OBS Business School

RED DE EDUCACIÓN SUPERIOR
Planeta Formación y Universidades

Fermín Lozano Rodríguez

Machine Learning II: Deep Learning y Redes Neuronales

Apuntes teóricos

Índice

1. Introducción

1.1. Qué son las redes neuronales

Las redes neuronales son una familia de algoritmos pertenecientes al campo del Machine Learning que están **inspirados en cómo funciona el cerebro humano**. De manera similar a las neuronas de nuestros cerebros, que se comunican mediante conexiones sinápticas, las redes neuronales artificiales están compuestas por una serie de nodos (neuronas) conectadas entre sí en un grafo.

Más allá de su inspiración biológica, estos algoritmos **no son más que modelos matemáticos** que, con un proceso de entrenamiento, se adaptan a una serie de datos para representar una función para resolver un problema dado.

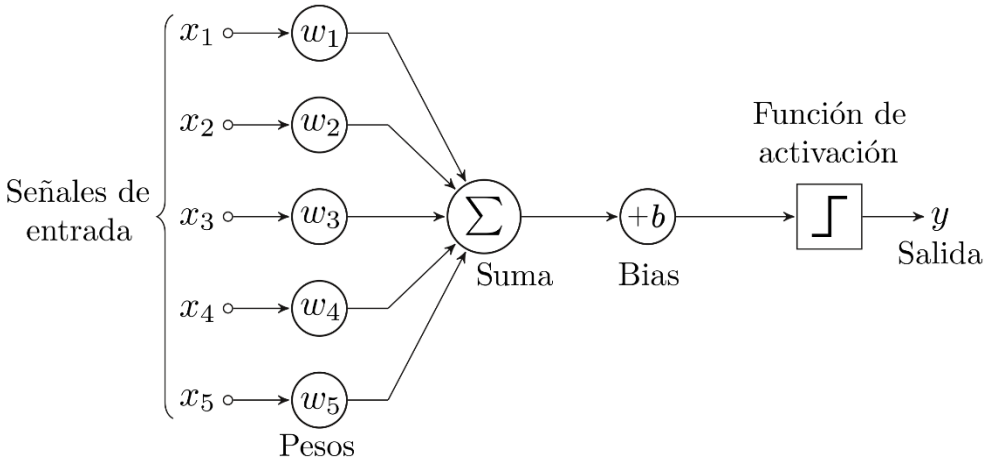
En los últimos años, el aumento de capacidad de procesamiento, particularmente usando GPUs, así como desarrollos de nuevas arquitecturas de redes, nuevas técnicas que han hecho posible el entrenamiento de redes cada vez más complejas, y la disponibilidad de datos de entrenamiento masivos, han posibilitado avanzar el estado del arte en campos como el de la visión artificial o el procesamiento de lenguaje, entre otros.

2. Componentes de redes neuronales

2.1. La neurona artificial. El Perceptrón

La red neuronal más básica posible es la compuesta por una sola neurona, la llamada **Perceptrón**. Esta neurona está compuesta por:

- Varias señales de entrada x_i
- Una serie de pesos w_i, cada uno correspondiéndole a cada entrada x_i
- Un sesgo, o *bias*, b
- Una función de activación $u(x)$ de tipo escalón



El Perceptrón

Esta neurona representa el siguiente operador matemático:

$$y = u\left(b + \sum_{i=0}^{n} w_i\, x_i\right)$$

O, en forma vectorial:

$$y = u(b + \overline{w} \cdot \overline{x})$$

Siendo $u(x)$ la función escalón.

$$u(x) = 1 \quad si \quad x \geq 0$$
$$u(x) = 0 \quad si \quad x < 0$$

En la comparación con la neurona biológica, x_i serían las señales sinápticas de las neuronas vecinas, w_i un factor que define cuánto afecta cada neurona vecina a la activación, y b el umbral a partir del cual la neurona se activa o “dispara”, esto es, lo "propensa" que es la neurona a activarse.

Este tipo de red neuronal actúa como un clasificador binario: en función de su entrada y los valores de sus parámetros (pesos w_i y *bias* b), da como resultado 1 ó 0.

2.2. Capas

Más allá del perceptrón como ejemplo de la red más simple posible, las redes neuronales están compuestas por multitud de neuronas (miles de millones en las redes más grandes) interconectadas entre sí.

El número de neuronas y cómo están dispuestas sus conexiones determina cómo es la red. Estas neuronas se agrupan en capas, y el tipo de capa especifica como son las conexiones. Así, **la arquitectura de una red neuronal se define como una serie de capas**, que se ejecutan de manera secuencial: la salida de una capa se usa como la entrada de la siguiente, y así hasta el final de la red. En las arquitecturas de red más complejas, pueden verse además varios caminos en paralelo, conexiones que se saltan capas, o conexiones cíclicas (de una capa posterior a una capa anterior de la red).

Estas capas representan operaciones matemáticas con tensores, y las definen una serie de parámetros e hiperparámetros.

2.3. Parámetros e Hiperparámetros

Los **parámetros** de una red neuronal son un conjunto de valores numéricos que se ajustan durante el proceso de entrenamiento. Estos son elementos como los pesos y *biases* de cada neurona, los componentes de un *kernel* de convolución, los parámetros γ y β de normalización, etc.

Además de los parámetros, las redes neuronales presentan una serie de **hiperparámetros** que definen su configuración, pero no se ajustan durante el proceso de entrenamiento, si no que se ajustan a mano por la persona diseñando la red. Entre ellos están el número de capas y cuántas neuronas tiene cada una, los parámetros de regularización, el porcentaje de *dropout*, el tamaño del *kernel* en capas convolucionales o de *pooling*, el *learning rate* y cómo evoluciona, etc.

2.4. Tensores

Los **tensores** son unas estructuras de datos que generalizan conceptos como el escalar, el vector o la matriz, y pueden tener un número arbitrario de dimensiones. Un vector se puede considerar un tensor de dimensión 1, y una matriz uno de dimensión 2, pero el tensor permite representaciones complejas de todo tipo, como, por ejemplo, un set de entrenamiento con 100 imágenes de 1920x1080 píxeles a color, que sería un tensor de 4 dimensiones:

Dimensión 1: Imágenes en el set de datos, en este caso 100
Dimensión 2: Canales de color de cada imagen: 3 en una imagen a color RGB
Dimensión 3: Alto de la imagen en píxeles, 1080
Dimensión 4: Ancho de la imagen en píxeles, 1920

En este ejemplo, el tensor sería de forma o tamaño [100, 3, 1080, 1920].

Hay que destacar que una dimensión puede tener tamaño 1, como sería el caso de ejemplo si fueran imágenes en escala de grises, donde el tensor sería de tamaño [100, 1, 1920, 1080]. En este caso, el número de valores alojados en la memoria del tensor sería el mismo que si la Dimensión 2 no existiera y el tensor fuera de tamaño [100, 1920, 1080] (en total habría 100*1920*1080 valores en ambos tensores), pero es importante recalcar que ambos tensores no serían iguales; el número, tamaño y orden de las dimensiones afecta a los cálculos.

En el caso del Perceptrón mencionado arriba, habría un tensor para los datos de entrada x, uno para los pesos de la neurona w, uno para el *bias* b, y uno para el resultado tras la activación, y, y su ecuación quedaría:

$$y = u(b + w \odot x)$$

Donde $u(x)$ es la función escalón definida anteriormente, calculada elemento a elemento, y el operador $\odot$ representa el **producto de Hadamard**, también conocido como multiplicación elemento a elemento o componente a componente. El resultado del producto de Hadamard es un nuevo tensor en el que cada componente es igual al producto de los componentes en esa misma posición en los tensores.

$$z_i = w_i \cdot x_i \qquad \text{para toda posición } i \text{ en el tensor}$$

Estos tensores son **la estructura de datos fundamental** que usan las librerías de redes neuronales como PyTorch o TensorFlow, y son necesarios para ejecutar los cálculos de manera eficiente en GPUs.

2.5. Funciones de activación

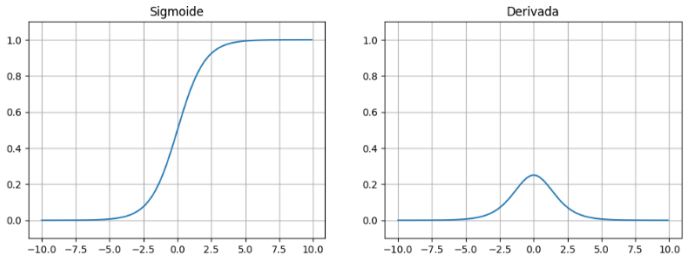
Cuando se introdujo el Perceptrón, se vio que al final incluye una función escalón, que hace las veces de función de activación. **Las funciones de activación en las redes neuronales aportan la capacidad de ajustarse a funciones no lineales.**

Una red neuronal, con múltiples neuronas con sus pesos y *biases* como hemos visto, pero sin función de activación, resultaría en una combinación de operaciones lineales, con lo no sería más que un modelo de regresión lineal, apropiado para representar funciones lineales o casi lineales. Sin embargo, la mayoría de las funciones que se quieren modelar con redes neuronales son altamente no lineales. La existencia de un término no lineal en las redes hace que estas tengan una mayor capacidad de aproximarse a estas funciones no lineales. Este término es la función de activación, y al resultado de calcular esta función sobre la salida de una neurona se le llama activación.

En el caso del Perceptrón la función de activación usada era la función escalón o función de Heaviside, pero esta función no se usa en la actualidad. En su lugar, los tipos de función de activación más comunes son:

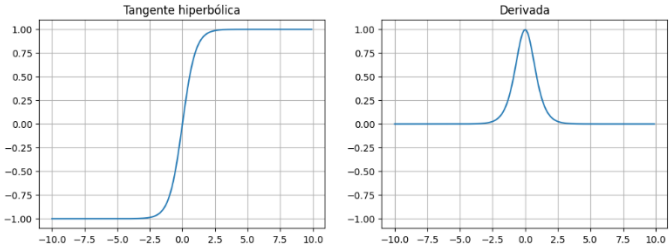
Sigmoide o Logística: Transforma la entrada a un rango entre 0 y 1. Se usa en la capa de salida en problemas de clasificación binaria, donde su valor representa la probabilidad de que la entrada pertenezca a una clase. En los extremos esta función se satura, pudiendo provocar problemas de *vanishing gradient*, en los que el gradiente deja de propagarse a través de la red.

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



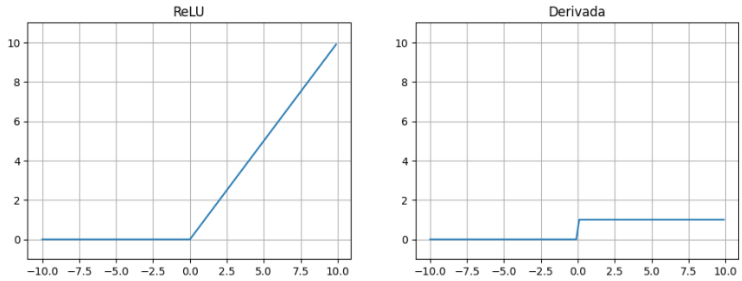
Tanh, Tangente hiperbólica: Similar a la sigmoide, pero su salida varía entre –1 y 1. Como ventaja respecto a la sigmoide, el que esté centrada en 0 puede ayudar a reducir el problema del *vanishing gradient.*

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



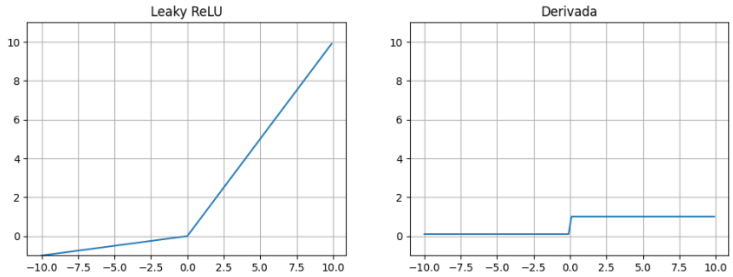
ReLU, Rectified Linear Unit: Es la más usada actualmente. Es ampliamente utilizada en capas ocultas debido a su simplicidad y que empíricamente da buenos resultados. A diferencia de la sigmoide y tanh, ReLU no satura, por lo que no presenta el problema del *vanishing gradient* para valores de entrada positivos. Pero si durante el entrenamiento alcanza un punto donde para todo el dataset la entrada fuera negativa, su gradiente sería 0, y no podría salir de esa situación; en ese momento dejarían de fluir gradientes por esa neurona, habría "muerto". A este fenómeno se le denomina *Dead* ReLU.

