Módulo 4: Aprendizaje Automático

Temario de la Clase 6
13 de septiembre del 2024
Redes Neuronales Artificiales – Parte II

- Propagación hacia adelante
- Backpropagation
- Estabilidad numérica
- Gradientes que explotan y se desvanecen
- Métodos de inicialización
- Detención temprana
- Dropout
- Clipping del gradiente

Propagación hacia adelante

Hace referencia al cálculo y almacenamiento de variables intermedias de la red desde la entrada hasta la salida. Si tenemos una red con d entradas y una capa oculta con h neuronas, la matriz de pesos de la capa oculta será:

$$\boldsymbol{W}^{(1)} \in \mathbb{R}^{hxd}$$
$$\boldsymbol{x} \in \mathbb{R}^{d}$$

Entonces la variable intermedia de la capa oculta se calcula:

$$\boldsymbol{z} = \boldsymbol{W}^{(1)}\mathbf{x} + \boldsymbol{b_1}$$

Sobre la cual se aplica la función de activación:

$$\boldsymbol{h} = \emptyset(\boldsymbol{z}) \in \mathbb{R}^{h}$$

Si q es el número de salidas, tendremos una segunda matriz de pesos:

$$\boldsymbol{W}^{(2)} \in \mathbb{R}^{qxh}$$

Y la salida de la red será:

$$\boldsymbol{o} = \boldsymbol{W}^{(2)}\mathbf{h} + \boldsymbol{b_2}$$

Propagación hacia adelante

Por cada ejemplo tendremos un error asociado a una función de pérdida que denotamos como:

$$L = l(o, y)$$

Si también consideramos el término de regularización ℓ_2:

$$s = \frac{\lambda}{2}\left(\left\|\boldsymbol{W}^{(1)}\right\|_F^2 + \left\|\boldsymbol{W}^{(2)}\right\|_F^2\right)$$

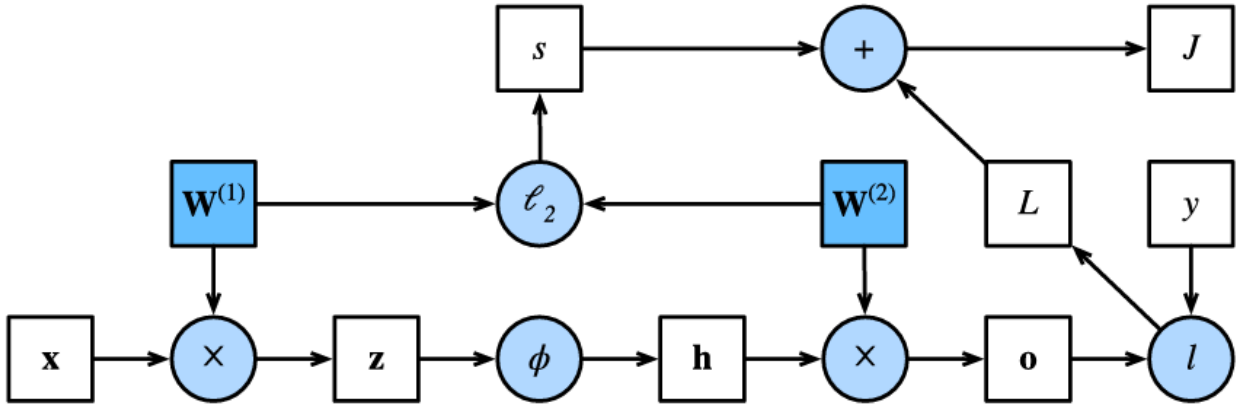
Donde la norma Frobenius era simplemente la norma ℓ_2 aplicada después de aplanar la matriz como un vector. Finalmente definíamos la función objetivo como:

$$J = L + s$$

Propagación hacia adelante

En cada iteración del ciclo de entrenamiento, el algoritmo realizará la propagación hacia adelante (*forward pass*), en base a los valores actuales de las matrices de pesos. Luego, se calculará el error mediante la función objetivo.

Una vez que el modelo ha sido creado y se han ajustado los hiperparámetros, la propagación hacia adelante será el único cálculo involucrado en la llamada al modelo.

