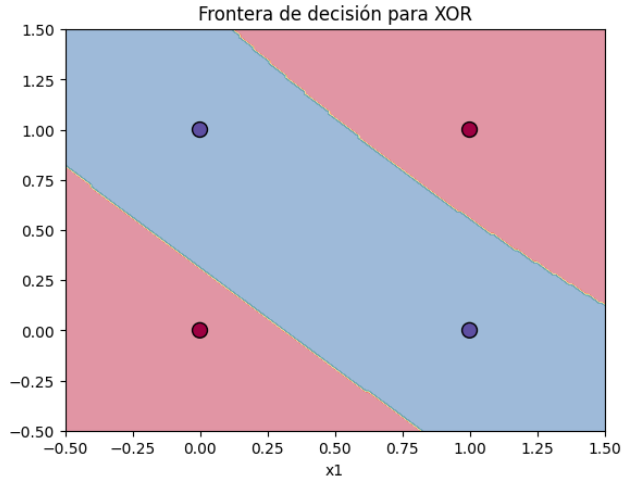
Perceptrones Multicapa y XOR

Hiperparámetros

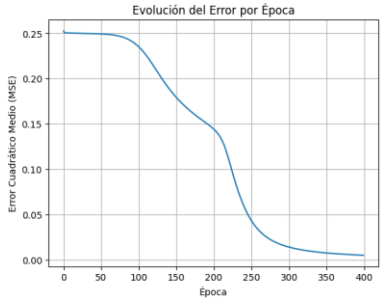


Relacionados con el entrenamiento

Función de pérdida	`binary_crossentropy`, `categorical_crossentropy`, `mean_squared_error`	Escoge según la tarea: clasificación binaria, multiclase o regresión.
Optimizador	`SGD`, `Adam`, `RMSProp`	Adam suele ser la opción más estable y rápida para la mayoría de problemas.
Tasa de aprendizaje (learning rate)	0.001–0.5	Depende del optimizador y tamaño de red; learning rate demasiado alto puede hacer que no converja.
Épocas (epochs)	50–1000	Depende del dataset; usar early stopping ayuda a evitar sobreentrenamiento.
Tamaño de batch (batch size)	4–256	Pequeños → más ruido, exploración; grandes → aprendizaje más estable, pero mayor memoria.



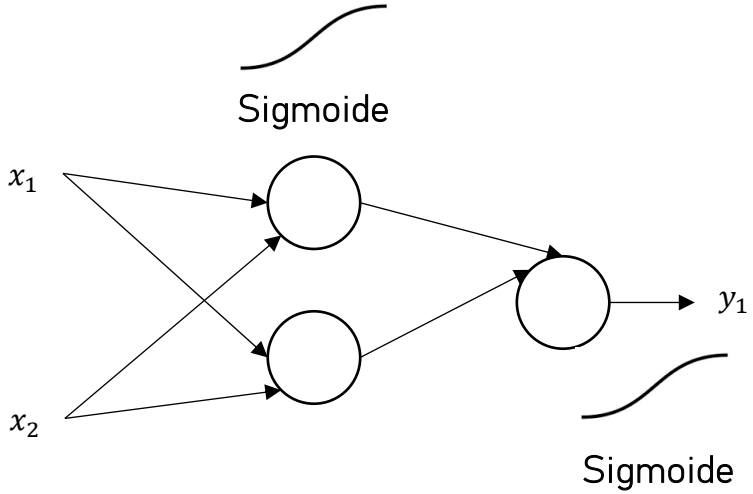
¿Funcionará para una red con 1 capa oculta y 2 neuronas y 400 épocas?



jupyter

Backprop_TF_XOR.ipynb

Perceptrones Multicapa y XOR



Hiperparámetros

Relacionados con el entrenamiento

Función de pérdida	`binary_crossentropy`, `categorical_crossentropy`, `mean_squared_error`	Escoge según la tarea: clasificación binaria, multiclase o regresión.
Optimizador	`SGD`, `Adam`, `RMSProp`	Adam suele ser la opción más estable y rápida para la mayoría de problemas.
Tasa de aprendizaje (learning rate)	0.001–0.5	Depende del optimizador y tamaño de red; learning rate demasiado alto puede hacer que no converja.
Épocas (epochs)	50–1000	Depende del dataset; usar early stopping ayuda a evitar sobreentrenamiento.
Tamaño de batch (batch size)	4–256	Pequeños → más ruido, exploración; grandes → aprendizaje más estable, pero mayor memoria.

$$\text{Backpropagation} \longrightarrow \textit{Cálculodegradientes}$$

$$\text{Descenso por gradiente} \longrightarrow \textit{Actualizacióndepesos}$$

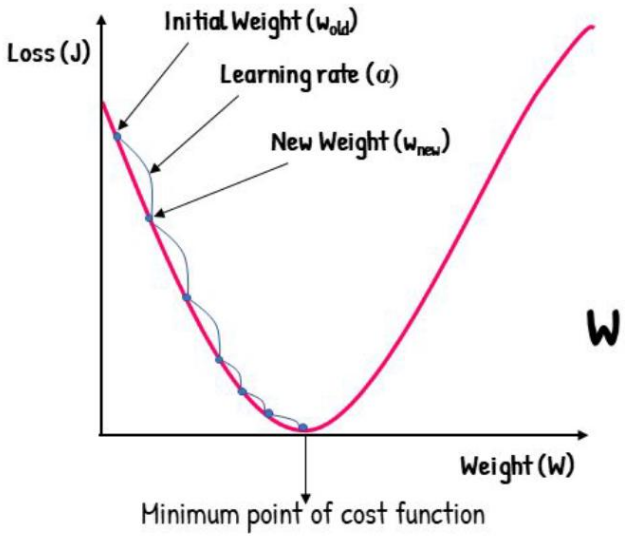
Reglas

- Todos los optimizadores disponibles en Keras/TensorFlow (PyTorch) se basan en el cálculo de gradientes, obtenidos mediante backpropagation.
- Existen en la literatura métodos alternativos (p.ej. optimización evolutiva, algoritmos genéticos, búsqueda aleatoria, simulated annealing), pero esos no están en Keras por defecto, porque Keras está diseñado en torno a redes neuronales entrenadas por gradiente.