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



Leaky ReLU: Es una variante de ReLU que permite que fluya gradiente para entradas negativas, evitando así el problema del *Dead* ReLU. El hiperparámetro α es la pendiente de la función para entradas negativas y es un valor pequeño, típicamente 0.01.

$$Leaky\ ReLU(z) = \max(\alpha z, z)$$

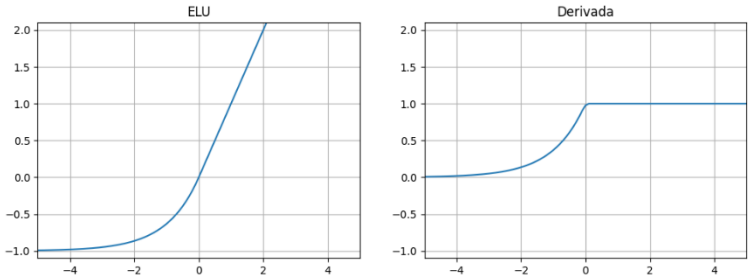


Parametric ReLU, PReLU: Es una variante del Leaky ReLU en la que α es un parámetro de la red que cambia durante el entrenamiento. En datasets pequeños corre el riesgo de provocar *overfitting*.

$$PReLU(z) = z \quad si \quad z \geq 0$$
$$PReLU(z) = \alpha z \quad si \quad z < 0$$

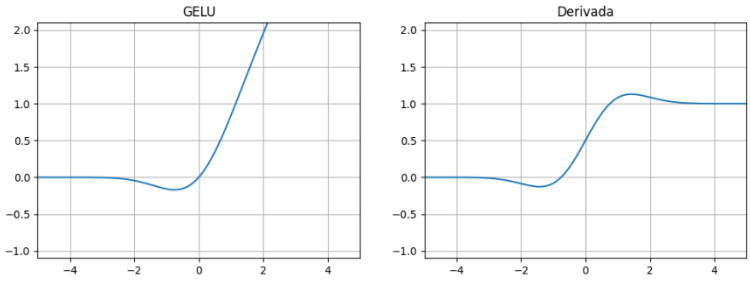
Exponential Linear Unit, ELU: Es una variante de ReLU con una transición suave en entradas negativas. El hiperparámetro α controla el valor al que satura ELU para valores muy negativos de la entrada. Su desventaja respecto a las ReLU es que es computacionalmente más cara, más lenta de calcular.

$$ELU(z) = z \quad si \quad z > 0$$
$$ELU(z) = \alpha(e^z - 1) \quad si \quad z \leq 0$$



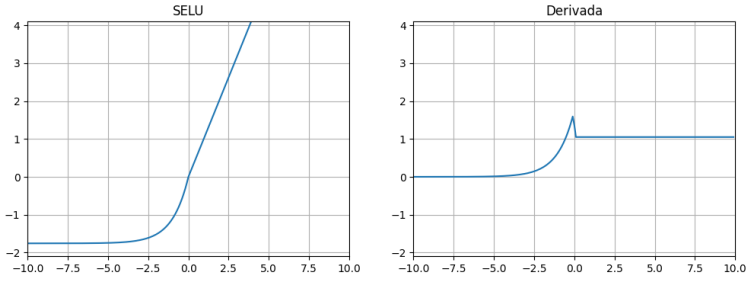
Gaussian Error Linear Unit, GELU: Esta función ha ganado popularidad en arquitecturas transformer. Se define multiplicando la entrada por la distribución acumulativa de la distribución Gaussiana, que se puede aproximar de la siguiente forma:

$$GELU(z) = 0.5z\left(1 + \tanh\left(\sqrt{2/\pi}(z + 0.044715z^3)\right)\right)$$



Scaled Exponential Linear Unit, SELU: Esta variante de ReLU tiene la propiedad de que cuando se usa en redes MLP, la red tiende a autonormalizarse, a que la salida de cada capa tenga media 0 y desviación 1, lo que mitiga el problema de *vanishing gradient.*

$$SELU(z) = scale \cdot z \qquad si \quad z > 0$$
$$SELU(z) = scale \cdot \alpha \cdot (e^z - 1) \quad si \quad z \leq 0$$
$$\alpha = 1.673263242354377284817042991671 7$$
$$scale = 1.050700987355480493419334985294 6$$



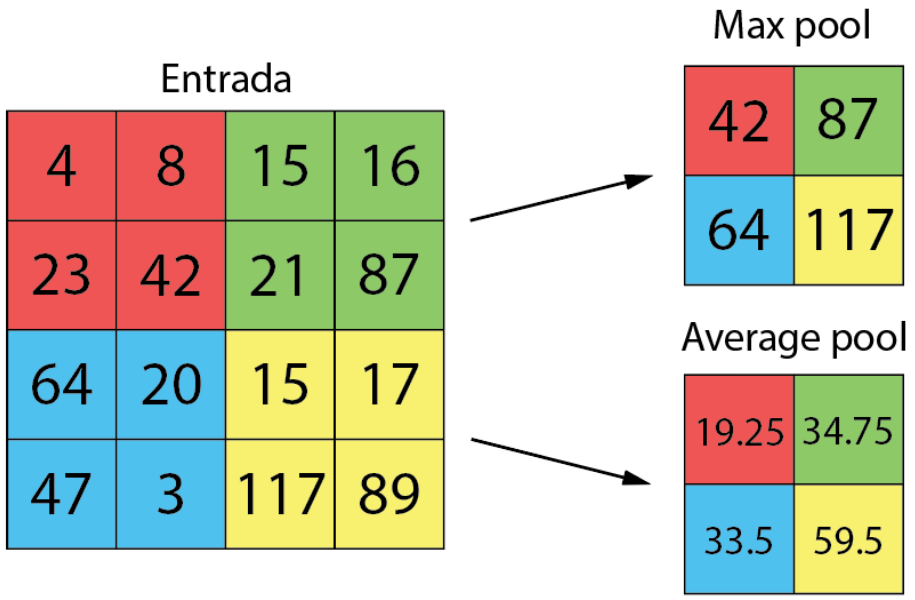
La elección de una u otra función de activación depende del problema a resolver y la arquitectura de la red neuronal usada. **Experimentar con distintos tipos de activación y comparar sus resultados es la mejor manera de determinar cuál funciona mejor para cada caso**.

Softmax: La función Softmax es un tipo de función de activación que no se suele utilizar para las capas ocultas de la red, si no que se utiliza en la capa de salida en problemas de clasificación multiclase, donde asigna probabilidades a cada clase. Esta función transforma el tensor de entrada con números reales en un tensor de probabilidades, normalizando la salida para que todas esas probabilidades sumen 1. Se define con la ecuación:

$$Soft\max(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

2.6. Pooling

La operación de *pooling* o **agrupación** consiste en combinar la salida de varias neuronas en una, reduciendo así las dimensiones de la siguiente capa y el coste computacional. El resultado de esta operación actúa como resumen de sus entradas, manteniendo la información más importante. Puede ser de una o varias dimensiones y se define por su tamaño de *kernel* (que define cuantas neuronas se agrupan) y si es *max pooling* (tomando el valor máximo del kernel) o *average pooling* (tomando la media de los valores). De manera semejante a las convoluciones, el *pooling* ayuda a la que la red sea invariante a traslaciones.



Capas de *pooling*

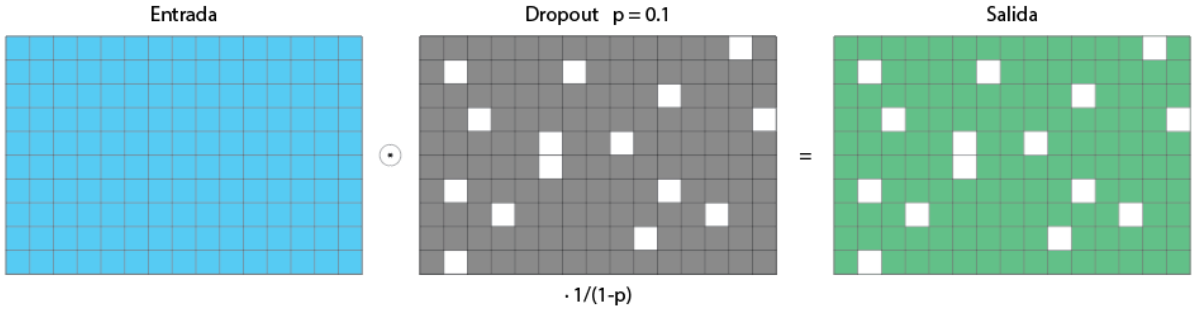
2.7. Dropout

El ***dropout*** es una técnica de regularización que se usa para mitigar el *overfitting* durante el entrenamiento. Es una capa que consiste en desactivar de manera aleatoria en cada iteración del entrenamiento algunas de sus entradas, reescalando después la salida.

La capa de *dropout* se configura con un hiperparámetro p que indica la probabilidad de que cada neurona sea desactivada. Las neuronas que no se desactivan se multiplican por $1/(1-p)$ para compensar la reducción de las activaciones de la capa.

Como en cada iteración las neuronas desactivadas cambian, introduce un grado de aleatoriedad al proceso que hace que la red aprenda patrones más robustos y generalizables. De esta manera, cada neurona aprende a contribuir cuando algunas de sus compañeras estén desactivadas.

La capa de *dropout* solo se utiliza durante la fase de entrenamiento. En la inferencia no se desactiva ninguna neurona.



Capas de *dropout*

2.8. Normalización

La normalización de redes neuronales sirve para ajustar los valores de las activaciones para darle estabilidad numérica, facilitando el proceso de entrenamiento y mitigando problemas como el desvanecimiento o explosión de gradiente (*vanishing/exploding gradient*).

La idea detrás de la normalización es **hacer más estable el entrenamiento**: un cambio de escala en una de las primeras capas puede resultar en cambios muy drásticos en las capas posteriores, hasta el punto de generar gradientes tan grandes que dificulten mucho el entrenamiento. La normalización ha hecho posible que se puedan entrenar redes profundas, con más capas de lo que antes era posible.

Batch normalization: Una capa de normalización por lotes o *batch normalization*, normaliza las activaciones de su capa anterior. Son muy usadas tras capas con multiplicaciones, como las lineales y convolucionales.

Durante el entrenamiento de la red, en cada iteración, se calculan la media μ_B y desviación σ_B de las activaciones de cada *mini-batch* de entrenamiento. Un *mini-batch* o mini-lote es un conjunto de datos del set de entrenamiento que se utiliza durante una iteración del entrenamiento. Siendo x_i los valores de activación:

$$\mu_B = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \sigma_B^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_B)^2$$

Con estos valores calculados, se ajustan las activaciones para que sigan una distribución normal de media β y desviación γ, que son los parámetros de la capa de *batch normalization* y se actualizan en el proceso de entrenamiento, como los demás parámetros, haciendo uso de un optimizador como *gradient descent* u otros.

$$y_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \cdot \gamma + \beta$$

Durante la inferencia no podemos calcular la media y desviación del *mini-batch*, puesto que no hay ningún *mini-batch*, si no que se calcula la predicción para un solo caso de datos de entrada. Por lo tanto, en lugar del μ_B y γ_B, durante la inferencia se usan unas medias móviles $E[x]$ y $Var[x]$. Estas medias móviles se calculan durante el proceso de entrenamiento y se pueden considerar otros parámetros de la capa de batch normalization, pero estos parámetros no se actualizan durante el proceso de optimización ni participan en el proceso de *backpropagation*.

Layer normalization: Usadas habitualmente en redes recurrentes y *transformers*, las capas de *layer normalization* normalizan las activaciones de la capa anterior, de manera similar al *batch normalization*, pero en vez de normalizar a lo largo del *mini-batch*, en *layer normalization* se normaliza a lo largo de las características de la capa (el total de neuronas de una capa oculta, o los canales de salida de una convolución, por ejemplo) para cada caso de entrenamiento por separado. Siendo H las características o dimensiones de la entrada x :

$$\mu_L = \frac{1}{H}\sum_{i=1}^{H} x_i \qquad \sigma_L^2 = \frac{1}{H}\sum_{i=1}^{H}(x_i - \mu_L)^2$$

Al igual que en *batch normalization*, con estos valores se ajustan las activaciones para que sigan una distribución normal de media β y desviación γ , que son los parámetros de la capa de *layer normalization* y se actualizan durante el proceso de entrenamiento.

$$y_i = \frac{x_i - \mu_L}{\sqrt{\sigma_L^2 + \epsilon}} \cdot \gamma + \beta$$

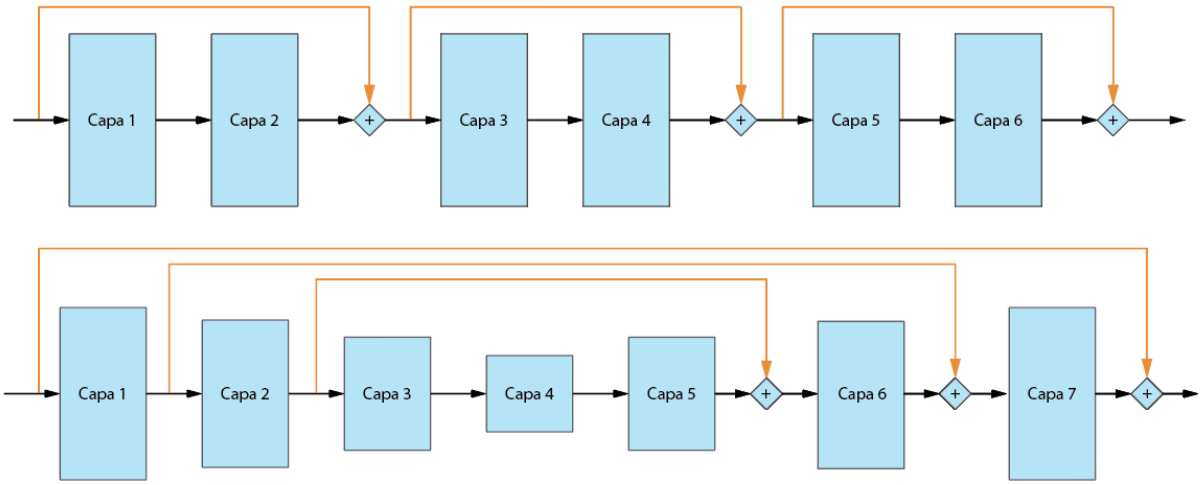
Debido a que en *layer normalization* se normaliza cada caso de entrada por separado, no es necesario tener unas medias móviles con la media y desviación. Durante la inferencia toda la información para hacer estos cálculos está disponible.

