Sistemas Cognitivos Artificiales

Roberto Casado Vara

Tema 4: Aspectos prácticos en el entrenamiento de redes neuronales profundas

Recordad…

- Repasad la teoría, no lo dejéis para el último día.
- Haced los test de cada tema.
- Actividad → 10 de mayo
- Apuntarse en los grupos, 26 de abril último día.
- *Trabajo en grupo* → 24 mayo

En este tema...

Semana 5:

- Unidades de activación
- Inicialización de parámetros
- *Batch normalization*

Semana 6:

- Optimización avanzada
- Regularización

Entrenamiento de una red neuronal

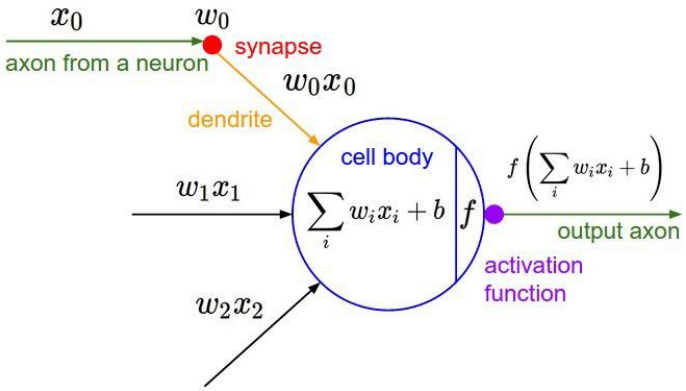
- Como sabemos, el entrenamiento de una red neuronal es un **problema complejo de optimización** con un gran número de parámetros, para el que obtener una solución adecuada no es sencillo.
- Las redes dependen de un **gran número de parámetros y de hiperparámetros**. Es fácil cometer errores o elegir estrategias erróneas que dificultan el entrenamiento de las redes (una de las causas por las que el deep learning tardó en desarrollarse).
- En este tema veremos aspectos prácticos y avanzados para conseguir un entrenamiento satisfactorio y eficiente de redes neuronales.

Roberto Casado Vara

Tema 4.1: Unidades de activación

Unidades de activación

- Uno de los elementos más críticos en una red neuronal es la elección de **unidad de activación** o ***non-linearity***.
- Hasta ahora, hemos visto la utilización de la unidad más clásica, la unidad **sigmoid**.
- La irrupción de nuevas y más efectivas unidades de activación ha sido uno de los grandes avances en el mundo del *deep learning*.

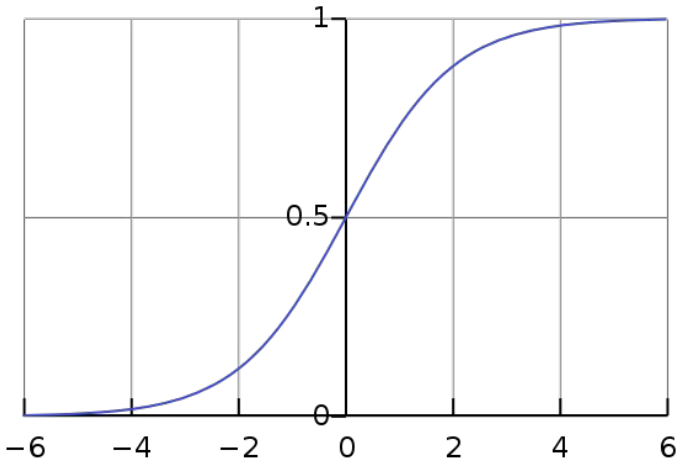


Fuente: http://cs231n.github.io/neural-networks-1/

Unidad sigmoid

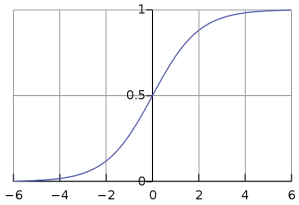
- Como ya sabemos, la función sigmoid se define de la siguiente manera:

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



- La unidad convierte cualquier número real en un número entre 0 y 1, tendiendo a 1 en el cuadrante positivo y a 0 en el cuadrante negativo.

Problemas de sigmoid

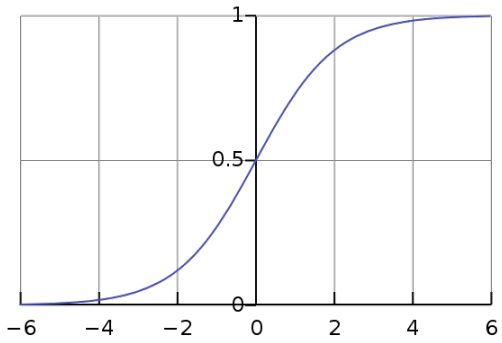


- La unidad sigmoid ha caído en desuso en las arquitecturas modernas debido a una serie de problemas:

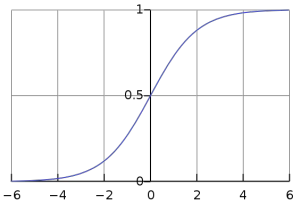
1. Las unidades sigmoid saturadas “matan” los gradientes.