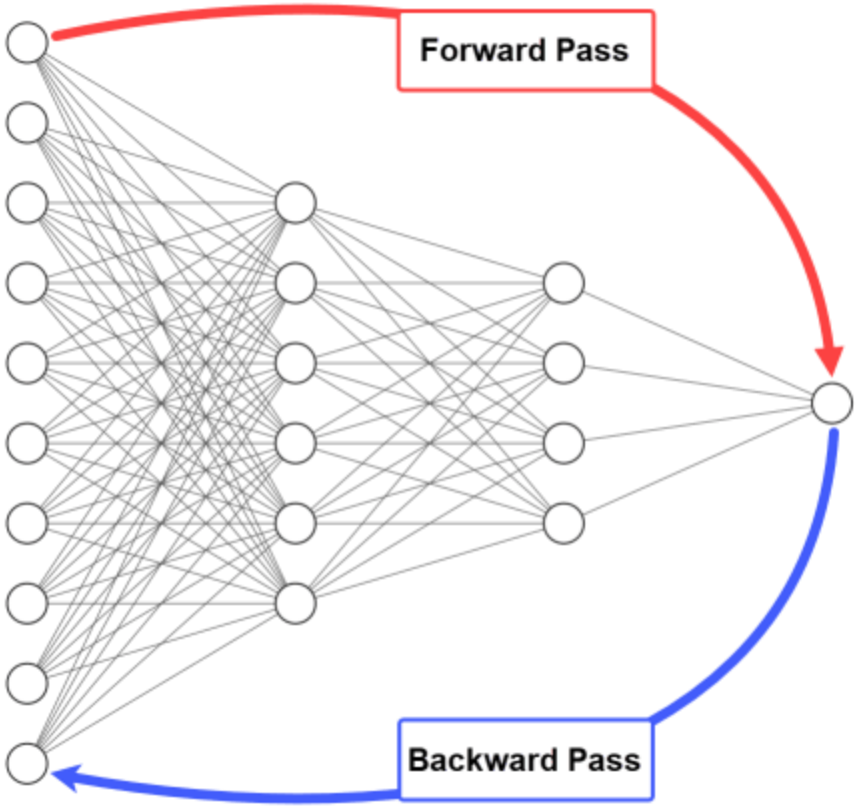


Backpropagation

La propagación hacia atrás es un paso clave del aprendizaje automático, ya que es cuando se ajustan los parámetros del modelo.

El término backpropagation se refiere al método para calcular los gradientes de los parámetros de la red. Para ello se recorre la misma en sentido inverso, desde la salida hacia la entrada, de acuerdo a la *regla de la cadena*.

Existen distintas implementaciones del backpropagation, a menudo llamadas **optimizadores**, que siguen distintas reglas para acelerar la convergencia.



Backpropagation

Durante el entrenamiento, la propagación hacia adelante y atrás se realiza de forma alternada.

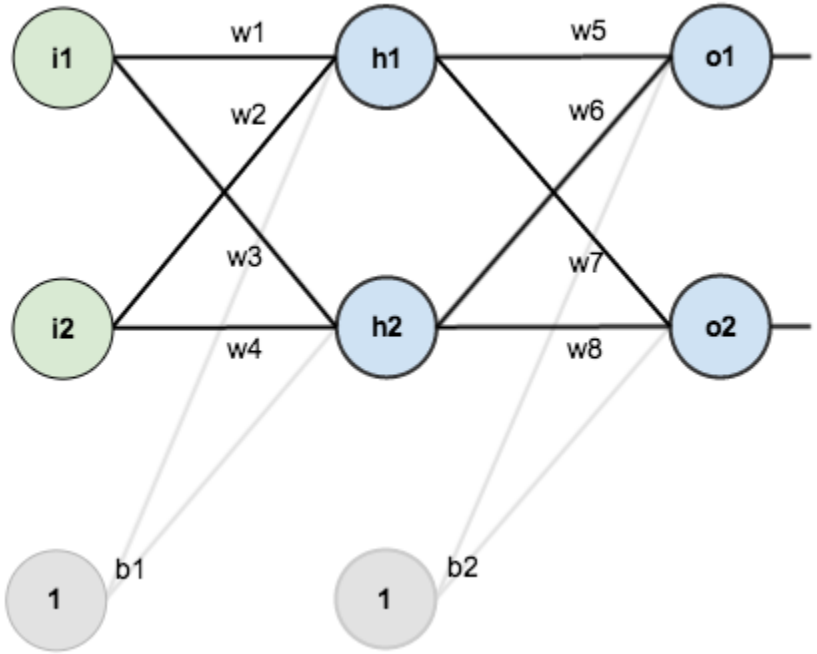
Los valores intermedios almacenados durante la forward pass servirán para los cálculos en la backpropagation. Las implementaciones de las librerías de ML buscan evitar realizar cálculos repetidos.

El gradiente consistirá en las derivadas parciales de la función objetivo con respecto a cada parámetro de la red, tomando los valores actuales de las salidas de los nodos.

El hecho de que haya que almacenar los valores intermedios determina que el entrenamiento requiera mucha más memoria que una simple predicción por parte del modelo. Esto está directamente relacionado con el número de capas y el tamaño del batch.

Backpropagation - Ejemplo

Si en esta red queremos calcular la componente del gradiente para los parámetros w5 y w1, es decir $\partial L/\partial W5$ y $\partial L/\partial W1$, debemos propagar el error desde atrás hacia adelante. Suponemos que no aplicamos regularización ℓ_2.

