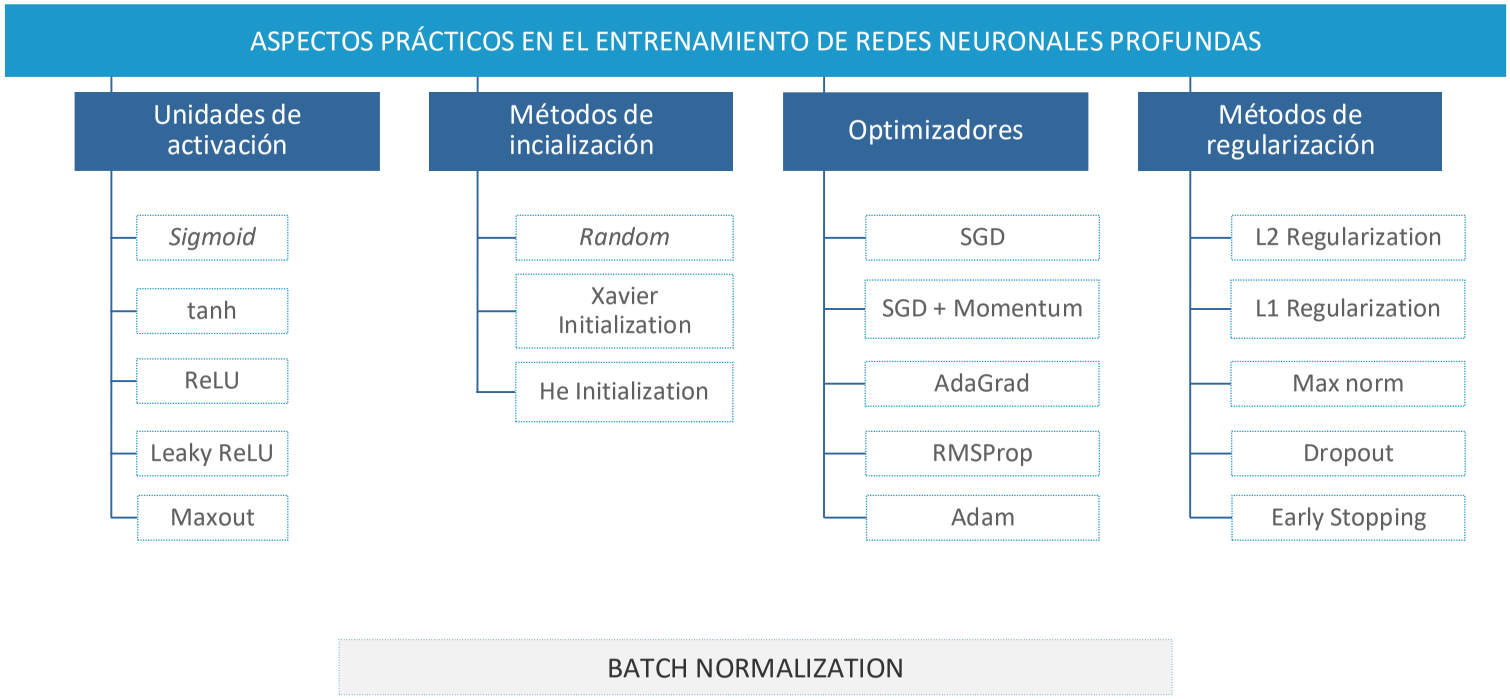
Sistemas Cognitivos Artificiales

Aspectos prácticos en el entrenamiento de redes neuronales profundas

Índice

Esquema



Ideas clave

4.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás leer las **Ideas clave** que se desarrollan a continuación.

Hasta ahora hemos estudiado los conceptos matemáticos básicos que rigen el funcionamiento de las redes neuronales. Como sabemos, una red neuronal es un problema complejo de optimización con muchos parámetros en el que obtener una solución adecuada no es sencillo. En este tema veremos varios de los problemas que aparecen al entrenar redes neuronales profundas y estudiaremos técnicas para entrenarlas de manera más efectiva y eficiente.

Este tema nos ayudará a verificar si hemos comprendido los conceptos básicos vistos en los temas anteriores. Es importante comprender:

- Las trabas al aprendizaje que aparecen en ciertas unidades de activación, en las formas de inicializar los parámetros de la red y en el algoritmo básico SGD.
- Cómo otras alternativas facilitan el entrenamiento de las redes neuronales profundas.
- Qué es el concepto de *overfitting* y cómo las técnicas de regularización ayudan a combatirlo.

4.2. Unidades de activación

Uno de los elementos más críticos en las redes neuronales son las unidades de activación o *non-linearities* presentes en las neuronas de la red. Hemos visto ya el funcionamiento de la unidad *sigmoid*, utilizada de manera clásica en el mundo de las redes neuronales. Uno de los avances en el mundo del *deep learning* que ha permitido el entrenamiento de redes más profundas y complejas ha sido la irrupción de nuevas unidades de activación que solventan varios de los problemas que esta unidad presenta.

Sigmoid

Como ya sabemos, la función *sigmoid* se define de la siguiente manera:

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

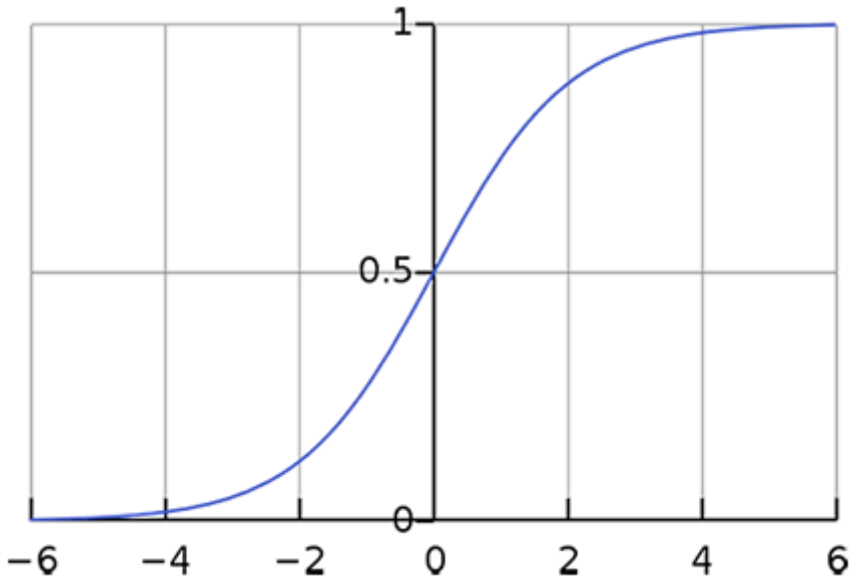


Figura 1. Gráfica de la función *sigmoid*.
Fuente: https://github.com/stanfordnlp/cs224n-winter17-notes/blob/master/notes1/fig/sigmoid.png

La unidad se caracteriza por convertir cualquier número real en un número entre 0 y 1, haciendo que valores positivos grandes sean prácticamente 1, mientras que valores negativos grandes tienden a 0.

Las unidades *sigmoid* han caído bastante en desuso en las arquitecturas modernas debido a una serie de **desventajas**:

Las unidades *sigmoid* saturadas «matan» los gradientes. Como podemos apreciar en la gráfica, para valores altos positivos o negativos, la unidad queda saturada en valores cercanos a 1 y -1, donde la pendiente de la gráfica es nula. En otras palabras, la derivada o gradiente de la función es 0. Como sabemos, durante la propagación hacia atrás, el gradiente local de la unidad es multiplicado por el gradiente de la salida de la unidad. Esto implica que, si el gradiente de la función *sigmoid* es casi 0, el valor del producto también lo será, «matando» de manera efectiva el gradiente que se propaga y por tanto eliminando la señal e impidiendo el aprendizaje.

