KIBERNUM
IT Academy

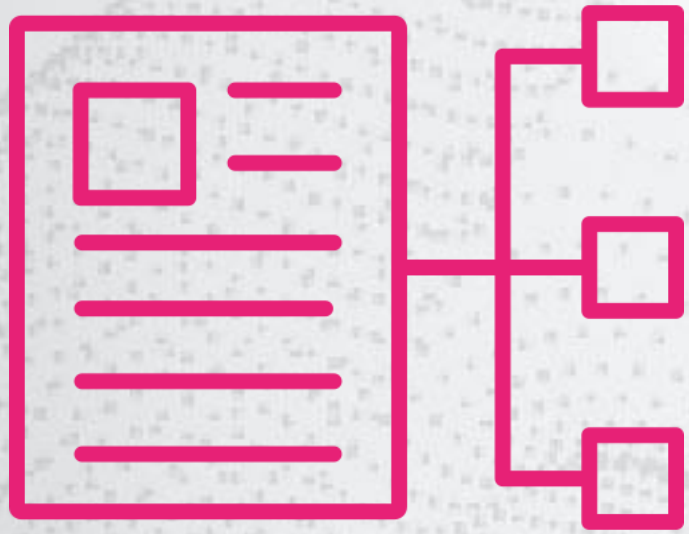
Módulo 7
Clase 3-1

Perceptrón Multilayer

Introducción Deep Learning

Aprendizajes

Describir los conceptos fundamentales las redes neuronales y su utilidad para la resolución de problemas de aprendizaje de máquina



KIBERNUM

Contenido

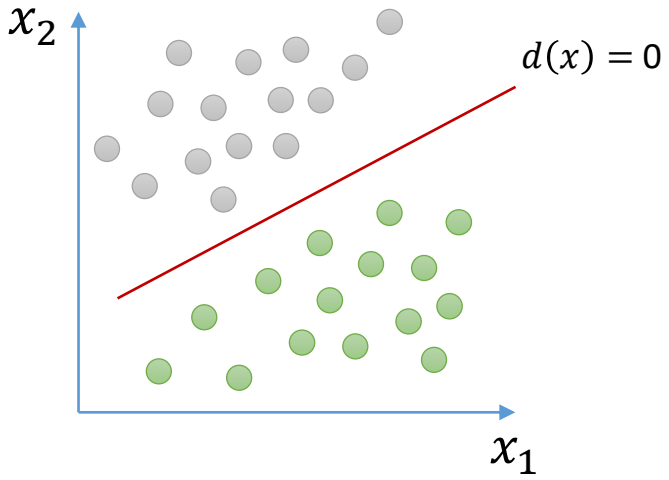
- Perceptrón multilayer
- Arquitectura de una red neuronal
- Entrenamiento de una red neuronal
- Función de costo
- Backpropagation
- Ventajas y desventajas

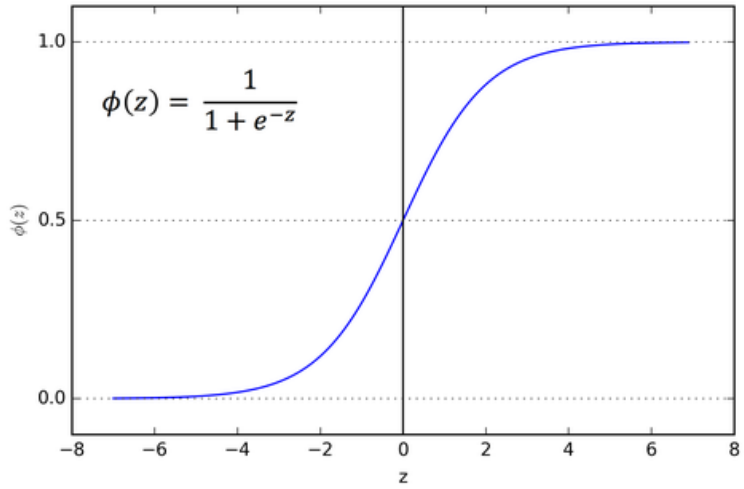
KIBERNUM

Perceptrón Multilayer (MLP)

Regresión Logística

- Predice la probabilidad de una clase (¿es verde ? [1: verde , 0: no verde])
- Función de decisión $d(x) = w_0 + w_1 x_1 + w_2 x_2$
- Algoritmo $z(x) = \phi(d(x))$

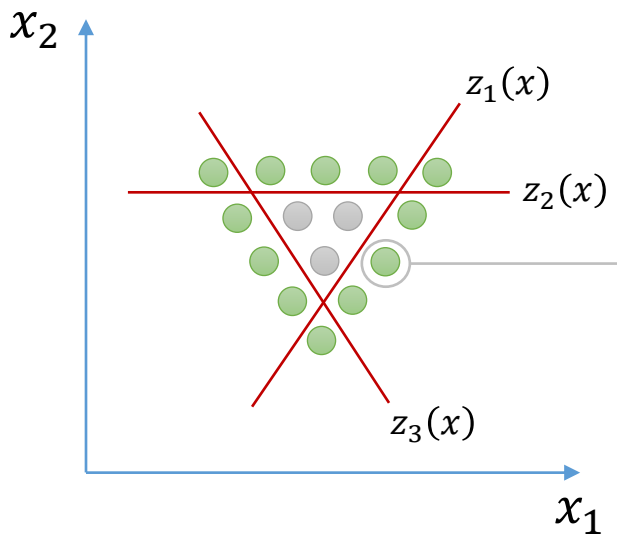




KIBERNUM

El problema del triángulo

Supongamos una figura que no es linealmente separable.