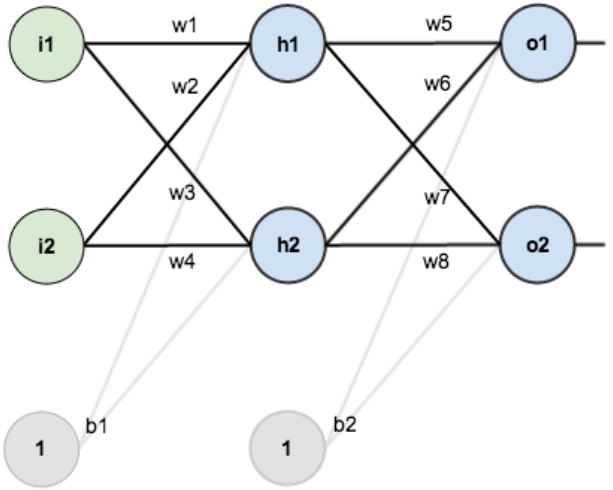


Backpropagation - Ejemplo

$$\frac{\partial L}{\partial W5} = \frac{\partial L}{\partial out_{o1}} \frac{\partial out_{o1}}{\partial in_{o1}} \frac{\partial in_{o1}}{\partial w5}$$

$$L = \frac{1}{2}(target_{o1} - out_{01})^2 + \frac{1}{2}(target_{o2} - out_{02})^2$$

$$\frac{\partial L}{\partial out_{o1}} = -(target_{o1} - out_{01})$$

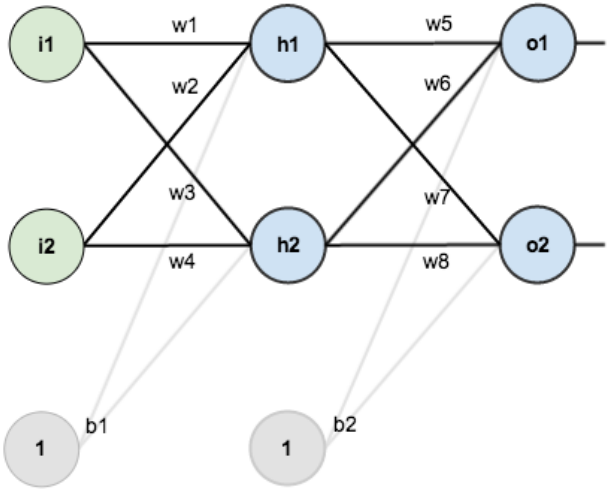


Backpropagation - Ejemplo

$$\frac{\partial L}{\partial W5} = \frac{\partial L}{\partial out_{o1}} \frac{\partial out_{o1}}{\partial in_{o1}} \frac{\partial in_{o1}}{\partial w5}$$

$$out_{o1} = \frac{1}{1 + e^{-in_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial in_{o1}} = out_{01}(1 - out_{01})$$

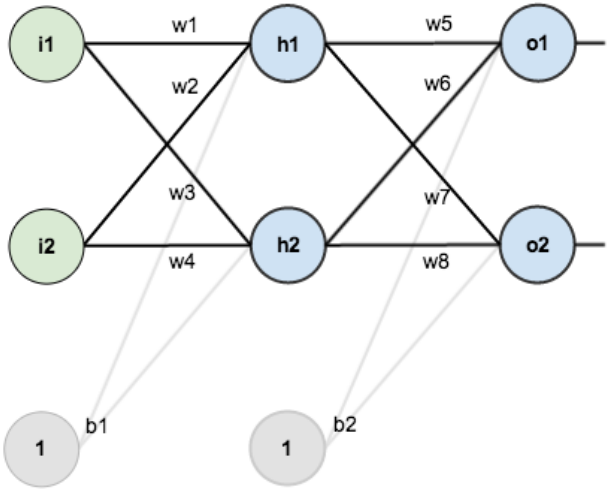


Backpropagation - Ejemplo

$$\frac{\partial L}{\partial W5} = \frac{\partial L}{\partial out_{o1}} \frac{\partial out_{o1}}{\partial in_{o1}} \frac{\partial in_{o1}}{\partial w5}$$

$$in_{o1} = w_5 out_{h1} + w_6 out_{h2} + b_2$$

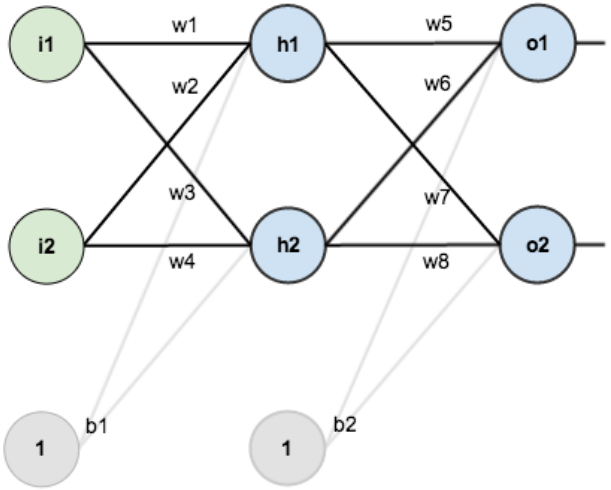
$$\frac{\partial in_{o1}}{\partial w5} = out_{h1}$$



