Tema 3: Entrenamiento de redes neuronales profundas

Aprendizaje
Profundo

Grado en Ingeniería y Ciencia de datos (Universidad de Oviedo)

Pablo González, Pablo Pérez
{gonzalezgpablo, pabloperez}@uniovi.es
Centro de Inteligencia Artificial, Gijón

Proceso básico de entrenamiento de una red

Proceso básico de entrenamiento de una red

Entrenar una red neuronal consiste en ajustar los pesos de la red mediante la presentación repetida de datos de entrenamiento, de modo que la red pueda aprender a realizar predicciones y generalizar sobre datos no vistos anteriormente. Para poder entrenar una red neuronal primero debemos:

- **Preparar el conjunto de datos** y realizar una separación en un conjunto de **entrenamiento**, de **validación** y de **test**, realizando un preprocesado de los datos si es necesario.
- Definir una **arquitectura** para la red.
- Definir la **función de pérdida** a utilizar que sea adecuada para nuestro problema.
- Elegir un **optimizador** a utilizar para definir como se actualizarán los pesos en la red en el proceso de entrenamiento.
- **Decidir cuando parar** el proceso de entrenamiento y elegir el mejor modelo hasta el momento.

Entrenamiento de una red: separación de los datos

Conjunto de entrenamiento (Training set)

El conjunto de entrenamiento es utilizado para ajustar los parámetros del modelo durante el proceso de entrenamiento.

Conjunto de validación (Validation set)

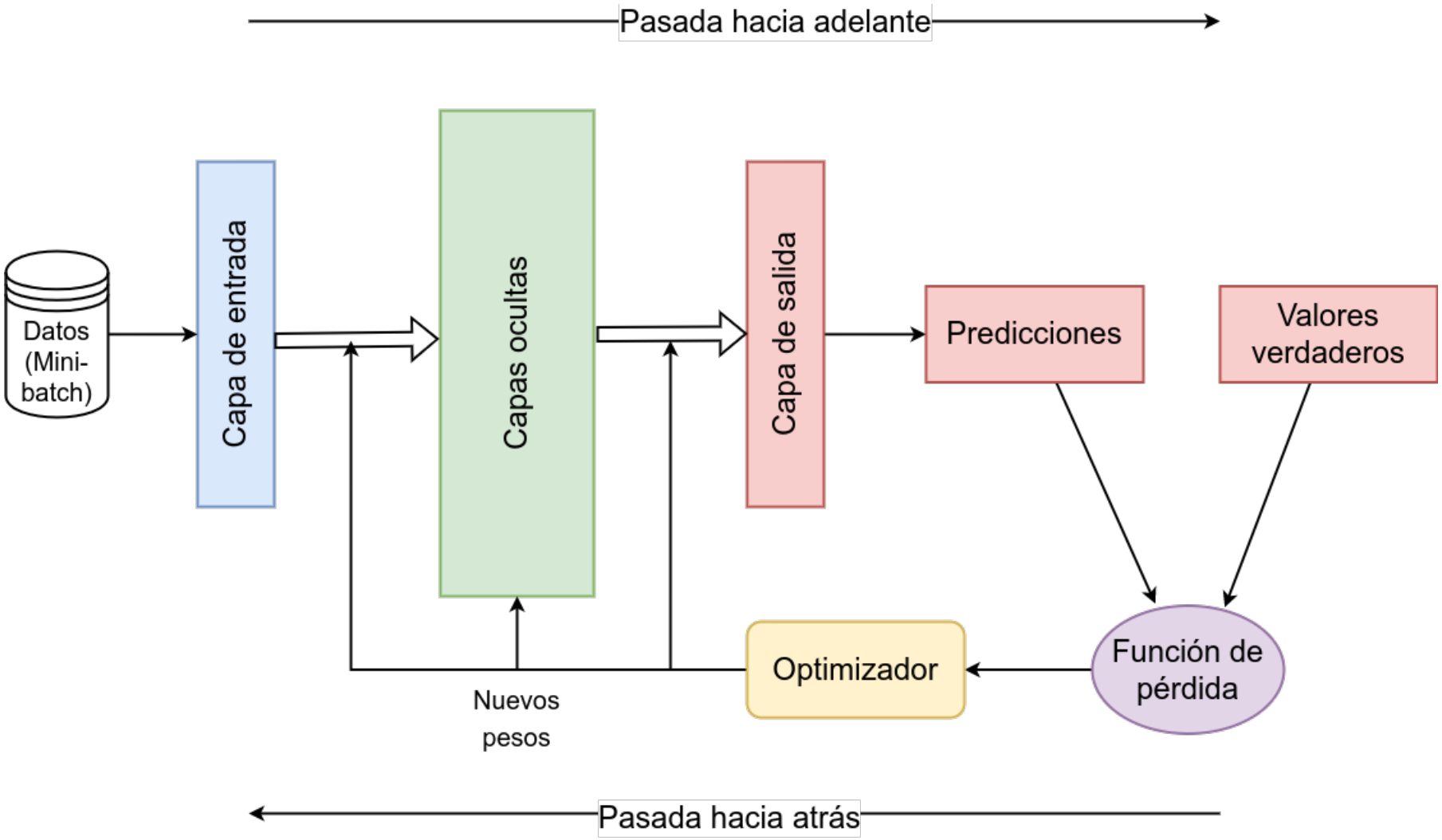
El conjunto de validación se utiliza para evaluar el rendimiento del modelo durante el entrenamiento y ajustar los hiperparámetros. No se utiliza para ajustar los parámetros del modelo. Ayuda a detectar el sobreajuste, a seleccionar el mejor modelo y a decidir cuando parar.

Conjunto de prueba (Test set)

El conjunto de prueba se utiliza para evaluar el rendimiento final del modelo después de que el entrenamiento ha finalizado. Representa datos no vistos por el modelo y proporciona una medida objetiva del rendimiento en situaciones del mundo real.

Entrenamiento de una red

El proceso de entrenamiento de una red se puede resumir en el siguiente gráfico:



Entrenamiento de una red: ciclo completo

De manera general se realizan los siguientes pasos al entrenar una red:

- Se van realizando iteraciones de actualizaciones de pesos, pasando los datos del conjunto de entrenamiento en lotes hasta completar una **época**. Una época se refiere a cuando la red ha visto todos los ejemplos del conjunto de entrenamiento.
- Una vez finalizada la época, se evalúan los ejemplos del conjunto de validación. Esto puede hacerse cada una o varias épocas. Así tendremos el **error en validación**.
- Con el error en validación podremos analizar aspectos como el **sobreajuste del modelo** o su capacidad de **generalización**. Utilizamos este error para decidir cuando parar de entrenar la red (parada temprana o early-stopping).
- Una vez parado el entrenamiento, probaremos el modelo en el conjunto de test para tener una estimación final de su rendimiento.

Funciones de pérdida

Elección de la función de pérdida

La **función de pérdida** reduce todos los aspectos positivos y negativos de un sistema posiblemente complejo a **un único número**, un valor escalar, que permite clasificar y comparar las soluciones candidatas.

Es importante que la función de pérdida **disminuya cuando el error cometido por el modelo sea menor**, ya que el algoritmo de entrenamiento de las redes neuronales tratará de minimizar la misma.

Importancia

Es importante, por tanto, que la función represente fielmente nuestros objetivos de diseño. Si hacemos una mala elección de la función de pérdida, obtendremos un modelo insatisfactorio aun cuando nuestra red converja. Es por tanto una **decisión muy importante en el diseño de nuestra red** y su proceso de entrenamiento.

Clasificación de las funciones de pérdida

Las funciones de pérdida compararán la salida esperada de la red y con la salida predicha $\hat{y}$. Las funciones de pérdida más usadas se pueden clasificar en los siguientes tipos:

- Clasificación
  - Binary Cross-Entropy Loss
  - Categorical Cross-Entropy Loss
- Regresión
  - Error Absoluto Medio (MAE)
  - Error Cuadrático Medio (MSE)
  - Error Porcentual Absoluto Medio (MAPE)

Binary Cross-Entropy Loss

Binary Cross-Entropy Loss se utiliza comúnmente en el aprendizaje profundo para tareas de clasificación binaria. Está basada en el concepto de **entropía cruzada** y mide la disimilitud entre dos distribuciones de probabilidad. En este caso, entre las probabilidades predichas $\hat{\mathbf{y}}$ (valor real entre 0 y 1) y las etiquetas verdaderas $\mathbf{y}$ (0 o 1).

Se define de la siguiente manera:

$$\text{BCELoss}(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Binary Cross-Entropy Loss

Binary Cross-Entropy Loss se utiliza comúnmente en el aprendizaje profundo para tareas de clasificación binaria. Está basada en el concepto de **entropía cruzada** y mide la disimilitud entre dos distribuciones de probabilidad. En este caso, entre las probabilidades predichas $\hat{\mathbf{y}}$ (valor real entre 0 y 1) y las etiquetas verdaderas $\mathbf{y}$ (0 o 1).

Se define de la siguiente manera:

$$\text{BCELoss}(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

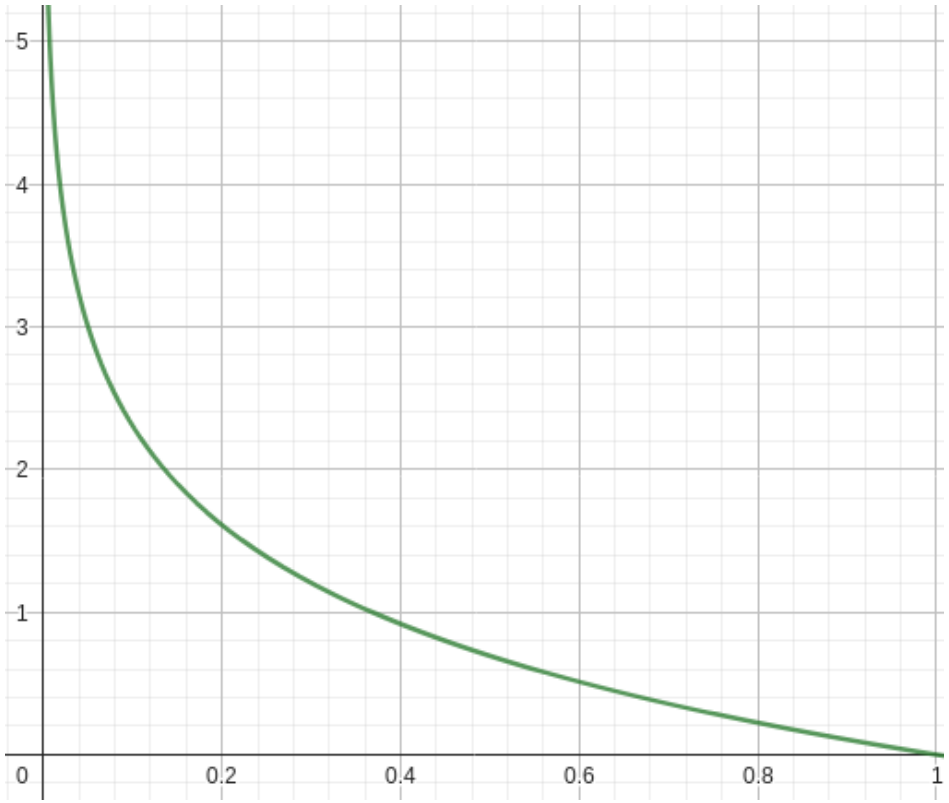
Es importante destacar lo siguiente:

- Cuando $y_i = 0$ solo la segunda parte se tiene en cuenta $-\log(1 - \hat{y}_i)$
- Cuando $y_i = 1$ solo la primera parte se tiene en cuenta $-\log(\hat{y}_i)$.

Binary Cross-Entropy Loss

$$\text{BCELoss}(\hat{\mathbf{y}}, \mathbf{y}) = -[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Función $-\log(x)$:



Binary Cross-Entropy Loss

Como podemos ver anteriormente, la función Binary Cross-Entropy Loss penaliza de manera muy fuerte cuando el modelo se equivoca de manera clara:

- Cuando $y = 0$ pero $\hat{y}$ está cerca de 1.
- Cuando $y = 1$ pero $\hat{y}$ está cerca de 0.

Importante

Recuerda que para que esta función de pérdida funcione correctamente los valores predichos por la red $\hat{y}$ tienen que estar entre 0 y 1. Normalmente esto se consigue utilizando una función **sigmoide** antes de la salida de la red.

Binary Cross-Entropy Loss

Algunas propiedades interesantes de la función Binary Cross-Entropy Loss son las siguientes:

- Es una función **diferenciable**, con lo cual es puede usar en propagación hacia atrás.
- Da una **interpretación probabilística** del modelo, de modo que la salida $\hat{y}$ puede ser interpretada como la confianza del modelo en que un ejemplo es positivo o no.
- Favorece **probabilidades bien calibradas**, penalizando mucho probabilidades cercanas a 0 o 1 si el modelo no está totalmente seguro de acertar.
- Realiza un **tratamiento simétrico de ambas clases** (positiva y negativa), por tanto es útil incluso en problemas no balanceados.

Categorial Cross-Entropy Loss

Categorical Cross-Entropy Loss es una **generalización** del Binary Cross-Entropy Loss **para el caso multiclase**.
La fórmula para Categorical Cross-Entropy Loss es la siguiente:

$$CE = -\sum_{i=1}^{c} y_i log(\hat{y}_i)$$

donde:

- c es el número de clases
- y_i es el valor del vector one-hot para la clase i (solo tendremos un 1).
- $\hat{y}_i$ es la probabilidad devuelta para la clase i-ésima.

Importante

Ten en cuenta que en frameworks como PyTorch, la entrada esperada para esta función no es la salida de una función **softmax**, sino la salida de la última capa directamente. Softmax es aplicado por defecto dentro de la función de pérdida.

Categorial Cross-Entropy Loss: Ejemplo

Supongamos que tenemos un problema con tres clases: coches, motos y aviones.

- Si tenemos una imagen de un coche, podemos codificar su etiqueta usando **one-hot-encoding** como $[1, 0, 0]$.
- Digamos que la salida de nuestra red es $[0.8, 0.1, 0.1]$, lo que nos está indicando que la red cree que el ejemplo es un coche con una probabilidad de 0.8.
- La función de pérdida en este caso (de manera similar que para el caso binario), tendrá el valor $-log(0.8) = 0.223$. Un valor bajo ya que la red ha predicho correctamente que el ejemplo era un coche.
- Si por el contrario la salida fuese $[0.1, 0.1, 0.8]$, el valor sería $-log(0.1) = 2.3$

Cross-Entropy Loss: Interpretación

Es importante entender como se interpretan los valores devueltos por estas funciones de pérdida:

- En estas funciones de pérdida, valores cercanos a cero indican que el modelo está devolviendo probabilidades muy cercanas a las reales.
- Por otro lado, valores próximos a 1 indican que el modelo no está funcionando bien.
- Si tenemos valores mayores que uno, las probabilidades devueltas son muy malas y probablemente tengamos algún error en nuestro código.