- Decimos que la unidad se satura cuando sus valores se aproximan a 1 o a 0. El régimen de saturación se caracteriza porque pequeños cambios en x no implican apenas variación en y.

- Visto de otra manera, cuando la unidad está saturada, la derivada de la función es casi 0.

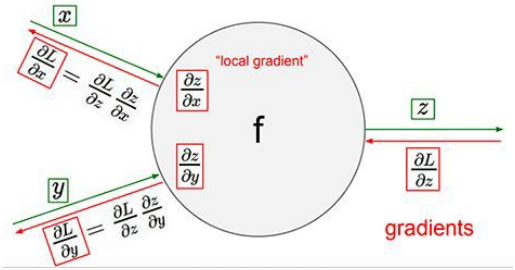


Problemas de sigmoid



1. Las unidades sigmoid saturadas “matan” los gradientes.

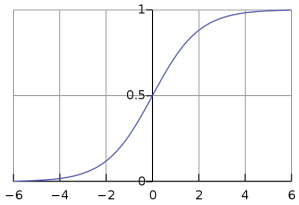
- Esto implica que la derivada local es aproximadamente 0. Recordemos de backpropagation que la derivada local se multiplica por el gradiente que fluye desde arriba, y el resultado se propaga hacia atrás:



- Por tanto, en régimen de saturación, el gradiente resultante es aproximadamente 0, por lo que se dice que la unidad está “matando” el gradiente.

- Al “matar” el gradiente eliminamos la señal de aprendizaje y por tanto estamos dificultando el entrenamiento.

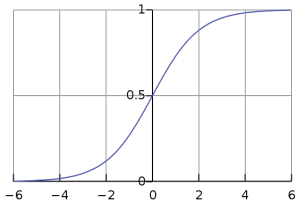
Problemas de sigmoid



2. La salida de sigmoid no está centrada en 0.

- Los valores de salida de *sigmoid* son **estrictamente positivos**.
- Si una neurona recibe todos sus inputs con valores positivos, las derivadas respecto de los pesos *w* tendrán siempre el mismo signo.
- Esto provoca ciertas ineficiencias en el entrenamiento.
- Las salidas positivas de sigmoid provocan este fenómeno en las siguientes capas.

Problemas de sigmoid

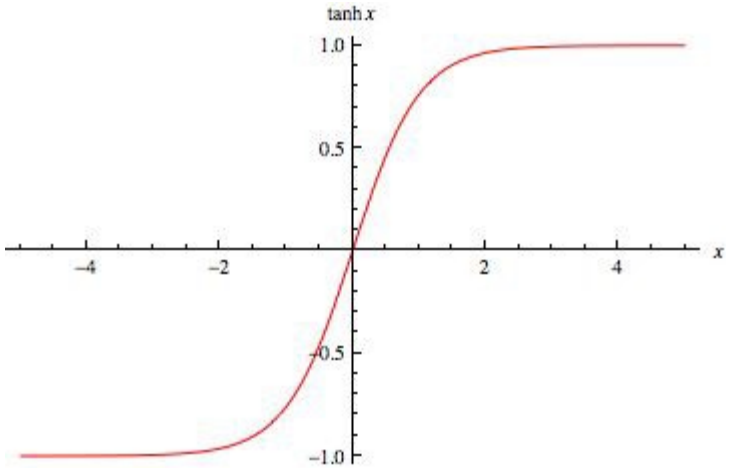


3. Implica el cálculo de una función exponencial (e^x)

- Otro pequeño inconveniente, ya que el cálculo de una función exponencial conlleva cierta complejidad en comparación con otras operaciones.

tanh (tangente hiperbólica)

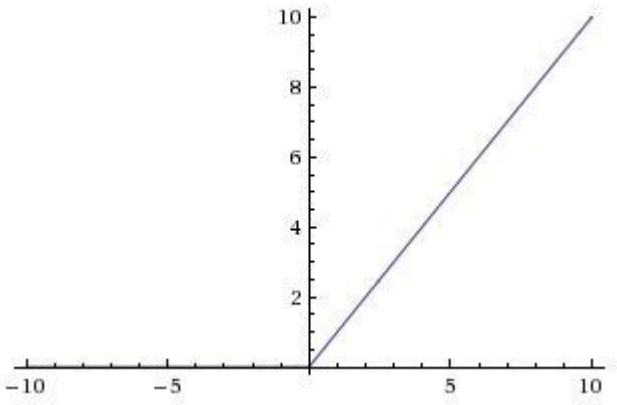
$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



- Similar a sigmoid, pero ahora con valores entre -1 y 1.
- Sigue “matando” gradientes, pero la salida está centrada en 0.
- En la práctica, tanh es preferible a sigmoid.