2.9. Skip connections

Las *skip connections*, o conexiones de salto, son una técnica que consiste en **conectar la salida de una capa a la entrada de una capa más profunda en la red, saltándose esa conexión algunas capas**. Un ejemplo son las conexiones residuales, que combinan esta señal con una suma, pero también se pueden combinar de otras formas, como concatenando la señal o multiplicándola.

La idea de esta técnica es asegurar que, aunque haya algún gradiente que se haga cero en alguna capa intermedia, **los gradientes se puedan propagar por estas conexiones**, evitando así problemas de desvanecimiento de gradiente.

Las *skip connections* permiten la construcción de redes muy profundas, con incluso cientos de capas, y son un elemento fundamental de arquitecturas como las redes residuales y los *transformers*.



Skip connections

3. Entrenamiento de redes neuronales

3.1. Introducción

En el campo de las redes neuronales, y más generalmente en *Machine Learning*, lo que se persigue es crear un modelo que cumpla una función dada. Para ello, se hace uso de un set de datos de entrenamiento, que consisten en una serie de inputs a ese modelo, y, en el caso de aprendizaje supervisado, el output correspondiente a esos inputs.

Durante el proceso de entrenamiento se ajustan los parámetros del modelo (pero no los hiperparámetros): los pesos y *biases* de sus neuronas principalmente, pero también otros elementos como los parámetros de normalización, para que la red neuronal se comporte de manera similar a los datos del set de entrenamiento.

La métrica que se utiliza para saber cómo de bien se ajusta el modelo al set de entrenamiento es la función de pérdida (*loss function* en inglés), que es menor cuanto más se aproxima el modelo a los datos de entrenamiento. **El proceso de entrenamiento de la red neuronal consiste en encontrar los parámetros de la red que minimicen esta función de pérdida**.

3.2. Sets de datos

Para comprobar que el proceso de entrenamiento está siendo exitoso y el modelo está funcionando correctamente, el set de datos de entrada disponible se parte en tres *subsets*: de entrenamiento, de validación y de test. Con esta partición nos aseguramos de que el modelo es capaz de generalizar y funcionar bien en datos diferentes a los de entrenamiento. Una partición típica es que el 80% del tamaño total del set de datos se dedique a entrenamiento, 10% a validación y 10% a test.

El set de **entrenamiento** es el que se utiliza en el proceso de entrenamiento para ajustar los parámetros de la red.

Después de haber entrenado la red, se comprueba que tal funciona contra el set de **validación**, con la intención de modificar a mano los hiperparámetros y volver a entrenar la red, para mejorar el rendimiento de la red. Se puede interpretar que con el set de entrenamiento el algoritmo de optimización ajusta los parámetros, y con el de validación el humano diseñando la red ajusta los hiperparámetros.

Por último, se comprueba que tal funciona la red contra el set de datos de **test**. Típicamente, esta comprobación se hace menos veces, para evitar caer en el error de modificar los hiperparámetros para mejorar el rendimiento contra estos datos (para eso está el set de validación).

En algunos casos, para simplificar, los datos se dividen en dos subsets: el de entrenamiento y el de test, haciendo el de test en este caso las funciones tanto de validación como de test.

3.3. Overfitting y underfitting

Si la red neuronal tiene mucha capacidad (es muy compleja, tiene demasiadas neuronas) para el set de datos de entrenamiento, y se entrena durante mucho tiempo, corremos el riesgo de provocar ***overfitting***, que ocurre cuando la red se ajusta demasiado bien a los datos de entrenamiento y no es capaz de generalizar para otros datos. En este caso, el modelo tendría muy buen rendimiento contra los datos de entrenamiento, pero malo contra los de test. Se puede mitigar disminuyendo la complejidad de la red, aumentando el set de datos, o usando regularización.

Por otra parte, el fenómeno contrario es el del ***underfitting***, que ocurre cuando el modelo es muy simple y no es capaz de ajustarse a los datos de entrenamiento; por lo que tendrá un mal rendimiento contra tanto los datos de entrenamiento como los de test. Se puede combatir aumentando la complejidad de la red o dejando que entrene durante más tiempo.

3.4. Función de pérdida

La función de pérdida, *loss function* en inglés, también llamada función de coste, es una métrica que define cuánto se aleja la predicción del modelo de los valores del set de datos de entrenamiento. Durante el entrenamiento, los parámetros de la red se ajustan para minimizar está función de pérdida, para que su valor sea el menor posible, haciendo por lo tanto que la predicción del modelo se acerque a los valores del set datos.

Algunos de los tipos más comunes de funciones de pérdida son las siguientes ($\hat{y}_i$ es el valor de salida del modelo y y_i es el valor objetivo en el set de datos):

Para problemas de regresión, donde la salida del modelo es un valor numérico:

Error cuadrático medio, *Mean Squared Error*, MSE:

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

Error absoluto medio, *Mean Absolute Error*, MAE:

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

Para problemas de clasificación multiclase y modelos de generación de texto:

Entropía cruzada, Cross-Entropy: Es una medida de la discrepancia o diferencia entre dos distribuciones de probabilidad. Es comúnmente usada tras una función softmax, por lo que a veces la fórmula que se usa es una combinación de las dos. Aquí se presenta la función de la entropía cruzada por separado.

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

Para problemas de clasificación binaria, donde la salida del modelo es si la entrada pertenece o no a una clase:

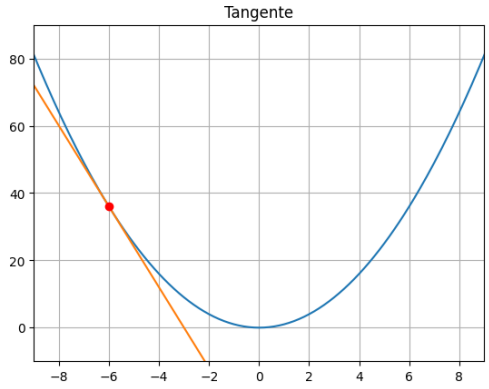
Entropía cruzada binaria, *Binary Cross-Entropy*, BCE: Es un caso particular de la entropía cruzada, con solo dos "clases" (pertenencia o no a la clase en cuestión).

$$-\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

3.5. Minimización de la pérdida

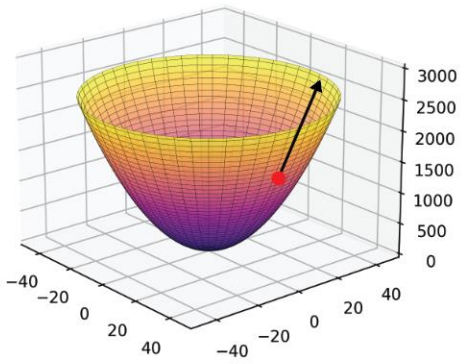
Para encontrar un mínimo de la función de pérdida, donde nuestro modelo de red neuronal se ajusta mejor a los datos de entrenamiento, se usan técnicas de optimización matemática.

En el caso más sencillo de optimización, las funciones de una variable, se usa la derivada para obtener la tangente de la función en un punto, y con ella se puede avanzar en la dirección donde está el mínimo local.



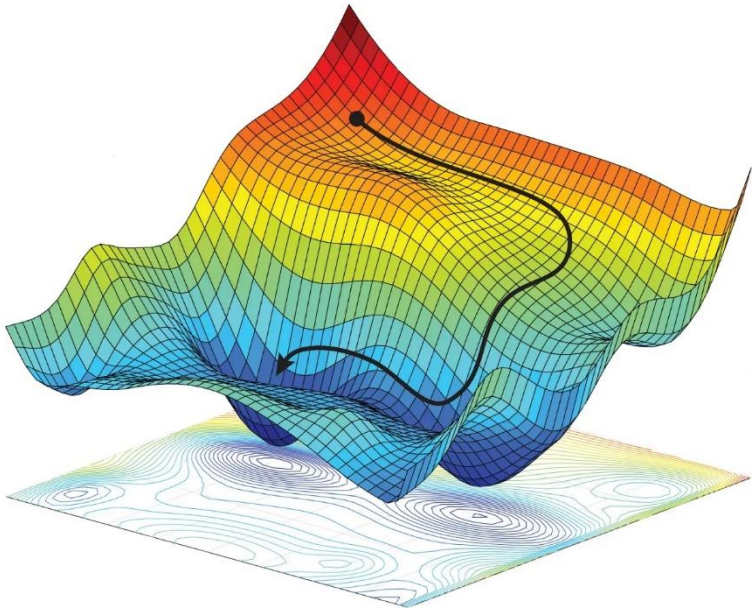
Uso de la tangente para encontrar el mínimo

La generalización de la derivada en varias dimensiones es el **gradiente**, que está definido como un vector/matriz/tensor con las derivadas parciales de la función con respecto a cada una de sus variables. En cualquier punto de la función, su gradiente es el vector que apunta en la dirección de máxima pendiente, por lo tanto, para encontrar el mínimo hay que moverse en dirección opuesta al gradiente.



El gradiente apunta en la dirección de máxima pendiente

En el problema que nos ocupa, el entrenamiento de una red neuronal, la función que se quiere minimizar es la función de pérdida, que depende de todos los parámetros de la red. Es un problema altamente multidimensional, por lo que la intuición de las 3 dimensiones se pierde, pero en esencia el proceso es el mismo: **obtener el gradiente de la función, y avanzar en dirección opuesta para acercarse al mínimo local**.



Uso del gradiente para alcanzar el mínimo

3.6. Backpropagation

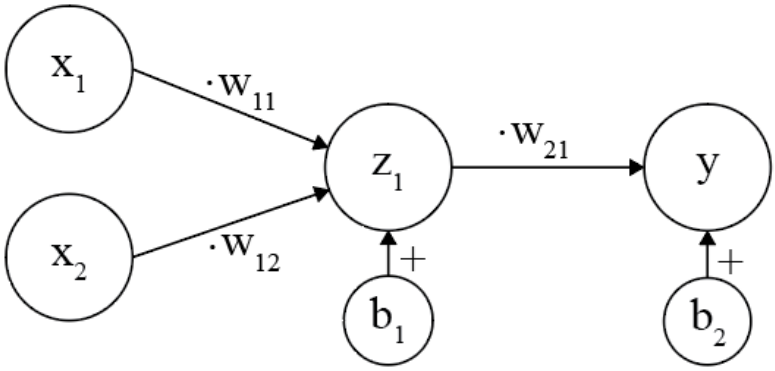
Para obtener el gradiente en cada punto, necesario en el proceso de encontrar el mínimo de la función de perdida, se hace uso de un mecanismo llamado ***backpropagation***. Consiste en, haciendo uso de la regla de la cadena, *chain rule*,

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

La derivada de y respecto a x es igual a
la derivada de y respecto a u, multiplicada por la derivada de u respecto a x.

calcular las derivadas con respecto a todos los parámetros de la red, haciendo uso de las derivadas con respecto a otros parámetros.