Importante

Las funciones de pérdida se calculan para todo el lote (**batch**) y su valor es la media de los valores individuales para cada ejemplo.

Error Absoluto Medio (MAE)

El error absoluto medio (MAE) es una medida de **regresión**. En este caso tanto la salida esperada y como la predicha $\hat{y}$ serán dos **valores reales**. La fórmula para calcular el MAE es:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

donde:

- n es el número total de ejemplos.
- y_i es el valor real del ejemplo i.
- $\hat{y}_i$ es el valor predicho por el modelo para el ejemplo i.

Interpretación

Es una medida fácilmente interpretable ya que nos da una magnitud del error cometido en la **misma escala** que la variable objetivo. Cuanto más cerca se encuentre de cero mejor será el modelo.

Error Cuadrático Medio (MSE)

El error cuadrático medio (MSE) es otra medida de **regresión**.

La fórmula para calcular el MSE es:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

donde:

- n es el número total de ejemplo.
- y_i es el valor real del ejemplo i.
- $\hat{y}_i$ es el valor predicho por el modelo para el ejemplo i.

Interpretación

En MSE, al elevar los errores al cuadrado, **se penalizan más los errores grandes**. Un valor de MSE más bajo indica una mayor precisión del modelo.

Error Porcentual Absoluto Medio (MAPE)

El Error Porcentual Absoluto Medio (MAPE) representa el promedio de los errores porcentuales absolutos entre los valores reales y las predicciones del modelo.
La fórmula para calcular el MAPE es:

$$MAPE = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right| \times 100$$

donde:

- n es el número total de ejemplo.
- y_i es el valor real del ejemplo i.
- $\hat{y}_i$ es el valor predicho por el modelo para el ejemplo i.

Interpretación

El MAPE proporciona una medida de la precisión **relativa** del modelo en términos de errores porcentuales. Un valor más bajo indica una mayor precisión del modelo, ya que los errores porcentuales son menores.

Funciones de pérdida para regresión: ejemplo

Ejemplo

Supongamos que queremos crear un modelo para predecir el volumen de ventas de un determinado producto para un día concreto.

Día	Ventas reales (y)	Ventas predichas $\hat{y}$
1	90	100
2	50	25
3	30	28
4	60	75

$$MAE = \frac{|90-100| + |50-25| + |30-28| + |60-75|}{4} = 13$$

$$MSE = \frac{(90-100)^2 + (50-25)^2 + (30-28)^2 + (60-75)^2}{4} = 238.5$$

$$MAPE = \left[\left|\frac{90-100}{90}\right| + \left|\frac{50-25}{50}\right| + \left|\frac{30-28}{30}\right| + \left|\frac{60-75}{60}\right|\right]\frac{100}{4} = 23.19\%$$

Algoritmos de optimización

Algoritmos de optimización: repaso

En secciones previas hemos hablado de el **descenso de gradiente** como algoritmo para optimizar los pesos de una red con respecto a una función de pérdida. Del descenso de gradiente habíamos identificado tres tipos:

- Descenso de gradiente por lotes
- Descenso de gradiente por mini-lotes
- Descenso de gradiente estocástico

La actualización de pesos en esta familia de algoritmos se puede resumir como:

$$\theta_{t+1} = \theta_t - \eta \left(\frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t) \right)$$

donde:

- θ_t representa un peso de la red en un instante determinado
- η es el learning rate
- $\nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t)$ representa el gradiente para el peso θ sobre un ejemplo i determinado.

Algoritmos de optimización: repaso

$$\theta_{t+1} = \theta_t - \eta \left(\frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t) \right)$$

Recuerda que la idea consistía en seguir la **dirección opuesta a la apuntada por el gradiente**.

Por otra parte, las tres variantes de este algoritmo listadas anteriormente se diferencian en los ejemplos que usan en cada iteración para calcular el gradiente:

- Descenso de gradiente por lotes: todo el conjunto de entrenamiento.
- Descenso de gradiente por mini-lotes: un subconjunto de ejemplos del conjunto de entrenamiento.
- Descenso de gradiente estocástico: un único ejemplo aleatorio.

Descenso del gradiente basado en Momento

Conocido en inglés como **Momentum-Based Gradient Descent** (MBGD). Partiendo de la formulación anterior:

$$\theta_{t+1} = \theta_t - update_t,$$

donde $update_t$ sería la actualización basada en el learning rate y el gradiente, podemos incorporar la noción de momento de la siguiente manera:

$$update_t = \gamma update_{t-1} + \eta \left(\frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t) \right),$$

Intuición del concepto de momento

Si me piden repetidamente que me mueva en la misma dirección, probablemente debería ganar algo de confianza y empezar a dar pasos más grandes en esa dirección. Igual que una pelota coge impulso al rodar por una pendiente.

Descenso del gradiente basado en Momento

$$update_t = \gamma update_{t-1} + \eta \left(\frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t) \right),$$

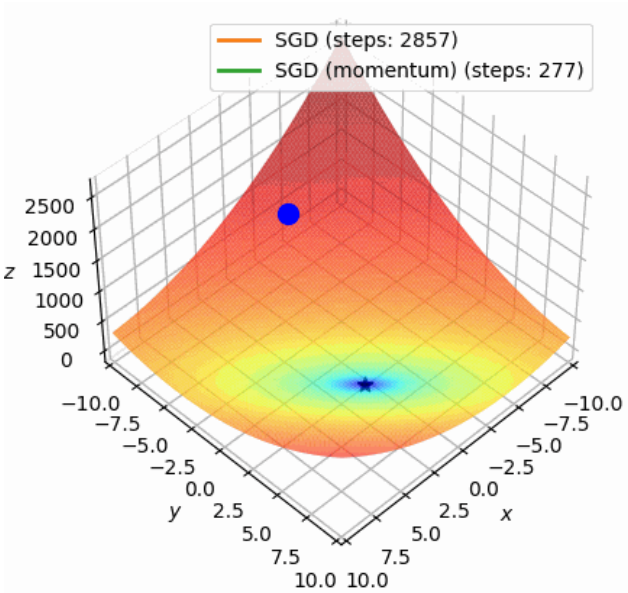
El descenso del gradiente basado en momento tiene las siguientes características:

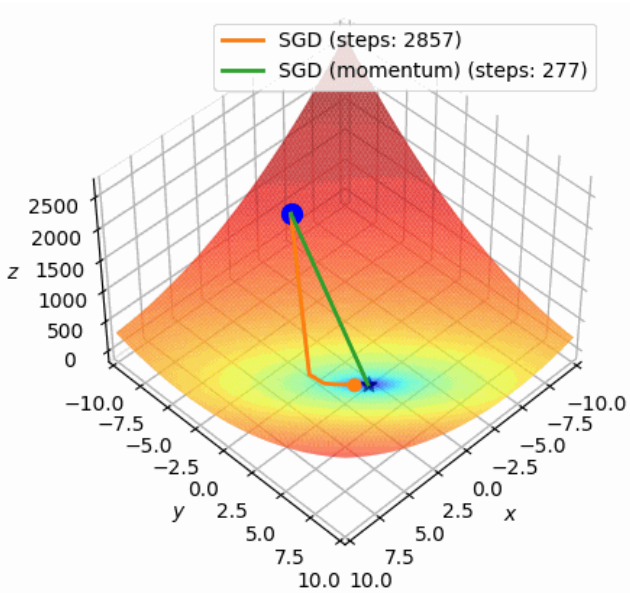
- La actualización de los pesos no depende solo del gradiente, sino que **depende también de las actualizaciones anteriores**.
- En regiones con gradientes consistentes permite al algoritmo ir **más rápido**.

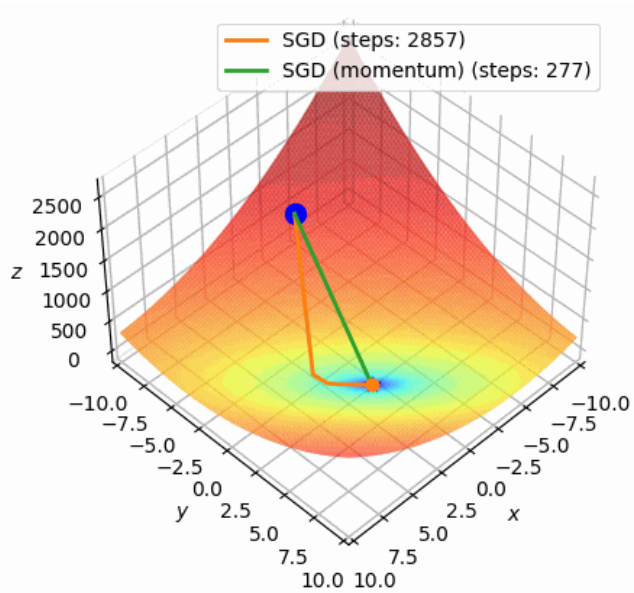
Importante

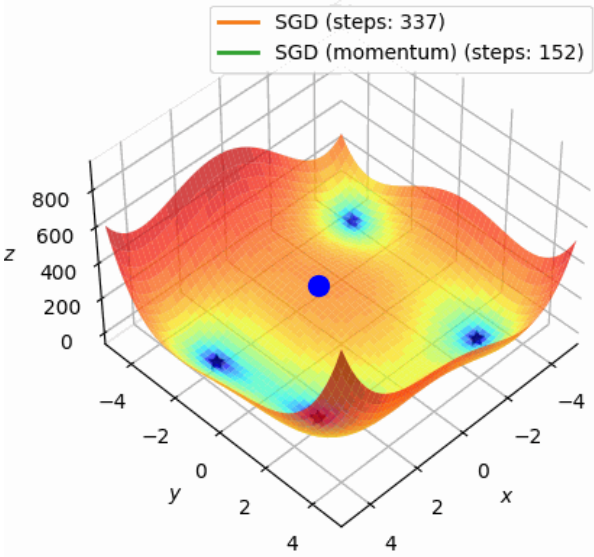
Los Frameworks de aprendizaje profundo más conocidos ya implementan el descenso del gradiente con momento por defecto.

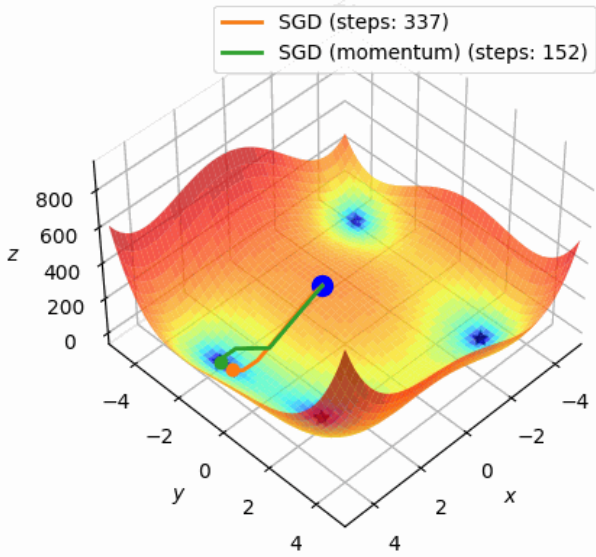
Descenso del gradiente basado en Momento

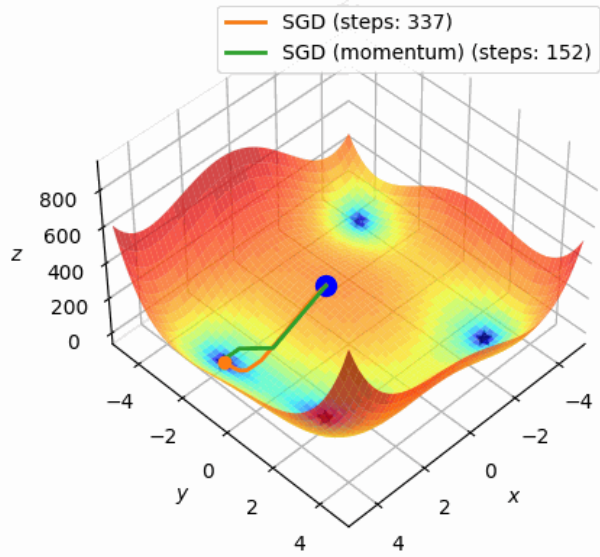












Algoritmos de optimización: AdaGrad

El descenso de gradiente tiene un parámetro crítico que es la tasa de aprendizaje, learning rate (η). Además, **este parámetro es compartido por todos los pesos de la red**. La idea de AdaGrad es utilizar un learning rate por cada peso de la red, basado en la historia de los gradientes calculados para ese peso:

$$\theta_{t+1} = \theta_t - \eta' g_t$$

$$g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t)$$

$$\eta' = \frac{\eta}{\sqrt{\epsilon + G_t}},$$

donde ϵ es un número pequeño para evitar la división por cero y G_t es la acumulación de todos los gradientes al cuadrado acumulados hasta ahora para este parámetro en concreto:

$$G_t = G_{t-1} + g_t^2$$

Algoritmos de optimización: AdaGrad

$$\eta' = \frac{\eta}{\sqrt{\epsilon + G_t}},$$

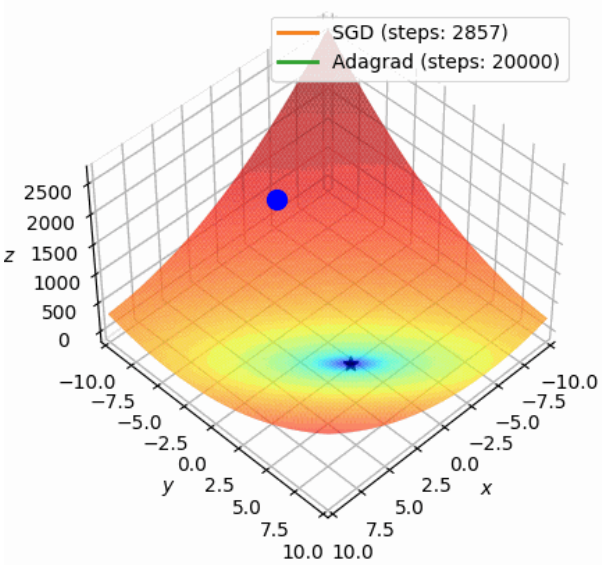
Ventajas de AdaGrad:

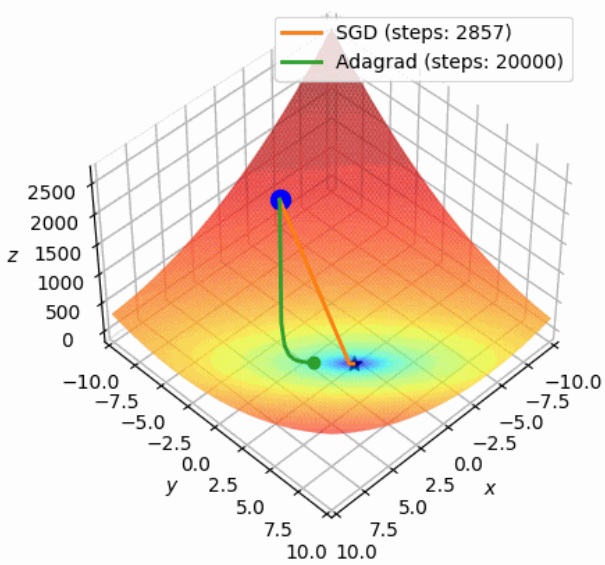
- Puede manejar gradientes grandes, reduciendo el learning rate para parámetros que son actualizados frecuentemente.
- El learning rate inicial η se convierte en este algoritmo en un parámetro menos importante.