Backpropagation - Ejemplo

$$\frac{\partial L}{\partial W1} = \frac{\partial L}{\partial out_{h1}} \frac{\partial out_{h1}}{\partial in_{h1}} \frac{\partial in_{h1}}{\partial w1}$$

$$\frac{\partial L}{\partial out_{h1}} = \frac{\partial E_{O1}}{\partial out_{o1}} + \frac{\partial E_{O2}}{\partial out_{o2}}$$

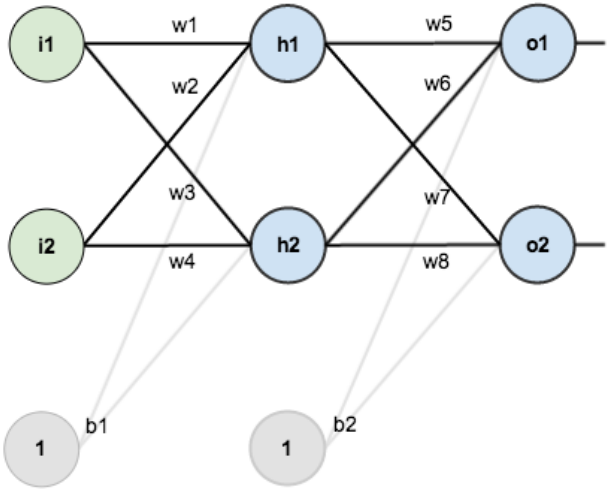


Backpropagation - Ejemplo

$$\frac{\partial L}{\partial W1} = \frac{\partial L}{\partial out_{h1}} \frac{\partial out_{h1}}{\partial in_{h1}} \frac{\partial in_{h1}}{\partial w1}$$

$$out_{h1} = \frac{1}{1 + e^{-in_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial in_{h1}} = out_{h1}(1 - out_{h1})$$

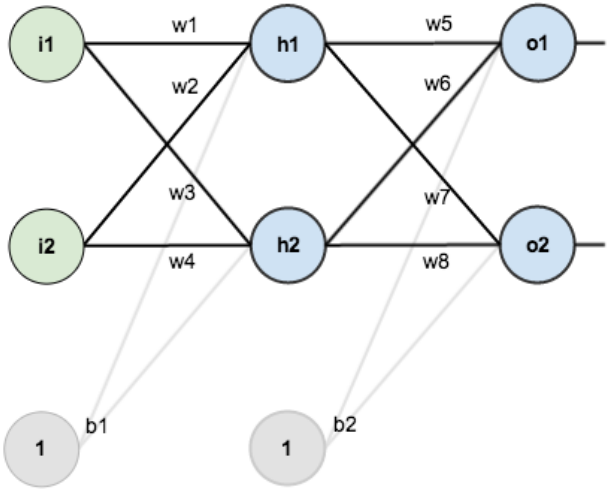


Backpropagation - Ejemplo

$$\frac{\partial L}{\partial W1} = \frac{\partial L}{\partial out_{h1}} \frac{\partial out_{h1}}{\partial in_{h1}} \frac{\partial in_{h1}}{\partial w1}$$

$$in_{h1} = w_1 i1 + w_2 i2 + b_1$$

$$\frac{\partial in_{h1}}{\partial w1} = i1$$



Estabilidad numérica

Hasta ahora no se prestó demasiada atención a la forma de inicializar los parámetros de una red neuronal. Sin embargo, dicha inicialización juega un rol muy importante durante el entrenamiento y puede ser crucial para mantener la estabilidad numérica en la ejecución de los algoritmos.

La elección de la forma de inicialización también se termina relacionando con la elección de las funciones de activación. Ambos factores van a influir en la rapidez de la convergencia de los algoritmos de optimización.

Si no se hacen elecciones correctas, la dinámica del modelo puede llevar a ciertos fenómenos conocidos como el **exploding gradient** (gradiente que explota) o el **vanishing gradient** (gradiente que se desvanece).