La salida no está centrada en 0. Los valores de *sigmoid* son estrictamente positivos, lo cual crea ciertas dinámicas no deseables en el entrenamiento de las redes. En particular, si una neurona recibe todos sus *inputs* con valores positivos, la derivada respecto de los pesos *w* será siempre positiva o negativa, creando una ineficiencia en el aprendizaje. El hecho de que en este caso la salida no esté centrada en 0 provoca estas dinámicas en las siguientes capas de la red.

Implica el cálculo de una función exponencial e^x. Si bien esto no es un cuello de botella a la hora de hacer cálculos, el cálculo de una exponencial conlleva cierta complejidad en comparación con otras operaciones.

Tanh

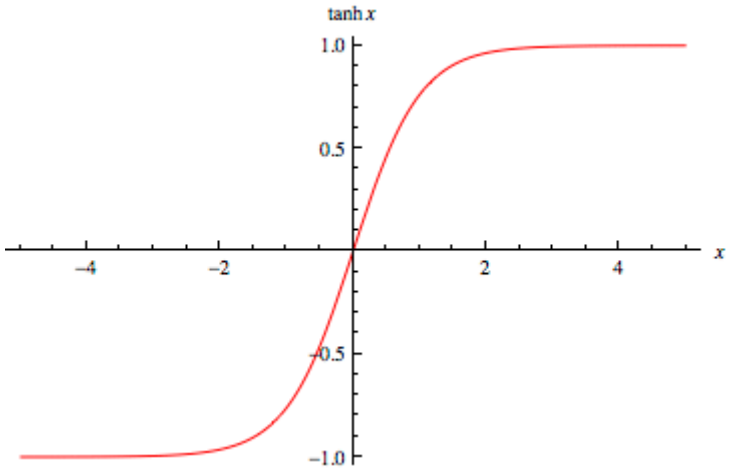


Figura 2. Gráfica de la tangente hiperbólica.
Fuente: http://mathworld.wolfram.com/HyperbolicTangent.html

La **tangente hiperbólica**, $tanh(x)$, presenta una forma bastante similar a *sigmoid*: de nuevo saturando por los lados, pero esta vez entre los valores -1 y 1. La gran ventaja respecto a *sigmoid* es que la salida está centrada alrededor de 0. Sin embargo, debido a la saturación de valores, la unidad sigue teniendo el problema de «matar» a los gradientes. En la práctica, es siempre preferible usar tanh antes que *sigmoid*.

ReLU

La unidad ReLU (*rectified linear unit*) tiene la siguiente fórmula:

$$f(x) = x^{+} = \max(0, x)$$

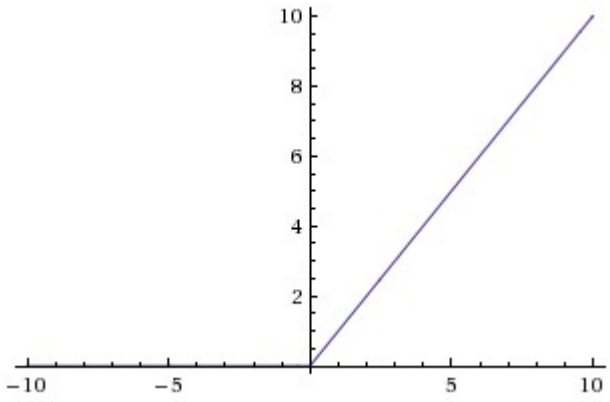


Figura 3. Gráfica de la unidad ReLU.
Fuente: https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning

La unidad ReLU tiene salida 0 para valores menores que 0 y la función identidad para valores mayores que 0. Es la unidad de activación más usada en la práctica.

Sus **ventajas** son:

- No satura en el régimen positivo. La tendencia a «matar» gradientes que veíamos en *sigmoid* y tanh desaparece para valores mayores que 0.
- Computacionalmente muy eficiente. La unidad ReLU no requiere del cálculo de exponenciales. Es suficiente con aplicar un *threshold* y convertir todos los números negativos en 0. El resto de valores se quedan igual.
- Acelera la convergencia de *Stochastic Gradient Descent*. Hasta seis veces más rápida que las unidades *sigmoid* o *tanh*, según Krizhevsky et al. (2012).

Por otro lado, las ReLU también tienen algunos **problemas**:

- Como pasaba con *sigmoid*, la salida no está centrada en 0.
- Las ReLU son una unidad frágil y pueden «morir» durante el entrenamiento. Es posible que un cambio grande en el gradiente haga que sus *weights* cambien de manera que la unidad no consiga volver a activarse nunca, es decir, no producir nunca más un valor mayor que 0. Esto provoca que el gradiente que fluye por la unidad sea siempre 0 a partir de ese momento, efectivamente convirtiendo a la neurona con la ReLU en una neurona «muerta». De hecho, es normal que al

entrenar redes neuronales con ReLU veamos que un gran número de ellas nunca se activan ante ningún *input* del *training set*, si bien esto no tiene por qué desembocar en un mal rendimiento de la red. Normalmente, el problema de las unidades muertas mejora con *learning rates* pequeños.

Leaky ReLU

La Leaky ReLU es una variante de la ReLU que intenta solucionar el problema de las ReLU «muertas». Básicamente, la función añade una pequeña pendiente en el régimen negativo, evitando que la unidad pueda morir:

$$f(x) = \begin{cases} x \\ 0.01x \end{cases} if\ x > 0\ otherwise$$

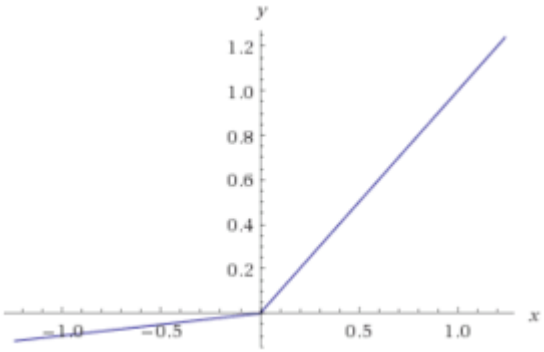


Figura 4. Gráfica de la unidad ReLU.
Fuente: https://datascience.stackexchange.com/questions/5706/what-is-the-dying-relu-problem-in-neural-networks

Una variante de la Leaky ReLU es la PReLU, la cual añade un parámetro (que también es aprendido) en vez de 0.01 como pendiente, permitiendo mayor flexibilidad.

$$f(x) = \begin{cases} x \\ ax \end{cases} if\ x > 0\ otherwise$$

Finalmente, otra variante es la **ELU**, que intenta resolver el problema de las activaciones no centradas en 0, lo cual hace que el entrenamiento sea más lento.

$$f(x) = \begin{cases} x \\ a(e^x - 1) \end{cases} if\ x > 0\ otherwise$$

Maxout

Maxout es otra unidad propuesta para solucionar el problema de las ReLU que mueren durante el entrenamiento. La unidad generaliza tanto a la ReLU como a la Leaky ReLU y presenta una forma un tanto distinta de las clásicas unidades que hemos visto de la forma $f(w \cdot x + b)$. Su fórmula es:

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Como vemos, la unidad introduce dos parámetros w en vez de uno. Esta unidad se convertiría en ReLU con $w_1 = 0$ y con $b_1 = 0$. Si bien es más versátil y no «muere» como las ReLU, esta unidad tiene el problema de que necesita el doble de parámetros.