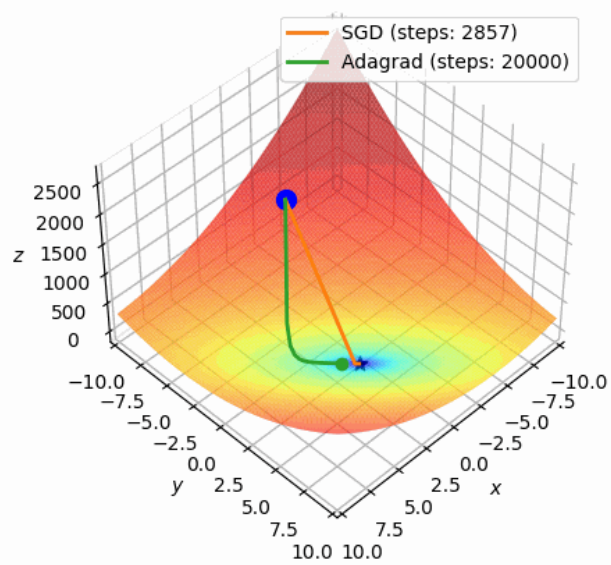
Problemas

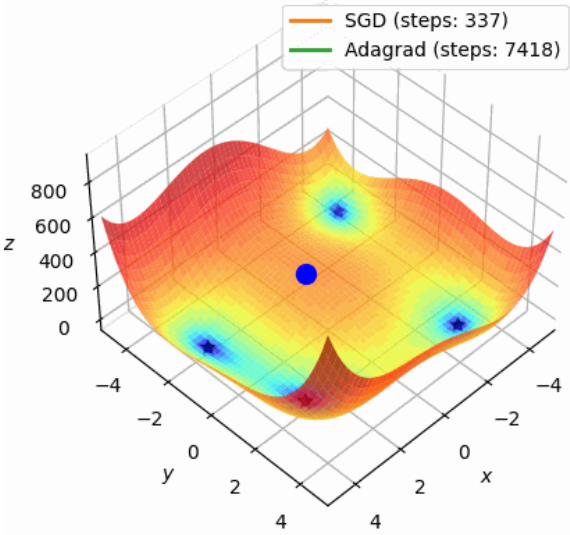
Uno de los problemas de AdaGrad es que según el entrenamiento progresa, el denominador se vuelve mayor (hay más gradientes acumulados), y por tanto corremos el riesgo de que el learning rate para ese parámetro sea demasiado pequeño para aprender nada. Este problema lo resuelve el algoritmo de optimización **RMSProp**.

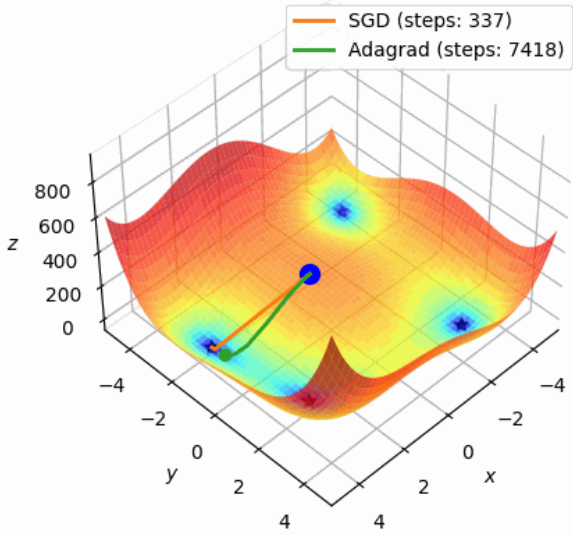
AdaGrad

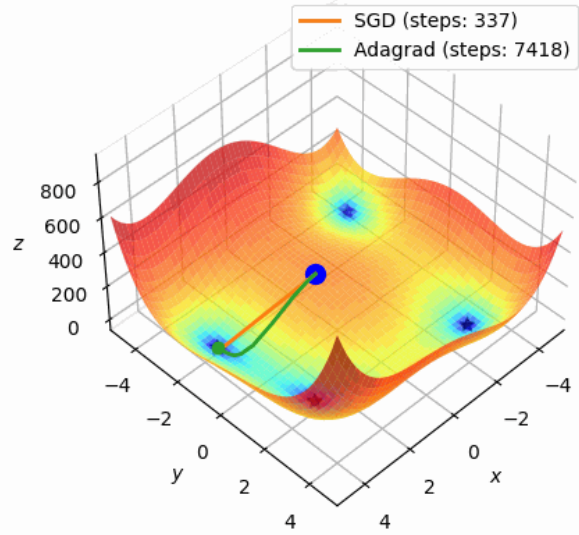

SGD (steps: 2857)
Adagrad (steps: 20000)
z
x
y


SGD (steps: 2857)
Adagrad (steps: 20000)
z
x
y


SGD (steps: 2857)
Adagrad (steps: 20000)
z
x
y


SGD (steps: 337)
Adagrad (steps: 7418)
z
x
y


SGD (steps: 337)
Adagrad (steps: 7418)
z
x
y


SGD (steps: 337)
Adagrad (steps: 7418)
z
x
y

Algoritmos de optimización: RMSProp

AdaGrad reduce el learning rate de manera muy agresiva (según el denominador va creciendo). Por ello, los pesos más activos van recibiendo según pasa el entrenamiento actualizaciones menores. La idea de RMSProp es reducir el denominador para evitarlo.

$$\theta_{t+1} = \theta_t - \eta' g_t$$

$$g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t)$$

$$\eta' = \frac{\eta}{\sqrt{\epsilon + G_t}},$$

$$G_t = \beta G_{t-1} + (1 - \beta) g_t^2$$

donde G_t es actualizado con una media móvil exponencial en base al hiperparámetro β (típicamente entre 0 y 1).

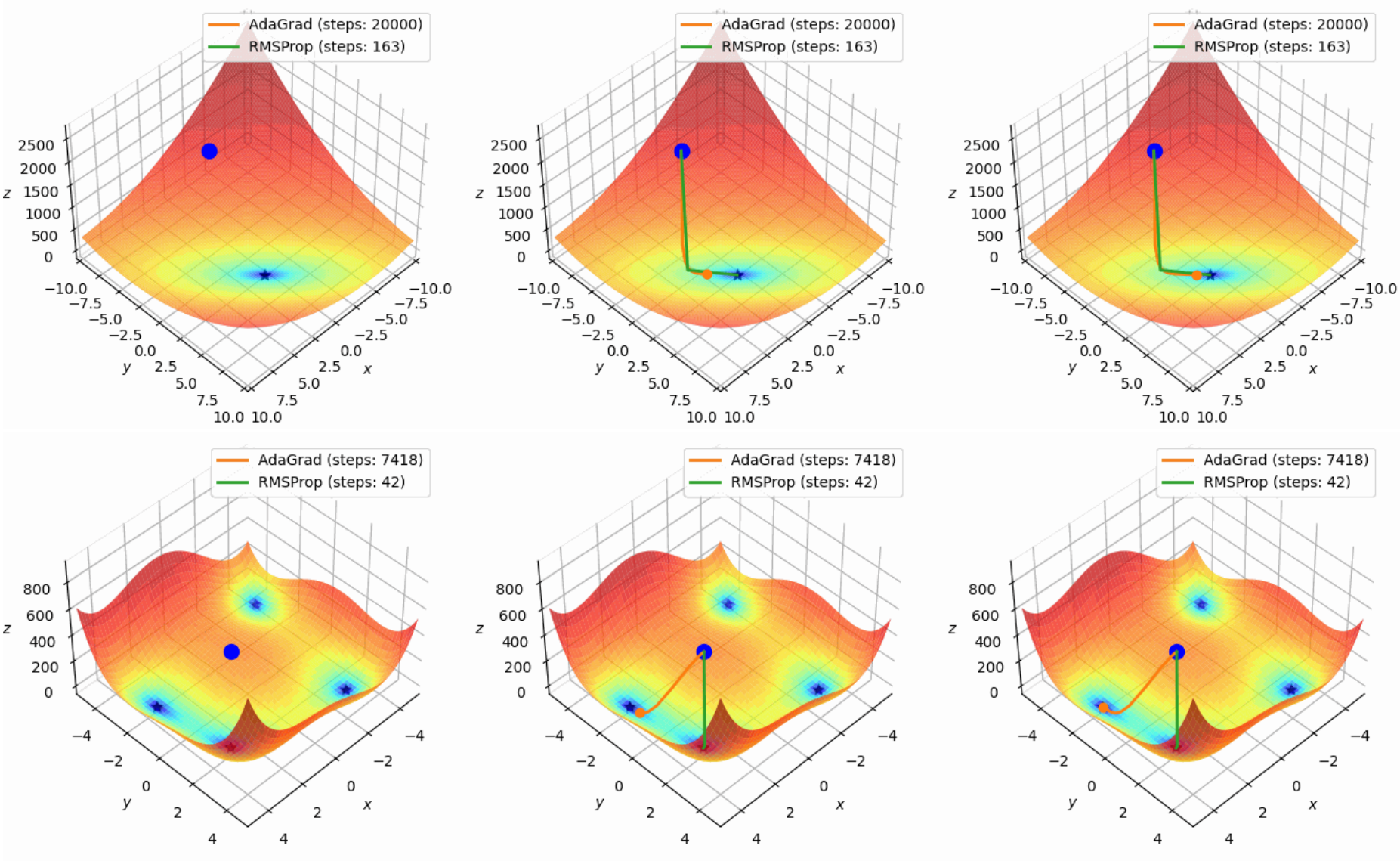
Algoritmos de optimización: RMSProp

$$G_t = \beta G_{t-1} + (1 - \beta) g_t^2$$

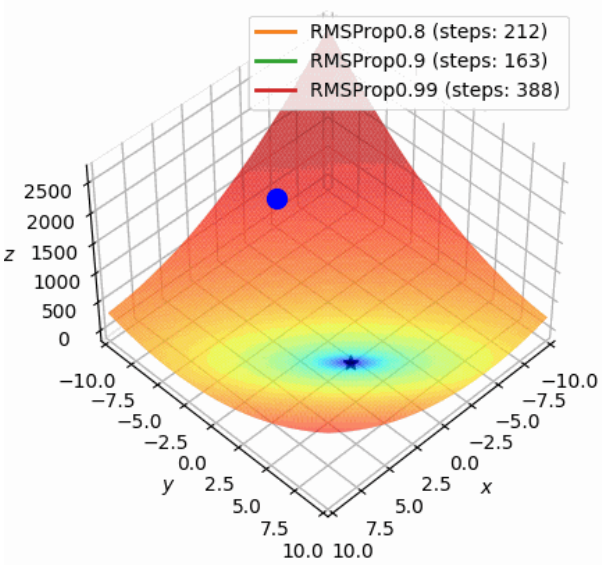
Algunas consideraciones sobre el hiperparámetro β:

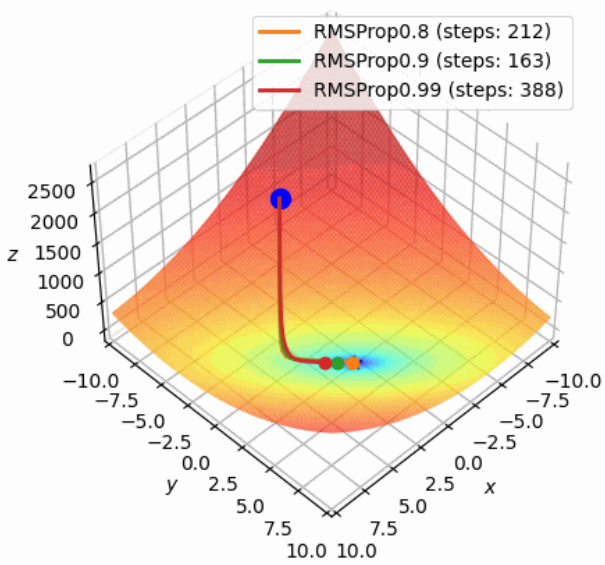
- Un valor cercano a 1 da más importancia a los gradientes antiguos, haciendo que el algoritmo **se adapte más lentamente a los cambios en los gradientes**.
- Un valor cercano a 0 da más importancia al gradiente actual, siendo por tanto el algoritmo **más sensible a los cambios en los gradientes**.
- La elección del valor óptimo de β depende del problema específico, el conjunto de datos y la arquitectura del modelo.
- En la práctica, a menudo se elige un valor de beta en el **rango de 0.9 a 0.999**, lo que proporciona un equilibrio entre la estabilidad y la adaptabilidad en la optimización de la tasa de aprendizaje.