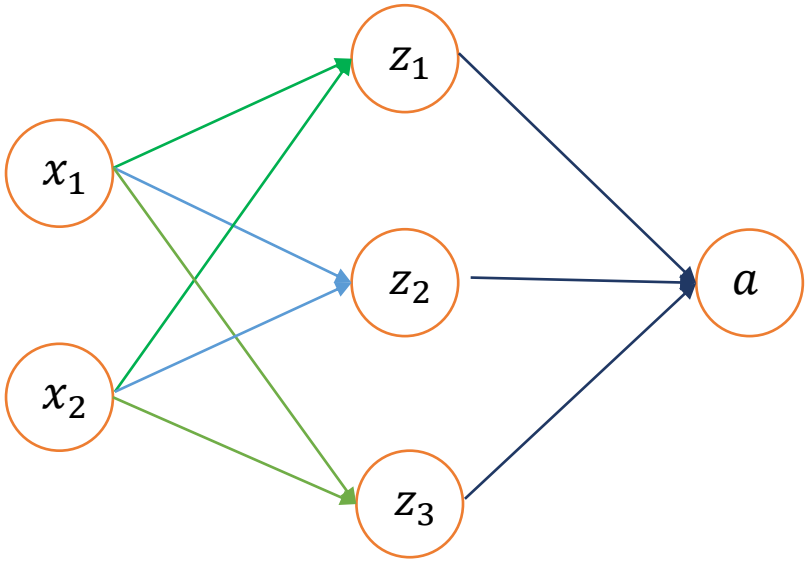
$z_1(x)$	$z_2(x)$	$z_3(x)$	y
0.4	0.7	0.8	0
0.6	0.7	0.3	1
0.1	0.4	0.4	0
...	...	...	...

Ahora podemos construir un nuevo modelo lineal con los features z_1 , z_2 y z_3

$$a(x) = \phi(w_0 + w_1 z_1(x) + w_2 z_2(x) + w_3 z_3(x))$$

Multi Layer Perceptron

Al llevar el procedimiento anterior a un grafo, obtenemos el siguiente modelo:

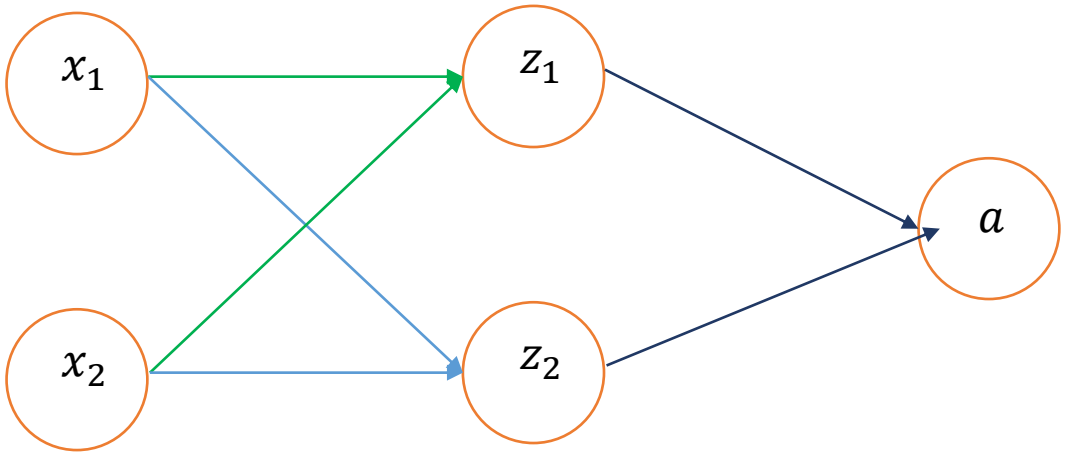


: nodo, recibe información

: relación, lleva un peso w

Quizz

¿Qué pasa si no utilizamos una función de activación?



Nuestro algoritmo es una simpática función lineal!!

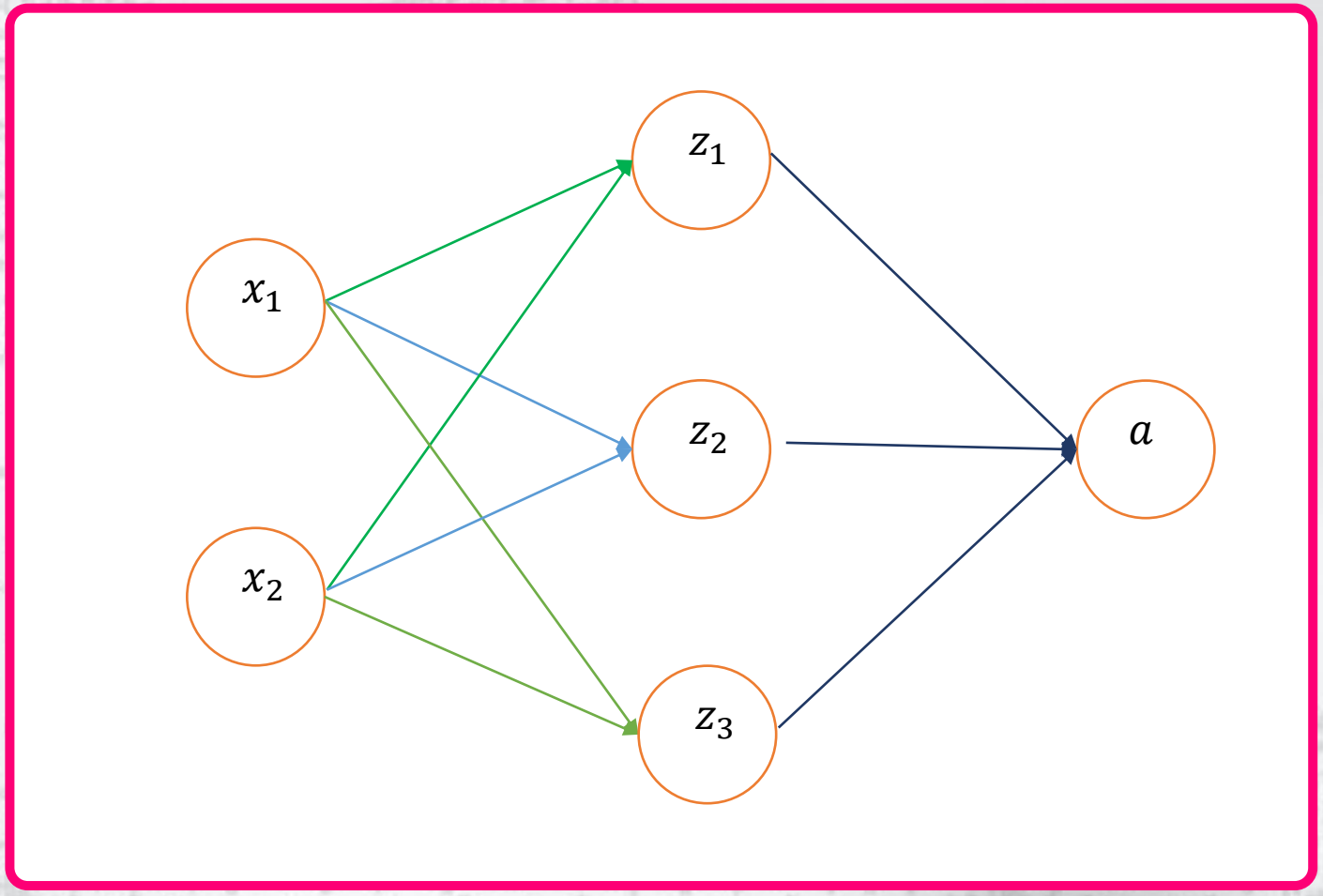
$$z_1 = w_{0,1} + w_{1,1}x_1 + w_{2,1}x_2$$