Descenso por Gradiente

```
optimizer = tf.keras.optimizers.SGD(learning_rate=4.5, momentum=0.0, nesterov=False)
history = model.fit(x, y, epochs=200, batch_size=len(x), verbose=2)
```

Usa todo el dataset para el cálculo del error, y actualizar pesos y sesgos.



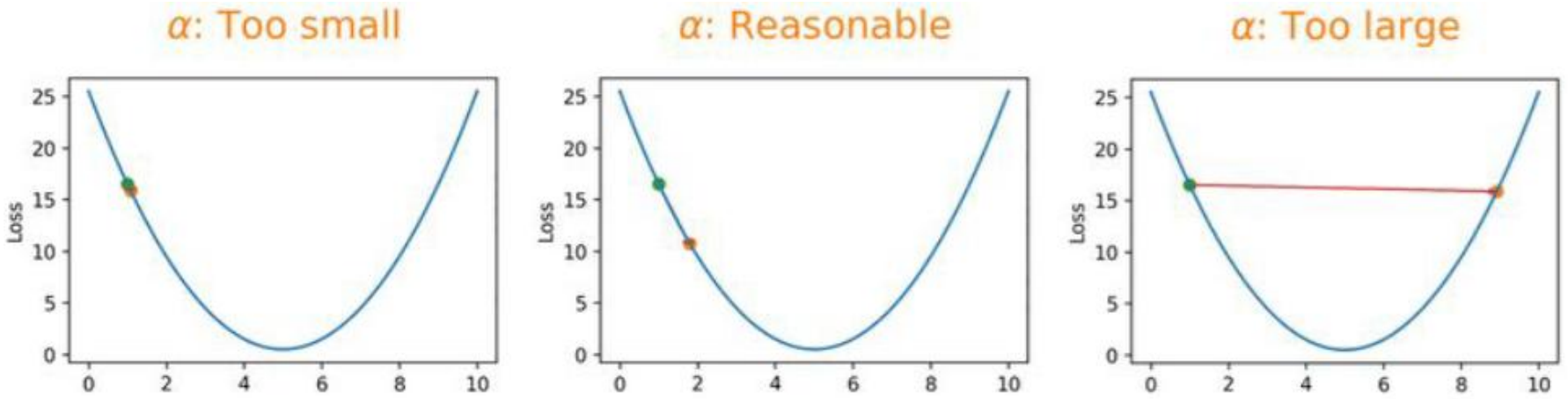
Error/Pérdida

$$w_{new} = w_{old} - \alpha \frac{\delta J}{\delta w}$$

Tasa de aprendizaje

Elección de la tasa de aprendizaje:

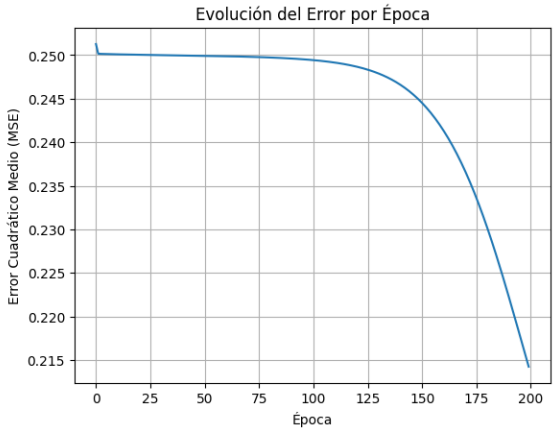
- Si es demasiado **grande**, el algoritmo puede **oscilar** o **divergir** en lugar de converger.
- Si es demasiado **pequeña**, la convergencia será muy lenta.



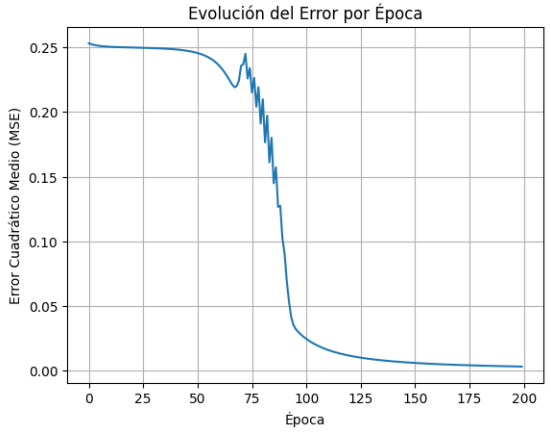
Descenso por Gradiente

```
optimizer = tf.keras.optimizers.SGD(learning_rate=4.5, momentum=0.0, nesterov=False)
history = model.fit(x, y, epochs=200, batch_size=len(x), verbose=2)
```