Estabilidad numérica

Si tenemos una red profunda con **L** capas, una entrada **x** y una salida **o** tendremos:

$$h^{(l)} = f(h^{(l-1)})$$
$$o = f_L * \cdots \ldots \ldots . * f_1(x)$$

El gradiente de la salida con respecto a cualquier conjunto de parámetros será:

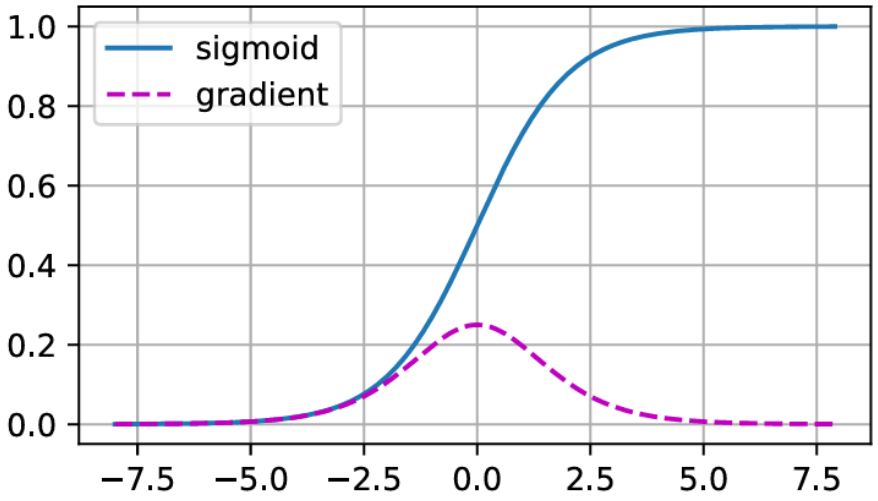
$$\frac{\partial o}{\partial W^{(l)}} = \frac{\partial h^{(L)}}{\partial h^{(L-1)}} \ldots \ldots \frac{\partial h^{(l+1)}}{\partial h^{(l)}} \frac{\partial h^{(l)}}{\partial W^{(l)}}$$

Por lo que tendremos el producto de *L-l* matrices lo que puede llevar a un resultado con números muy grandes (exploding gradient) o muy pequeños (vanishing gradient).

El problema de la estabilidad se hace más notorio a medida que la red se hace más profunda.

Vanishing Gradient

Debido a su eficacia para representar fenómenos que se disparan o no (como las neuronas) la función sigmoidea fue una de las primeras favoritas para su uso en RNA. Sin embargo puesto que su gradiente se *desvanece* para valores que son muy positivos o muy negativos, en una red profunda, puede que el gradiente disminuya a cero a partir de una determinada capa. Es por eso que la función ReLU ganó popularidad en redes de aprendizaje profundo.



Exploding Gradient

Se puede ver como, si se multiplican 100 matrices inicializadas con valores tomados de una distribución Gaussiana (con varianza = 1), el producto de las mismas tiende hacia valores muy grandes, por lo tanto, a *explotar*. Cuando una red profunda se inicializa de dicha forma, no hay forma de que converja un método basado en descenso del gradiente.

```
M = torch.normal(0, 1, size=(4, 4))
print('a single matrix \n',M)
for i in range(100):
    M = M @ torch.normal(0, 1, size=(4, 4))
print('after multiplying 100 matrices\n', M)
```

```
a single matrix
 tensor([[-0.8755, -1.2171,  1.3316,  0.1357],
        [ 0.4399,  1.4073, -1.9131, -0.4608],
        [-2.1420,  0.3643, -0.5267,  1.0277],
        [-0.1734, -0.7549,  2.3024,  1.3085]])
after multiplying 100 matrices
 tensor([[-2.9185e+23,  1.3915e+25, -1.1865e+25,  1.4354e+24],
        [ 4.9142e+23, -2.3430e+25,  1.9979e+25, -2.4169e+24],
        [ 2.6578e+23, -1.2672e+25,  1.0805e+25, -1.3072e+24],
        [-5.2223e+23,  2.4899e+25, -2.1231e+25,  2.5684e+24]])
```

Inicialización de parámetros

Una inicialización cuidadosa puede ayudar a mitigar los efectos anteriormente mencionados.

Hasta ahora vimos una inicialización basada en tomar valores aleatorios de alguna distribución (Gaussiana, uniforme, etc.), lo cual funciona muy bien en modelos de tamaño moderado. Este era el enfoque más utilizado en la década del 2000.

Para redes más profundas, se han presentado en los últimos 15 años artículos científicos que buscan mejorar la performance de las RNA basados en análisis estadísticos de la propagación de los valores numéricos.

Inicialización Xavier

Presentada en el artículo “Understanding the difficulty of training deep feedforward neural networks” (2010) por Xavier Glorot y Yoshua Bengio, consiste en inicializar los parámetros tomandolos de alguna distribución y teniendo en cuenta el número de entradas y salidas de cada capa.

Glorot y Bengio propusieron que para que la red fluya apropiadamente, en cada capa debía cumplirse que la varianza de las salidas debía ser igual a la varianza de las entradas.