$$z_2 = w_{0,2} + w_{1,2}x_1 + w_{2,2}x_2$$

$$\mathrm{a} = w_0 + w_1 z_1 + w_2 z_2 = w_0 + \left(w_1 w_{1,1} + w_2 w_{1,2}\right)x_1 + \left(w_1 w_{2,1} + w_2 w_{2,2}\right)x_2$$

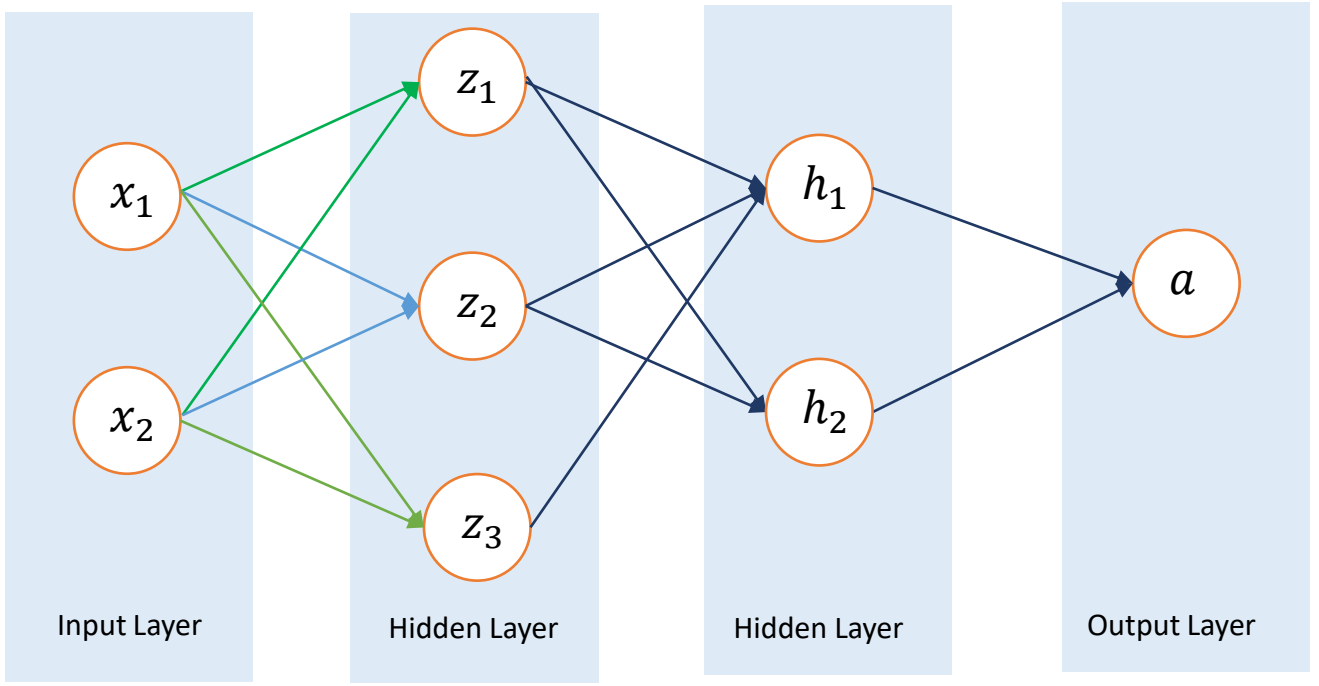
Quizz

¿Cuántos parámetros tiene el siguiente perceptrón multilayer?



Arquitectura de una RN

Es posible armar varias capas en el MLP.



Arquitectura de el MLP:

- Número de Layers
- Número de neuronas en cada layer
- Funciones de activación

Hidden Layer:

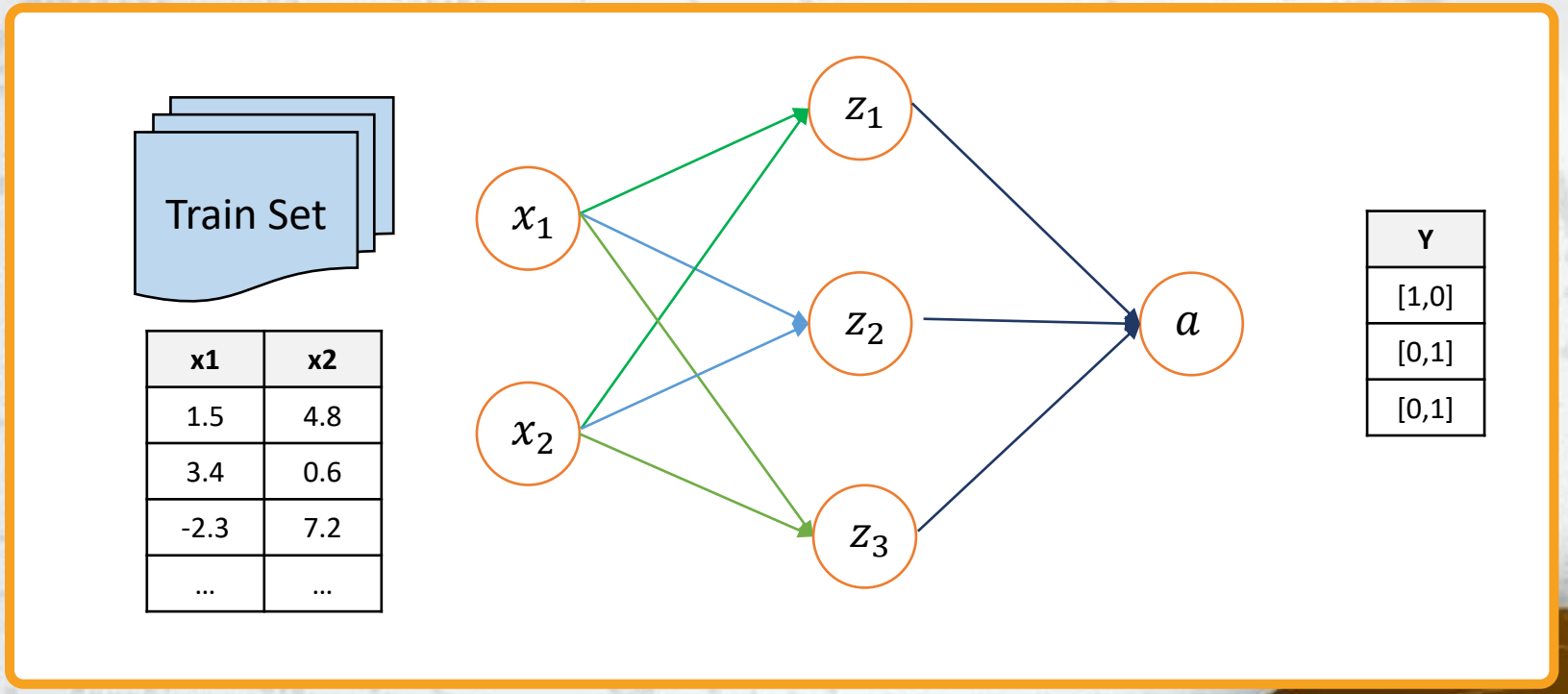
- Dense Layer (Fully Connected)
- Convolutional Layer
- Pooling Layer
- Etc…