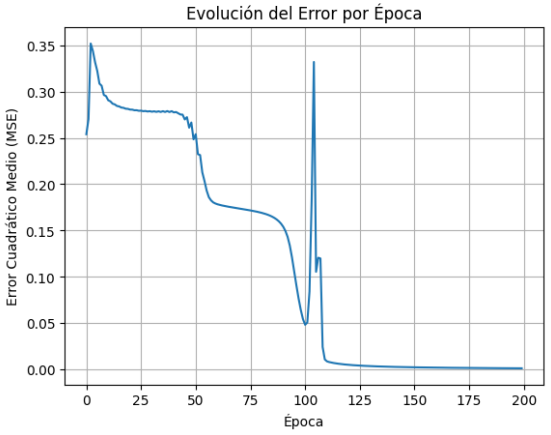
Backprop_TF_XOR.ipynb



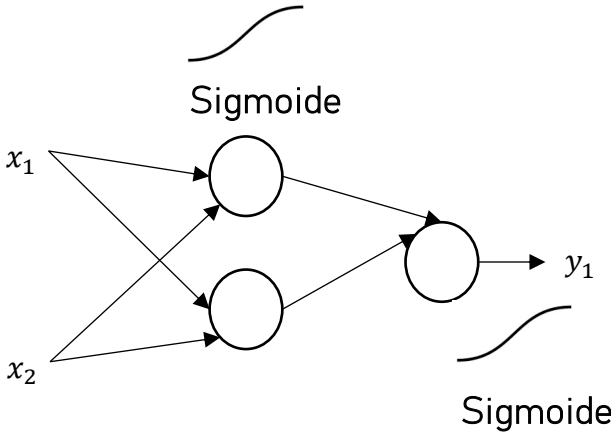
Tasa de aprendizaje = 4.5?

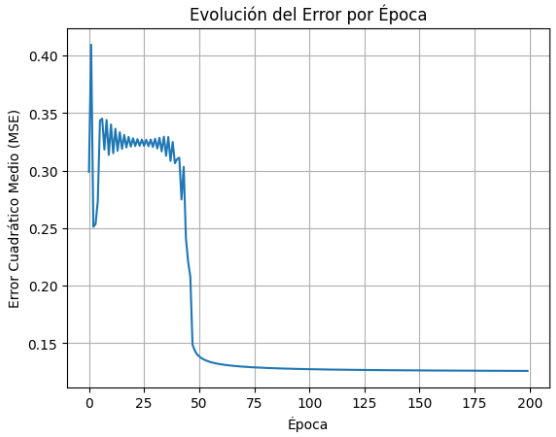


Tasa de aprendizaje = 9.0?

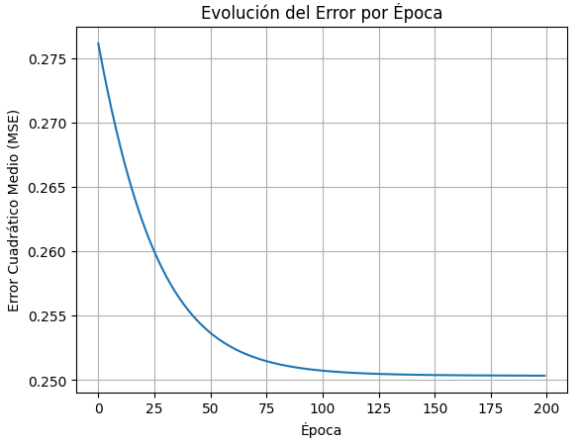


Tasa de aprendizaje 18.0





Tasa de aprendizaje = 25.0?



Tasa de aprendizaje = 0.1?

Descenso por Gradiente

```
optimizer = tf.keras.optimizers.SGD(learning_rate=4.5, momentum=0.0, nesterov=False)
history = model.fit(x, y, epochs=200, batch_size=len(x), verbose=2)
```

Usa todo el dataset para el cálculo del error

Ventajas:

1. La dirección del gradiente es exacta.

2. Convergencia más estable.

Desventajas:

1. Es muy costoso computacionalmente para datasets grandes.

2. Puede ser muy lento en converger, porque cada actualización de los pesos y sesgos requiere recorrer todos los datos de entrenamiento.

1 actualización por época → aprendizaje lento

Descenso por Gradiente Estocástico (SGD)

```
optimizer = tf.keras.optimizers.SGD(learning_rate=4.5)
history = model.fit(x, y, epochs=200)
```

o

```
history = model.fit(x, y, epochs=200, batch_size=m)
```

1<m<N ejemplos
#actualizaciones = N/m

Ejemplo: dataset de 1000 muestras con batch_size=100 → 10 actualizaciones por época.

Hace grupos de m ejemplos para calcular el error y actualizar pesos y sesgos.

- Sustituye el gradiente real (calculado a partir de todo el conjunto de datos) por una estimación del mismo (calculada a partir de un subconjunto de datos seleccionados aleatoriamente).

Ventajas:

1. Eficiencia computacional.
2. Aprendizaje en línea.

   Permite entrenar en escenarios donde los datos llegan en flujos (streaming), porque no necesita esperar a ver todo el dataset para actualizar.
3. Escalabilidad.

   Muy adecuado para datasets grandes y modelos profundos.

Desventajas:

1. Trayectorias ruidosas

   Como cada actualización se hace con pocos datos, la pérdida no decrece suavemente: da "saltos" irregulares y puede oscilar.
2. Convergencia más lenta o inestable

   Puede necesitar más épocas para converger, y a veces nunca estabilizarse en un mínimo exacto (se queda "rebotando" cerca).
3. Sensibilidad a la tasa de aprendizaje

   Requiere ajuste cuidadoso o esquemas adaptativos (schedule).