Además, los gradientes debían tener igual varianza antes y después de pasar por cada capa durante la brackpropagation. Esto evitaría que los gradientes exploten o se desvanezcan.

Inicialización Xavier

Se usa en redes no muy profundas o con funciones de activación sigmoide o tanh. Las dos formas más comunes de implementar dicho método son:

- **Inicialización Xavier Normal**: los pesos de la capa se obtienen a partir de una distribución Gaussiana con mediana igual a 0 y desviación estandard σ que cumpla la siguiente condición:

$$\sigma^2 = \frac{2}{n_{in} + n_{out}}$$

- **Inicialización Xavier Uniforme**: los pesos se obtienen de una distribución uniforme en el rango [-x, x], donde:

$$x = \sqrt{\frac{6}{n_{in} + n_{out}}}$$

Inicialización He

Introducida por He Kaiming en el paper “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification” (2015) como una mejora para redes profundas que utilicen ReLU (o sus variantes) como función de activación. Esta función hace cero los valores negativos por lo que disminuye la varianza. Funciona en redes profundas al escalar los valores con el número de entradas a la capa.

He Normal: los pesos se toman de una distribución normal con media igual a cero y varianza:

$$\sigma^2 = \frac{2}{n_{in}}$$

He Uniforme: los pesos se toman de una distribución uniforme en el rango [x, -x] donde:

$$x = \sqrt{\frac{6}{n_{in}}}$$

Otros métodos

Inicialización ortogonal: utilizada muy comúnmente en redes neuronales recurrentes. Consiste en tomar la matriz de pesos como una matriz ortogonal para que los mismos no presenten correlación.

Inicialización dispersa: un porcentaje de los parámetros toman valores iniciales nulos, lo que permite una mayor rapidez en los cálculos.

Inicialización Dirac: utilizada en redes neuronales convolucionales.

torch.nn.init

- `calculate_gain()`
- `uniform_()`
- `normal_()`
- `constant_()`
- `ones_()`
- `zeros_()`
- `eye_()`
- `dirac_()`
- `xavier_uniform_()`
- `xavier_normal_()`
- `kaiming_uniform_()`
- `kaiming_normal_()`
- `trunc_normal_()`
- `orthogonal_()`
- `sparse_()`

Generalización

Objetivo del aprendizaje automático:
Descubrir patrones generales que permitan predecir en nuevos datos.
Optimización como herramienta, no fin en sí misma.

Rendimiento de redes neuronales profundas:
Generalizan bien en diversas áreas:
Visión por computadora, lenguaje natural, salud, entre otros.

Desafíos en la teoría del aprendizaje profundo:
La optimización y la generalización aún no están completamente entendidas.
Los modelos ajustan bien los datos de entrenamiento, pero generalizar sigue siendo el mayor reto.

Evolución en la práctica:
Nuevas heurísticas y enfoques prácticos permiten mejorar los modelos.
Las técnicas desarrolladas por profesionales ayudan a producir modelos efectivos en la práctica.

Situación actual:
La teoría está en desarrollo y aún falta una explicación completa.
La investigación y las técnicas aplicadas avanzan rápidamente

Generalización

Teorema "No Free Lunch"
Algoritmos de aprendizaje dependen de distribuciones de datos.
Sesgos inductivos reflejan cómo los humanos interpretan el mundo.

Proceso de Entrenamiento
Ajustar datos de entrenamiento.
Estimar error de generalización con datos de validación.

Brecha de Generalización
Diferencia entre ajuste en entrenamiento y validación.
El sobreajuste ocurre cuando la brecha es grande.

Deep Learning y Complejidad
Modelos pueden ajustarse perfectamente a datos de entrenamiento.
Complejidad adicional puede reducir el error de generalización ("doble descenso").

Limitaciones de la Teoría Tradicional
No explica completamente cómo las redes neuronales profundas generalizan.

Generalización

Modelos no paramétricos:
Crecen en complejidad con más datos.
Ejemplo: k-nearest neighbors (k-NN).

k-NN (k-nearest neighbors):
Memoriza datos de entrenamiento.
Predicciones basadas en la distancia a los vecinos más cercanos.

Redes neuronales sobreparametrizadas:
Interpolan perfectamente los datos.
Comportamiento similar a modelos no paramétricos.

Investigación reciente:
Jacot et al. (2018): Redes neuronales grandes son equivalentes a métodos de kernel no paramétricos.
Introducción del kernel tangente neural.