Considerando la siguiente red MLP con dos neuronas en la entrada, una en la capa oculta y otra en la salida (se omiten las funciones de activación para simplificar):



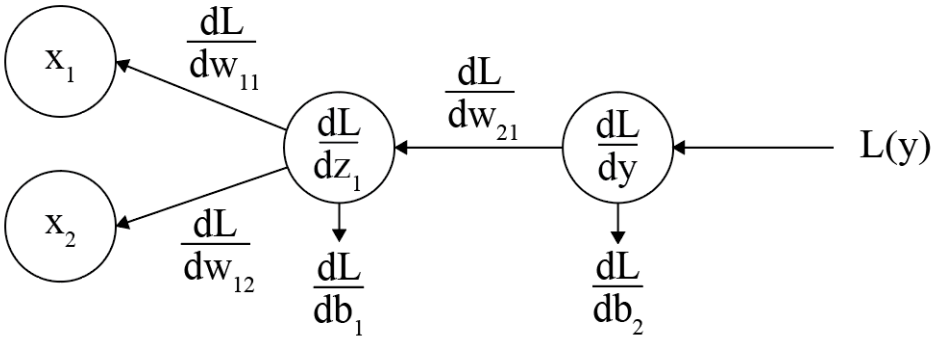
Forward pass

La salida y de la red queda definida por la función:

$$y = w_{21}(x_1 w_{11} + x_2 w_{12} + b_1) + b_2$$

Para calcular la salida de la red se ejecuta lo que se denomina ***forward pass***, en el que se recorre la red "hacia delante", empezando por las entradas, y avanzando hasta la salida. En el ejemplo del diagrama, se calcularía primero $x_1 w_{11}$ y $x_2 w_{12}$, y estos se sumarían con b_1, con lo que tendríamos la salida de la neurona oculta z_1, y este valor se multiplicaría por w_{21}, y tras esa multiplicación se sumaría b_2, obteniendo como resultado la salida de la red y .

Con esto tenemos las activaciones de las neuronas y a la salida de la red, que sería suficiente para obtener las predicciones de la red (inferencia) o calcular la pérdida. Pero en el proceso de entrenamiento, necesitamos también el gradiente de la función de pérdida, como se mencionaba antes. Necesitamos por lo tanto la derivada de la función de pérdida con respecto a cada uno de los parámetros de la red, los pesos y los *biases,* para lo que usamos el *backward pass,* en el que recorre la red "hacia atrás".



Backward pass

Usando la regla de la cadena anteriormente mencionada, podemos obtener la derivada de la pérdida con respecto a cualquier parámetro a partir de las derivadas respecto a otros parámetros que están más adelante en la red.

$$\frac{dL}{db_2} = \frac{dL}{dy} \cdot \frac{dy}{db_2} \qquad \frac{dL}{dw_{21}} = \frac{dL}{dy} \cdot \frac{dy}{dw_{21}} \qquad \frac{dL}{dz_1} = \frac{dL}{dy} \cdot \frac{dy}{dz_1}$$

$$\frac{dL}{db_1} = \frac{dL}{dz_1} \cdot \frac{dz_1}{db_1} \qquad \frac{dL}{dw_{11}} = \frac{dL}{dz_1} \cdot \frac{dz_1}{dw_{11}} \qquad \frac{dL}{dw_{12}} = \frac{dL}{dz_1} \cdot \frac{dz_1}{dw_{12}}$$

Cada uno de los operadores a la derecha, las derivadas locales, son triviales en este caso, puesto que hay sólo operadores de suma y multiplicación:

$$y = z_1 w_{21} + b_2 \qquad \frac{dy}{db_2} = 1 \qquad \frac{dy}{dw_{21}} = z_1 \qquad \frac{dy}{dz_1} = w_{21}$$

$$z_1 = x_1 w_{11} + x_2 w_{12} + b_1 \qquad \frac{dz_1}{db_1} = 1 \qquad \frac{dz_1}{dw_{11}} = x_1 \qquad \frac{dz_1}{dw_{12}} = x_2$$

Donde los valores numéricos de x_1 y x_2 son la entrada para un dato del set de entrenamiento, w_{21} es uno de los pesos de la red en ese instante del proceso de entramiento, y z_1 la salida de la primera neurona, que se calculó al hacer el *forward pass*. Con esto y la derivada de la función de pérdida con respecto a la salida de la red, tenemos todo lo necesario para calcular el gradiente de la función. Para una función de perdida MSE:

$$\frac{dL}{dy} = 2(y_d - y)(-1)$$

Siendo y_d el valor target para la entrada dada en el set de datos de entrenamiento.

Poniendo unos valores numéricos de ejemplo, el proceso sería el siguiente:

$$x_1 = 1 \quad x_2 = -2 \quad w_{11} = 3 \quad w_{12} = 2 \quad b_1 = 2 \quad w_{21} = 4 \quad b_2 = -1 \quad y_d = 2$$

Forward Pass:

$$z_1 = x_1 w_{11} + x_2 w_{12} + b_1 = 1 \cdot 3 + (-2) \cdot 2 + 2 = 1$$
$$y = z_1 w_{21} + b_2 = 1 \cdot 4 + (-1) = 3$$

Backward Pass:

$$\frac{dL}{dy} = 2(y - y_d) = 2(3 - 2) = 2$$

$$\frac{dL}{db_2} = \frac{dL}{dy} \cdot \frac{dy}{db_2} = 2 \cdot 1 = 2 \qquad \frac{dL}{dw_{21}} = \frac{dL}{dy} \cdot \frac{dy}{dw_{21}} = 2 \cdot z_1 = 2 \cdot 1 = 2$$

$$\frac{dL}{dz_1} = \frac{dL}{dy} \cdot \frac{dy}{dz_1} = 2 \cdot w_{21} = 2 \cdot 4 = 8 \qquad \frac{dL}{db_1} = \frac{dL}{dz_1} \cdot \frac{dz_1}{db_1} = 8 \cdot 1 = 8$$

$$\frac{dL}{dw_{11}} = \frac{dL}{dz_1} \cdot \frac{dz_1}{dw_{11}} = 8 \cdot x_1 = 8 \cdot 1 = 8 \qquad \frac{dL}{dw_{12}} = \frac{dL}{dz_1} \cdot \frac{dz_1}{dw_{12}} = 8 \cdot x_2 = 8 \cdot (-2) = -16$$

Con estos cálculos ya tendríamos la derivada de la función de perdida con respecto a todos los parámetros, esto es, el gradiente, y con él podemos hacer actualizar los parámetros para acercarse al mínimo local de la función de pérdida.

Destacar que, en el ejemplo mostrado, por simplificar, se ha usado un solo dato del set de entrenamiento. Cuando se entrena una red, se usan varios de estos datos, lo que se llamaría un **lote o *batch***, de tal manera que a la función de pérdida contribuirían todos los casos del *batch*, haciendo que se tenga que calcular tanto el *forward* como el *backward pass* de cada uno de los casos del *batch* por separado. Por suerte, esto se hace de manera muy eficiente con las GPUs, que pueden ejecutar los cálculos de cada caso del *batch* a la vez, en paralelo.

Este ejemplo mostrado es el de una red extremadamente simple, pero el proceso de *backpropagation* funciona igual en redes de producción con millones de parámetros. Por supuesto, no es necesario programar los cálculos de cada derivada a mano, si no que las librerías de redes neuronales como PyTorch o TensorFlow incluyen esta funcionalidad.

Durante el proceso de entrenamiento, existe la posibilidad de que los gradientes se hagan muy pequeños (***vanishing gradients***) lo que ralentiza o para el entrenamiento. Por otro lado, puede pasar lo contrario: que los gradientes tomen valores muy altos (***exploding gradients***), lo que dificulta que se pueda converger a un mínimo. A lo largo de este texto se pueden ver varias estrategias para mitigar estos fenómenos, como el uso de distintas funciones de activación, la regularización o la normalización.

3.7. Funciones de optimización

Una vez se tiene calculado el gradiente gracias al proceso de *backpropagation*, se pueden actualizar los valores de los parámetros de la red, para acercarse poco a poco al mínimo de la función de pérdida. El proceso de entrenamiento consiste en repetir reiteradamente los *forward* y *backward passes* para calcular el gradiente, y con él dar un paso o *step* con el que actualizar los parámetros.

Hay varios métodos para actualizar los parámetros. A continuación, se detallan las funciones de optimización u optimizadores más comunes:

Gradient Descent: El más simple de los optimizadores es el *gradient descent* o descenso de gradiente. Consiste en sumar a los parámetros θ el gradiente en negativo $-g$, multiplicado por el *learning rate* γ , que define la magnitud del "paso" que se da.

$$\theta_t = \theta_{t-1} - \gamma \cdot g$$

Stochastic Gradient Descent (SDG): Para hacer el *gradient descent* real, se tendría que utilizar el set de datos de entrenamiento entero en cada paso del optimizador. El *stochastic gradient descent* es una variante en la que usa un solo dato del set en cada paso del optimizador.

Mini-Batch Gradient Descent (SDG): Mientras que en el gradient descent se utiliza el set de datos de entrenamiento completo en cada paso, y el SDG se utiliza una muestra en cada paso, el *Mini-Batch Gradient Descent* es una variante intermedia de los dos: **se usa una parte del set de datos, un lote o *batch***, normalmente de tal manera que todos los casos del *batch* caben en la

memoria de la GPU, de tal manera que se pueden calcular los *forward* y *backward passes* de los elementos del *batch* a la vez, en paralelo, de manera eficiente.

El gradiente que se calcula cuando se usan *batches* no es tan preciso como si se usara el dataset entero, y se pueden ver fenómenos como que durante el proceso de entrenamiento a veces la función de pérdida suba un poco, en lugar de estar siempre disminuyendo. Pero debido a que el uso de *batches* es mucho más eficiente computacionalmente, en el mismo tiempo podemos hacer muchas más iteraciones de entrenamiento, por lo que da mejor resultado. Es la mejor de las variantes del *gradient descent* y la que se usa siempre.

Cuando se ha ejecutado el optimizador con distintos *mini-batches* las veces necesarias para cubrir todo el set de entrenamiento, se dice que se ha completado un ***epoch***. Normalmente se usa esté termino para referirse a cómo de largo es un proceso de entrenamiento para una red.

Momentum: A las variantes del *gradient descent* se les puede añadir un término de momento que es similar conceptualmente al momento lineal de física, la inercia. Con él, el gradiente tiende a seguir en la dirección en la iba en el instante anterior.

$$\theta_t = \theta_{t-1} - \gamma \cdot v_t \qquad v_t = \mu \cdot v_{t-1} + g$$

Nesterov Accelerated Gradient: El *Nesterov Accelerated Gradient*, o NAG, es una variación que se puede hacer al *gradient descent* y otros tipos de optimizador, que hace que el optimizador "mire hacia delante": como gracias al momento sabemos hacia qué dirección se dirige el optimizador, calculamos el gradiente no donde estamos ahora si no donde vamos a estar en el siguiente paso.

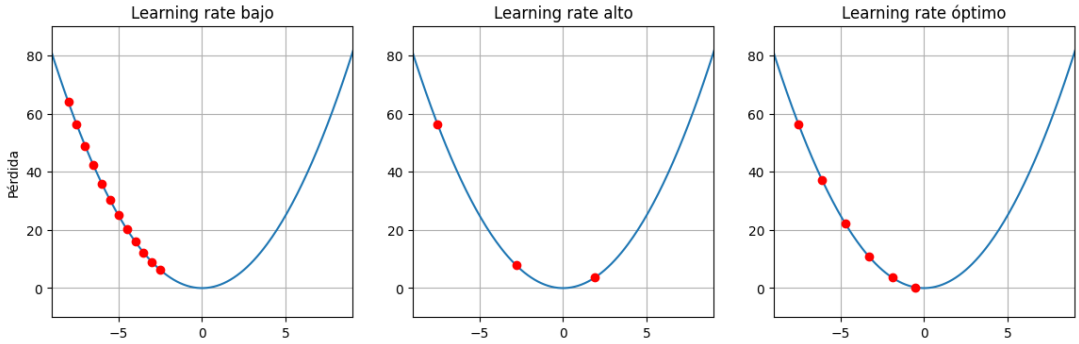
Adagrad: El algoritmo *Adagrad* ajusta el *learning rate* de cada parámetro de la red de manera independiente, según un acumulado de las derivadas anteriores de la pérdida con respecto a ese parámetro.

Adadelta y RMSprop (Root Mean Square Propagation): Similares a *Adagrad*, con capacidad de ajustar el *learning rate* de cada parámetro independientemente. La principal diferencia es que en ambos el *learning rate* no tiende a disminuir a lo largo del tiempo.

Adam (Adaptive Moment Estimation): Es el método más usado actualmente, puesto que da buenos resultados. Adapta el *learning rate* de cada parámetro haciendo uso de un promedio exponencial de los gradientes y del cuadrado de los gradientes anteriores.

3.8. Learning rate

El ***learning rate*** es un hiperparámetro que se multiplica por el gradiente en cada iteración del optimizador, definiendo cuán rápido se cambian los parámetros de la red. Si es muy grande, se corre el riesgo de no converger a un mínimo (los pasos son tan grandes que “nos pasamos” del mínimo). Si es muy pequeño, se puede tardar mucho en alcanzar el mínimo. Normalmente, se empieza con un valor mayor y se va disminuyendo según se van haciendo iteraciones.



3.9. Inicialización de parámetros

Antes de empezar el proceso de entrenamiento, los parámetros de la red comienzan con unos valores iniciales. Estos influyen en el tiempo y capacidad de convergencia del proceso de entrenamiento. Las técnicas más comunes de inicialización de parámetros son las siguientes:

Aleatoria: Los parámetros se inicializan de manera aleatoria siguiendo una distribución normal gaussiana o una distribución uniforme.