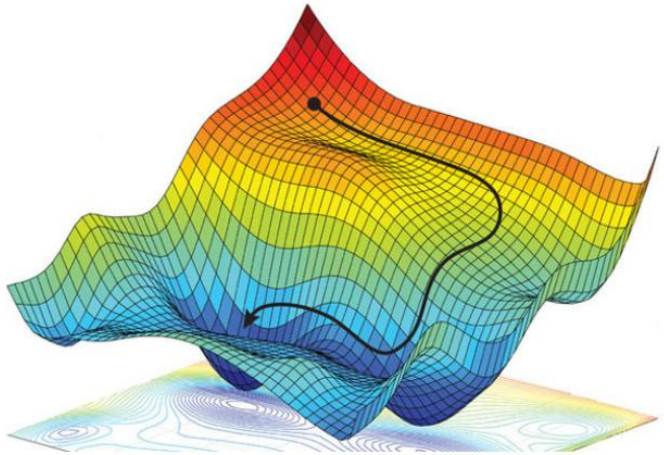
Descenso por Gradiente con momento

```
optimizer = tf.keras.optimizers.SGD(learning_rate=4.5, momentum=0.9, nesterov=False)
                                                                    # True para Nesterov Accelerated Gradient
```

- En el descenso por gradiente clásico, cada actualización depende únicamente del gradiente actual.
- Con **Momentum**, en lugar de movernos solo según el gradiente actual, acumulamos un "historial" de gradientes pasados, como si la actualización tuviera **inercia**.

Intuición

- Imagina una **bola rodando en un valle**:
  - El gradiente indica la pendiente inmediata.
  - El momentum le da a la bola **inercia**, de modo que sigue avanzando en la dirección previa, incluso si el gradiente cambia un poco.
- Esto permite:
  - **Avanzar más rápido** en aquellas direcciones en las que el gradiente se mantiene similar a lo largo de varias iteraciones.



El descenso por gradiente con momentum **combina el gradiente actual con un promedio de los anteriores**, lo que da un efecto de inercia que acelera la convergencia y estabiliza el entrenamiento.

Descenso por Gradiente con momento

```
optimizer = tf.keras.optimizers.SGD(learning_rate=4.5, momentum=0.9, nesterov=False)
                                                                    # True para Nesterov Accelerated Gradient
```

Ventajas:

1. Convergencia más rápida que GD puro.

2. Menos oscilaciones en superficies con valles estrechos.

3. Puede escapar más fácilmente de mínimos locales poco profundos.

Desventajas:

1. Añande un hiperparámetro adicional.

2. Si se ajusta mal, el algoritmo puede pasarse del mínimo en lugar de detenerse en él.

Nesterov:

Mejora el momentum calculando el gradiente "mirando un paso adelante"

Antes de calcular el gradiente, se da un **paso provisional en la dirección del momentum acumulado** y el gradiente se evalúa ahí.

Se considera una de las mejoras más efectivas al momentum clásico.

Descenso por Gradiente RMSProp

Root Mean Square Propagation

```
tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9 momentum=0.0)
```

Opcional

Controla cuánto pesan los gradientes pasados frente a los nuevos en el promedio móvil

Valores altos (ej. 0.9–0.99): se da más peso al historial → actualizaciones más suaves y estables

Valores bajos (ej. 0.5): se da más peso al gradiente actual → el optimizador responde más rápido, pero puede ser inestable.

Problema que soluciona

En **SGD puro** todos los parámetros comparten la misma tasa de aprendizaje.

- Si un gradiente es muy grande, el paso puede ser inestable.
- Si un gradiente es muy pequeño, el aprendizaje será demasiado lento.

Necesitamos un método que **ajuste automáticamente la magnitud del paso de cada parámetro**.

Consiste en **ajustar dinámicamente la tasa de aprendizaje para cada parámetro** dividiendo el gradiente entre la raíz cuadrada de la media móvil de los gradientes al cuadrado. Esto lo hace más robusto y eficiente que el descenso por gradiente estándar.

Adam

Adaptive Moment Estimation

Opcionales y estos son sus valores típicos.

```
tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
```

Momento

RMSProp-like

Es uno de los métodos más populares en *deep learning* porque combina lo mejor de **Momentum** y **RMSProp**.

Problema que soluciona

El descenso por gradiente clásico tiene dos problemas:

- **Gradientes inestables** (pueden ser muy grandes o muy pequeños según el parámetro).
- **Sensibilidad al learning rate**: si no eliges bien el valor, el entrenamiento puede ser muy lento o divergir.

Adam soluciona esto **ajustando la tasa de aprendizaje de manera adaptativa para cada parámetro** y además introduciendo inercia con momentum.

Adam

Adaptive Moment Estimation

Opcionales y estos son sus valores típicos.

```
tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
```

Momento

RMSProp-like

Ventajas:

1. Funciona bien sin mucho ajuste de hiperparámetros.

2. Convergencia rápida en muchos problemas.

3. Escala bien con datos grandes y redes profundas.

4. Generalmente más eficiente que el SGD puro.

Desventajas:

1. Puede **aprender demasiado rápido y adaptarse demasiado bien a los datos de entrenamiento (overfit/sobreajuste)**, en comparación con optimizadores más "conservadores",

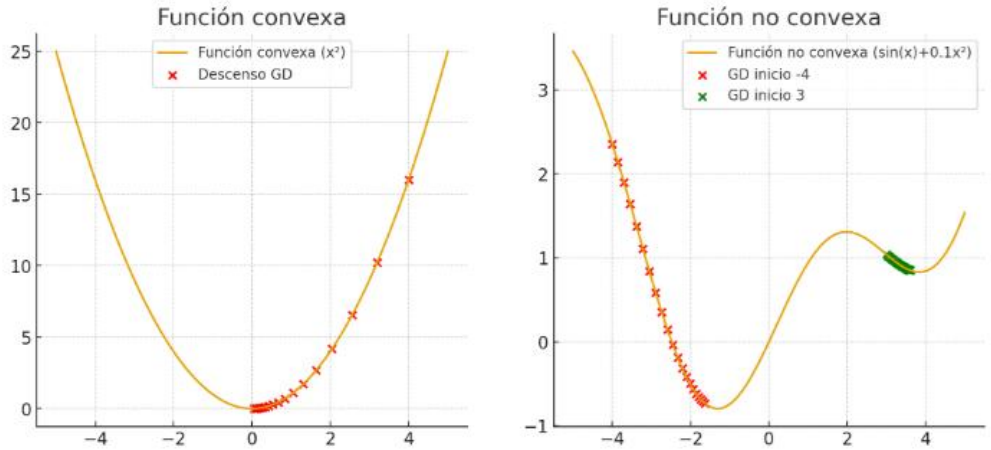
2. A veces da soluciones que generalizan peor que SGD clásico en visión por computadora, aunque converge más rápido.