Entrenamiento de una Red Neuronal

Objetivo

Dado un set de datos de entrenamiento (mediciones y etiquetas), determinar los parámetros (pesos e interceptos) que minimizan el costo.

¿Qué costo? Cross-Entropy



KIBERNUM

Cross-Entropy

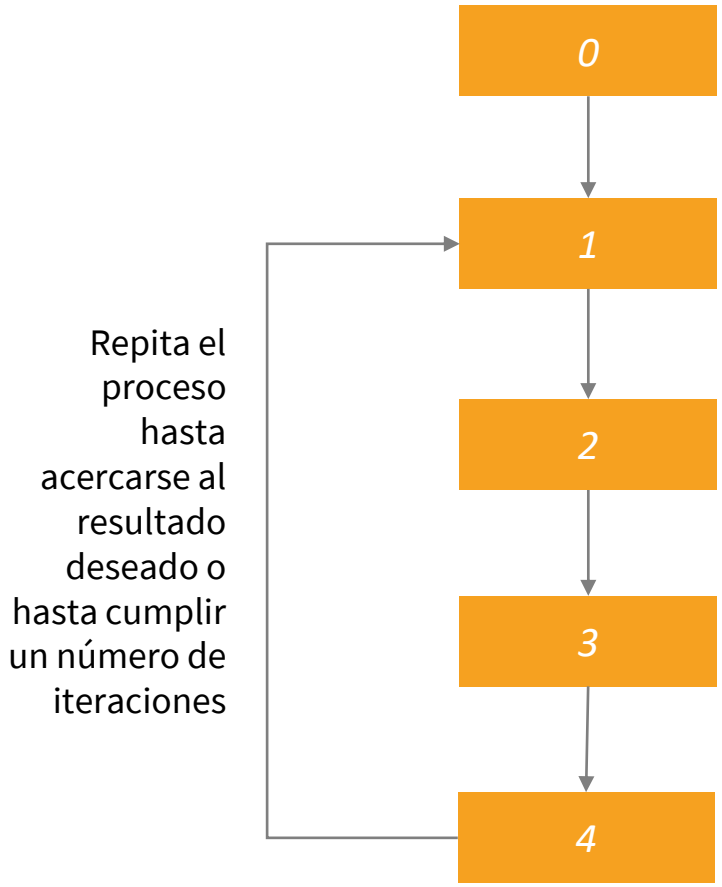
La función de costo a optimizar es **Cross-Entropy**. Esta métrica mide la disimilaridad entre un valor predicho y un valor real.

$$CrossEntropy = -\sum_{k=1}^{K}[y = k]log\frac{e^{z_k}}{\sum_{j=1}^{K}e^{z_j}} = -log\frac{e^{z_y}}{\sum_{j=1}^{K}e^{z_j}}$$

Supongamos que es un problema con 3 clases (K=3), y para un valor de x, se calcula mediante un MLP las probabilidades para cada clase:

Valor predicho	Valor real	Cross-Entropy
(1.0, 0.0, 0.0)	(1, 0, 0)	-1*log1 – 0*log 0 – 0*log 0 = 0
(0.0, 1.0, 0.0)	(1, 0, 0)	-1*log 0 – 0*log 1 – 0*log 0 = +infinito
(0.5, 0.25, 0.25)	(1, 0, 0)	-1*log 0.5 – 0*log 0.25 – 0*log 0.25 = 0.693 (aprox)

Entrenando un MLP



Asignación inicial de los pesos

Provea al perceptrón de una entrada de datos de entrenamiento (mediciones y etiquetas)