¿Cuál usar?

Tenemos un gran surtido de unidades de activación para elegir. En general, en la actualidad se recomienda utilizar ReLU, teniendo cuidado con la *learning rate* y monitorizando el entrenamiento para que no haya demasiadas unidades muertas. Si esto es un problema se puede probar alguna variante de Leaky ReLU o Maxout. En general, debería evitarse el uso de *sigmoid* por la gran cantidad de problemas que presenta.

4.3. Inicialización de parámetros

Hemos visto que las redes neuronales tienen un gran número de parámetros que aprender: en particular, los pesos w y los biases b. También hemos repasado cómo el proceso de aprendizaje se realiza mediante *stochastic gradient descent*. Sin embargo, ¿con qué valores inicializamos todos estos parámetros? Esta pregunta es muy importante, ya que los valores iniciales de los parámetros tienen una gran repercusión en el proceso de entrenar una red neuronal.

Error: inicializar todo a 0

Podríamos pensar que una forma de inicializar nuestros parámetros es en un punto medio, por ejemplo con todo 0. Sin embargo, esto sería un grave error. Si todas las neuronas de la red tienen la misma salida (lo cual pasaría si inicializamos todo a 0), todas ellas tendrán el mismo gradiente durante la *backpropagation* y, por tanto, cambiarán los parámetros de igual manera. Es necesaria cierta asimetría en los valores de inicialización para evitar este fenómeno.

Mejor: inicialización aleatoria

Para evitar el problema de la inicialización a todo 0, necesitamos «romper la simetría» en la red. Una opción sencilla es inicializar los pesos de manera aleatoria usando una distribución normal centrada en 0 y con un valor pequeño de desviación típica. Por ejemplo:

```
W = 0.01* np.random.randn(fan_in, fan_out)
```

Al hacer todos los pesos aleatorios y por tanto distintos, conseguimos que los *updates* del gradiente sean distintos y, así, la red consiga aprender. Es importante destacar que los números tienen que ser pequeños (de aquí el valor de 0.01), ya que unos

pesos grandes podrían hacer que las funciones de activación se saturasen frecuentemente.

Si bien esta solución es correcta, aún tiene ciertos problemas. En redes muy profundas, el hecho de que los pesos se inicialicen con números pequeños puede dar problemas a la hora de entrenar, haciendo que los valores en la red se vuelvan más y más pequeños y que los gradientes tiendan a 0.

Xavier *initialization* y He *initialization*

El caso de arriba es un ejemplo claro de la dificultad de trabajar con redes neuronales profundas, donde cualquier problema a la hora de inicializar los parámetros puede impedir que la red entrene correctamente. La inicialización de parámetros es un campo de estudio activo, con continuos avances en busca de mejores estrategias de inicialización.

Una solución al problema visto en el apartado anterior es la conocida como **Xavier initialization**. La idea aquí es intentar que la varianza de salida de una neurona sea igual a la varianza de entrada. Esto hace que todas las neuronas de la red tengan aproximadamente la misma distribución de salida, lo que acelera la convergencia al entrenar. En particular, para un peso w individual una formulación de esta inicialización es:

```
w = np.random.randn(n) * sqrt(2/(n_in + n_out))
```

Donde:

- `n_in` es el número de *inputs* de la neurona.
- `n_out` el número de neuronas en la capa siguiente.

Un estudio más reciente (He, Zhang, Ren y Sun, 2015), deriva una inicialización específica para unidades ReLU:

```
w = np.random.randn(n) * sqrt(2/n)
```

Donde:

- `n` es el número de *inputs* de la neurona.

En la práctica, se recomienda utilizar esta inicialización cuando entrenamos con redes neuronales con ReLUs.

Inicialización de los *biases*

Hasta ahora, hemos hablado de inicializar los pesos w, pero no sobre los biases b de la red neuronal. Estos no son tan críticos una vez que los pesos w son inicializados aleatoriamente, lo que ya rompe la simetría de la red. Es muy común inicializarlos simplemente a 0.

4.4. *Batch normalization*

Uno de los mayores avances en el entrenamiento de redes neuronales profundas en los últimos tiempos ha sido la técnica de ***batch normalization***. En gran medida, esta técnica evita muchos de los problemas creados por una mala inicialización de parámetros y posibilita una mejor convergencia al entrenar redes neuronales.

En los puntos anteriores hemos explicado que la distribución de valores de salida en las unidades de activación tiene una gran repercusión en el entrenamiento de una red neuronal. La idea de *batch normalization* consiste en forzar que la entrada de las unidades de activación en todas las capas siga una distribución normal estándar. Para

ello, se obtiene la media y la varianza empíricas de los valores por ***training batch***. Esto es, por cada batch que entrenamos, obtenemos una media y las varianzas a partir de todos los *inputs* en esa batch. Esta normalización se hace por cada dimensión del *input* mediante la siguiente fórmula:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

Normalmente, la operación de *batch normalization* se ejecuta **antes de la aplicación de la función de activación** y, por tanto, justo después de la operación lineal $Wx + b$, donde x es el *input* de la neurona.

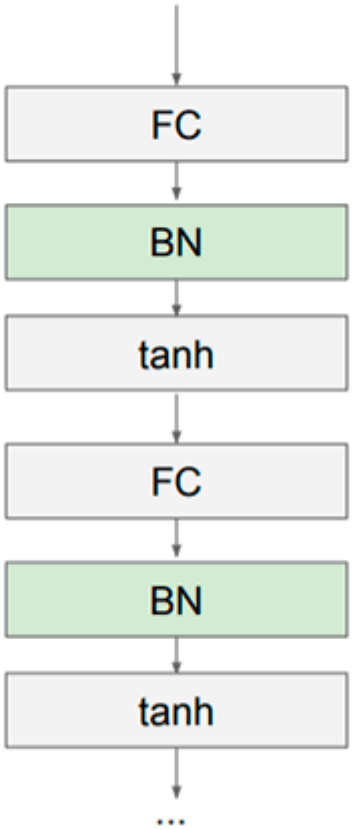


Figura 5. Aplicación de *batch normalization*.
Fuente: http://forums.fast.ai/t/lesson-2-using-batch-normalization-after-non-linearity-or-before-non-linearity/4817

Si normalmente tenemos una función de activación $f(Wx + b)$, la normalización se ejecuta al calcular $Wx + b$ y antes de aplicar la *non-linearity*.

Esta normalización capa a capa puede reducir el poder de expresión de una red neuronal. Por tanto, es habitual reemplazar los valores de la normalización por una nueva parametrización con dos nuevos parámetros:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

Los nuevos parámetros introducidos son aprendidos durante el entrenamiento como si fueran unos pesos, es decir, también se aplican las técnicas de optimización de *gradient descent* sobre ellos. De este modo, la red tiene la capacidad teórica de aplicar la función identidad y por tanto «deshacer» la *batch normalization* si así quisiera.