Xavier: Los parámetros se inicializan siguiendo una distribución uniforme o normal, en la que los máximo y mínimo para la uniforme, o la desviación para la normal, se calculan en base al número de entradas y salidas de la capa. Ayudan a mitigar el problema de *vanishing* y *exploding* gradients.

Kaiming/He: Similar a la de Xavier. Aquí el máximo y mínimo de la distribución uniforme o la desviación de la normal se calculan a partir del número de entradas o del número de salidas de la capa (a elegir uno de los dos). Se usa con función de activación ReLU o Leaky ReLU.

3.10. Regularización

La regularización es una técnica utilizada en el entrenamiento de redes neuronales para mitigar el *overfitting*. Para ello, penaliza la complejidad en los modelos (que sus parámetros tengan valores muy altos), que suelen ser causantes de *overfitting*. Así, a la función de pérdida se le añade un término de regularización, asociado a la complejidad de la red, de forma que se minimiza el conjunto de ambos componentes.

Regularización L1 (Lasso): El término de regularización es proporcional a la suma de los valores absolutos de los parámetros de la red. Tiende a que algunos parámetros alcancen un valor cero, lo que equivaldría a eliminar dimensiones del modelo, y, por lo tanto, simplificarlo.

$$\lambda \sum_i |\theta_i|$$

Regularización L2 (Ridge): El término de regularización es proporcional a la suma de los cuadrados de los parámetros de la red. Tiende a suavizar los parámetros, evitando que llegue a valores extremos.

$$\lambda \sum_i \theta_i^2$$

Elastic Net: Es una combinación de L1 y L2, añadiendo sus dos términos:

$$\lambda_1 \sum_i |\theta_i| + \lambda_2 \sum_i \theta_i^2$$

En ambos casos, el hiperparámetro de regularización λ controla la intensidad de este término. Si λ es muy bajo, se corre el riesgo de que la regularización no haga efecto y se provoque *overfitting*. Si λ es muy alto, la red va a tender sobre todo a minimizar los parámetros y se corre el riesgo de que haya *underfitting*.

3.11. Transfer Learning

El aprendizaje por transferencia o *transfer learning* es una técnica que consiste en usar un **modelo ya entrenado**, que ha aprendido características y generalizaciones de un set de datos de entrenamiento grande, y **entrenarlo para una nueva tarea con una cantidad limitada de datos**.

Hay dos estrategias para llevar a cabo este transfer learning. Una es utilizar la red pre-entrenada como un **extractor de características** (*feature extractor*), en el que las primeras capas de la red se mantienen fijas (no se entrenan), y las últimas se reentrenan o incluso se sustituyen por capas nuevas, que pueden ser diferentes. Por ejemplo, en redes convolucionales las primeras capas pueden detectar patrones, texturas... y ese aprendizaje es válido para la nueva tarea; pero se sustituye la capa de clasificación por una nueva que se entrene con los datos de la nueva tarea. El número de parámetros a entrenar es menor que en la red original, lo que acelera el entrenamiento.

La otra estrategia es el ***fine-tuning***, en el que se usa el modelo pre-entrenado como una inicialización de sus parámetros, pero se continua el proceso de entrenamiento a partir de ese estado, haciendo uso de los datos de entrenamiento de la nueva tarea.

Este tipo de técnicas ahorra el tener que generar gran cantidad datos de entrenamiento etiquetados. Cuando el número de casos en el set de entrenamiento de la nueva tarea es muy limitado o incluso sólo uno, se le llama ***few-shot learning*** o ***one-shot learning***.
Un caso todavía más extremo es el del ***zero-shot learning***, donde se entrena un modelo para realizar una tarea para la cual no hay ejemplos de entrenamiento. Para ello, a un modelo ya entrenado, se le proporciona información descriptiva sobre la nueva tarea, pero sin añadir datos de entrenamiento etiquetados asociados. Por ejemplo: en un modelo de *computer vision* que detecta distintos tipos de plantas, se quiere usar *transfer learning* para que aprenda a detectar

una nueva especie de flor, pero no se tienen imágenes de ella; así que se le proporciona información al modelo sobre qué color tiene, tamaño, forma de sus hojas....

Cuando el proceso de *transfer learning* resulta en obtener peores resultados en la tarea nueva que los que se tenían antes, se dice que hay ***negative transfer.*** Pasa en casos donde la tarea original de la red pre-entrenada y la nueva tarea no están lo suficientemente relacionadas, o que la manera de hacer el *transfer learning* no aprovecha bien la relación entre ambas tareas.

3.12. Knowledge distillation

La destilación de conocimiento (*knowledge distillation*) es una técnica para trasferir conocimiento de un modelo grande y complejo ya entrenado (el modelo profesor o ***teacher model***) a uno más pequeño y simple (el modelo estudiante o ***student model***). Para ello, al proceso de entrenamiento del modelo estudiante se le pone como objetivo en la función de pérdida, en lugar de las etiquetas correspondientes en el set de datos de entrenamiento, la salida del modelo profesor.

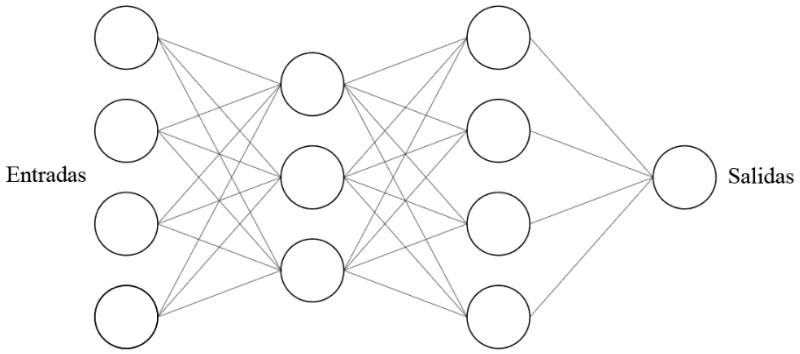
La idea detrás de esta técnica es que el modelo estudiante aprenda a imitar no sólo las predicciones del profesor, sino también su conocimiento interno, más profundo.

4. Arquitecturas de redes neuronales

4.1. Perceptrón Multicapa

El Perceptrón Multicapa, ***Multi Layer Perceptron*, MLP** por sus siglas en inglés, o también llamado ***Deep Feedforward Network***, es una red neuronal compuesta por un conjunto de neuronas con al menos tres capas: una capa de entrada, una o varias capas ocultas y una capa de salida.

Las capas de un MLP **son transformaciones lineales totalmente conectadas**, separadas por funciones de activación. Esto quiere decir que cada neurona de una capa tiene como entradas las salidas de todas las neuronas de la capa anterior (de ahí lo de *totalmente conectadas*) y lineal porque son conexiones que siguen una formula lineal $w * x + b$ como la que vimos en el Perceptrón.

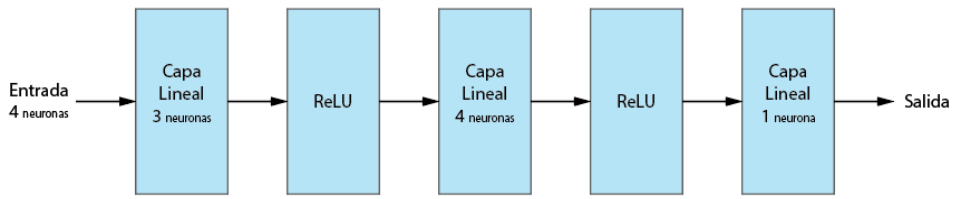


Red Multi Layer Perceptron

Los parámetros de cada capa lineal en un MLP son los pesos y biases de cada neurona, y su único hiperparámetro es el número de neuronas en la capa.

Aunque a estas redes se les llama Perceptrón Multicapa, no están realmente compuestas por perceptrones, ya que se suelen usar otro tipo de funciones de activación (el Perceptrón usa la función escalón como activación como se vio anteriormente).

Los MLP se definen por el número de capas, el número de neuronas de cada capa, y el tipo de función de activación que se usa tras cada capa, como en el diagrama siguiente:



Arquitectura esquemática de una red Multi Layer Perceptron

Que haya varias capas añade niveles de abstracción y permite resolver problemas más complejos. Por ejemplo, en un clasificador que detecta si en una imagen hay un coche o no:

- La entrada sería la imagen
- La primera capa oculta podría detectar donde hay cambios de color en la imagen y que forma tienen
- La segunda capa oculta podría detectar ruedas, la carrocería, ventanas... a partir de la salida de la primera capa oculta
- La capa de salida podría inferir si hay o no un coche en la imagen a partir de la información de la segunda capa oculta

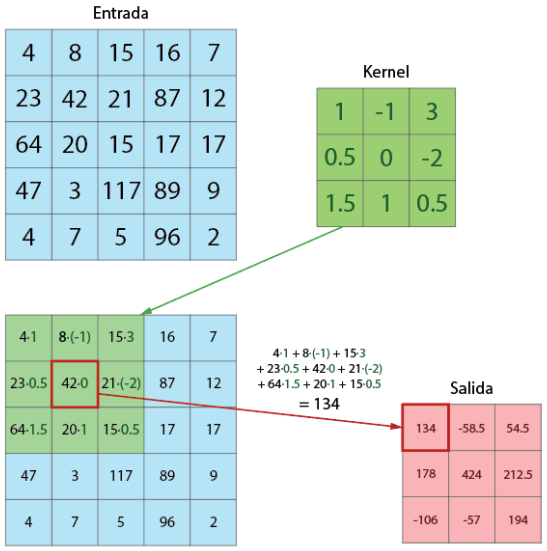
Este razonamiento de añadir niveles de abstracción conforme haya más capas en la red neuronal es válido para todos los tipos de arquitectura que vamos a ver, y es de donde viene el nombre ***Deep Learning***; ese *deep* hace referencia a que una red tiene muchas capas, a que es *profunda*.

Los MLP se pueden usar cuando las dimensiones de la señal a procesar no son muy grandes. Es muy común verlos al final de arquitecturas de red más complejas, donde las primeras capas reducen el tamaño de la señal, y al final un MLP hace la tarea de clasificación.

4.2. Redes Neuronales Convolucionales

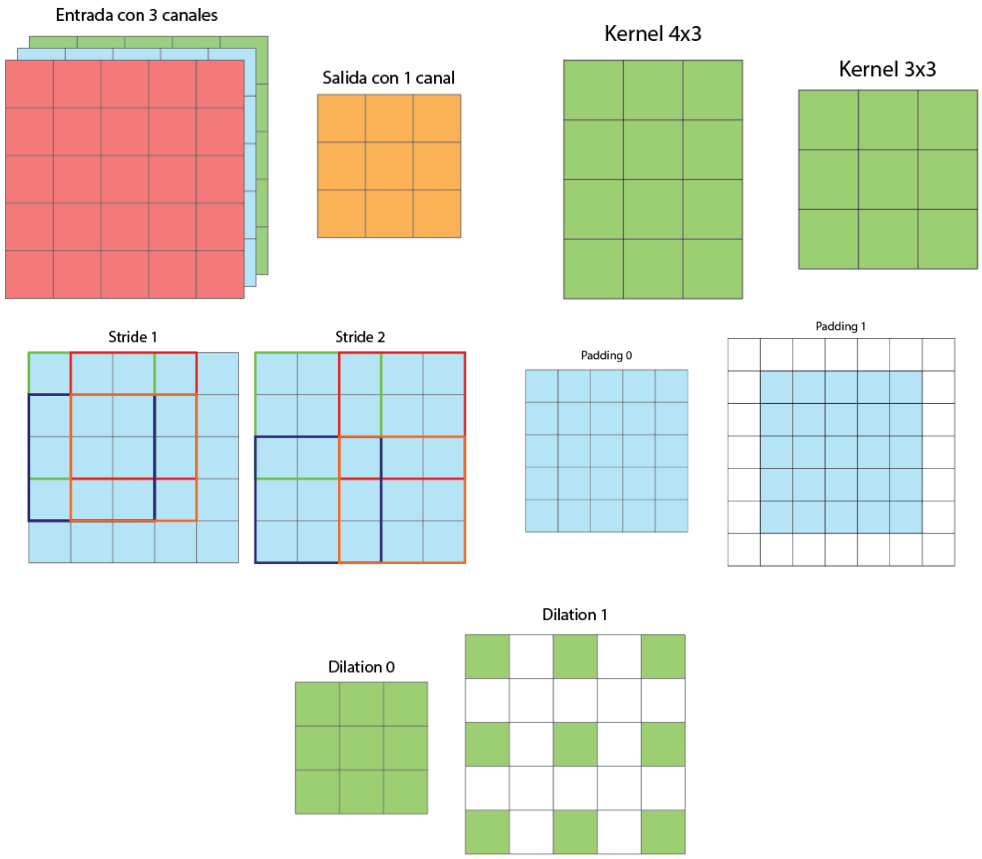
Las redes neuronales convolucionales (***Convolutional Neural Networks***, ***CNN***, en inglés) son muy eficientes en problemas donde los datos tienen una cierta estructura espacial, como por ejemplo las imágenes, por lo que este tipo de redes es muy usado en problemas de *computer vision*.