Early Stopping

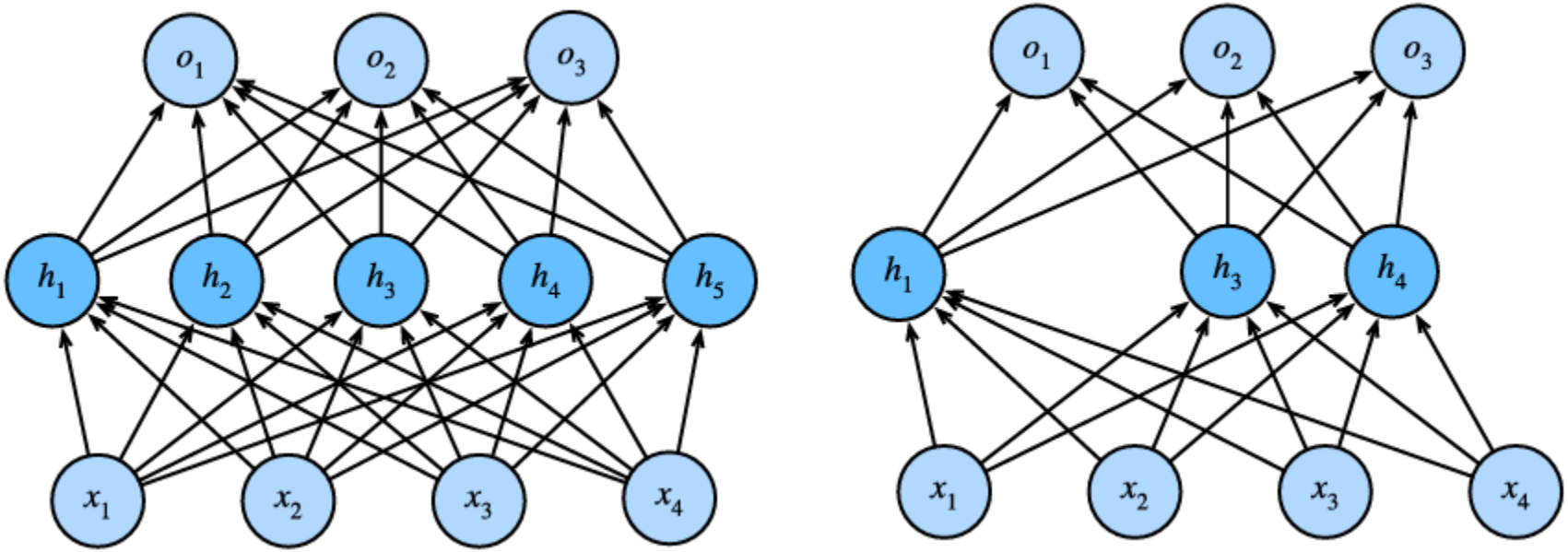
Es una técnica básica de regularización que se basa en restringir el número de epochs de entrenamiento en base a un criterio de parada. Dicho criterio se puede establecer monitoreando el error de validación y detener el entrenamiento cuando dicho error no ha disminuido una determinada cantidad durante un determinado número de epochs. También se denomina **criterio de paciencia**.

Mejora el potencial de una mejor generalización, sobre todo cuando los labels o los datos contienen ruido. Distintos papers han encontrado que en la presencia de datos ruidosos o mal etiquetados, las redes interpolan primero los datos correctos y luego los datos corrompidos, por lo que el early stopping es importante para no perder generalización.

Para modelos muy grandes de Deep learning, los criterios basados en early stopping pueden ahorrar tiempo valioso de ejecución en GPUs.