En total, el **proceso de batch normalization** queda resumido en la siguiente imagen:

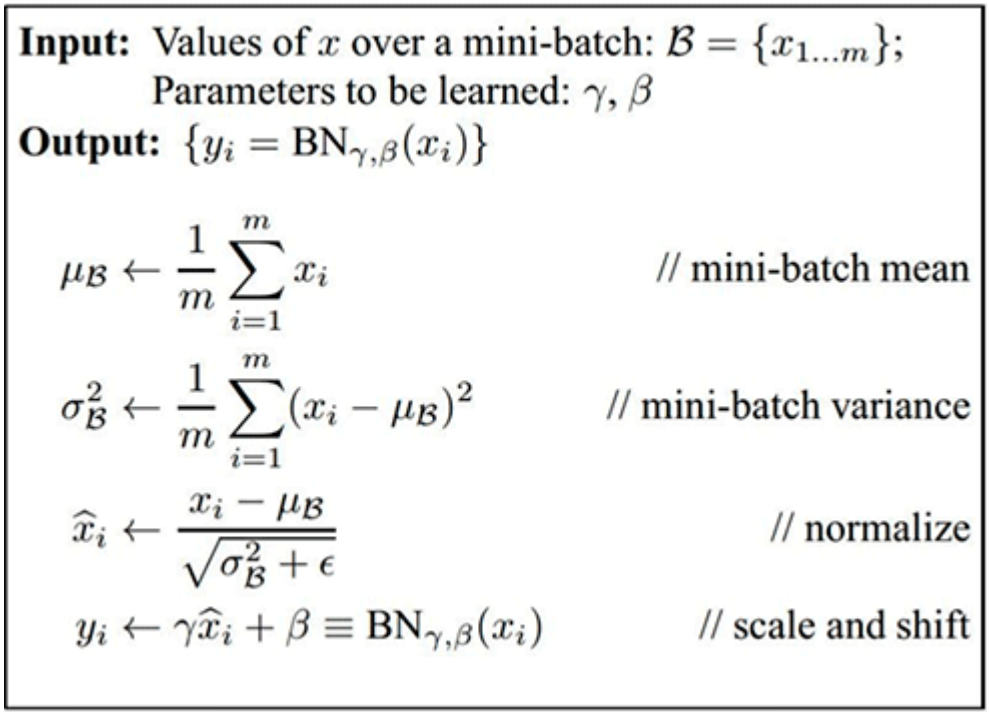


Figura 6. Proceso de *batch normalization*.
Fuente: https://arxiv.org/pdf/1502.03167.pdf

Durante la inferencia (esto es, una vez la red neuronal ya ha sido entrenada y la estamos usando para predecir), las medias y varianzas se fijan con valores totales obtenidos durante el entrenamiento.

Las **ventajas** de la *batch normalization* son las siguientes:

- Mejora el flujo de gradientes por la red durante el aprendizaje, aumentando por tanto la velocidad de convergencia.
- Permite *learning rates* mayores.
- Reduce la dependencia en la inicialización de parámetros que tiene el proceso de entrenamiento.
- Puede interpretarse como una forma de regularización.

Con todas estas, la técnica de *batch normalization* es muy usada en la práctica.

4.5. Optimización avanzada

En temas anteriores, hemos visto cómo el entrenamiento de una red neuronal se fundamenta en solucionar un problema de optimización. Para ello, estudiamos *stochastic gradient descent*, un algoritmo sencillo que consiste en aproximar localmente de manera lineal la función que intentamos minimizar, y realizar pequeños pasos en la dirección de máximo descenso. En esta sección, veremos algunos problemas de este método de optimización y algunas técnicas más avanzadas para entrenar redes neuronales.

Problemas de SGD

Fórmula de *update* de SGD:

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

Si bien *stochastic gradient descent* es un algoritmo que funciona correctamente en general, tiene una serie de problemas que dificultan en ocasiones el entrenamiento de redes neuronales. Por ejemplo, supongamos que tenemos una región donde el

coste o *loss* disminuye rápidamente en una dirección y muy lentamente en otra, como se ve en la siguiente imagen.

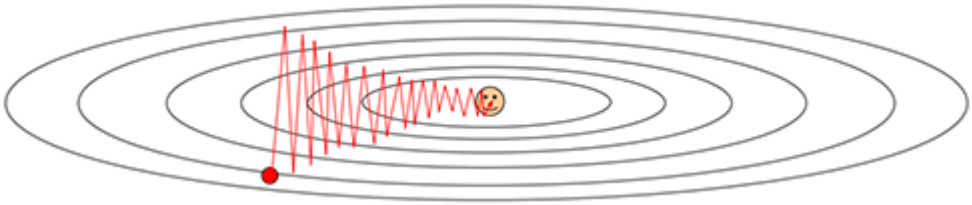


Figura 7. Ejemplo de problema de SGD.
Fuente: http://cs231n.github.io/

En la imagen, el punto rojo es el punto de inicio, mientras que el punto central representa el mínimo a alcanzar. Como vemos, SGD empieza un movimiento en zigzag siguiendo la dirección de máximo cambio, encontrando el mínimo de manera muy ineficiente. El camino óptimo habría sido seguir una línea directa de bajada hacia el punto central. Este problema es aún peor en el caso de una red neuronal real, donde en vez de dos dimensiones como en la imagen, podemos llegar a tener millones de variables, dando lugar a millones de posibles direcciones que el gradiente puede tomar.

Otro gran problema es el de los **mínimos locales**: puntos mínimos de una función que no son el mínimo global de esta (figura 8).

Figura 8. Ejemplo de mínimo local.
Fuente: http://cs231n.github.io/

De manera similar, los **puntos de silla** *(saddle points)* representan el mismo problema. Si bien no tienen la forma de un mínimo, en ellos la derivada también es cero:

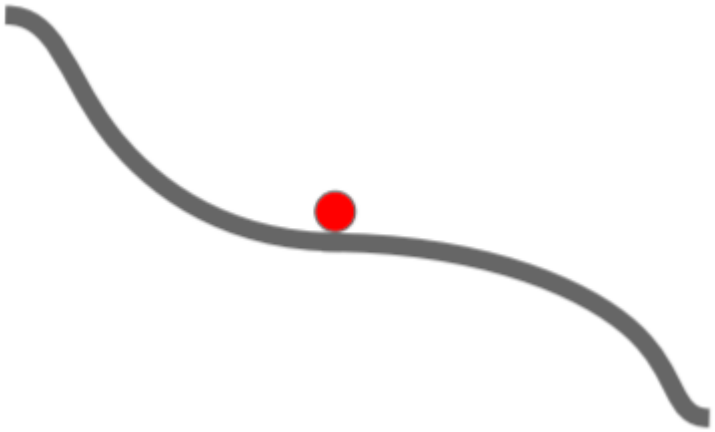


Figura 9. Ejemplo de mínimo local.
Fuente: http://cs231n.github.io/

En el caso particular de las redes neuronales, donde tenemos una gran cantidad de parámetros, los puntos de silla aparecen de manera mucho más frecuente que los mínimos locales.

Es importante mencionar aquí que el entrenamiento de redes neuronales es un **problema de optimización no convexo**, para el cual encontrar la solución óptima no está garantizado; el hecho de que tengamos un gran número de parámetros en la red hace más complejo aún encontrar la solución óptima. Hablamos de funciones con, potencialmente, millones de variables a optimizar, donde es prácticamente imposible de visualizar un proceso de minimización.

Lo que sí podemos imaginar es que encontrar el mínimo global de una función de este estilo es computacionalmente intratable y, en muchas ocasiones, tenemos que conformarnos con que el mínimo encontrado sea lo suficientemente bueno. Hay que resaltar aquí que al inicializar los pesos de la red de manera aleatoria, las soluciones encontradas suelen ser distintas unas a otras. En la práctica por suerte, al entrenar una red desde distintos valores iniciales se tiende a encontrar mínimos parecidos,

aunque este no tiene por qué ser el caso al ser un proceso dependiente del punto inicial, de la función a tratar (la topología de la red junto con sus parámetros) y del algoritmo de optimización en uso.