Hacen uso de capas convolucionales, que, en lugar de conectar todas las neuronas de una capa con la capa siguiente, aplican un filtro convolucional (también llamado *kernel*) sobre la entrada. Este filtro consiste en un tensor que cubre una región de los datos de entrada, y se desliza para cubrir toda la imagen, pedazo a pedazo. Por cada posición en la que se aplica el filtro se obtiene un valor de la salida.



Filtro convolucional

Los parámetros de una capa convolucional son los pesos y biases de cada una de las posiciones del *kernel*, y sus hiperparámetros son el **número de canales de entrada y salida**, el **tamaño del *kernel***, el ***stride*** (lo grande que se dan los pasos al deslizar el *kernel* sobre la entrada), el ***padding*** (un margen alrededor de la entrada) y ***dilation*** (el espacio entre posiciones contiguas del *kernel*). Dependiendo de la configuración de la capa, esta puede reducir en mayor o menor medida las dimensiones de la señal. Por simplificación se muestran en los diagramas solo *kernels* de dos dimensiones, pero también existen las convoluciones en 1 y 3 dimensiones.



Hiperparámetros de una capa convolucional

La idea de estas capas es que extraen patrones locales de los datos de entrada y con ello generan un mapa de características. Estas características detectadas pueden ser elementos como bordes, texturas o formas; y la convolución puede encontrarlas sin importar su posición en la imagen. Esta propiedad de **invarianza a translaciones** es importante en problemas de visión artificial, puesto que se puede querer detectar un objeto, aunque esté en distintas posiciones. Una red MLP, sin embargo, tendría que aprender a detectar un objeto de manera independiente en cada posición de la imagen.

En problemas de clasificación, estas capas convolucionales, combinadas con capas de agrupación, **se suelen usar para reducir el tamaño de la señal** para, en las últimas etapas de la red, poder hacer uso de bloques MLP que realizan la clasificación a partir del mapa de características obtenido por las convoluciones.

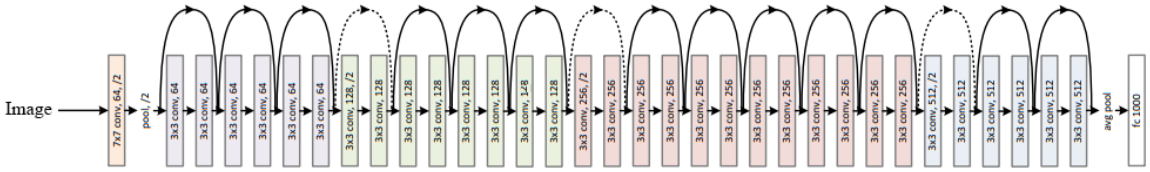
4.3. Convolución Traspuesta

La convolución traspuesta es una capa que, al contrario que la convolución, **amplía el tamaño de la señal**. Es comúnmente utilizada en tareas como el *upsampling* de imágenes y en redes como las GAN y VAE.

De manera similar a la convolución, se definen con un filtro o *kernel* que se desplaza por los datos de entrada, solo que en este caso en lugar de disminuir el tamaño de la señal lo aumentan. Sus parámetros e hiperparámetros son similares a los de la convolución: canales de entrada y salida, tamaño de *kernel*, *stride*, *padding* y *dilation*. Puede entenderse como la operación inversa a la convolución.

4.4. Redes Residuales

Las redes convolucionales tienen dificultad para extenderse a muchas capas, debido al problema del *vanishing gradient*. Para combatir esto, las **redes residuales** o ***ResNets*** introducen conexiones de salto (*skip connections* en inglés), que se saltan varias capas de la red, mapeando capas con otras capas posteriores sin realizar ninguna transformación. En las capas de "destino" de la conexión convergen tanto esta conexión de salto como las conexiones estándar que atraviesan el resto de las capas de la red. La suma de estos dos caminos se utiliza como dato de entrada para la siguiente capa, lo que hace que estas conexiones de salto sean de tipo residual.



ResNet-34

Estas redes residuales posibilitan el entrenamiento de redes muy profundas y han tenido mucho éxito en tareas de visión artificial.

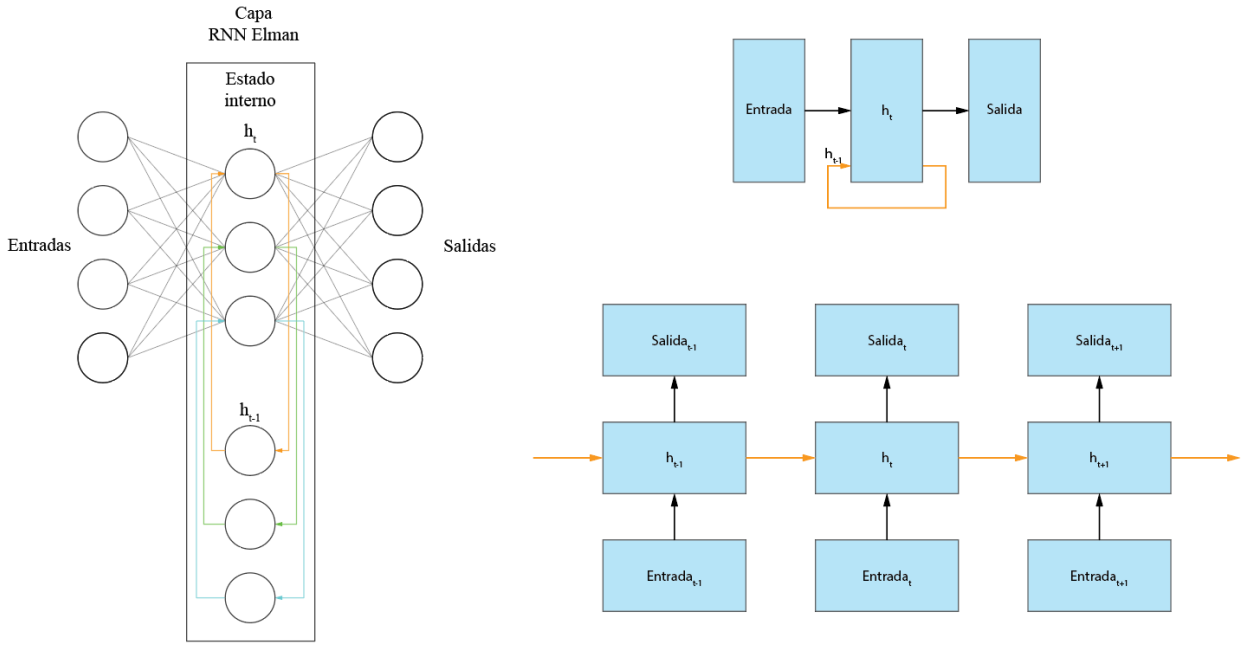
4.5. Redes Neuronales Recurrentes

Las redes neuronales recurrentes, ***Recurrent Neural Networks*** o **RNN** en inglés, son un tipo de redes que presentan conexiones retroactivas, en las que capas de la red se conectan con capas anteriores de la misma, formando así conexiones cíclicas.

En las RNN, no solo se procesa la entrada, si no que **se mantiene un estado interno o memoria**, que se actualiza con cada nueva entrada de la secuencia. De esta manera, se puede utilizar información de eventos anteriores, lo que hace este tipo de redes especialmente efectivas para modelar secuencias, como texto, audio o series temporales.

El tipo básico de RNN es la ***Elman Network***, *y* el estado de su memoria interna se define con la siguiente formula:

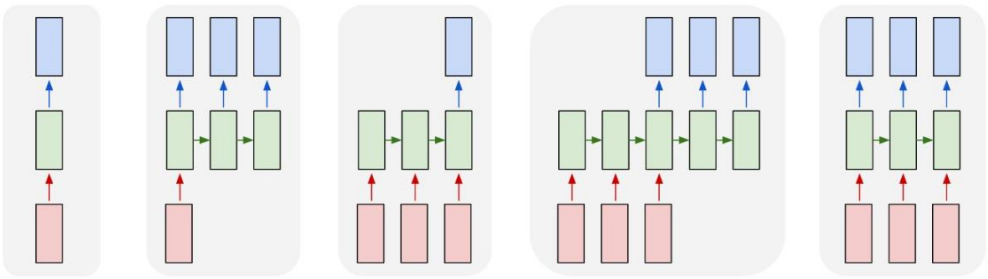
$$h_t = \tanh(x_t w + b + h_{t-1} w_h + b_h)$$



Diagramas de una capa Elman. Izquierda: diagrama completo.
Derecha y arriba: diagrama comprimido. Derecha y abajo: diagrama desdoblado en el tiempo.

Donde h_t es el nuevo estado de la memoria, h_{t-1} el estado de la memoria en el instante anterior, x_t la nueva entrada de datos, w y b los pesos y biases de la señal de entrada, y w_h y b_h los pesos y biases de la memoria interna. Se puede usar en lugar de la tangente hiperbólica otra función de activación, como ReLU.

En las arquitecturas de red presentadas anteriormente, las MLP y CNN, los datos de entrada y salida de la red tienen que tener una forma fija: son tensores con unas dimensiones y tamaño dados. Las RNN pueden, sin embargo, recibir como **entrada de datos secuencias de tamaño arbitrario**. De la misma manera, su salida puede ser una secuencia de tamaño arbitrario.



Las RNNs son muy flexibles en cuanto a la forma de sus entradas y salidas. De izquierda a derecha:
(1) Tamaño fijo a tamaño fijo (eg.: clasificación de imagen). (2) Tamaño fijo a secuencia (eg.: descripción en texto de una imagen).
(3) Secuencia a tamaño fijo (eg.: análisis de sentimiento de un texto). (4) Secuencia a secuencia (eg.: traducción de texto de un idioma a otro).
(5) Secuencia a secuencia, sincronizadas (eg.: clasificación frame a frame de un video).

Antes del desarrollo de los *transformers*, las RNN eran el método más usado para procesamiento de lenguaje natural. Pero debido a los problemas del *vanishing* y *exploding gradients,* tienen dificultades para aprender memorias a largo plazo. Para mitigar esto, existen variantes de la RNN como las LSTM (*Long Short-Term Memory*) y GRU (*Gated Recurrent Unit*) que ayudan a gestionar mejor la memoria a largo plazo.

4.6. Long Short Term Memory

Las *Long Short-Term Memory* o LSTM, son un tipo de RNN diseñadas para aprender dependencias a largo plazo. Incluyen unas unidades de memoria, llamadas celdas, y cuatro puertas que regulan su tránsito de información. La ***Forget Gate*** define que información antigua se borra y que se mantiene. La ***Input Gate*** y la ***Cell Gate*** definen que información nueva se añade a la celda. Y la ***Output Gate*** define que información de la celda se usa como output.

Así, las LSTM pueden almacenar información y usarla de forma selectiva a lo largo de una secuencia de datos. Las operaciones que rigen la celda de memoria del LSTM son:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Donde h_t es el nuevo output de la memoria, h_{t-1} el output de la memoria en el instante anterior, c_t el nuevo estado interno de la celda, c_{t-1} el estado interno de la celda en el instante anterior, x_t la nueva entrada de datos, i_t, f_t, g_t y o_t las puertas *input, forget, cell* y *output* respectivamente, y los distintos w y b los pesos y *biases* de cada puerta. σ es la función sigmoide y $\odot$ el producto de Hadamard.