Problemas de Backpropagation

1. Mínimos locales y puntos silla

- En funciones del error (pérdida) no convexas (como en redes neuronales profundas):
  - el gradiente puede quedarse atorado en mínimos locales o en punto silla (donde la pendiente es casi nula, pero no es un mínimo).
  - Esto retrasa o impide encontrar una buena solución global.
  - La convergencia depende fuertemente de la inicialización de los pesos, la tasa de aprendizaje

Una función de pérdida no convexa es aquella que **no garantiza encontrar el mínimo global fácilmente** porque su superficie tiene una geometría complicada.



- Tienen múltiples mínimos locales (pero no todos son óptimos globales),
- Existen puntos silla (lugares en donde la pendiente es cero, pero no son mínimos ni máximos), y pueden atrapar al algoritmo.
- Hay mesetas o regiones planas (gradientes muy pequeños) que hacen que el entrenamiento avance muy lento.

Problemas de Backpropagation

2. Explosión del gradiente

Es un problema que ocurre durante el entrenamiento de redes neuronales profundas cuando los **valores de los gradientes se vuelven extremadamente grandes** al propagarse hacia atrás en el algoritmo de **backpropagation**.

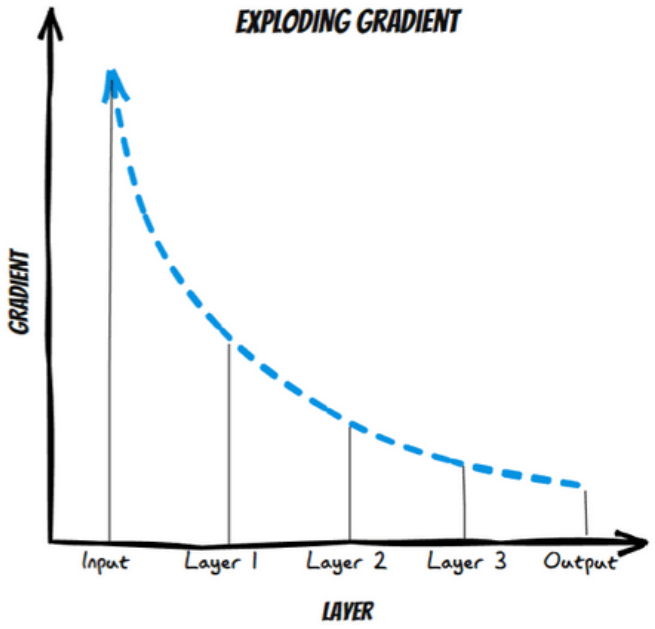
¿Por qué ocurre?

Durante backpropagation, los gradientes se calculan usando la **regla de la cadena**.

En redes muy profundas o con pesos grandes:

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial h^{(L)}} \cdot \frac{\partial h^{(L)}}{\partial h^{(L-1)}} \cdot \cdots \cdot \frac{\partial h^{(1)}}{\partial W^{(1)}}$$

- Cada derivada puede ser mayor que 1.
- Al multiplicarse muchas veces, los valores pueden crecer exponencialmente.
- Esto da lugar a gradientes enormes.



Consecuencias

- **Actualizaciones inestables:** los pesos cambian demasiado bruscamente.
- **Divergencia:** la pérdida no disminuye, incluso puede crecer indefinidamente.
- **Valores NaN:** si los gradientes o pesos llegan a infinito o a números no representables, el entrenamiento

Problemas de Backpropagation

2. Explosión del gradiente

Es un problema que ocurre durante el entrenamiento de redes neuronales profundas cuando los **valores de los gradientes se vuelven extremadamente grandes** al propagarse hacia atrás en el algoritmo de **backpropagation**.

Dónde es común

- En redes **muy profundas**.
- En **RNNs/LSTMs**, donde los gradientes se multiplican a lo largo de muchas secuencias de tiempo.

Estrategias para mitigarlo

1. **Gradient clipping** (recorte de gradientes):
   - Limitar la magnitud de los gradientes. Ejemplo en Keras:

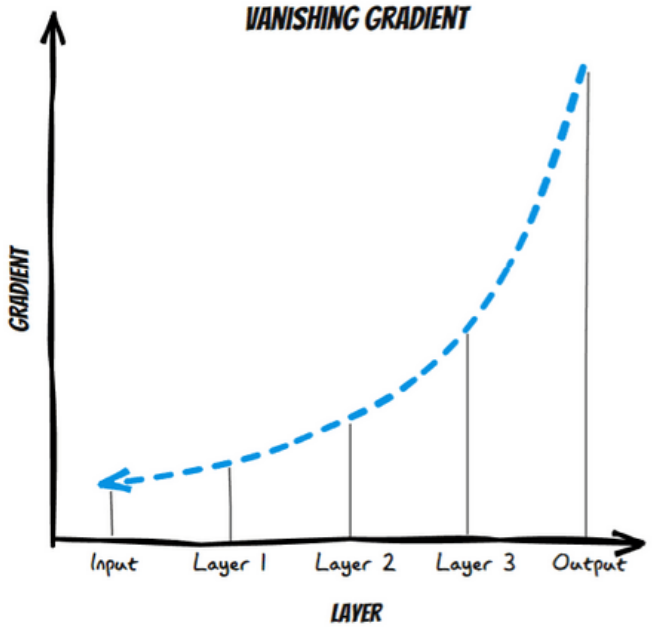
```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, clipnorm=1.0)
```

2. **Inicialización adecuada de pesos** (Xavier, He initialization).
3. **Normalización** (Batch Normalization, Layer Normalization).
4. **Optimizadores adaptativos** (Adam, RMSProp), que ajustan la escala de los gradientes.
5. **Reducir learning rate** para que los pasos no sean tan agresivos.