Es también importante mencionar que, sin embargo, encontrar un mínimo global o un valor demasiado cercano al mínimo global no es lo que queremos siempre. Las redes neuronales tienen una gran capacidad de representación, por lo que podríamos encontrarnos con que pueden explicar perfectamente los datos de entrenamiento, pero no unos datos fuera del *training set*. Esto es lo que se conoce como ***overfitting***, y lo veremos en el siguiente punto del tema.

SGD + Momentum

Una variante de SGD para resolver los problemas que hemos vistos en el apartado anterior consiste en añadir *momentum*. En SGD con *momentum* tenemos un vector de velocidad que es una especie de media ponderada de gradientes anteriores:

$$v_{t+1} = pv_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

Como vemos en las ecuaciones, la regla de *update* del punto x depende ahora también de un nuevo término, la **velocidad**. Por su parte, el parámetro `rho` corresponde a la fricción y su valor suele ser 0.9 o 0.99.

La idea es que el descenso va adquiriendo «velocidad» en las direcciones correctas a través de todos los *updates*, acelerando el proceso de convergencia. SGD con *momentum* soluciona, además, los problemas que hemos visto anteriormente.

En el caso de los mínimos locales y los puntos de silla, si bien el gradiente es 0, al tener el vector velocidad información de los gradientes anteriores podemos «pasarnos» estos puntos conflictivos:

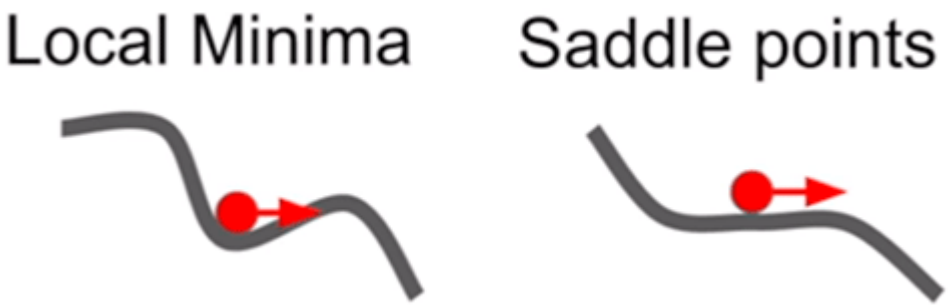


Figura 10. Ejemplo de SGD + Momentum ante mínimo local y punto de silla.
Fuente: http://cs231n.github.io/

De la misma manera, los ineficientes zigzags que veíamos antes se compensan unos a otros gracias al vector de velocidad, permitiendo encontrar una solución al problema más efectiva:

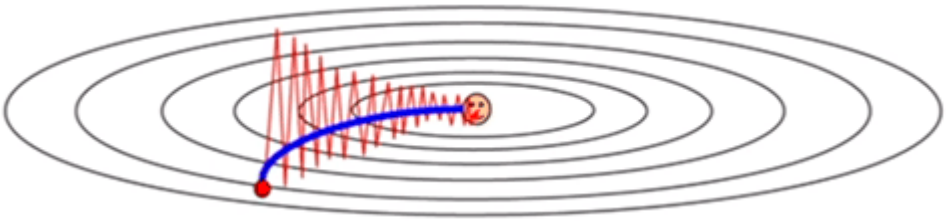


Figura 11. Ejemplo de SGD + Momentum como solución.
Fuente: http://cs231n.github.io/

Una variante más moderna de SGD con momentum es el denominado **Nesterov Momentum**. Esta técnica goza de mejores propiedades de convergencia teóricas para funciones convexas y, aunque el entrenamiento de redes neuronales no optimiza dicho tipo de funciones, en la práctica parece que también se comporta mejor. La idea de Nesterov Momentum es aplicar el gradiente sobre el *update* que hace el vector velocidad, puesto que este va a afectar a la posición igualmente.

Ecuaciones de Nesterov Momentum:

$$v_{t+1} = pv_t - \alpha \nabla f(x_t + pv_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

A continuación, una comparación entre las *update rules* de SGD con Momentum y con Nesterov Momentum:

Figura 12. Nesterov *update* y Nesterov Momentum.
Fuente: http://cs231n.github.io/

AdaGrad

AdaGrad es un método que modifica la *learning rate* de manera independiente para cada parámetro. Encontrar una *learning rate* adecuada es una tarea compleja, por lo que el atractivo de este método, junto con varios de los que veremos a continuación, es que tenemos que preocuparnos menos de ello. Las ecuaciones de AdaGrad pueden simplificarse visualmente en código NumPy de la siguiente manera:

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Figura 13. Ecuaciones AdaGrad con NumPy.
Fuente: http://cs231n.github.io/

Aquí, las variables `x` y `grad_squared` son vectores, por lo que las operaciones se realizan elemento a elemento. La idea de AdaGrad consiste en dividir el valor del *update* de `x` por el cuadrado del gradiente:

- Si el valor del gradiente es grande, estamos haciendo que el avance en esa coordenada sea reducido.
- Si es más pequeño, hacemos que el avance sea amplificado.

En nuestra ya conocida imagen, estamos desacelerando el movimiento en la dirección vertical mientras que lo aceleramos en la horizontal

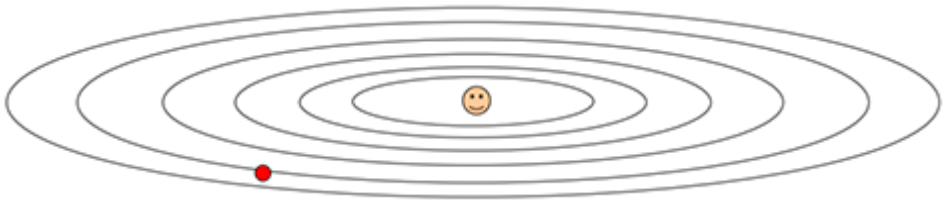


Figura 14. Ejemplo con AdaGrad.
Fuente: http://cs231n.github.io/

Es importante fijarse en que `grad_squared` va sumando valores, por lo que a lo largo del entrenamiento los pasos que demos serán cada vez más pequeños. La idea aquí es que, una vez estemos más cerca del mínimo, necesitamos hilar más fino, mientras que al principio podemos avanzar de manera más rápida en busca de las direcciones adecuadas. Pero esto crea un problema para AdaGrad y es que esta reducción de velocidad llega a ser muy agresiva y podría detener el aprendizaje demasiado pronto.

RMSProp y Adam

RMSProp y Adam son dos **métodos de optimización** que elaboran las ideas de AdaGrad.

RMSProp, en vez de sumar los cuadrados de los gradientes, introduce un `decay`, de manera que los valores no se acumulen y evitando que el entrenamiento se pare en cierto momento:

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Figura 15. Comparación entre AdaGrad y RMSProp.
Fuente: http://cs231n.github.io/

Adam es un optimizador que, utilizado en su forma por defecto, presenta muy buenos resultados. Combina RMSProp con la idea de Momentum vista en SGD.

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Figura 16. Método Adam.
Fuente: http://cs231n.github.io/

Learning rate decay

Todos los algoritmos que hemos visto hasta ahora utilizan *learning rates*, que es probablemente el hiperparámetro más crítico a elegir a la hora de entrenar una red neuronal. En la siguiente imagen podemos ver una gráfica de *loss* por época de entrenamiento. Una buena elección en *learning rate* nos debería llevar a una reducción de la *loss* hasta converger en un valor bajo. Una *learning rate* alta también

lleva a convergencia, pero el hecho de que no sea lo suficientemente pequeña hace que converja en una solución no óptima. Si la *learning rate* es demasiado alta, nos encontramos probablemente dando saltos sin sentido en el espacio de parámetros (*overshooting*) y por tanto, la *loss* crece en vez de disminuir. Finalmente, en el caso de una *learning rate* demasiado baja, la red neuronal se entrena correctamente, pero a un ritmo demasiado lento.