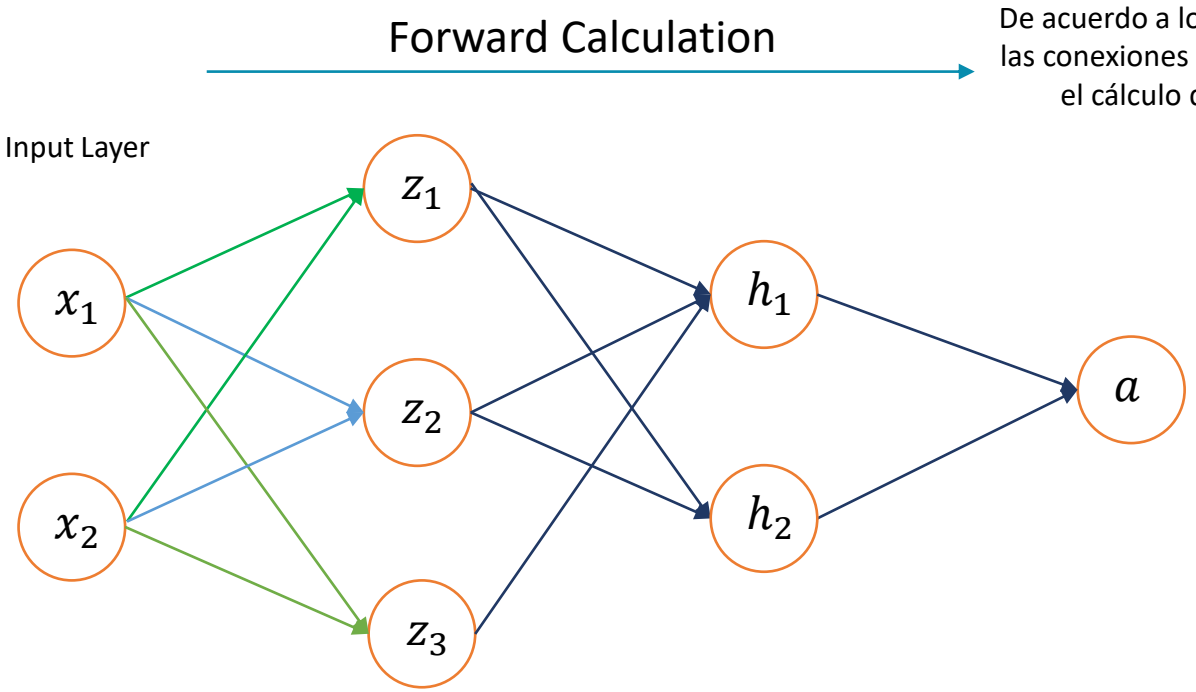
A partir de las mediciones, computar la respuesta en la red (forward)

Contabilice el error. (¿cuán alejado quedamos de las respuestas correcta?)

Ajuste los pesos de acuerdo al error para disminuirlo (backward)

Back Propagation

Backpropagation es un algoritmo utilizado en el entrenamiento de redes neuronales artificiales. Su objetivo es ajustar los pesos y sesgos de la red para minimizar el error de la salida en relación a las salidas deseadas.

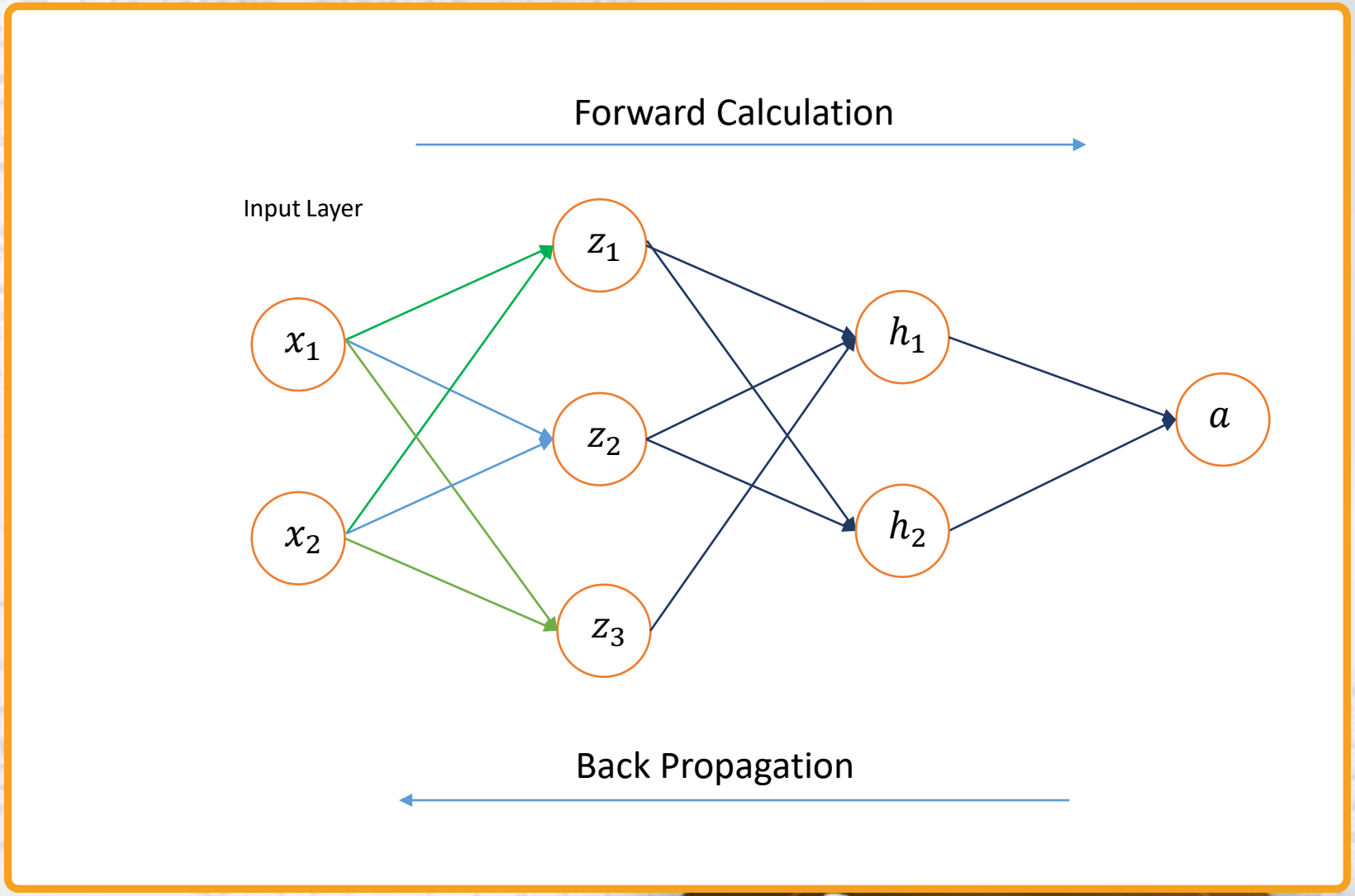


Back Propagation

El proceso comienza propagando la entrada de la red neuronal hacia adelante, capa por capa, calculando las salidas de cada neurona y pasándolas a la siguiente capa. Una vez que se ha calculado la salida de la red, se compara con la salida deseada y se calcula el error.