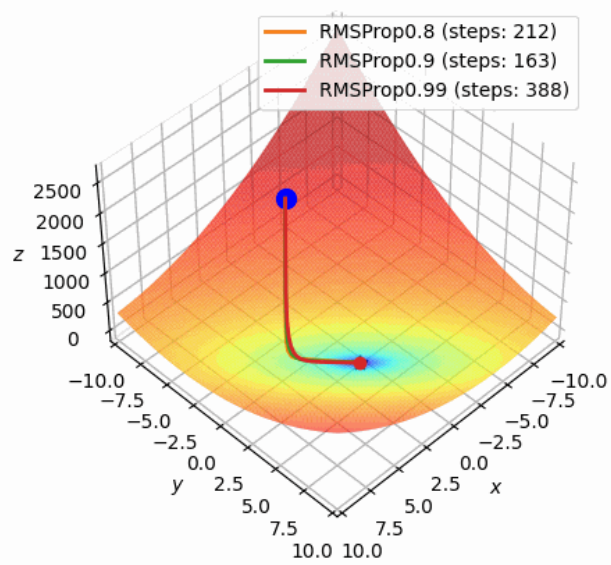
RMSProp

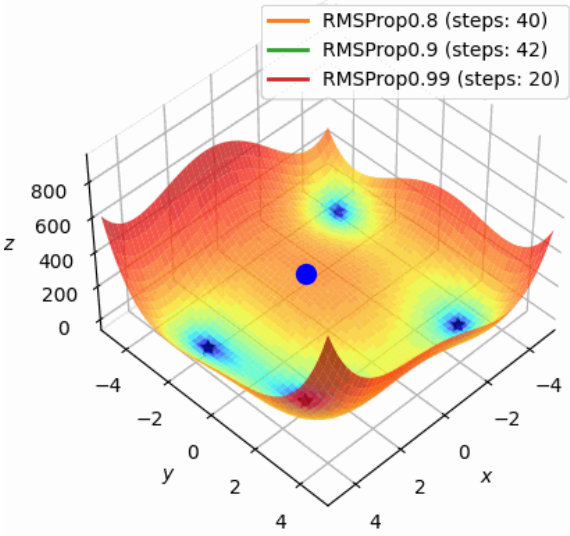

AdaGrad (steps: 20000)
RMSProp (steps: 163)
AdaGrad (steps: 20000)
RMSProp (steps: 163)
AdaGrad (steps: 20000)
RMSProp (steps: 163)
AdaGrad (steps: 7418)
RMSProp (steps: 42)
AdaGrad (steps: 7418)
RMSProp (steps: 42)
AdaGrad (steps: 7418)
RMSProp (steps: 42)

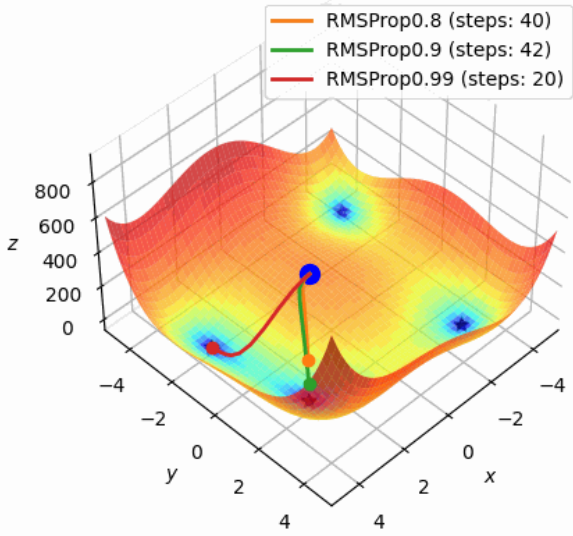
RMSProp

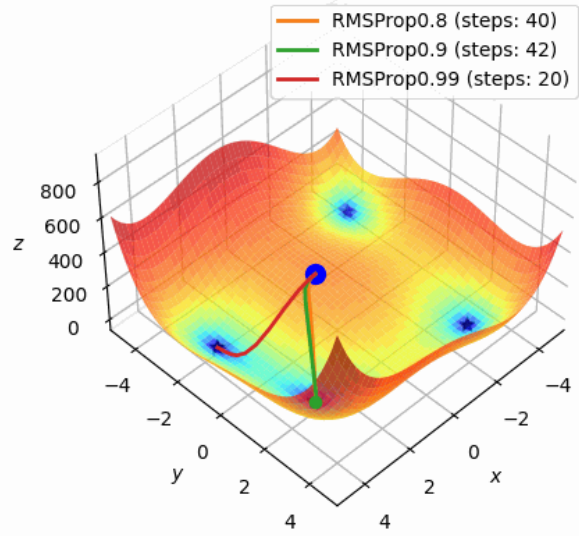

RMSProp0.8 (steps: 212)
RMSProp0.9 (steps: 163)
RMSProp0.99 (steps: 388)
z
x
y


RMSProp0.8 (steps: 212)
RMSProp0.9 (steps: 163)
RMSProp0.99 (steps: 388)
z
x
y


RMSProp0.8 (steps: 212)
RMSProp0.9 (steps: 163)
RMSProp0.99 (steps: 388)
z
x
y


RMSProp0.8 (steps: 40)
RMSProp0.9 (steps: 42)
RMSProp0.99 (steps: 20)
z
x
y


RMSProp0.8 (steps: 40)
RMSProp0.9 (steps: 42)
RMSProp0.99 (steps: 20)
z
x
y


RMSProp0.8 (steps: 40)
RMSProp0.9 (steps: 42)
RMSProp0.99 (steps: 20)
z
x
y

Algoritmos de optimización: Adam

Adam (Adaptative Moment Estimation) es una evolución de RMSProp y AdaGrad. La idea principal es actualizar de manera adaptativa el learning rate de cada parámetro basado en la estimación del primer momento (media) y segundo momento (varianza) de los gradientes.

Momentos

- Primer momento (media): captura la **dirección** general del gradiente así como su **magnitud**.
- Segundo momento (varianza): captura la información sobre la magnitud de los cambios o **fluctuaciones del gradiente** durante el entrenamiento.

Con la utilización de ambos momentos, Adam combina las ventajas de optimizadores basados en momento (SGD basado en Momento), con optimizadores con un learning rate adaptativo (RMSProp).

Algoritmos de optimización: Adam

Cálculo de los momentos:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

donde:

$$g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t)$$

Corrección del sesgo para los momentos: $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$ y $\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$

Siendo la regla de actualización de pesos:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

Valores para β_1 y β_2

Los valores típicos para β_1 y β_2 son 0.9 y 0.999

Algoritmos de optimización: Adam

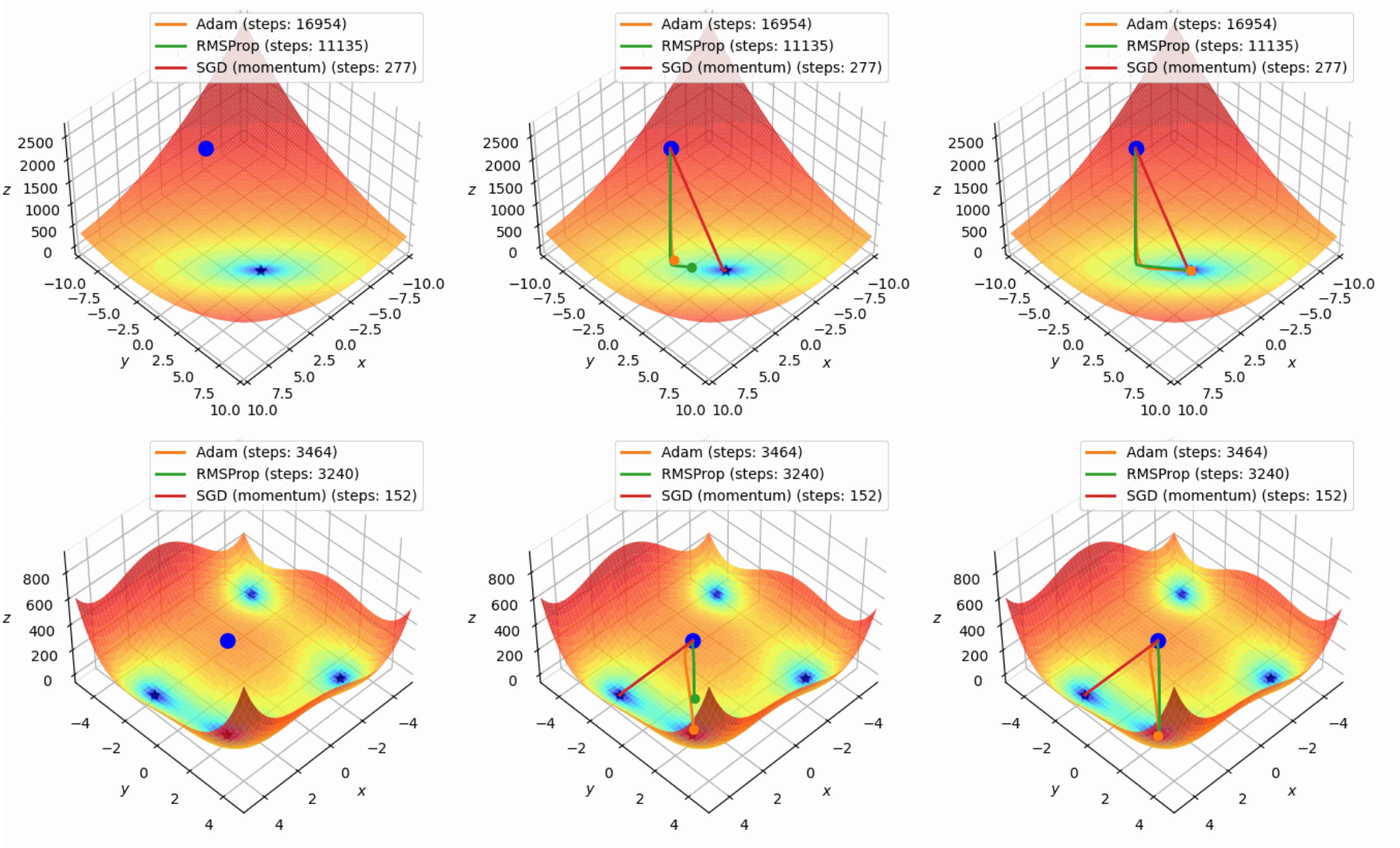
Corrección del sesgo en Adam

Si quieres profundizar más para entender porque la corrección del sesgo es necesaria en Adam, tienes una muy buena explicación [aquí].

- Adam es en general **uno de los mejores optimizadores** existentes para el aprendizaje profundo, siendo utilizado en muchísimas aplicaciones hoy en día.
- SGD con Momento también es una buena opción que puede llevar a convergencias más lentas pero con mejor capacidad de generalización.

No existe realmente una regla de oro sobre que optimizador escoger (así como sus parámetros). El proceso habitual es la prueba/error para ver que es lo que mejor funciona para nuestro conjunto de datos y modelo en particular.

Adam


Adam (steps: 16954)
RMSProp (steps: 11135)
SGD (momentum) (steps: 277)
Adam (steps: 3464)
RMSProp (steps: 3240)
SGD (momentum) (steps: 152)
z
x
y