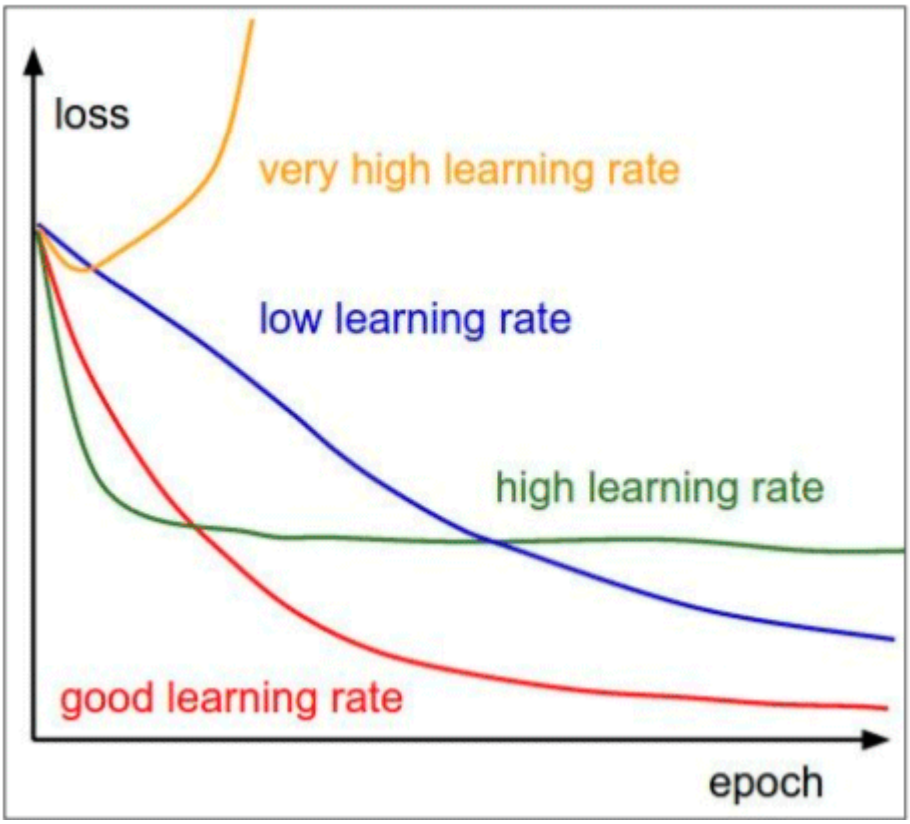


Figura 17. Método Adam.
Fuente: http://cs231n.github.io/

Una idea para tratar de una manera más eficiente la *learning rate* consiste en usar *learning rate decay*, haciendo que el *learning rate* se vaya reduciendo a medida que avanza el entrenamiento. De este modo, el entrenamiento comienza con una *learning rate* mayor, haciendo que la *loss* se reduzca rápido, y poco a poco esta se va reduciendo permitiendo buscar mejor el espacio de soluciones.

Hay varias estrategias posibles para ejecutar el *decay*:

- *Step decay*: reducir, por ejemplo a la mitad, la learning rate cada cierto número de epochs.
- *Exponential decay*: donde la *learning rate* sigue un decrecimiento como el de una función exponencial.

4.6. Regularización

Las redes neuronales son modelos con un gran poder de representación. El gran número de parámetros y capas que podemos añadir a una red neuronal hace que sea fácil que esta aprenda demasiado bien los datos con los que estamos entrenándola. En ocasiones, es hasta posible que la red sea capaz de memorizar perfectamente un conjunto de datos de entrenamiento, llegando a clasificarlos a la perfección (100 % de *accuracy*). Sin embargo, al hacer esto, el modelo pierde capacidad de generalización y, al ser evaluado sobre un conjunto de datos no vistos durante el entrenamiento o test set, veremos una baja capacidad de predicción.

Este fenómeno se conoce como ***overfitting*** y es un fenómeno común en *machine learning*: el algoritmo está modelando demasiado bien los datos de entrenamiento mientras que pierde capacidad de generalización en datos no vistos. Más que aprender las características generales de un dataset, el algoritmo está fijándose y aprendiendo los detalles particulares de los datos con los que es entrenado.

El fenómeno puede apreciarse fácilmente en la siguiente gráfica donde, durante el entrenamiento, se van obteniendo los *prediction error* tanto para los datos de entrenamiento como para un test set que el algoritmo no está viendo. Según avanza el entrenamiento, tanto el *training error* como el *test error* disminuyen. Sin embargo, llega un punto en el que el error sobre los datos de entrenamiento sigue bajando

mientras que el test error empieza a aumentar. En ese momento, se dice que el modelo está *overfitting* y perdiendo capacidad de generalización.

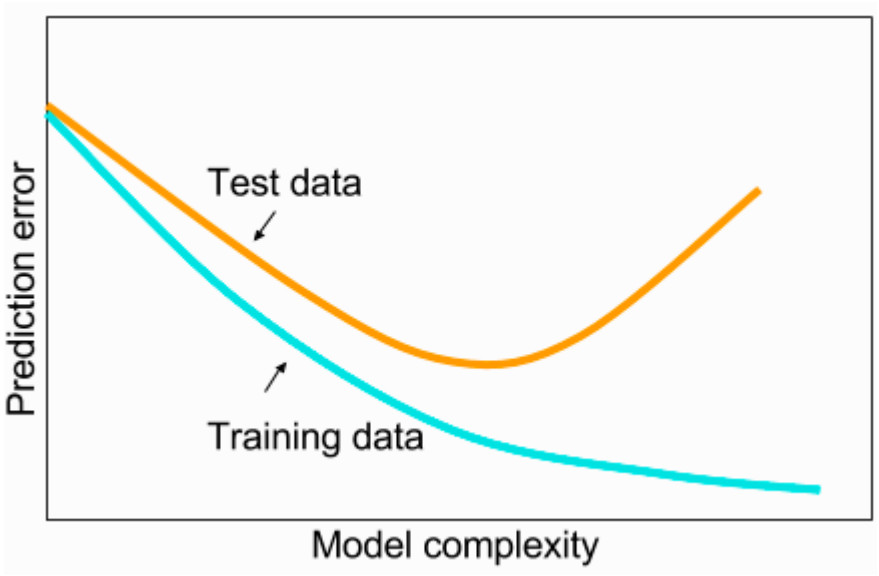


Figura 18. *Overfitting*.
Fuente: http://gluon.mxnet.io/chapter02_supervised-learning/regularization-scratch.html

Las técnicas de **regularización** (*regularization*) intentan solucionar este problema. Su objetivo es conseguir reducir este *gap*. Ahora veremos algunas de las técnicas de regularización más comunes.

Regularización L2

Probablemente la forma más común de aplicar regularización. Consiste en penalizar la norma al cuadrado (de ahí el nombre de L2) de los *weights* en una red neuronal en la función de coste. Dicho de otro modo, por cada parámetro w en la red, añadimos el valor cuadrado a la función de coste multiplicado por un parámetro de regularización lambda. Por ejemplo, usando *cross entropy loss*, la función de coste quedaría así:

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + \left(1 - y_j\right)\ln\left(1 - a_j^L\right)\right] + \frac{\lambda}{2n}\sum_{w} w^2$$

El parámetro de regularización lambda es un hiperparámetro de la red y aumentarlo implica aumentar la regularización.