Problemas de Backpropagation

3. Desvanecimiento del gradiente

Es un problema que ocurre cuando, durante el entrenamiento de redes neuronales profundas, los gradientes que se propagan hacia las primeras capas se vuelven **muy pequeños** (tienden a cero).
Como consecuencia, los pesos de esas capas apenas cambian y dejan de aprender.



¿Por qué ocurre?

Se debe principalmente a la **regla de la cadena en backpropagation**:

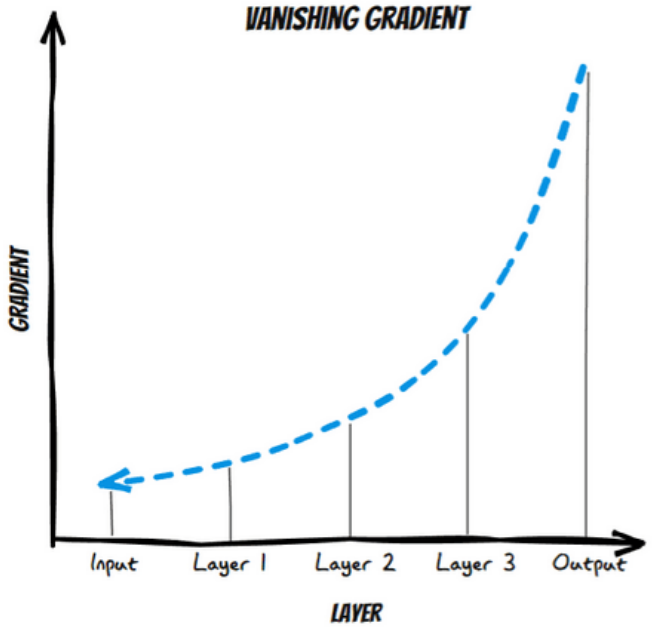
$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial h^{(L)}} \cdot \frac{\partial h^{(L)}}{\partial h^{(L-1)}} \cdot \cdots \cdot \frac{\partial h^{(2)}}{\partial h^{(1)}} \cdot \frac{\partial h^{(1)}}{\partial W^{(1)}}$$

- Si cada derivada intermedia es **menor que 1** (ej. activaciones sigmoide o tanh), al multiplicarlas repetidamente el resultado se vuelve **exponencialmente pequeño**.
- En redes profundas o RNNs con largas secuencias, este efecto se acumula y el gradiente que llega a las primeras capas es casi cero.

Problemas de Backpropagation

3. Desvanecimiento del gradiente

Es un problema que ocurre cuando, durante el entrenamiento de redes neuronales profundas, los gradientes que se propagan hacia las primeras capas se vuelven **muy pequeños** (tienden a cero).
Como consecuencia, los pesos de esas capas apenas cambian y dejan de aprender.



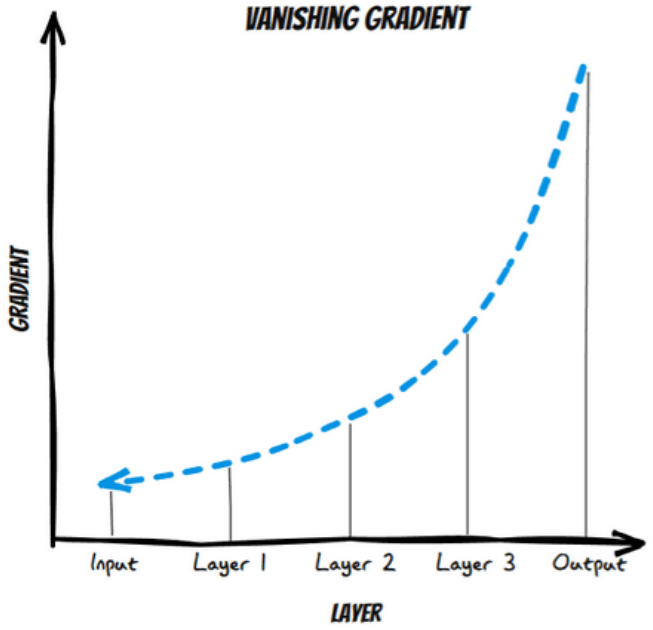
Consecuencias

1. **Capas iniciales dejan de aprender** → solo las capas cercanas a la salida se ajustan.
2. **Entrenamiento extremadamente lento** → aunque aumentes épocas, el modelo no mejora.
3. **Modelos con pobre representación** → porque las primeras capas no logran aprender características útiles.

Problemas de Backpropagation

3. Desvanecimiento del gradiente

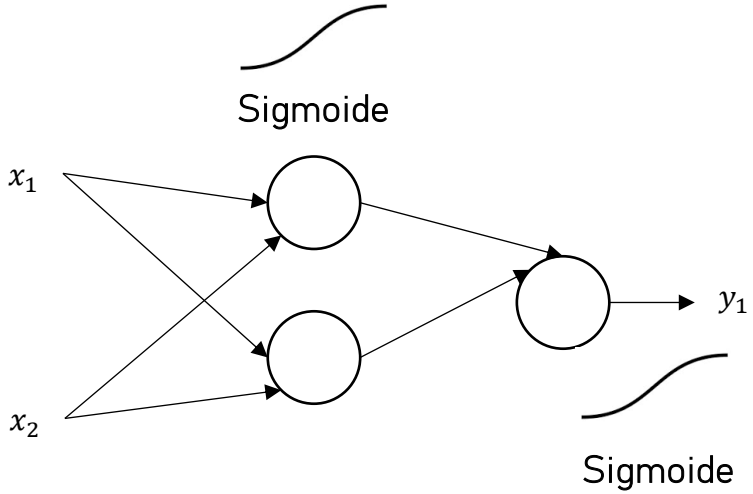
Es un problema que ocurre cuando, durante el entrenamiento de redes neuronales profundas, los gradientes que se propagan hacia las primeras capas se vuelven **muy pequeños** (tienden a cero).
Como consecuencia, los pesos de esas capas apenas cambian y dejan de aprender.