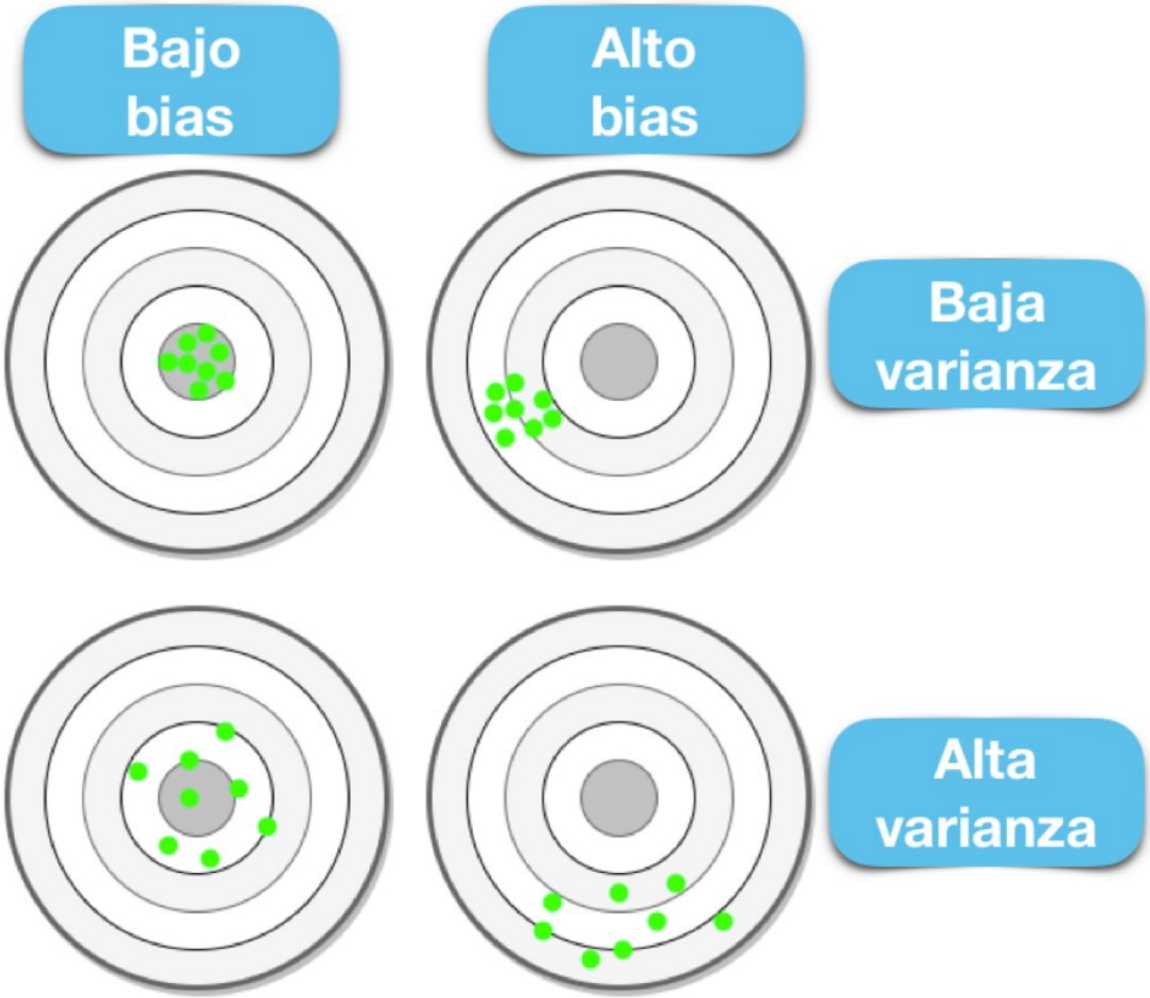
Compromiso entre el sesgo y la varianza

Compromiso entre el sesgo y la varianza

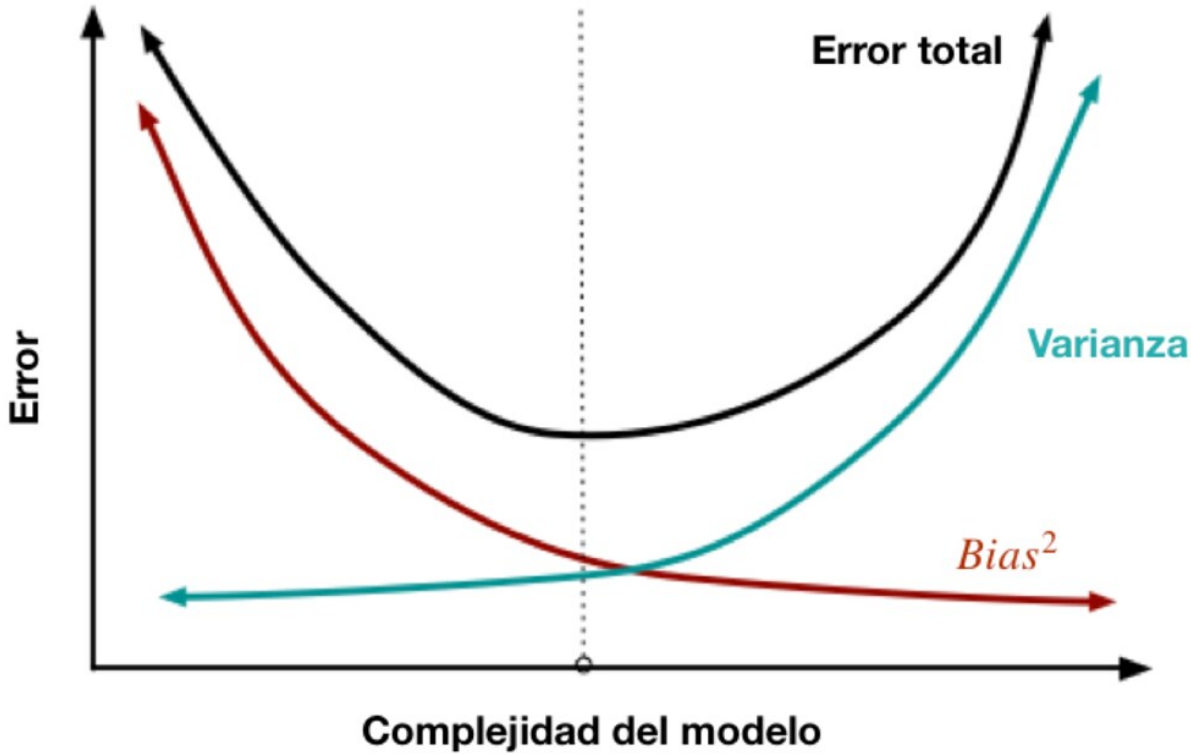
El compromiso entre el sesgo y la varianza (**bias-variance tradeoff**) es un concepto clave en el aprendizaje automático y también en el aprendizaje profundo. Este concepto se refiere al balance entre la capacidad del modelo para representar los datos (sesgo) y su sensitividad a variaciones en los datos (varianza)

- Un **sesgo grande** puede causar que el algoritmo se esté perdiendo relaciones importantes entre las variables de entrada y la de salida (**subajuste**)
- Una **varianza alta** puede causar que el algoritmo esté aprendiendo ruido de los datos de entrenamiento (**sobreajuste**).

Compromiso entre el sesgo y la varianza: gráficamente



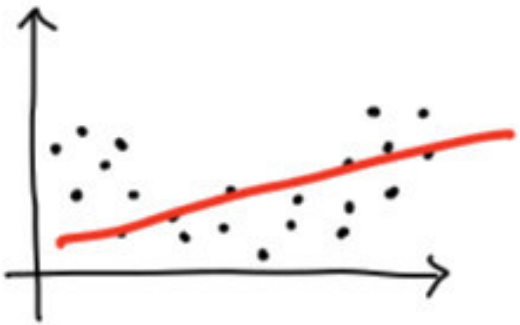
Compromiso entre el sesgo y la varianza: complejidad



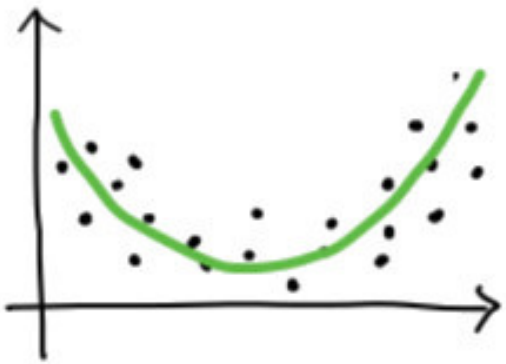
Importante

Como regla general, si intentamos disminuir el sesgo aumentando la complejidad del modelo, aumentará la varianza. Si por el contrario, intentamos minimizar la varianza disminuyendo la complejidad, aumentará el sesgo.

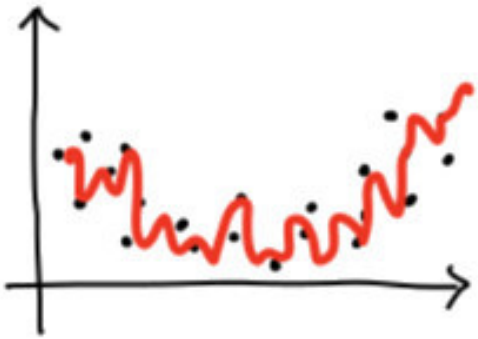
Compromiso entre el sesgo y la varianza: complejidad



Modelo demasiado simple



Modelo de complejidad adecuada



Modelo con demasiada complejidad

Compromiso entre sesgo y varianza: resumen

En resumen:

- Si nuestro modelo es muy **simple** y tiene pocos parámetros, es probable que tenga un **sesgo alto y una varianza pequeña** (posible subajuste).
- Si nuestro modelo es muy **complejo** y tiene un gran número de parámetros entonces probablemente tenga una **varianza alta y un sesgo bajo** (posible sobreajuste).

Las redes neuronales **tienen tendencia a sobreajustar** ya que se trata de modelos muy complejos, generalmente con muchos parámetros. Algunas técnicas que veremos para paliar este problema son:

- Regularización L1 y L2
- Dropout
- Parada temprana
- Aumento de datos
- Normalización del lote (batch normalization)

Regularización L1 y L2

¿Qué es la regularización?

La **regularización** se refiere a una serie de técnicas que tratan de disminuir la complejidad de los modelos durante el entrenamiento y por tanto tratan de **prevenir el sobreajuste**.

La idea de la regularización es muy sencilla: consiste en añadir un término a la función de pérdida cuyo valor es más grande cuanto más complejo es el modelo.

En esta sección trataremos de la regularización L1 y L2 (también conocida como **weight decay**).

Compromiso entre sesgo y varianza

La **regularización** en práctica normalmente conlleva **aumentar el sesgo** a costa de **reducir la varianza** de manera significativa.

Regularización L1

Recordemos por ejemplo la función de pérdida MSE para problemas de regresión:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

siendo $\hat{y}_i$ la predicción retornada por el modelo para el ejemplo x_i: $f_w(x_i)$
Esta función regularizada tendría la siguiente forma:

$$\mathcal{L} = \mathcal{C}||w||_1 + \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

donde $||w||_1 = \sum_i \sum_j |w_{ij}|$, es decir, la suma de los elementos de las matrices de pesos en valor absoluto y $\mathcal{C}$ es un **hiperparámetro** que controla la regularización.

Regularización L1

Algunos datos importantes sobre la regularización L1:

- Si establecemos $\mathcal{C}$ a cero, el modelo se convierte en un modelo no regularizado normal.

Regularización L1

Algunos datos importantes sobre la regularización L1:

- Si establecemos $\mathcal{C}$ a cero, el modelo se convierte en un modelo no regularizado normal.
- Si $\mathcal{C}$ es un valor muy grande, el algoritmo de aprendizaje tratará de tener pesos muy pequeños para minimizar la función de pérdida, lo que generalmente llevará a una situación de **subajuste**.

Regularización L1

Algunos datos importantes sobre la regularización L1:

- Si establecemos $\mathcal{C}$ a cero, el modelo se convierte en un modelo no regularizado normal.
- Si $\mathcal{C}$ es un valor muy grande, el algoritmo de aprendizaje tratará de tener pesos muy pequeños para minimizar la función de pérdida, lo que generalmente llevará a una situación de **subajuste**.
- La regularización L1 generalmente produce un modelo **disperso**, donde muchos pesos valen cero.
  - Podemos considerar que L1 realiza una especie de selección de atributos, decidiendo cuales son esenciales para la predicción.
  - Esto puede ser interesante si queremos incrementar la **explicabilidad del modelo**.

Regularización L2

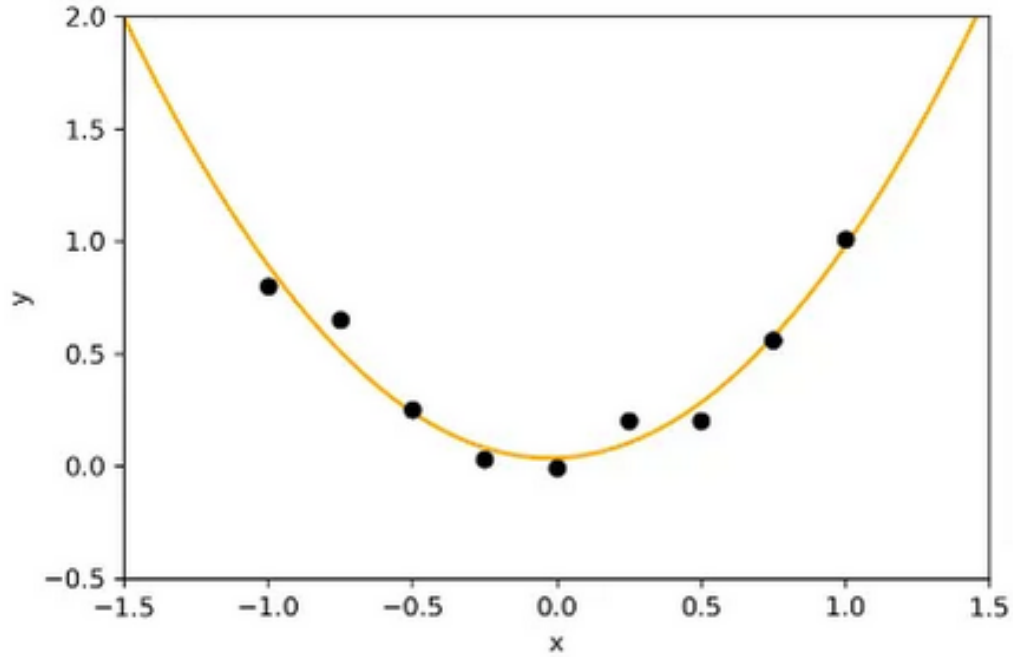
La regularización L2 generalmente conlleva mejores resultados que la regularización L1. En este caso la idea es la misma pero utilizando la norma 2 al cuadrado (también conocida como norma euclídea). Para la función de perdida MSE tendríamos:

$$\mathcal{L} = \mathcal{C}||w||_2^2 + \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

donde $||w||_2^2 = \sum_i \sum_j w_{ij}^2$, es decir, la suma de los elementos de las matrices de pesos al cuadrado y $\mathcal{C}$ es un **hiperparámetro** que controla la regularización.

¿Por qué funcionan la regularización L1 y L2?

Generalmente tener pesos más pequeños conlleva tener un modelo más simple. Veamos un ejemplo:



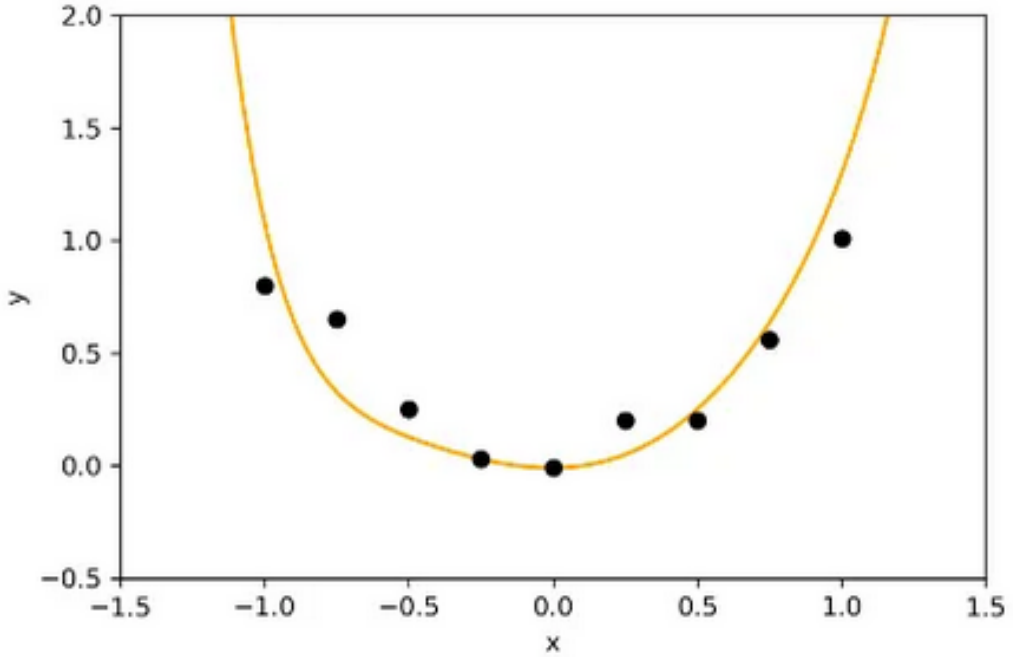
$$\hat{y} = 0.04 + 0.04x + 0.9x^2$$

MSE=0.006

Norma L2=0.9

Norma L1=0.98

¿Por qué funcionan la regularización L1 y L2?