Al penalizar el cuadrado de los parámetros en la función de coste (recordemos que lo queremos minimizar) estamos favoreciendo que los parámetros tengan valores absolutos pequeños. Durante el aprendizaje, la red solo asignará valores grandes a los parámetros (que son penalizados) si al hacer esto consigue reducir la *loss* en una cantidad mayor que esta penalización.

Intuitivamente, este tipo de regularización está reduciendo el espacio de posibles soluciones. Al forzar al espacio de parámetros a tener un valor pequeño, la red neuronal es menos libre para asignar valores cualesquiera y, por lo tanto, pierde cierta capacidad de expresividad, haciendo que sea más complicado fijarse en los detalles particulares en el *training data* que llevan al *overfitting*.

En la práctica, la regularización L2 es muy utilizada gracias a su buen funcionamiento. No obstante, es importante mencionar que esta regularización solo es aplicada a los pesos w y no a los *biases* b.

Regularización L1

La regularización L1 es muy similar a la L2, pero en vez de sumar los valores al cuadrado de los pesos, suma los valores absolutos.

$$C = C_0 + \frac{\lambda}{n}\sum_{w}|w|$$

La idea es similar a la regularización L2. Sin embargo, la regularización L1 tiene la interesante capacidad de hacer que los parámetros sean *sparse* (dispersos): muchos parámetros se quedan en 0 o valores muy cercanos a 0. De este modo, las neuronas acaban utilizando solo una serie de sus inputs, siendo por tanto resistentes al ruido.

En comparación, la regularización L2 tiende a asignar valores pequeños a todos los pesos.

Si bien la regularización L1 es muy común en tareas de *machine learning*, en el caso de redes neuronales la regularización L2 tiende a dar mejores resultados y es más utilizada en la práctica.

Max norm *regularization*

Otra idea utilizada a menudo en *deep learning* es la de constreñir los valores de los parámetros a tener una norma vectorial menor de cierto tamaño. Por ejemplo, forzar a los vectores de parámetros a tener norma 1 o menor que 1. Esta es otra forma de reducir la libertad del modelo para encontrar posibles soluciones, actuando así como fenómeno regularizador.

Dropout

Dropout es una técnica bastante moderna y que ha encontrado una gran acogida, muy utilizada en la práctica. La idea consiste en aplicar una salida 0 a un porcentaje de neuronas de la red durante el entrenamiento de manera aleatoria. De este modo, en cada *batch*, un número aleatorio de neuronas se desactivará. La probabilidad p de que una neurona se desactive es otro hiperparámetro de la red y es común utilizar valores de 0.25 o 0.5. En este último caso, significa que durante el entrenamiento la mitad de las neuronas se van desactivando en cada *batch* (nótese que las neuronas inactivas cambian en cada *batch*).

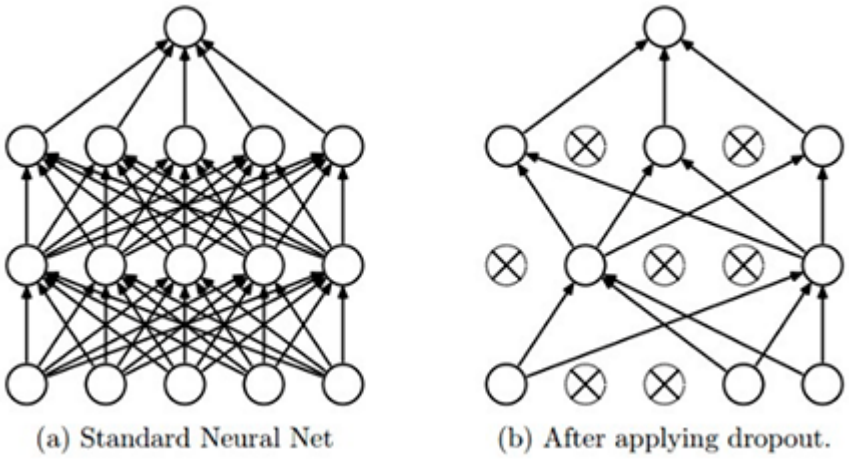


Figura 19. Efecto de aplicar *dropout*.
Fuente: http://cs231n.github.io/

Dropout impide la coadaptación de las *features*. Al forzar que ciertas neuronas tengan salida 0, la red tiene que aprender nuevas formas de correlacionar las *features*. Sin *dropout*, es posible que la red memorice la presencia continua de ciertas neuronas activadas para cierto valor de salida. Con *dropout*, la red es forzada a aprender nuevas relaciones entre neuronas, ya que muchas de las neuronas han quedado apagadas. Esto tiene un efecto regularizador, ya que se le impide a la red memorizar resultados.

Otra interpretación de *dropout* es la de que actúa como un *ensemble* de modelos. Durante el entrenamiento, estamos obteniendo en cada iteración una «nueva» red neuronal aleatoria según las neuronas que no quedan desactivadas. Según esta interpretación, estaríamos entrenando a la vez un gran número de redes neuronales ligeramente distintas.

Durante la inferencia o *test time* (cuando no estamos entrenando), *dropout* no se aplica y todas las neuronas están activadas.

Early stopping

Finalmente, otra técnica muy utilizada en la práctica es la de *early stopping*. Esta técnica consiste en utilizar un *validation set* de datos que son evaluados durante el entrenamiento, pero que no son utilizados para entrenar. De este modo, podemos

ver cuándo la red está empezando a *overfit*. Así, podemos parar el entrenamiento en este momento u obtener una idea de cuántas *epochs* hace falta entrenar el modelo en entrenamientos futuros.

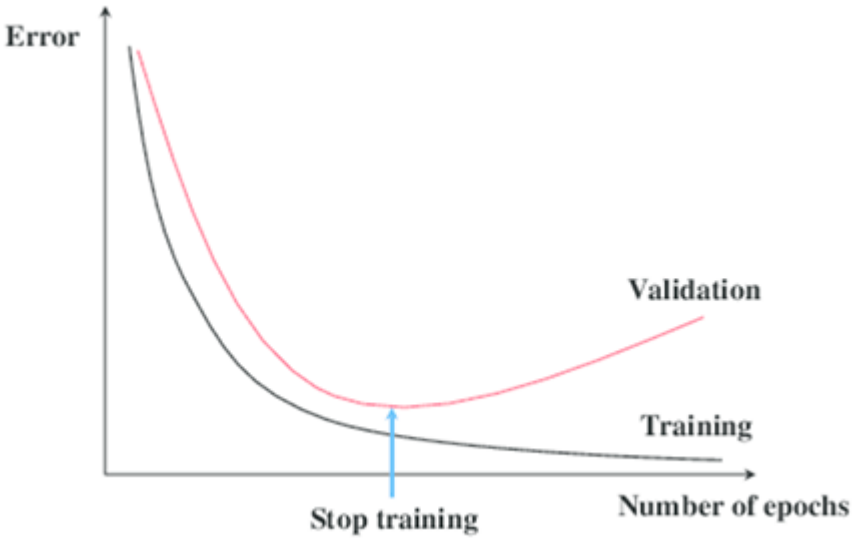


Figura 20. Efecto de aplicar *dropout*.
Fuente: https://www.researchgate.net/figure/Early-stopping-method_fig3_283697186

4.7. Referencias bibliográficas

He, K., Zhang, X., Ren, S. y Sun, J. (2015). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. [Lugar desconocido]: Microsoft Research. Recuperado de https://arxiv.org/pdf/1502.01852.pdf

Krizhevsky, A., Sutskever, I. y Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems*, *25*(2). DOI: 10.1145/3065386. Recuperado de https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Lo + recomendado