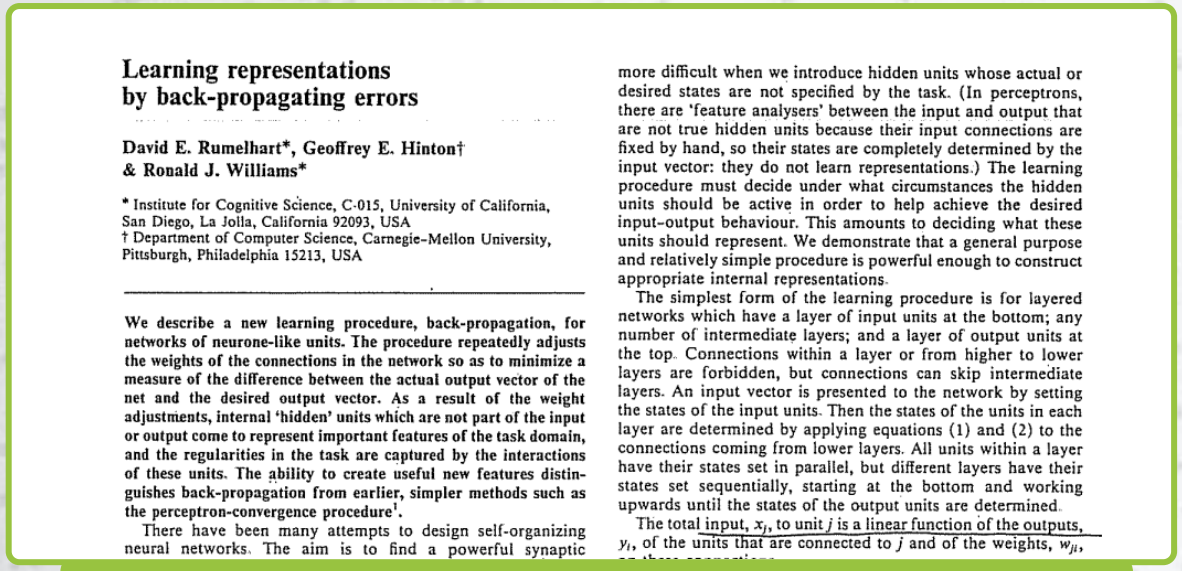
Luego, se propaga el error hacia atrás a través de la red, capa por capa, ajustando los pesos y sesgos en cada neurona para reducir el error en la salida. Este proceso se repite muchas veces hasta que el error de la red se reduce a un nivel aceptable.

Para calcular cómo ajustar los pesos y sesgos, se utiliza el gradiente descendente, que se basa en la derivada de la función de error con respecto a los pesos y sesgos. El gradiente descendente indica la dirección en la que se deben ajustar los pesos y sesgos para minimizar el error, y la magnitud del ajuste se determina mediante la tasa de aprendizaje.



Back Propagation

El algoritmo de backpropagation se introdujo originalmente en la década de 1970, pero su importancia no se apreció completamente hasta un famoso artículo de 1986 de David Rumelhart, Geoffrey Hinton y Ronald Williams. Este documento describe varias redes neuronales en las que la retropropagación funciona mucho más rápido que los enfoques de aprendizaje anteriores, lo que hace posible utilizar redes neuronales para resolver problemas que anteriormente habían sido insolubles. Hoy, el algoritmo de backpropagation es el caballo de batalla del aprendizaje en redes neuronales.



Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji},

https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

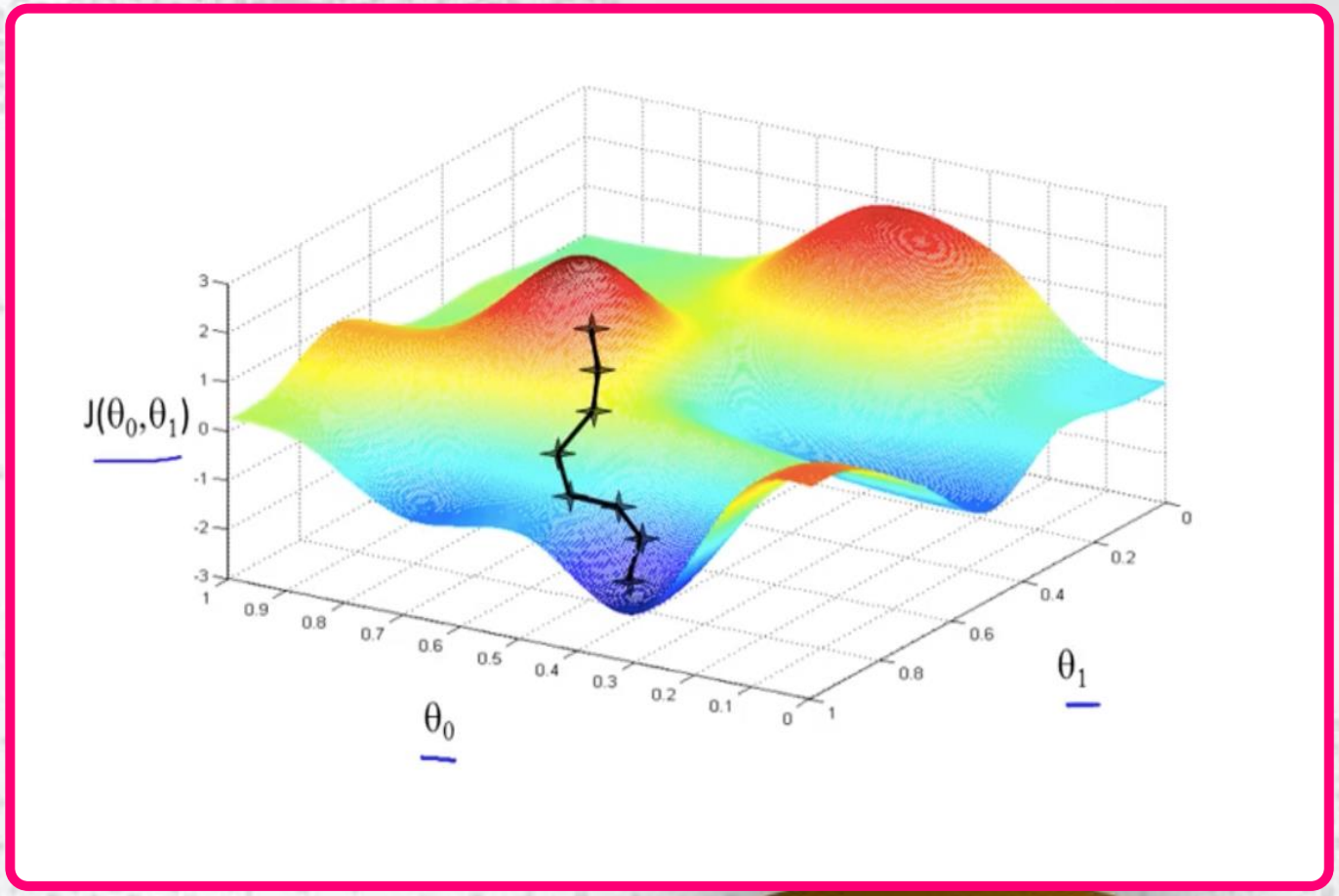
Geoffrey Hinton

KIBERNUM

Matemática de Back Propagation

El algoritmo de backpropagation utiliza cálculo diferencial y álgebra lineal para ajustar los pesos y sesgos de una red neuronal durante el proceso de entrenamiento. A continuación, se describen los conceptos matemáticos involucrados en este proceso.

Supongamos que tenemos una red neuronal con una función de error E, que se define como la diferencia entre las salidas deseadas y las salidas reales de la red. El objetivo del algoritmo de backpropagation es ajustar los pesos y sesgos de la red para minimizar esta función de error E. Para hacerlo, se utiliza el **gradiente descendente**, que es una técnica de optimización que se basa en la **derivada parcial de la función de error con respecto a cada peso y sesgo de la red**.



KIBERNUM

Matemática de Back Propagation

El gradiente descendente indica la dirección en la que se deben ajustar los pesos y sesgos para minimizar la función de error. Para actualizar los pesos y sesgos, se utiliza la regla de actualización de pesos de la siguiente manera:

$$W_{(t+1)} = W_{(t)} - \alpha \frac{\partial E}{\partial W}$$

Donde $W_{(t)}$ son los pesos de la red en el momento t, α es la tasa de aprendizaje y $\frac{\partial E}{\partial W}$ es la derivada parcial de la función de error E con respecto a los pesos W.

La regla de actualización de sesgos es similar:

$$b_{(t+1)} = b_{(t)} - \alpha \frac{\partial E}{\partial b}$$

Donde $b_{(t)}$ son los sesgos de la red en el momento t y $\frac{\partial E}{\partial b}$ es la derivada parcial de la función de error E con respecto a los sesgos b.

Matemática de Back Propagation

La derivada parcial $\frac{\partial E}{\partial W}$ se calcula utilizando la regla de la cadena de la derivada:

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial W}$$

Donde y es la salida de una neurona, z es la entrada de la siguiente neurona y W es el peso que conecta ambas neuronas.

El proceso se repite capa por capa, propagando el error hacia atrás y actualizando los pesos y sesgos de la red. Este proceso se realiza iterativamente hasta que se alcanza un nivel aceptable de precisión en la predicción de la red.

Matemática de Back Propagation

Si quieres ahondar más en la matemática y algoritmia de backpropagation, te recomendamos el siguiente enlace:

Compound expressions with chain rule

Lets now start to consider more complicated expressions that involve multiple composed functions, such as $f(x,y,z) = (x+y)z$. This expression is still simple enough to differentiate directly, but we'll take a particular approach to it that will be helpful with understanding the intuition behind backpropagation. In particular, note that this expression can be broken down into two expressions: $q = x + y$ and $f = qz$. Moreover, we know how to compute the derivatives of both expressions separately, as seen in the previous section. f is just multiplication of q and z, so $\frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$, and q is addition of x and y so $\frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$. However, we don't necessarily care about the gradient on the intermediate value q - the value of $\frac{\partial f}{\partial q}$ is not useful. Instead, we are ultimately interested in the gradient of f with respect to its inputs x, y, z. The **chain rule** tells us that the correct way to "chain" these gradient expressions together is through multiplication. For example, $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$. In practice this is simply a multiplication of the two numbers that hold the two gradients. Lets see this with an example:

```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
dqdx = 1.0
dqdy = 1.0
# now backprop through q = x + y
dfdx = dfdq * dqdx  # The multiplication here is the chain rule!
dfdy = dfdq * dqdy
```

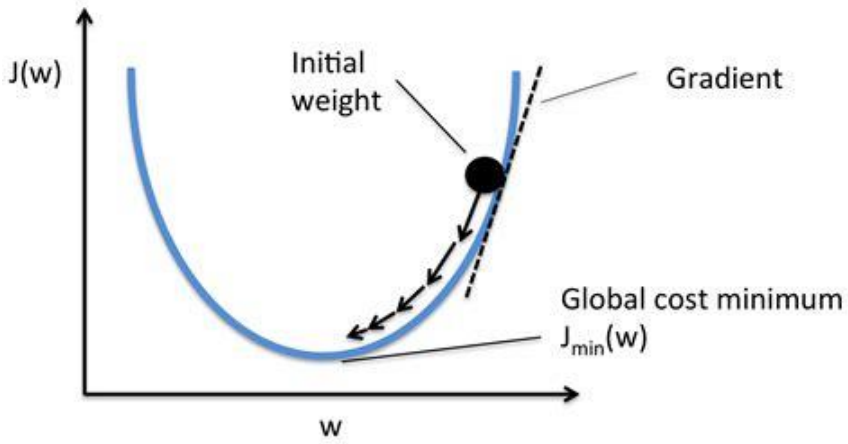
https://cs231n.github.io/optimization-2/

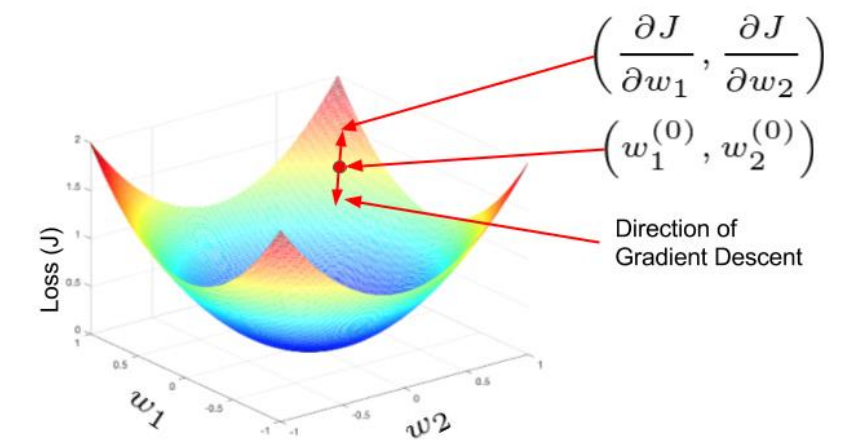
KIBERNUM

Algoritmos de Optimización

Los algoritmos de optimización en una red neuronal son técnicas utilizadas para ajustar los parámetros de la red, como los pesos y los sesgos, con el objetivo de minimizar una función de pérdida o de costo. Estos algoritmos son necesarios para que la red neuronal pueda aprender de los datos de entrenamiento y realizar predicciones precisas en nuevos datos.