Dónde es común

- **Redes profundas** con muchas capas ocultas.
- **RNNs/LSTMs** en secuencias largas (por eso surgieron LSTM y GRU, diseñadas específicamente para combatir este problema).
- Cuando se usan **funciones de activación sigmoide o tanh**, que saturan y generan derivadas muy pequeñas.

Funciones de activación



Hiperparámetros

Relacionados con la arquitectura de la red

Hiperparámetro	Rango típico / Valores sugeridos	Comentarios
Número de capas ocultas	1–5 para problemas simples, más para problemas complejos	Más capas aumentan capacidad, pero también riesgo de sobreajuste.
Número de neuronas por capa	4–512	Depende de la complejidad del problema y cantidad de datos. Se recomienda aumentar gradualmente y validar rendimiento.
Función de activación	`sigmoid`, `tanh`, `ReLU`, `Leaky ReLU`, `ELU`	ReLU es más eficiente para redes profundas; sigmoide/tanh se usan en redes pequeñas o salidas binarias.

Funciones de activación

- Son operadores diferenciables para transformar señales de entrada en salidas.
- La mayoría de ellos añaden no linealidad.
- Sin funciones de activación, las redes neuronales solo se enfocarían en operaciones lineales.
  - No distinguiría patrones complejos como el XOR, que son no linealmente separables.
  - En la práctica, sería equivalente a una regresión lineal (aunque con más parámetros inútilmente).

En una red neuronal, cada capa aplica:

$$h = Wx + b$$

Si no agregamos una función de activación, la siguiente capa recibe:

$$h' = W'h + b'$$

Sustituyendo:

$$h' = W'(Wx + b) + b' = (W'W)x + (W'b + b')$$

→ sigue siendo **una transformación lineal de la entrada.**

Si repites esto con muchas capas lineales apiladas, al final obtienes:

$$y = W^*x + b^*$$

que sigue siendo una **única transformación lineal**, por más capas que agregues.

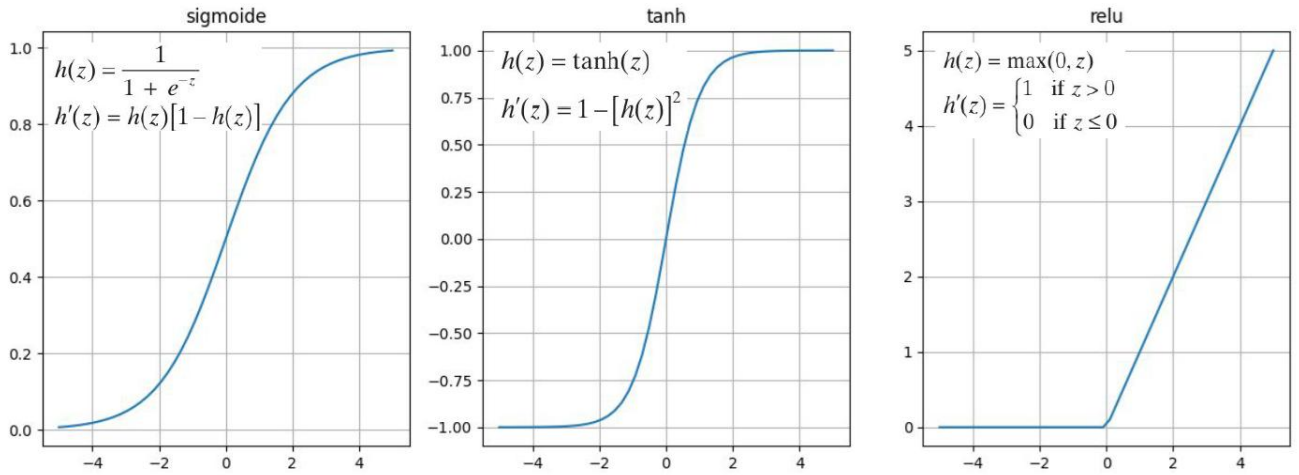
Funciones de activación

- Son operadores diferenciables para transformar señales de entrada en salidas.
- La mayoría de ellos añaden no linealidad.
- Sin funciones de activación, las redes neuronales solo se enfocarían en operaciones lineales.
  - No distinguiría patrones complejos como el XOR, que son **no linealmente separables**.
  - En la práctica, sería equivalente a una **regresión lineal** (aunque con más parámetros inútilmente).

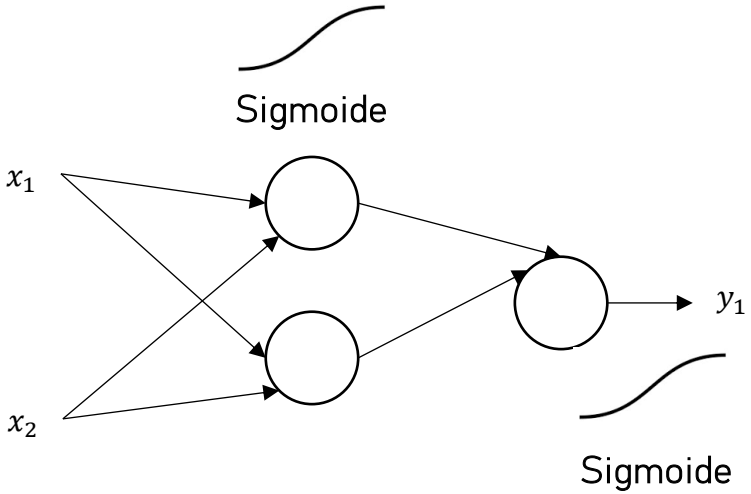
Las funciones de activación introducen **no linealidad** en cada capa:

$$h = f(Wx + b)$$

- Con funciones como **ReLU, sigmoide, tanh**, el modelo puede combinar hiperplanos lineales de forma no lineal.
- Eso le permite aproximar **cualquier función continua** (Teorema de aproximación universal).