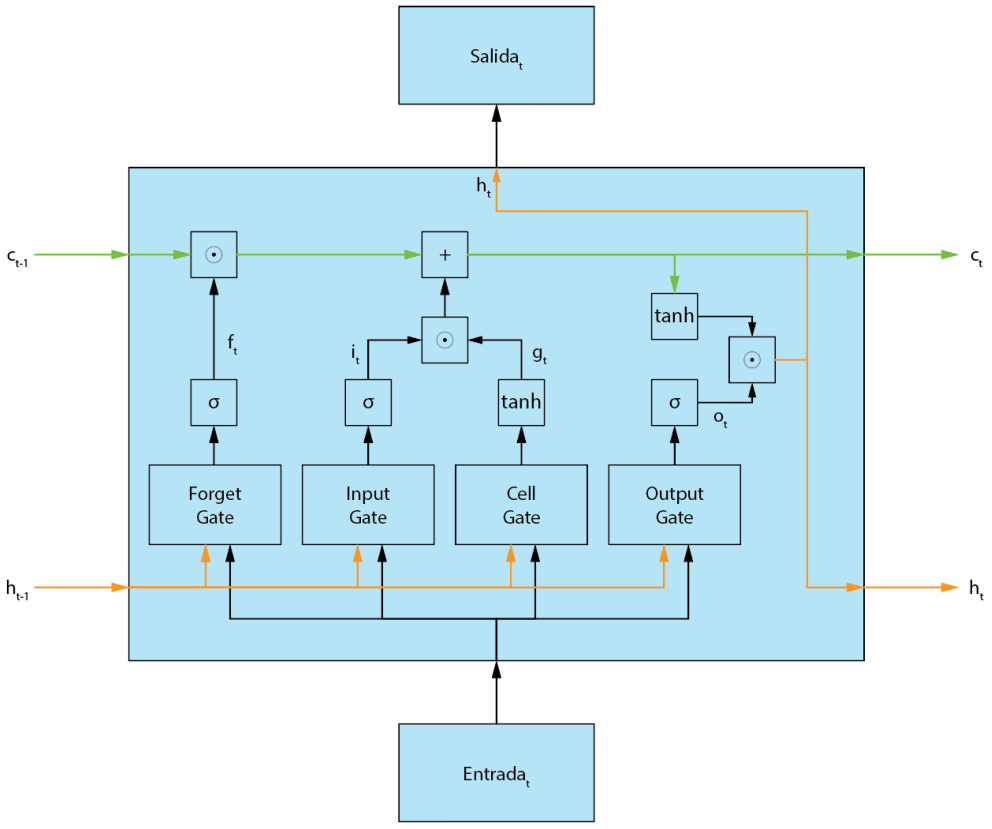


Diagrama de una capa LSTM.

4.7. Gated Recurrent Unit

Las GRU (**Gated Recurrent Units**) son similares a las LSTM, pero más simples y de menor coste computacional. Incluyen una ***Reset Gate***, que decide cuánta información pasada se debe olvidar, y una ***Update Gate***, que decide qué información de la memoria se debe actualizar.

Las operaciones que se ejecutan dentro de una GRU son las siguientes:

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)}
\end{aligned}
$$

Donde h_t es el nuevo estado de la memoria, h_{t-1} el estado de la memoria en el instante anterior, x_t la nueva entrada de datos, r_t y z_t, las puertas reset y update respectivamente, n_t la nueva información propuesta, y los distintos w y b, los pesos y *biases* de cada puerta. σ es la función sigmoide y $\odot$ el producto de Hadamard.

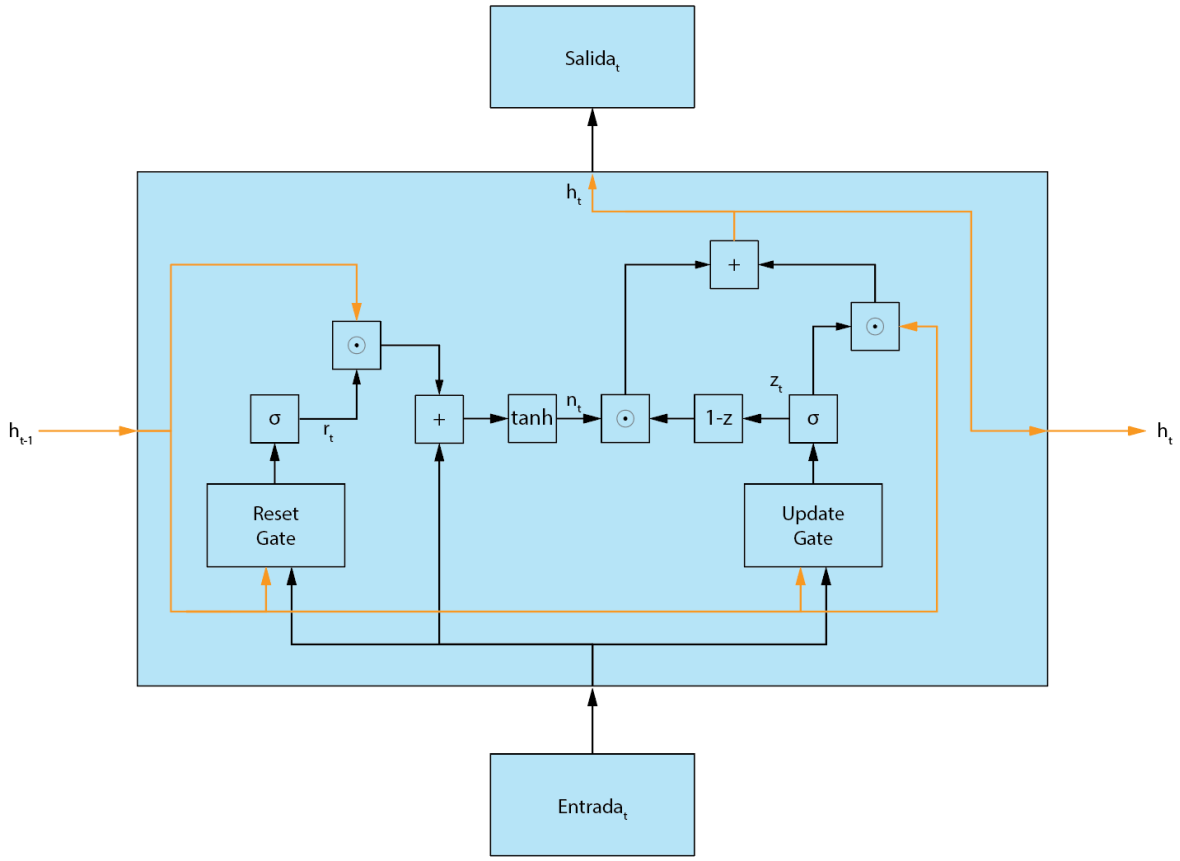


Diagrama de una capa GRU.

4.8. Transformers

Los *transformers* son una arquitectura de red neuronal que ha explotado en popularidad recientemente, por los avances que ha aportado al campo de procesamiento de lenguaje natural, habiendo hecho posible los últimos asistentes de IA como ChatGPT.
Lo que caracteriza a las arquitecturas transformer es el uso de un mecanismo de **atención**, que hace posible que la red aprenda relaciones entre distintas partes de los datos de entrada sin importar su posición relativa. A diferencias de las redes recurrentes, que procesan de manera secuencial los datos, los transformers procesan toda la entrada a la vez, lo que hace que sean más eficientes.

El mecanismo de atención asigna pesos a diferentes partes de la secuencia de entrada en función de lo relevante que sea para una tarea específica. Así, permite que cada posición en la secuencia atienda en mayor o menor medida a las demás posiciones, lo que posibilita establecer relaciones a larga distancia. Por ejemplo, en las frases:

Mario se comió la hamburguesa porque **estaba** hambriento.
Mario se comió la **hamburguesa** porque **estaba** muy buena.

En la primera frase, estaba hace referencia a Mario, y en la segunda, a hamburguesa. La capa de atención dará un peso mayor a Mario en el primer caso y a hamburguesa en el segundo.

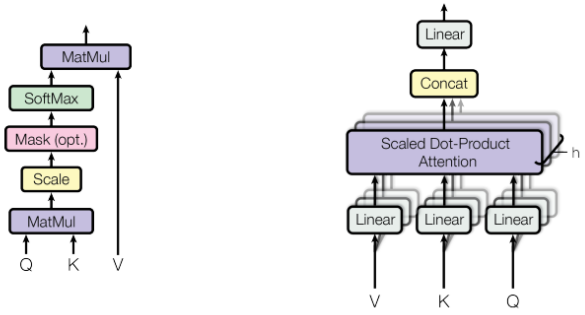
Al comienzo del operador de atención hay tres capas lineales que reciben la señal de entrada: ***query*** (la consulta que indica qué parte de la entrada queremos enfocar), ***key*** (las claves que se utilizan para calcular la similitud con la *query* y determinar qué partes de la entrada son relevantes) y ***value*** (la información asociada con esas partes relevantes); cada una de estas tres capas tiene sus pesos que se ajustan durante el entrenamiento. Después se calcula una matriz de puntuación de atención multiplicando matricialmente el resultado de *query* y *key*, escalándola con el número de dimensiones de la señal D y pasándola por una función de activación Softmax para normalizar las puntuaciones.

$$A_{q,k} = \frac{\exp\left(\frac{1}{\sqrt{D^{QK}}} Q_q \cdot K_k\right)}{\sum_l \exp\left(\frac{1}{\sqrt{D^{QK}}} Q_q \cdot K_l\right)}$$

La puntuación de atención normalizada se multiplica matricialmente por el resultado de *value* y con eso tenemos la salida del operador de atención.

$$Y_q = \sum_k A_{q,k} V_k.$$

Normalmente se utilizan varios operadores de atención en paralelo, formado lo que se llama ***Multi-Head Attention***. Cada "cabeza" tiene sus pesos para *query, key* y *value* independientes, y así se consigue añadir más complejidad al sistema para que sea capaz de establecer relaciones más ricas.

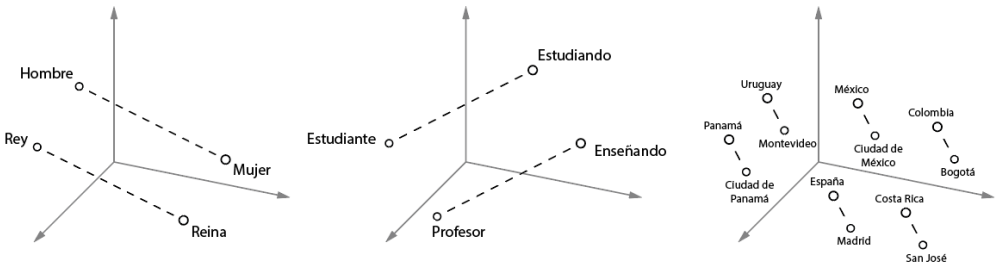


Izquierda: diagrama del operador de atención. Derecha: *Multi-Head Attention*

La arquitectura *transformer* original, propuesta en el paper "Attention is all you need", se diseñó para traducción de texto de uno a otro idioma, y está formada por un ***encoder***, que convierte la señal de entrada a un **espacio latente**, y un ***decoder*** que genera la salida a partir de la representación latente del *encoder* y la salida generada previamente.

Un **espacio latente** es un espacio abstracto que codifica la información significativa de los datos. En el ejemplo del traductor, el *encoder* convierte el texto de entrada en el idioma original a un espacio latente que es una representación abstracta de su significado, y el *decoder* usa esta representación latente y la convierte a otro idioma, el idioma de destino. La forma que adquiere el espacio latente y como se convierte hacia y desde él, se ajusta durante el proceso de entrenamiento.

Un ejemplo simplificado de cómo funciona el espacio latente es el siguiente: si hay una dimensión que define el género de un sujeto, moverse a lo largo de esa dimensión puede cambiar una palabra de “hombre” a “mujer”, de la misma manera que cambia el género a otros sujetos, como “rey” a “reina”.



Ejemplos de posibles dimensiones en un espacio latente.

Volviendo a la arquitectura transformer propuesta en "Attention is all you need", el ***encoder***, que convierte el texto de entrada a un espacio latente, está compuesto por una serie de bloques que empiezan con una capa de auto-atención (***self-attention***), en la que el texto de entrada "atiende" a otras partes de ese mismo texto; por lo tanto, el operador de atención relaciona cada palabra del texto de entrada con otras partes de ese mismo texto.

Tras este operador *Multi-Head Attention*, dentro de cada bloque se encuentra una *Feed Forward Network* (una red MLP). Además, hay conexiones residuales (una *skip connection* que se suma) y una normalización (de tipo *layer normalization*) tanto para el operador de atención como para la MLP.

Este bloque del *encoder* se repite varias veces, de tal manera que la salida de un bloque se usa como entrada para el siguiente. La entrada al primer bloque es el texto en el idioma original, pasado por una capa de *input embedding* a la que se añade un *positional encoding*.

La entrada de texto, antes de pasarla por el input embedding, tiene que ser convertida a *tokens.* Un ***token*** es una representación numérica de un fragmento de texto, que puede ser un carácter, un conjunto de caracteres o incluso palabras completas. Como ejemplo, en un modelo de lenguaje a nivel de caracteres, el diccionario o vocabulario (la colección de todos los *tokens* posibles) tendría un tamaño de 30 (las 27 letras del abecedario, el "espacio", más un símbolo de principio y otro de fin de secuencia), o más si se quieren añadir símbolos de puntuación, acentos, mayúsculas.... De tal manera que cualquier texto sería una secuencia de *tokens*, siendo cada *token* un valor numérico que hace referencia de a que letra corresponde. Normalmente, los modelos de lenguaje natural como ChatGPT tienen un *embedding* en el que cada token corresponde a un conjunto de letras, aproximadamente un par de sílabas.

El ***input embedding*** convierte la secuencia de entrada de *tokens* a una representación vectorial para que sea procesada por la red. Es un operador simple que relaciona cada *token* de la entrada con un punto en el espacio del *embedding*.

Por otra parte, el ***positional encoding*** añade información a cada elemento de la entrada (cada *token*) sobre qué posición ocupa en la secuencia. Como los *transformers* no tienen de manera nativa información sobre el orden la secuencia, se introduce este *positional encoding* para añadir a la red esta información.