Entre los algoritmos de optimización más comunes se encuentra el gradiente descendente, que se utiliza ampliamente en el aprendizaje profundo. Este algoritmo utiliza el cálculo del gradiente de la función de pérdida para ajustar los pesos y los sesgos de la red en pequeños incrementos, en la dirección opuesta al gradiente.





KIBERNUM

Algoritmos de Optimización

A continuación, se describen algunos de los algoritmos de optimización más comunes:

- **Gradiente descendente:** Es el algoritmo de optimización más simple y ampliamente utilizado en deep learning. Se actualizan los pesos en la dirección opuesta al gradiente de la función de pérdida.
- **SGD (Stochastic Gradient Descent):** Es una variante del gradiente descendente que utiliza un subconjunto aleatorio de los datos de entrenamiento en cada iteración. Esto lo hace más rápido y más escalable para grandes conjuntos de datos.
- **Momentum:** Es un algoritmo de optimización que utiliza un promedio móvil de los gradientes anteriores para ajustar los pesos de la red. Esto ayuda a acelerar la convergencia y suavizar la trayectoria de la optimización.
- **RMSprop (Root Mean Square Propagation):** Es un algoritmo de optimización que utiliza una tasa de aprendizaje adaptativa y una media móvil exponencial de los cuadrados de los gradientes para ajustar los pesos de la red. Esto ayuda a evitar que el algoritmo se atasque en mínimos locales y a mejorar la convergencia.

KIBERNUM

Algoritmos de Optimización

- **Adam (Adaptive Moment Estimation):** Es un algoritmo de optimización que combina los beneficios del momentum y RMSprop. Utiliza una tasa de aprendizaje adaptativa y una media móvil exponencial de los gradientes y sus cuadrados para ajustar los pesos de la red.

- **Adagrad:** Es un algoritmo de optimización que adapta la tasa de aprendizaje de cada peso de la red de acuerdo con la frecuencia con la que se ha actualizado. Esto ayuda a dar más importancia a los parámetros que se actualizan con menos frecuencia.

- **Adadelta:** Es una variante del algoritmo Adagrad que adapta la tasa de aprendizaje utilizando una media móvil exponencial de los cuadrados de los gradientes.

- **Nadam:** Es una variante del algoritmo Adam que utiliza la norma de las actualizaciones de los pesos para ajustar la tasa de aprendizaje y la dirección del momentum. Esto ayuda a evitar que el algoritmo se atasque en mínimos locales.

Como se puede apreciar, existen varios algoritmos de optimización utilizados en deep learning, cada uno con sus propias ventajas y desventajas. La elección del algoritmo adecuado depende del problema específico y de las características de los datos de entrenamiento.

KIBERNUM

Resumen

Resumen de conceptos

Los aspectos más relevantes de una red neuronal incluyen:

Arquitectura: La arquitectura de una red neuronal define la forma en que las neuronas se conectan entre sí y cómo se propagan las señales a través de la red (capas ocultas, neuronas)

Función de activación: La función de activación determina cómo las neuronas en la red producen su salida. Las funciones de activación más comunes incluyen la función sigmoide y la función ReLU.

Entrenamiento: El entrenamiento de una red neuronal implica ajustar los parámetros de la red para minimizar una función de pérdida en los datos de entrenamiento. Se utiliza el algoritmo de backpropagation.

Función de costo: también conocida como función de pérdida, es una medida de la diferencia entre los valores predichos y los valores reales de la salida de una red neuronal. Su función es determinar cuán bien está funcionando la red neuronal en términos de su capacidad para realizar predicciones precisas (MSE, cross-entropy, etc)

Algoritmos de optimización: Los algoritmos de optimización se utilizan para ajustar los parámetros de la red neuronal durante el entrenamiento. Algunos algoritmos de optimización comunes incluyen SGD, Adam y RMSProp.

KIBERNUM

Ventajas

Capacidad para aprender patrones complejos: Las redes neuronales son capaces de aprender patrones complejos en los datos de entrada que podrían ser difíciles de detectar con otras técnicas de aprendizaje automático.

Adaptabilidad: Las redes neuronales pueden adaptarse a diferentes tipos de datos, lo que las hace útiles para una amplia variedad de problemas en diferentes campos, desde la visión artificial hasta el procesamiento del lenguaje natural.

Capacidad de generalización: Las redes neuronales pueden generalizar bien a partir de un conjunto de datos de entrenamiento para hacer predicciones precisas sobre nuevos datos no vistos.

Tolerancia al ruido: Las redes neuronales pueden ser tolerantes al ruido en los datos, lo que las hace útiles en situaciones en las que los datos no están limpios o son incompletos.

Paralelismo: Las redes neuronales pueden procesar datos en paralelo, lo que las hace adecuadas para problemas que requieren un alto rendimiento computacional.

Desventajas

Complejidad computacional: Las redes neuronales pueden ser computacionalmente costosas de entrenar, especialmente si la red es grande o los datos son muy complejos.

Tendencia al sobreajuste: Las redes neuronales pueden tender a sobreajustar los datos de entrenamiento si no se controlan adecuadamente. Esto puede llevar a una disminución en el rendimiento en nuevos datos.

Dificultad en la interpretación: Las redes neuronales pueden ser difíciles de interpretar, lo que puede hacer que sea difícil entender cómo se están tomando las decisiones.

Requiere una cantidad significativa de datos de entrenamiento: Las redes neuronales requieren una cantidad significativa de datos de entrenamiento para poder hacer predicciones precisas. Si los datos de entrenamiento son limitados o sesgados, la red neuronal puede no ser capaz de generalizar adecuadamente.

Dudas y consultas

Gracias

KIBERNUM