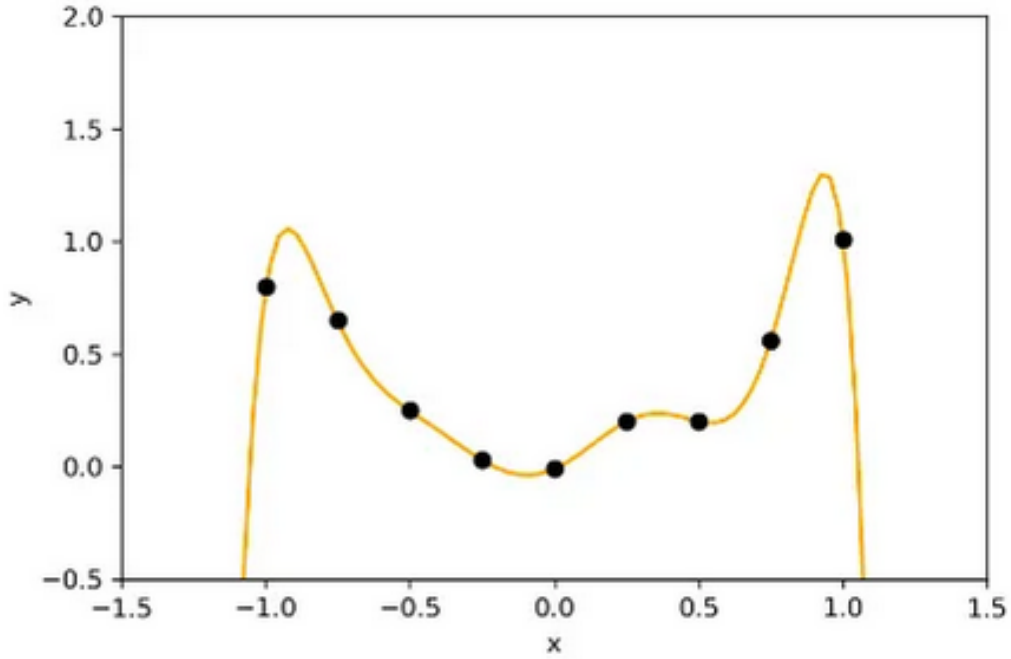
$$\hat{y} = -0.01 + 0.01x + 0.8x^2 + 0.5x^3 - 0.1x^4 - 0.1x^5 + 0.3x^6 - 0.3x^7 + 0.2x^8$$

MSE=0.035

Norma L2=1.06

Norma L1=2.32

¿Por qué funcionan la regularización L1 y L2?



$$\hat{y} = -0.01 + 0.57x + 2.67x^2 - 4.08x^3 - 12.25x^4 + 7.41x^5 + 24.87x^6 - 3.79x^7 - 14.38x^8$$

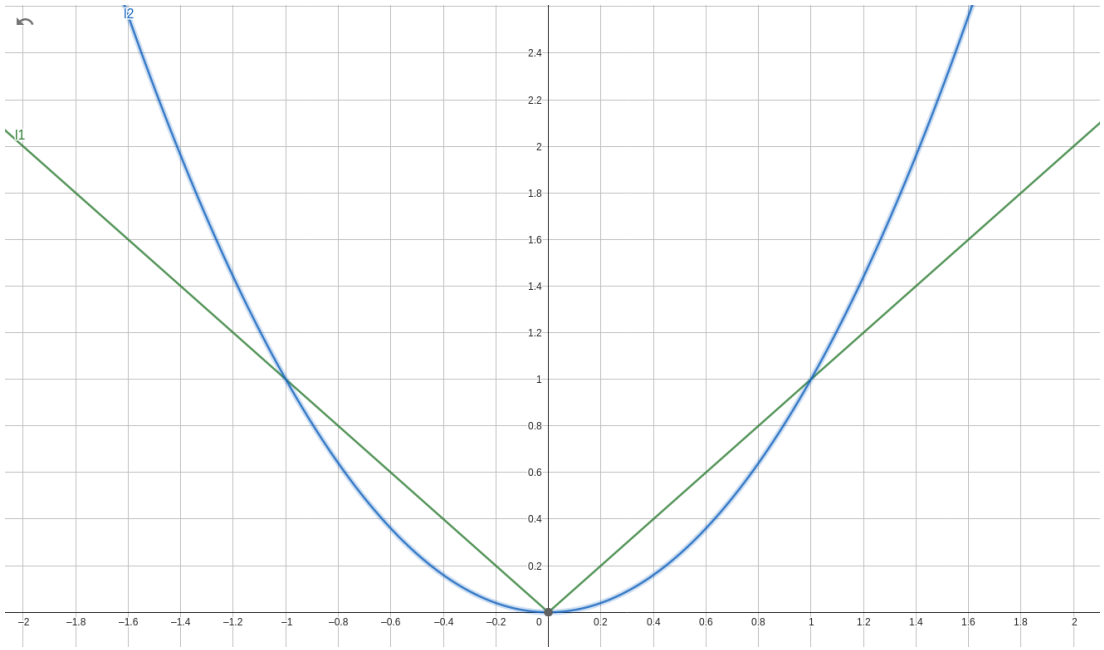
MSE=0

Norma L2=32.69

Norma L1=70.03

Diferencia entre L1 y L2

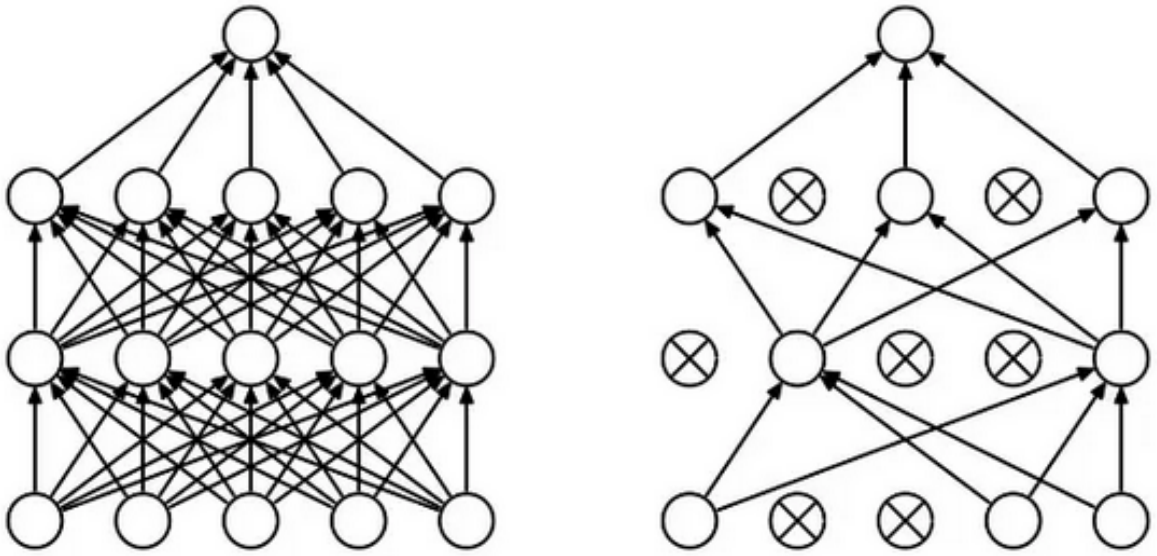
Ambos tipos de regularización hacen que los pesos sean más pequeños, sin embargo **L1** tiende a que **los pesos sean cero** mientras que **L2**, cuando los pesos ya son muy pequeños (debido a elevar los pesos al cuadrado), tenderá a que los pesos **no lleguen a valer cero**.



Dropout

Dropout

El concepto de **dropout** es muy sencillo. Consiste en **excluir de manera aleatoria un porcentaje de neuronas** de la computación. Estas neuronas son elegidas aleatoriamente para cada ciclo de evaluación y actualización de pesos.



¿Cúantas neuronas anular?

La tasa de neuronas a excluir en cada iteración es un **hiperparámetro** con valores en el rango $[0, 1]$.

Dropout

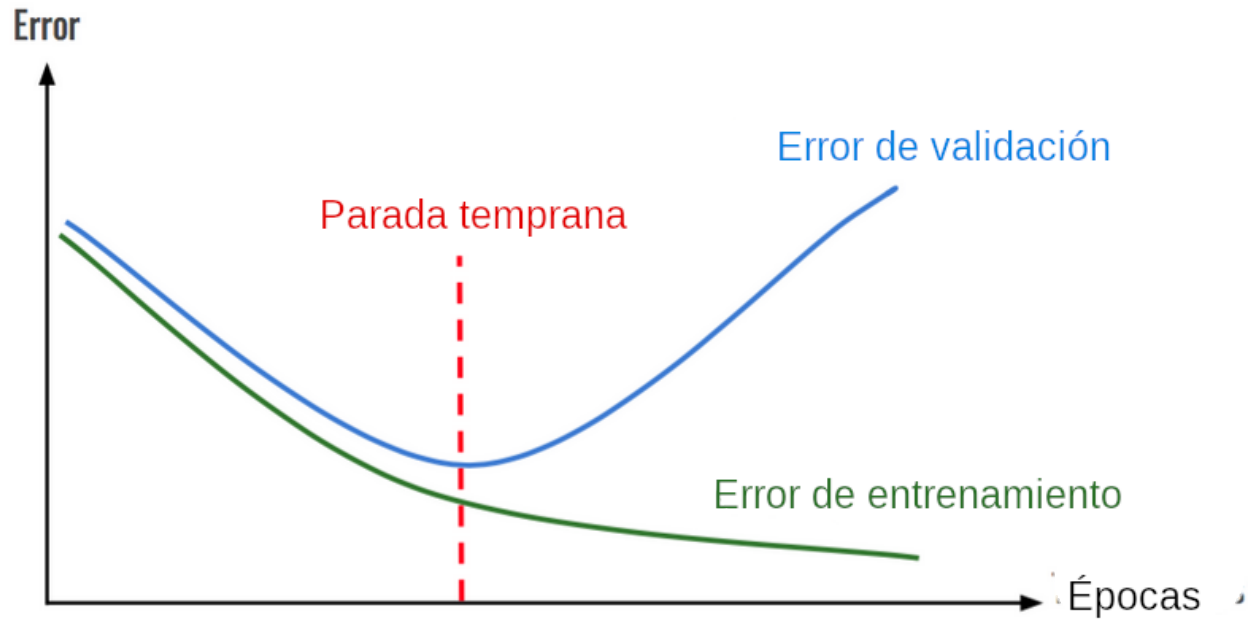
Algunos puntos claves del **dropout** son:

1. **Regularización:** Evita el sobreajuste al prevenir la dependencia excesiva de unidades individuales.
2. **Reducción de la co-adaptación:** Al desconectar aleatoriamente las unidades, se evita que se adapten en exceso a las otras unidades específicas con las que están conectadas.
3. **Aumento de la robustez:** Mejora la capacidad de generalización del modelo, ya que las unidades deben aprender características útiles de forma más independiente.
4. **Entrenamiento de redes más grandes:** Permite entrenar con éxito redes neuronales más grandes y más profundas al evitar el sobreajuste.

Parada temprana (Early Stopping)

Parada temprana

La **parada temprana** (early stopping) es una técnica utilizada en el entrenamiento de redes neuronales profundas para **evitar el sobreajuste** y **mejorar la generalización** del modelo.
Consiste en **detener el proceso de entrenamiento** basándonos en el valor de la función de pérdida para un **conjunto de validación**, no utilizado para la actualización de los pesos de la red.



Parada temprana: ventajas

A continuación, se presentan algunas razones por las cuales la parada temprana es importante:

1. **Evitar el sobreajuste:** La parada temprana ayuda a evitar que el modelo aprenda patrones específicos de los datos de entrenamiento que no se generalizan bien a nuevos datos.
2. **Mejorar la eficiencia computacional:** Al detener el entrenamiento antes de la convergencia completa, se ahorra tiempo y recursos computacionales.
3. **Evitar el estancamiento:** En algunos casos, el rendimiento del modelo puede dejar de mejorar después de cierto número de épocas. La parada temprana evita continuar el entrenamiento en estas situaciones.
4. **Flexibilidad en la elección del punto de parada:** La parada temprana permite seleccionar el punto de parada óptimo basado en algún criterio, como el rendimiento en un conjunto de validación.

Aumento de datos (data augmentation)

Aumento de datos

El aumento de datos es una técnica utilizada en el aprendizaje profundo para **aumentar la cantidad y variedad de datos de entrenamiento**.

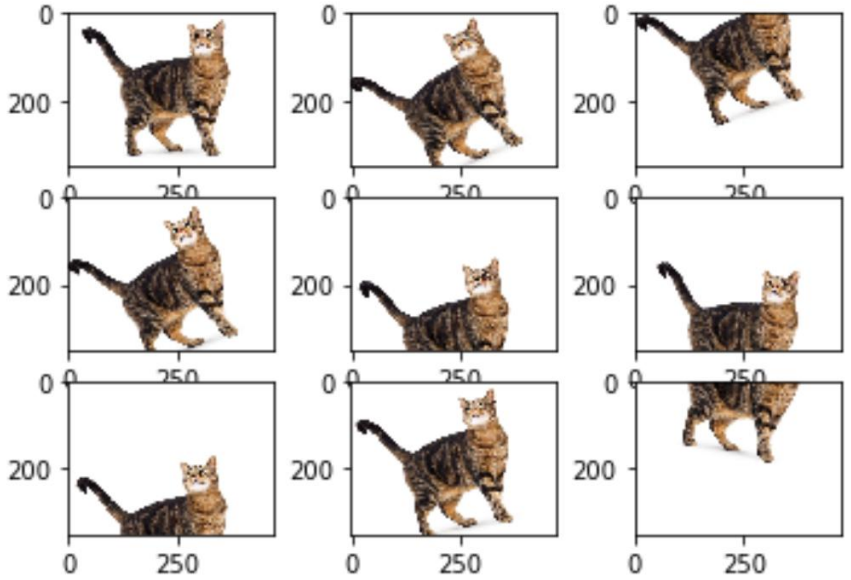
Consiste en aplicar transformaciones aleatorias a las muestras de datos existentes para **crear nuevas instancias sintéticas**.

Beneficios del aumento de datos:

- **Aumenta la cantidad de datos** de entrenamiento, lo que ayuda a evitar el sobreajuste.
- Mejora la **generalización** del modelo al introducir variabilidad y diversidad en los datos.