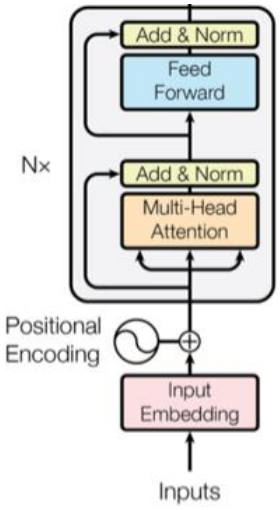


Diagrama del encoder de un transformer

La otra gran parte del *transformer* propuesto en "Attention is all you need", es el ***decoder***. El *decoder* convierte la representación latente generada por el *encoder* a texto en el idioma de destino.

La entrada del *decoder* es similar a la del *encoder*, la entrada pasa por un *embedding*, que aquí se le llama *output embedding*, porque la secuencia de entrada al *decoder*, es la secuencia de salida (el texto en el idioma de destino) del conjunto del *transformer*. A este *embedding* se le añade también su *positional encoding*.

Al igual que en el *encoder*, el *decoder* está compuesto por una serie de bloques que se ejecutan de manera secuencial, la salida de uno es la entrada del siguiente. Cada bloque comienza con una capa de *self-attention*, como la del *encoder*, en la que el texto se atiende a sí mismo.

Esta capa de *self-attention* tiene una diferencia respecto a la del *encoder:* presenta una **máscara**, que hace que un *token* de la secuencia no pueda atender a la parte de la secuencia posterior a él, esto es, no puede ver el futuro de la secuencia. En el caso del encoder tenía sentido poder ver a futuro, puesto que, en función del contexto, una palabra puede significar una u otra cosa. Pero en el caso del decoder, el futuro de la secuencia todavía no está escrito, por lo que no tiene sentido atender a esa parte. Cuando la capa de *self-attention* presenta una máscara, se le suele llamar ***masked self-attention***.

Después de la capa *masked self-attention,* el *decoder* presenta una capa de ***cross-attention***. En el *cross-attention*, cada *token* de la secuencia del *decoder* "atiende" a la secuencia del espacio latente generada por el *encoder.* A nivel matemático, el bloque es igual al de *self-attention*, con la diferencia de que la entrada *query* viene del *decoder,* y las *key* y *value* vienen de la salida del último bloque del *encoder,* como se muestra en el diagrama.

El bloque del decoder termina con una red MLP como la que había al final del encoder. Por último, tras repetir varias veces el bloque del decoder, hay una capa lineal y un softmax para obtener las probabilidades de cuál va a ser el siguiente token en la secuencia.

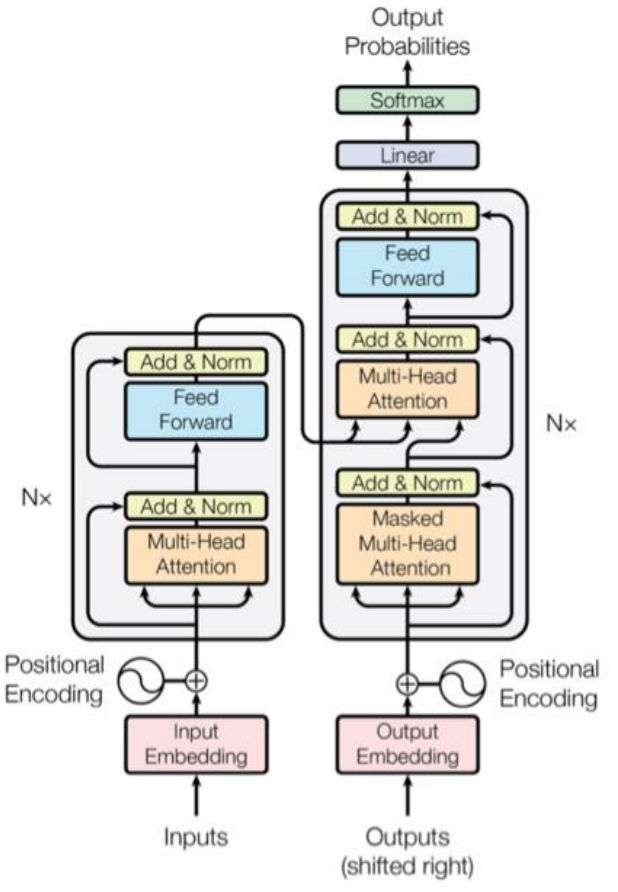


Diagrama de *transformer* con *encoder* y *decoder*.

La red mostrada es la presentada en el paper que dio a luz la arquitectura *transformer*, pero desde entonces se han encontrado otros usos de estos *transformers*, usando variaciones sobre la arquitectura detallada, demostrando su versatilidad y potencia.

Por ejemplo, en el campo de la visión artificial, el ***Vision Transformer*** o *ViT,* plantea una arquitectura con **sólo un *encoder***: la imagen de entrada se divide en parches a partir de los cuales se forma una secuencia que se usa como entrada de un *encoder* similar al ya mostrado. Este *encoder* transforma la entrada a un espacio latente, en el que, como vimos, está codificada la información significativa.

Como novedad del ViT, a la salida del *encoder* hay una red MLP que ejecuta la tarea de clasificación, de manera similar a una red convolucional con capas convolucionales para reducir la dimensión de la imagen y un MLP al final para la fase de clasificación.

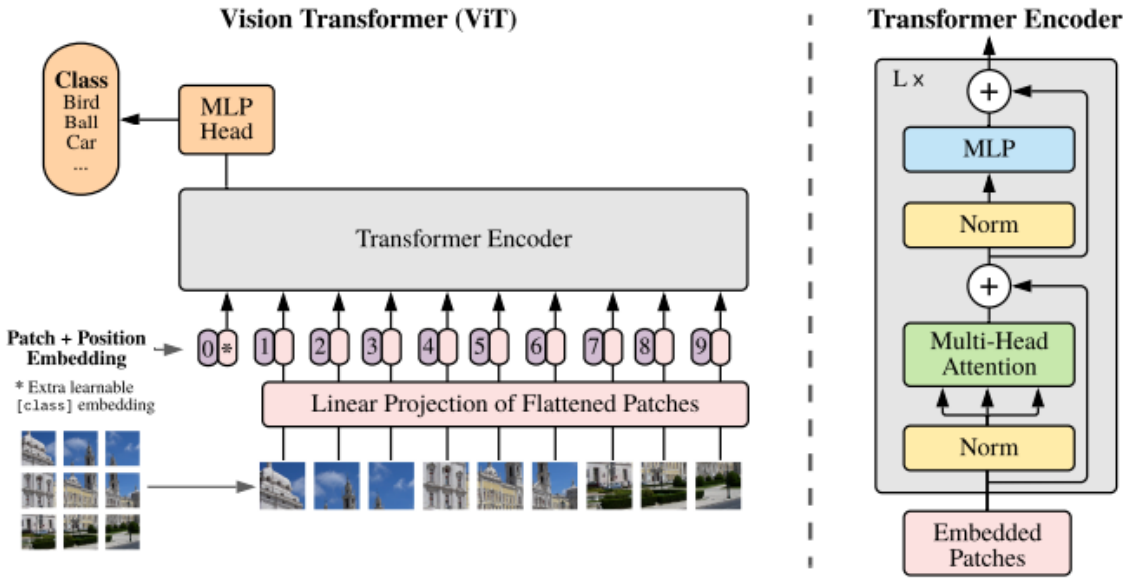


Diagrama de un vision transformer con sólo encoder.

Otro ejemplo interesante de uso de los *transformers* son los modelos generativos de texto, como **GPT (*Generative Pre-trained Transformer*)**. En GPT, se hace uso solo de un *decoder,* en el que no hay capa de *cross-attention*, al no haber *decoder* al que atender. La capa de *self-attention* es similar a la del *transformer* que hemos visto en detalle anteriormente, con la máscara para no atender "al futuro". Al no haber *encoder*, este GPT se entrena para predecir que texto vendrá después de la secuencia de entrada: si el primer *transformer* que vimos era un traductor, y ViT era un clasificador de imágenes, GPT es un "continuador" de texto.

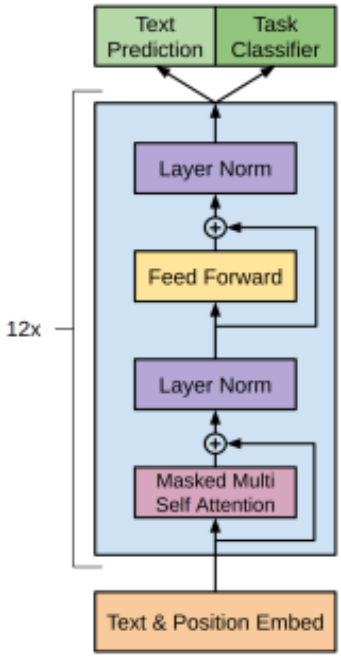


Diagrama de un GPT con sólo decoder.

4.9. Variational Autoencoders

Los *autoencoders* son modelos que mapean una señal de entrada, de alta dimensión, con una de **representación latente** de baja dimensión, y luego mapean de vuelta a la señal original, asegurando que la información se preserva. Se usan para eliminación de ruido (*denoising*), o para detección de anomalías o *outliers*.

Los ***Variational Autoencoders*** **(VAE)** son modelos generativos, de la familia de los *autoencoders*, con la particularidad de que el espacio latente se fuerza durante el entrenamiento a tener forma de una distribución normal, para que después, durante la inferencia, se pueda muestrear o *samplear* de manera aleatoria un punto en este espacio y al decodificar resulte en una salida coherente.

4.10. Generative Adversarial Networks

Las ***Generative Adversarial Networks*** **(GAN)** son modelos generativos que combinan dos redes: un **generador**, que toma un input aleatorio, con el produce una señal, como por ejemplo una imagen, y un **discriminador**, que clasifica si la imagen viene del set de entrenamiento (es real) o fue generada por el generador (es falsa).

Durante el entrenamiento de esta red, se introducen datos reales del set de entrenamiento con la etiqueta "real" y datos generados por el generador con la etiqueta "falso" para entrenar al discriminador.

Por otro lado, el generador se entrena introduciendo datos generados por él con la etiqueta "verdadero", para que aprenda cómo generar los datos de tal manera que engañe al discriminador: el proceso de *backpropagation* le informa de que pistas usa el discriminador para decidir si un dato es real o no, y el generador usa esa información para entrenarse.

Así, el entrenamiento de la GAN consiste en un ciclo en el que se van entrenando ambas redes, alternando una y otra. Una vez entrenado, el generador produce datos indistinguibles de los reales. Llegados a este punto, el rendimiento del discriminador será malo, porque no es capaz de diferenciar datos generados y reales. Esto puede suponer un problema, porque la información que le llegue al generador a través del *backpropagation* del discriminador será cada vez peor, lo que puede hacer que el generador empeore. Debido a este fenómeno las GANs puede ser difíciles de entrenar.

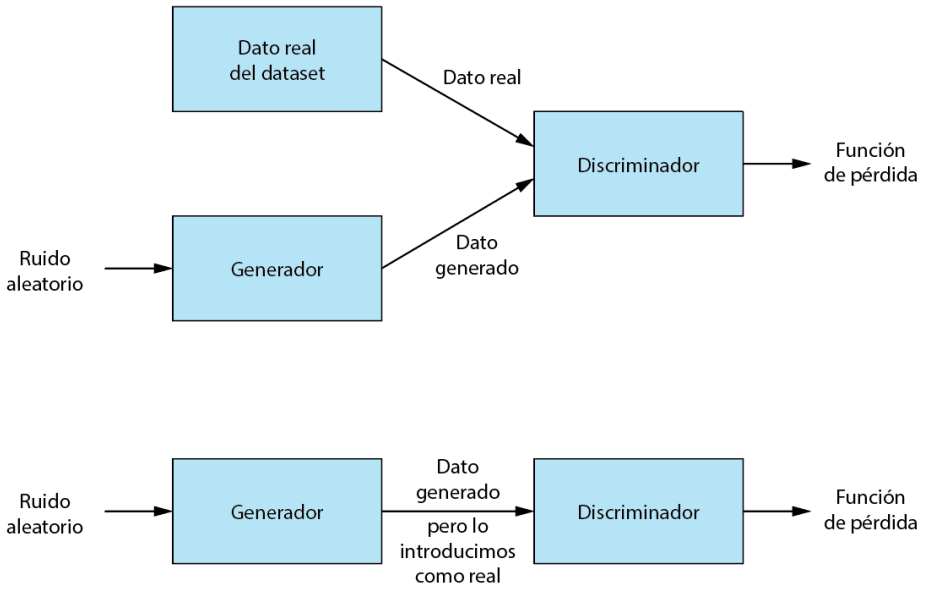


Diagrama de una red GAN. Arriba: entrenamiento del discriminador. Abajo: entrenamiento del generador.