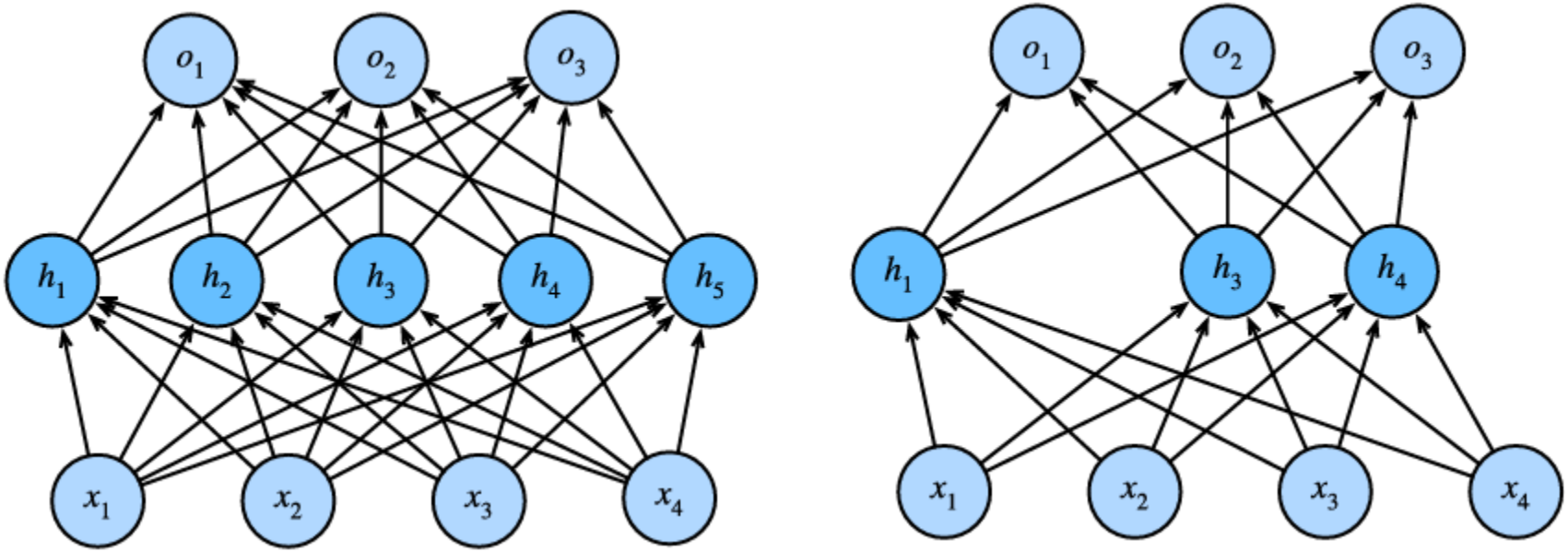
Dropout

Es una técnica de regularización que consiste en hacer cero la salida de ciertos nodos durante el entrenamiento, para evitar la co-adaptación de neuronas. Cuando esto sucede, los otros nodos deben adaptarse para compensar el efecto de los nodos perdidos, lo que genera múltiple representaciones internas aprendidas por la red.



Dropout

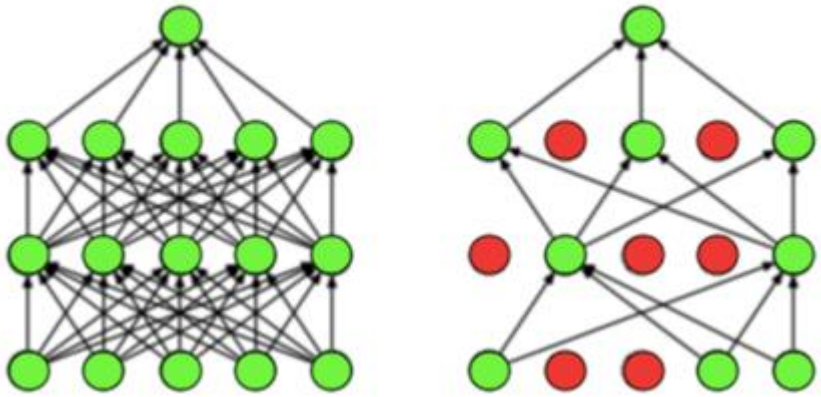
A pesar de parecer destructivo, el dropout mejora la eficacia del modelo al volverse menos sensible a los pesos específicos de la red, lo que mejora la generalización y evita el overfit a los datos de entrenamiento.



Dropout

En la práctica, el dropout se introduce como una capa intermedia en la red. Cuando esto se hace, durante el entrenamiento el algoritmo va a descartar un conjunto de nodos de manera aleatoria en base a una probabilidad p. Por ejemplo, si p=0.2, 1 de cada 5 nodos será descartado en cada iteración.

Para compensar el efecto de los nodos perdidos, la salida de esta capa será escalada en un factor $\frac{1}{1-p}$ para que el valor promedio del tensor se mantenga.



Gradient Clipping

Es una técnica que limita el valor de los gradientes a un rango específico, evitando que se vuelvan demasiado grandes durante la actualización de los pesos. De esta manera, se controla la magnitud del paso que realiza el optimizador en cada actualización, lo que ayuda a estabilizar el proceso de entrenamiento.

Además previene el desbordamiento numérico ya que cuando los gradientes explotan, los valores pueden exceder los límites que pueden ser representados por los números en el hardware, causando errores.

Se aplica para mejorar la convergencia de las redes neuronales profundas, como RNNs y LSTMs, y para que lo hagan de manera más estable.

Gradient Clipping

Existen dos implementaciones:

- **Clipping por valor** (Value Clipping):Los gradientes que superan un valor umbral se limitan a ese valor. Si un gradiente es mayor que el valor máximo permitido, se "corta" para que no exceda ese umbral. Ejemplo: si el umbral es 5, cualquier gradiente con un valor mayor a 5 se limitará a 5.

- **Clipping por norma** (Norm Clipping):Se limita la norma del vector de gradientes en lugar de los gradientes individuales. Si la norma total del vector de gradientes supera un valor umbral, se escala todo el vector para que su norma esté dentro del límite. Ejemplo: si la norma de los gradientes excede un valor como 1, el vector completo se escala para que su norma sea igual a 1, manteniendo las proporciones relativas de los gradientes.