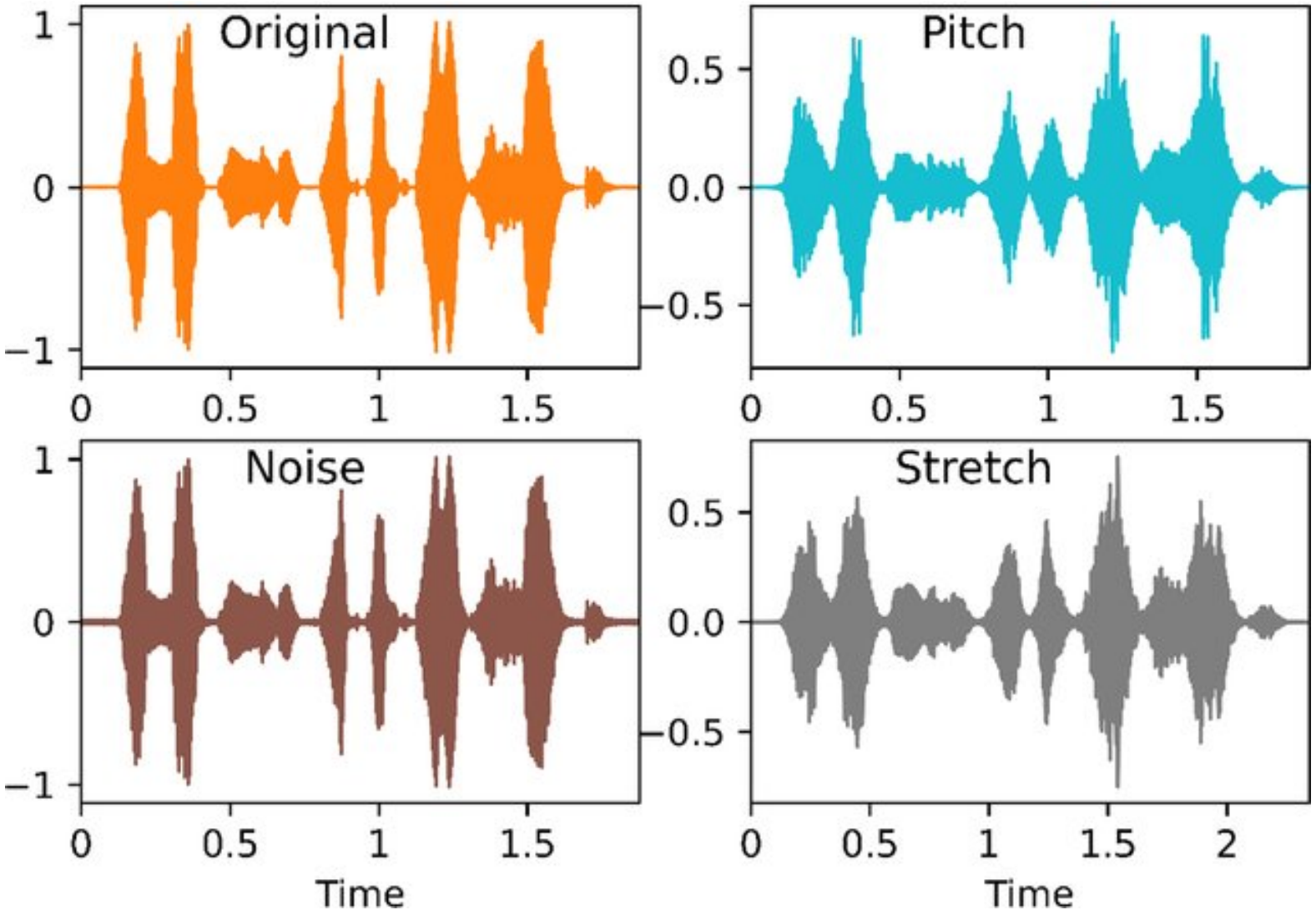
Aumento de datos

Ejemplo de aumento de datos en visión artificial:



Aumento de datos

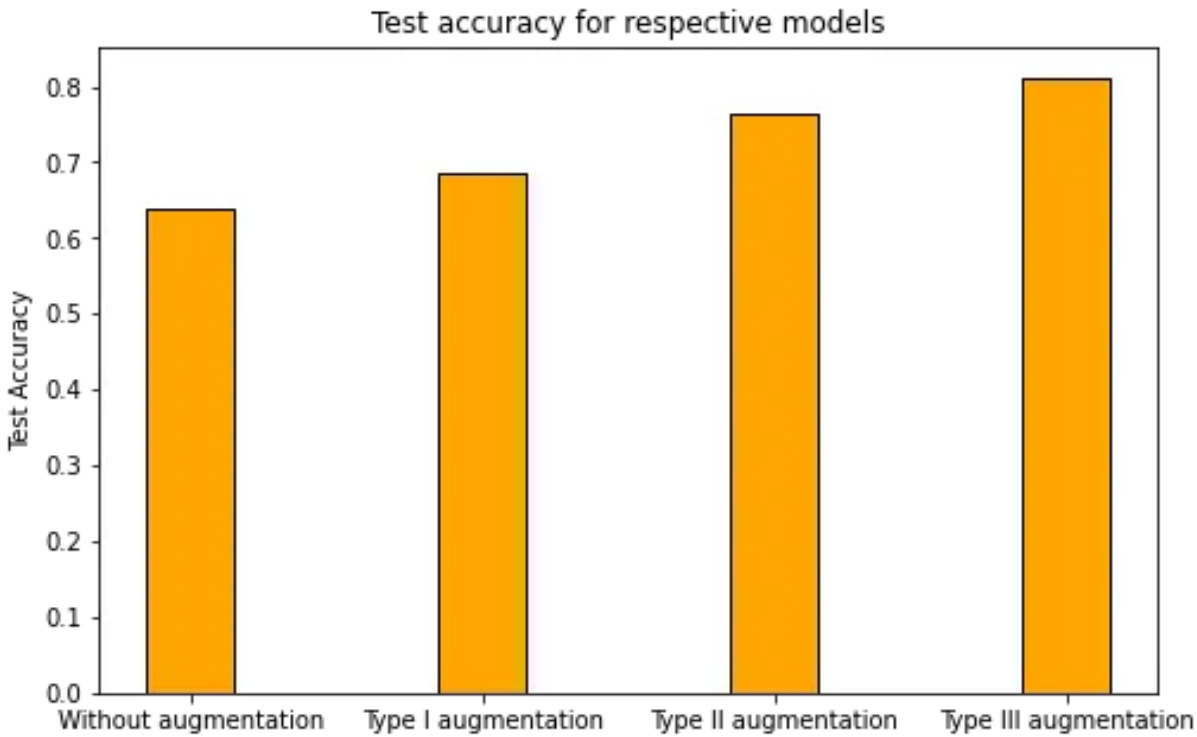
Ejemplo de aumento de datos en procesamiento de audio:



Aumento de datos

Ejemplo de tres tipos de aumento de datos para un conjunto de imagenes:

1. **Tipo I**: volteo horizontal alteatorio.
2. **Tipo II**: Tipo I + rotación alteatoria.
3. **Tipo III**: Tipo II + modificación de color aleatoria.



Normalización del lote: Batch Normalization

Normalización del lote

La **normalización del lote** es una técnica utilizada en redes neuronales para normalizar las activaciones de una capa, **mejorando la estabilidad y eficiencia del entrenamiento**.

Los modelos muy profundos implican la composición de varias funciones o capas. El gradiente indica cómo actualizar cada parámetro, suponiendo que **las demás capas no cambian**. En la práctica, **actualizamos todas las capas simultáneamente**.

Internal Covariate Shift

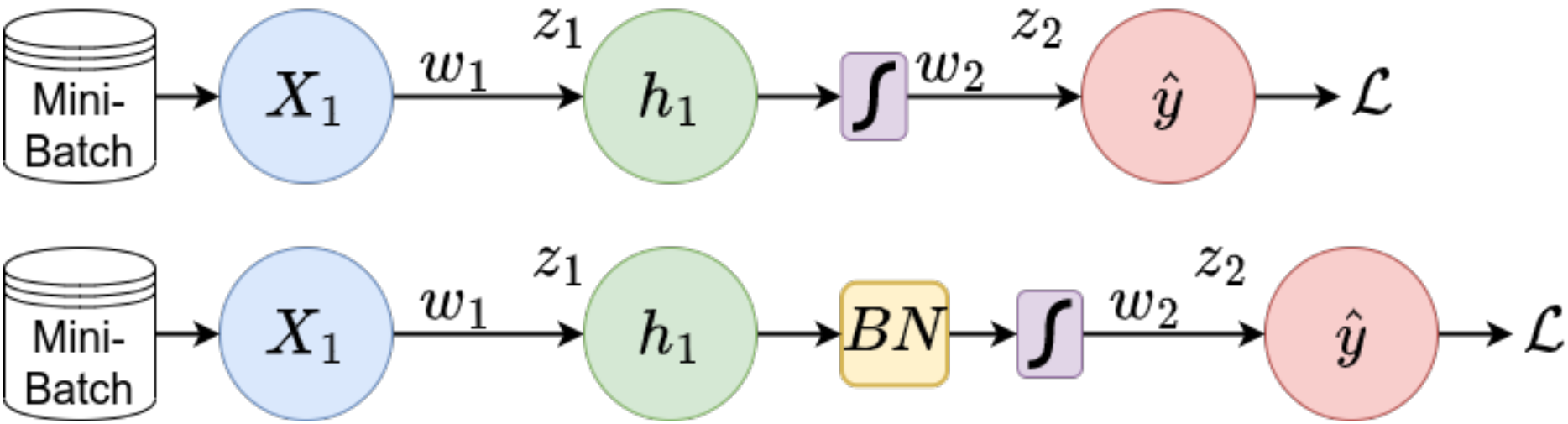
Internal Covariate Shift es el cambio de distribución de las activaciones de la red debido al cambio en los parámetros durante el entrenamiento.

Importante

La normalización del lote no se considera generalmente una técnica de regularización, aunque su uso suele causar este efecto.

Normalización del lote: ¿cómo funciona?

Veamos una red sencilla con una red neuronal muy sencilla con una sola neurona en la capa de entrada y una capas oculta con una neurona en la misma:



Normalización del lote: ¿cómo funciona?

Los pasos son los siguientes:

1. Cuando un mini-lote de ejemplos (mini-batch) llega a la red, este se propaga hacia adelante por la red.
2. Cuando llegamos a una capa de normalización del lote (BN), primero calculamos la media μ y la desviación típica σ^2 considerando todos los ejemplos del mini-batch.

$$\mu_j = \frac{1}{m} \sum_{i=1}^{m} h_{ij}$$

$$\sigma_j^2 = \frac{1}{m} \sum_{i=1}^{m} (h_{ij} - \mu_j)^2$$

donde m es el tamaño del mini-batch y h_{ij} es el valor de la salida de la neurona h_j para el ejemplo i.

Normalización del lote: ¿cómo funciona?

Continuación...

3. Una vez calculadas la media y la desviación típica, normalizamos la salida de la red, de manera que tengan **media 0 y varianza 1**:

$$\hat{h}_{ij} = \frac{h_{i_j} - \mu_j}{\sigma^2}$$

4. En el siguiente paso hacemos una operación de escalado y desplazamiento. Esta fase se hace para que la red no pierda flexibilidad y potencia por culpa de esta capa de normalización:

$$\hat{h}_{ij} = \gamma_j \hat{h}_{i_j} + \beta_j$$

donde γ_j y β_j son parámetros de la red (uno por cada canal j), que son **aprendidos** durante la fase de entrenamiento.

Normalización del lote: ¿cómo funciona?

Continuación...

5. Según va a avanzando el entrenamiento, se guarda una **estimación de la media y desviación** para cada canal, para poder usarla en inferencia.

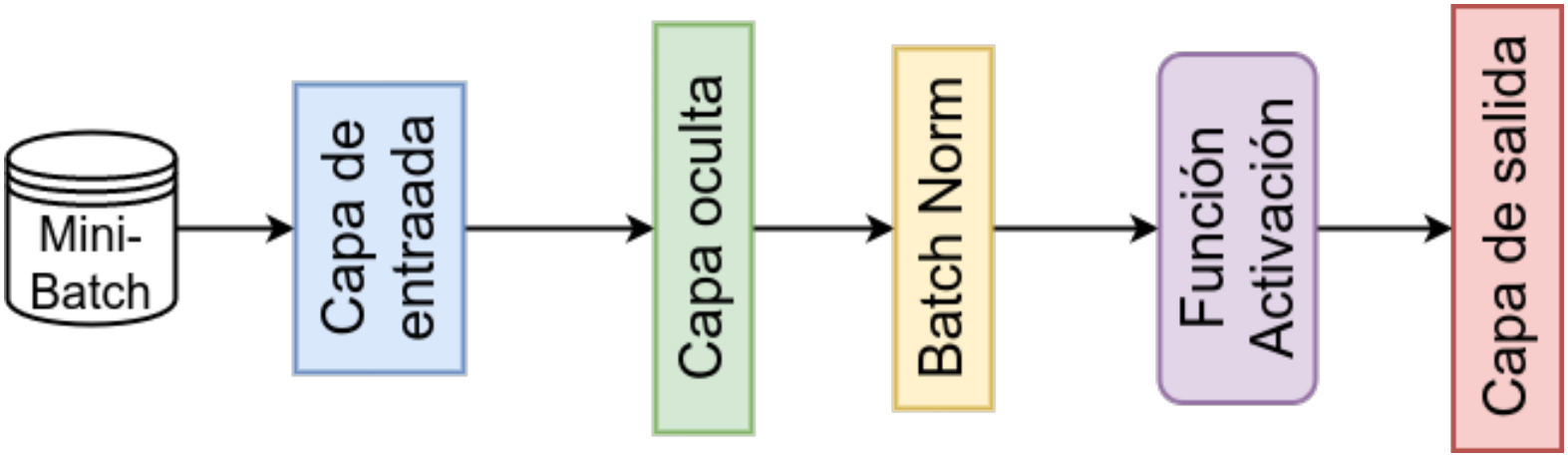
En **inferencia**, no nos llega un mini-batch de ejemplos, sino un único ejemplo. En este caso la capa de normalización usa la media y desviación estimadas para normalizar los valores producidos por el ejemplo de test.

Posición de la capa de normalización

Generalmente la capa de Batch Normalization se sitúa justo antes de la función de activación.

Normalización del lote

Ejemplo de donde iría colocada la capa de normalización en una red totalmente conectada:



Importante

Ten en cuenta que las capas de normalización del lote no solo se usan en redes totalmente conectadas sino en otras arquitecturas como redes convolucionales o transformers.