Lecciones magistrales

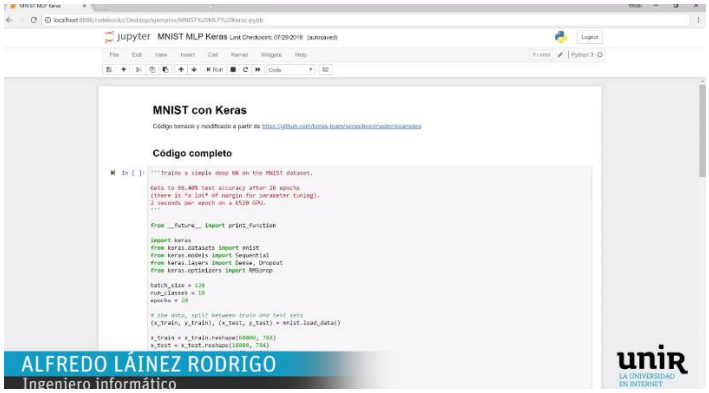
Aspectos prácticos del entrenamiento I

En esta magistral veremos aspectos prácticos del entrenamiento de redes neuronales, por ejemplo, cómo dividir los datos a la hora de entrenar un modelo o la monitorización de la función de pérdida (*loss function*) para saber si nuestro modelo está funcionando.

Accede a la lección magistral a través del aula virtual

Entrenamiento de una red neuronal con Keras

En esta magistral veremos aspectos prácticos del entrenamiento de redes neuronales, por ejemplo, cómo dividir los datos a la hora de entrenar un modelo o la monitorización de la función de pérdida (*loss function*) para saber si nuestro modelo está funcionando.



Accede a la lección magistral a través del aula virtual

Aspectos prácticos del entrenamiento II

En esta magistral se hablarán de una serie de buenas prácticas para lograr una configuración exitosa en el entrenamiento de redes neuronales.

Accede a la lección magistral a través del aula virtual

No dejes de leer

Why momentum really works

Goh, G. (Abril, 2017). Why momentum really Works. *Distill*. DOI: 10.23915/distill.00006.

Este artículo explica en profundidad el funcionamiento del momentum. Si bien no es necesario leérselo entero, al inicio hay una visualización con la que podemos jugar con distintos valores de momentum y *learning rate* para comprender su funcionamiento.

Accede al artículo a través del aula virtual o desde la siguiente dirección web:
https://distill.pub/2017/momentum/

+ Información

Webgrafía

Stanford CS231n Course Notes

Estos apuntes cubren en gran detalle gran parte de los contenidos que hemos visto hasta ahora.

Accede a la página web a través del aula virtual o desde la siguiente dirección:

http://cs231n.github.io/

Test

1. Marca todas las respuestas correctas acerca de la inicialización de parámetros en una red neuronal*:*

A. La inicialización de *biases* no es muy crítica una vez se ha definido una correcta inicialización de *weights*.

B. La inicialización de parámetros no es importante, cualquier conjunto de valores iniciales permitirá a la red converger correctamente.

C. No es buena idea inicializar todos los pesos con el mismo valor.

D. No es buena idea inicializar todos los *biases* con el mismo valor.

2. Si inicializamos todos los pesos *w* con valores muy grandes en valor absoluto (marca la respuesta correcta):

A. El uso de unidades *sigmoid* y tanh no sería una buena idea, ya que se encontrarán casi todo el tiempo en régimen de saturación.

B. En general no es un problema y es normal hacer esto al entrenar redes neuronales.

C. El entrenamiento de la red irá más rápido ya que los gradientes tendrán un valor mayor.

3. *Batch normalization* (marca todas las respuestas correctas):

A. Es una técnica que mejora el entrenamiento de redes neuronales.

B. Mejora los problemas que la inicialización de parámetros crea sobre las redes neuronales profundas.

C. Obtiene medias y varianzas a nivel de *batch*.

D. Funciona de manera distinta en *training time* (entrenamiento) que en *test time* (inferencia), ya que en el segundo podríamos querer evaluar *inputs* independientes sin formar parte de una *batch*.

4. Marca todas las respuestas correctas sobre SGD + Momentum:

A. SGD + Momentum es diferente a SGD porque no actúa por *batches*, sino utilizando todos los datos de entrenamiento por cada *update*.

B. El vector velocidad con Momentum ayuda a escapar de mínimos locales y puntos de silla.

C. El vector velocidad contiene información sobre los gradientes de todos los pasos anteriores.

D. `Rho = 0` en el vector de velocidad hace que SGD + Momentum sea idéntico a SGD.

5. En AdaGrad, los gradientes al cuadrado de los parámetros (marca la respuesta correcta):

A. Se calculan de nuevo en cada paso y, una vez usados, son desechados.

B. Han de ser guardados, ya que el valor en cada iteración depende del valor anterior. Por tanto, el número de elementos a guardar se duplica (por un lado parámetros, por otro lado suma de gradientes).

C. No han de ser guardados, se recalcula toda la suma desde el principio en cada iteración.

6. Marca todas las respuestas correctas acerca de *learning rate*:

A. Una *learning rate* demasiado grande hace que el coste o *loss* diverja.

B. En *learning rate decay*, la *learning rate* va aumentando poco a poco para agilizar el entrenamiento.

C. AdaGrad, RMSProp y Adam utilizan *learning rate* como hiperparámetro.

D. Es más seguro utilizar un valor pequeño para asegurar la convergencia. Sin embargo, esto podría hacer que el entrenamiento sea demasiado lento.

7. ¿Cuál de las siguientes es una razón por la que las unidades ReLU tienen menos tendencia a «morir» cuando se aplican *learning rates* pequeñas?

A. Los *learning rates* grandes suelen estar asociados con valores de *input* negativos. En el caso de la ReLU, esto nos lleva a caer en el régimen negativo con valor de salida 0.

B. El régimen positivo de la unidad ReLU se caracteriza por tener derivada 1. Esto hace que los gradientes «backpropagados» se magnifiquen con valores grandes de *learning rate*, llevando a la unidad a la «muerte».

C. Un valor de *learning rate* grande lleva a cambios mayores en los parámetros de la red. Esto implica que los pesos pueden cambiar mucho durante SGD e impedir que la unidad ReLU pueda volver a salir de su régimen negativo.

8. Marca todas la respuesta correcta acerca del *overfitting*:

A. Es un problema por el cual los algoritmos de *machine learning* no se comportan correctamente en datos no vistos durante el entrenamiento.

B. Es un problema por el que los algoritmos de *machine learning* pierden capacidad de generalización: no son capaces de generalizar a datos no vistos durante el entrenamiento.

C. Puede ser combatido utilizando varias técnicas a la vez, tales como *dropout* y regularización L2.

D. No es común en redes neuronales profundas.

9. ¿Qué ocurre con los gradientes en las neuronas desactivadas por *dropout*? (Marca la respuesta correcta):

A. El proceso de *backpropagation* es el mismo, anular la salida no implica cambios en la propagación del gradiente.

B. Los gradientes transmitidos hacia atrás multiplican por p la probabilidad de que la neurona se desactive.

C. Los gradientes transmitidos hacia atrás son 0 en estas neuronas, ya que la función de salida de la neurona es constante y vale 0.

10. *Early Stopping* (marca las respuestas correctas):

A. No impone explícitamente restricciones a la capacidad de representación del modelo. Simplemente marca un punto donde se deja de entrenar.

B. Puede utilizarse en conjunción con otras técnicas.

C. Garantiza que el *test error* es menor o igual que el *training error*.