Funciones de pérdida/error/costo



Hiperparámetros

Relacionados con el entrenamiento

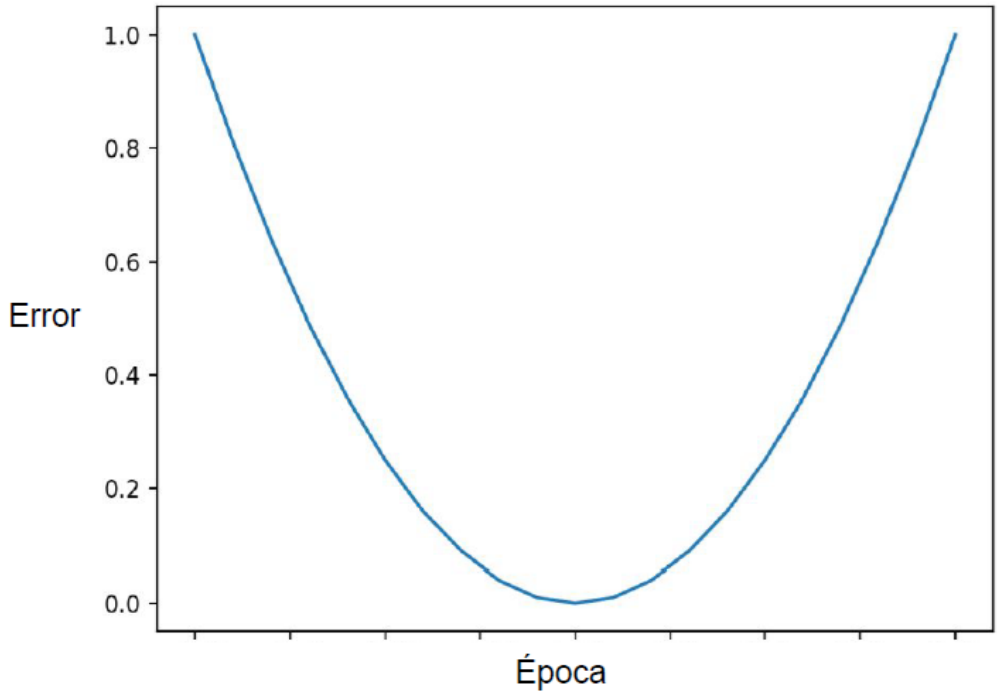
Función de pérdida	`binary_crossentropy`, `categorical_crossentropy`, `mean_squared_error`	Escoge según la tarea: clasificación binaria, multiclase o regresión.
Optimizador	`SGD`, `Adam`, `RMSProp`	Adam suele ser la opción más estable y rápida para la mayoría de problemas.
Tasa de aprendizaje (learning rate)	0.001–0.5	Depende del optimizador y tamaño de red; learning rate demasiado alto puede hacer que no converja.
Épocas (epochs)	50–1000	Depende del dataset; usar early stopping ayuda a evitar sobreentrenamiento.
Tamaño de batch (batch size)	4–256	Pequeños → más ruido, exploración; grandes → aprendizaje más estable, pero mayor memoria.

Funciones de pérdida/error/costo

Es una función matemática que mide **qué tan bien o mal está funcionando el modelo** comparando sus predicciones ($\hat{y}$) con las salidas reales (y).

$$L(y, \hat{y}) \quad = \quad \text{error entre salida esperada y salida del modelo}$$

El objetivo del entrenamiento es **minimizar esta función de pérdida** ajustando los parámetros (pesos y sesgos) del modelo.



Bloques de construcción de Redes Neuronales

Bloques matemáticos fundamentales

1. **Neuronas artificiales**
   - Cada neurona aplica una transformación lineal seguida de una no linealidad:

$$h = f(Wx + b)$$

2. **Funciones de activación**
   - Introducen la **no linealidad** necesaria para que la red modele funciones complejas.
   - Ejemplos: sigmoide, tanh, ReLU, Leaky ReLU, GELU.
3. **Funciones de pérdida (loss)**
   - Definen el objetivo a optimizar (ej. MSE en regresión, cross-entropy en clasificación).
4. **Algoritmos de optimización**
   - Usan los gradientes para actualizar los pesos.
   - Ejemplos: SGD, Momentum, RMSProp, Adam.
5. **Backpropagation**
   - El mecanismo que calcula los gradientes de la pérdida respecto a cada peso.

Bloques de construcción de Redes Neuronales

Bloques arquitectónicos básicos

1. **Capas densas (fully connected)**
   - El "bloque universal" de MLPs (Perceptrones Multicapa).
2. **Capas convolucionales (CNNs)**
   - Extraen patrones espaciales, útiles en imágenes.
3. **Capas recurrentes (RNN, LSTM, GRU)**
   - Modelan dependencias temporales o secuenciales.
4. **Normalización**
   - Ej. BatchNorm, LayerNorm → estabilizan los gradientes.
5. **Dropout y regularización**
   - Bloques que ayudan a reducir el sobreajuste.
6. **Atención (Attention blocks)**
   - Bloques que permiten ponderar dinámicamente la importancia de cada entrada (clave en Transformers).