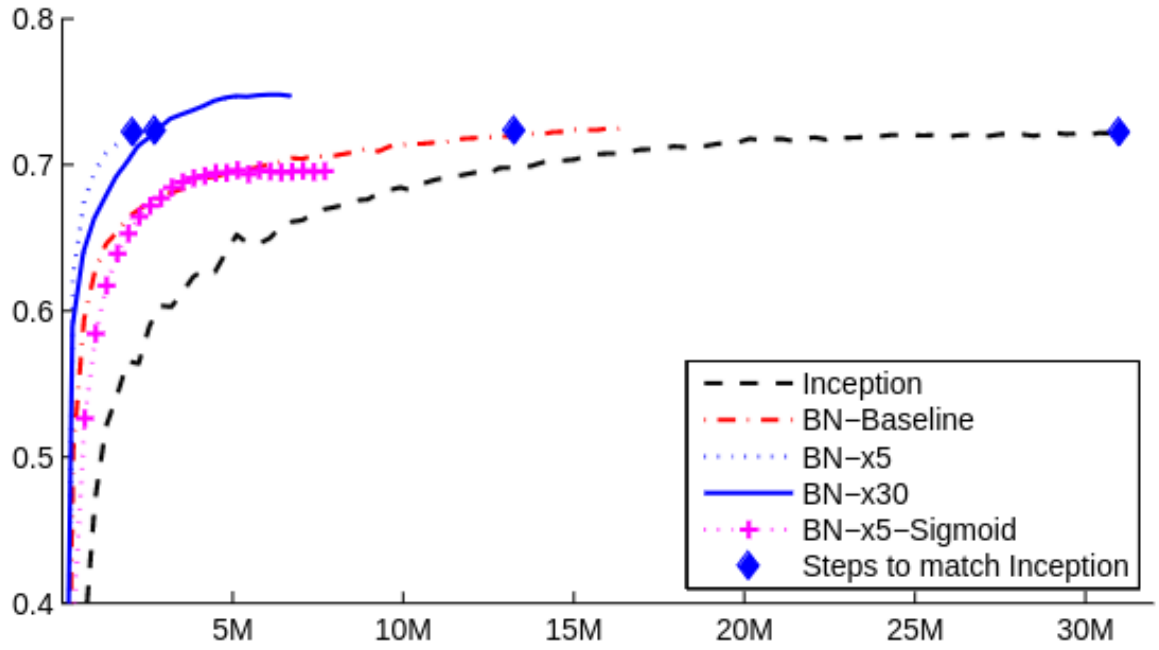
Normalización del lote: beneficios

Beneficios de la Batch Normalization:

- **Permite utilizar learning rates más altos** sin que la red colapse.
- Reduce la dependencia de la inicialización de los pesos.
- Ayuda a combatir el problema del **desvanecimiento y la explosión del gradiente**.
- Actúa como **regularizador** al introducir ruido en el cálculo de la media y la varianza.
- Puede llevar a una **convergencia más rápida**, al estabilizar las activaciones de la red.

Normalización del lote: beneficios

En la siguiente imagen, del [artículo original] donde se describe esta técnica, se puede ver como esta red converge de manera más rápida cuando se usa la normalización del lote.



x5 y x30 indican un learning rate 5 y 30 veces mayor respectivamente.

Inicialización de pesos

Inicialización de pesos: Introducción

La **inicialización de pesos** es una parte muy importante del entrenamiento de una red neuronal:

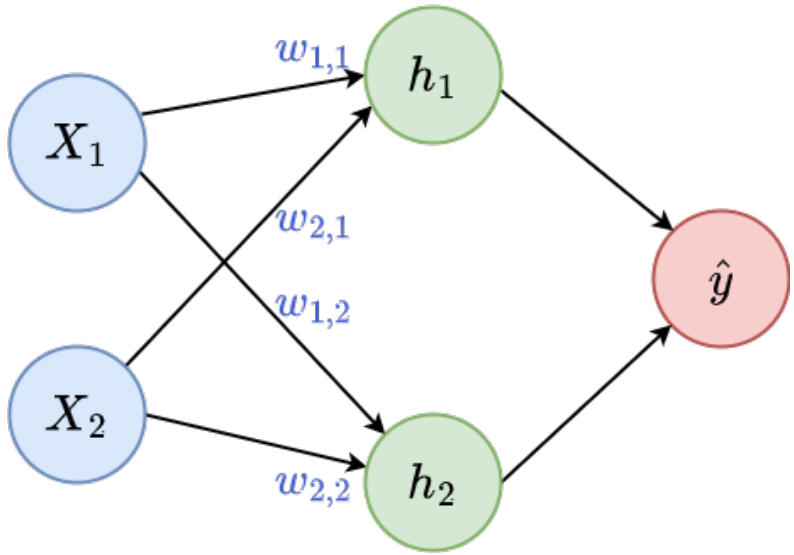
- El **valor inicial** de los pesos **afecta de manera importante** al proceso de entrenamiento de la red.
- La inicialización **puede determinar si el algoritmo converge o no**, habiendo ciertas situaciones iniciales que pueden hacer fallar completamente el entrenamiento.
- Diferentes incializaciones pueden llevar a **modelos diferentes, potencialmente con diferentes rendimientos**.

En la práctica se pueden usar diferentes métodos para inicializar los pesos:

1. Inicialización constante
2. Inicialización aleatoria
3. Xavier Initialization

Inicialización de pesos: Constante

Vamos a analizar como afecta la inicialización de los pesos de la red con **valores constantes**.

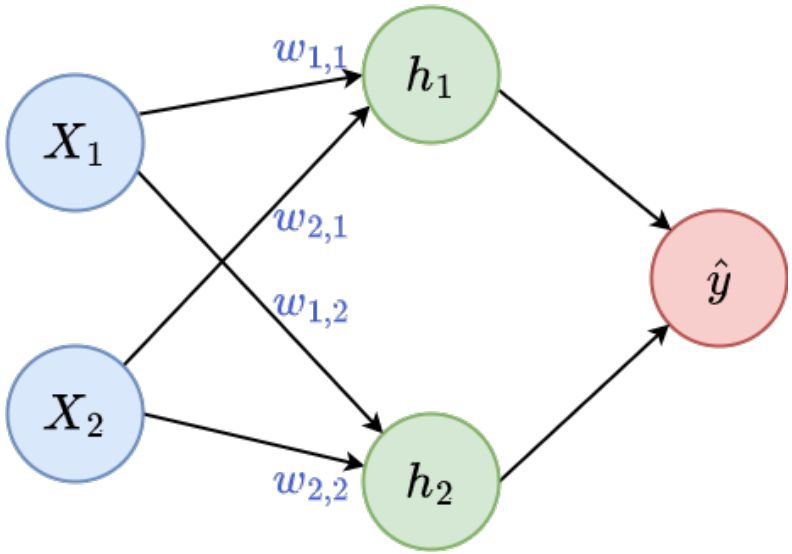


Supongamos que esta red utiliza una función de activación σ. Las activaciones de las neuronas h_1 y h_2 tendrían los siguientes valores:

$$h_1 = \sigma(w_{1,1}x_1 + w_{2,1}x_2)$$

$$h_2 = \sigma(w_{2,1}x_1 + w_{2,2}x_2)$$

Inicialización de pesos: Constante

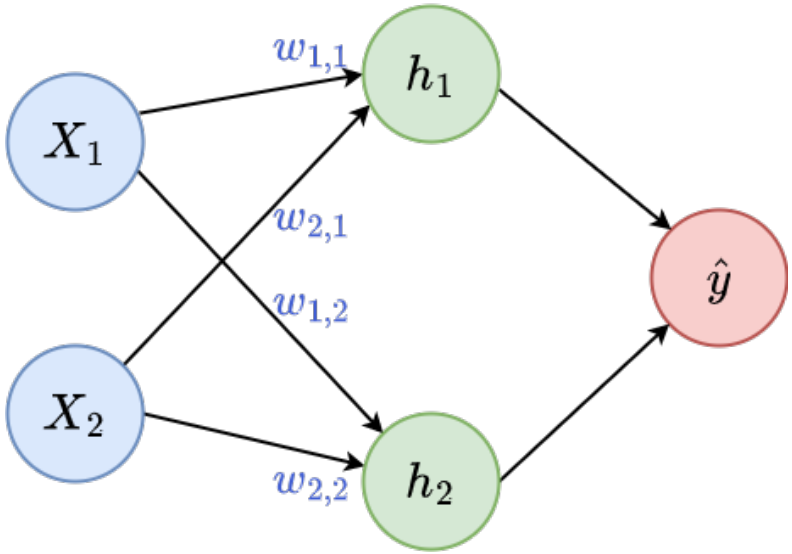


$$h_1 = \sigma(w_{1,1}x_1 + w_{2,1}x_2)$$

$$h_2 = \sigma(w_{2,1}x_1 + w_{2,2}x_2)$$

Si consideramos que $w_{1,1} = w_{1,2} = w_{2,1} = w_{2,2}$, es fácil comprobar que las activaciones h_1 y h_2 tienen el **mismo valor**.

Inicialización de pesos: Constante



$$h_1 = \sigma(w_{1,1}x_1 + w_{2,1}x_2)$$
$$h_2 = \sigma(w_{2,1}x_1 + w_{2,2}x_2)$$

Si ahora calculamos los **gradiente** para los pesos w:

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_j} \frac{\partial h_j}{\partial w_{i,j}}$$

podemos comprobar que todos los gradientes son iguales ya que $w_{i,j}$ son iguales por definición y las activaciones h_j también.

Inicialización de pesos: Constante

El efecto de utilizar inicialización constante de pesos es que todas las neuronas aprenden lo mismo, creandose una **simetría en la red** que va a ser imposible de romper según pasen las épocas de entrenamiento.

Importante

Obviamente, **esta simetría** en la red, donde todas las neuronas se comportan de la misma manera, **no es una propiedad deseable** para una red neuronal. Por tanto, existen técnicas de inicialización más adecuadas que veremos a continuación.

Inicialización de pesos: Aleatoria

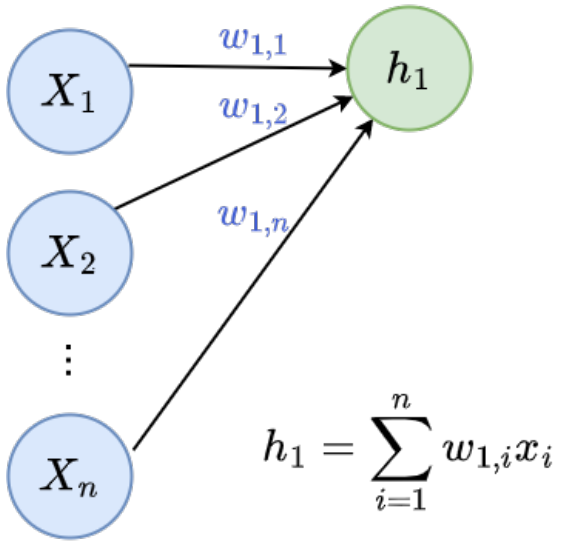
La solución al problema descrito anteriormente consiste en utilizar valores aleatorios para los pesos. Aquí nos enfrentamos potencialmente a dos problemas que ya hemos estudiado:

1. Valores aleatorios **demasiado pequeños**: en este caso existe el problema del desvanecimiento del gradiente, que hace que las primeras capas de la red apenas reciban actualizaciones de los pesos.
2. Valores aleatorios **demasiado grandes**: en este caso el problema es el "exploding gradient" que hace que el entrenamiento sea inestable.

La pregunta entonces es, ¿cómo encontrar los **valores óptimos** para inicializar los pesos aleatoriamente?

Inicialización de pesos: Aleatoria

Un heurístico muy utilizado para inicializar los pesos de manera aleatoria (utilizado por ejemplo por PyTorch por defecto), es utilizar una distribución uniforme $w \in \mathcal{U}(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}})$, donde n sería el número de neuronas en la capa anterior.



Intuición del heurístico

La escala de los pesos debe depender del número de conexiones que vienen de la capa anterior (más conexiones, valores más pequeños y viceversa)

Inicialización de pesos: Inicialización Xavier

Este es un **herístico** desarrollado por Xavier Glorot. La idea principal del heurístico es **tratar de conseguir dos cosas** con la inicialización de pesos:

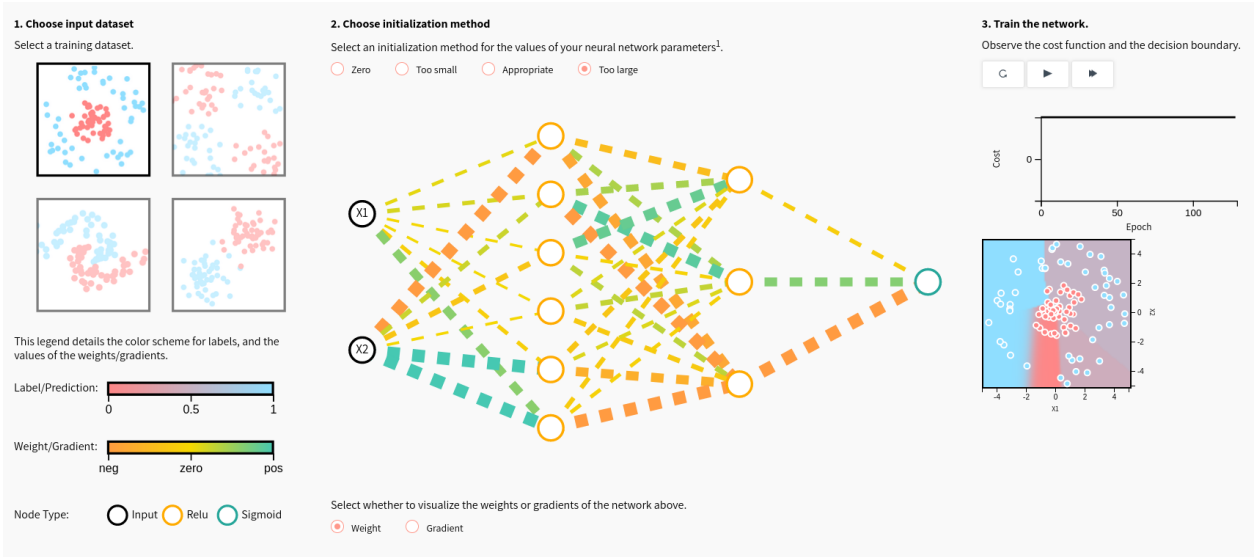
1. Durante la pasada hacia adelante, conseguir activaciones estén centradas en cero con una varianza de uno (aquí nos afecta el número de neuronas en al capa anterior n_{in}).
2. Durante la pasada hacia atrás, que los gradientes también estén centrados en cero con una varianza de uno (aquí nos afecta el número de neuronas en la siguiente capa n_{out}).

Para conseguir estas dos características, se utiliza una derivación matemática que depende del número de entradas n_{in} y de salidas de la capa n_{out}.

$$w \in \mathcal{U}\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}} \right)$$

Inicialización de pesos

Importante

La inicialización Xavier funciona bien para funciones de activación sigmoides o tanh, pero no para ReLU. La razón es que ReLU devuelve 0 para cualquier activación negativa. Cuando utilizamos esta función de activación se recomienda la inicialización de **He** en lugar de Xavier.

Pulsando en el link de la imagen siguiente puedes acceder a un sitio web donde jugar con las diferentes inicializaciones y ver sus efectos.