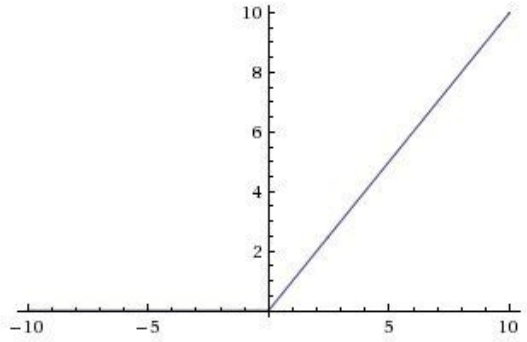
ReLU (rectified linear unit)

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



- Salida 0 para x < 0. Función identidad para x > 0.
- Es la unidad más usada en la práctica.

Ventajas de las ReLU

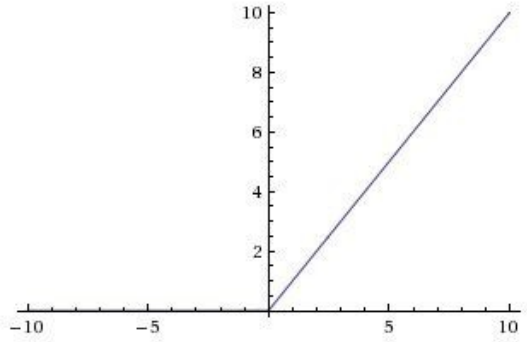


1. **No satura en el régimen positivo (x > 0)**. La unidad no mata gradientes en este régimen.

2. **Computacionalmente muy eficiente**. La computación es extremadamente sencilla. La salida es y(x) = x salvo que x < 0, en cuyo caso es 0.

3. **Acelera la convergencia de SGD**. Diversos estudios confirman que el entrenamiento converge mucho más rápido si usamos ReLU en comparación con sigmoid o tanh.

Problemas de las ReLU



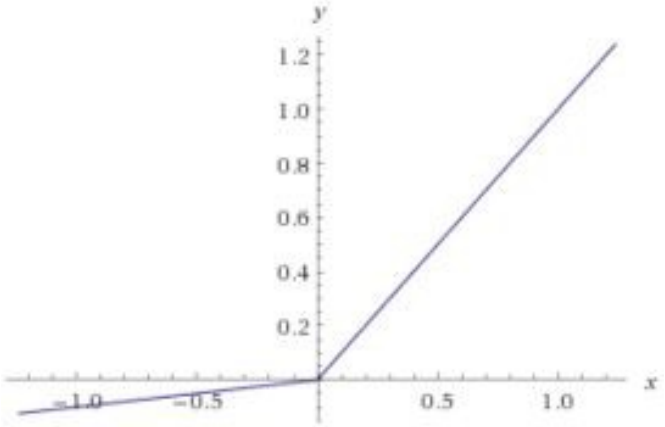
1. **La salida no está centrada en 0**, al igual que pasaba en sigmoid.

2. **Las ReLUs son frágiles y pueden "morir" durante el entrenamiento**.

   - Un cambio grande puede hacer que la unidad no vuelva a activarse nunca, esto es, que nunca vuelva a producir valores mayores que 0.
   - Esto provoca que el gradiente que pasa por la neurona sea siempre 0, por lo que la neurona se considera "muerta".
   - Es normal ver redes neuronales con ReLUs muertas.
   - Esto no suele ser un problema, pero conviene monitorizar las activaciones por si hay demasiadas unidades muertas.
   - La "muerte" de las ReLUs se puede aliviar con *learning rates* pequeños.

Leaky ReLU

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



- Intenta solucionar el problema de las ReLUs “muertas” añadiendo una pequeña pendiente en el régimen negativo, por lo que el gradiente no es 0 en este régimen.

- Otras variantes, añadiendo un nuevo parámetro a la red, que es aprendido:

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

PReLU

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

ELU

Maxout

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

- Otra unidad que intenta solucionar el problema de las ReLUs "muertas". Es una generalización de las ReLU y Leaky ReLU.
- La unidad tiene dos sets de parámetros *w* y *b* en vez de uno.
- Si la red aprende $w_1 = 0$ y $b_1 = 0$, tenemos ReLU.
- Es una unidad más versátil pero más compleja. Requiere el doble de parámetros.

En la práctica

- En la actualidad lo mejor suele ser **utilizar ReLUs**, teniendo cuidado con el *learning rate* y monitorizando el entrenamiento para que no haya demasiadas unidades “muertas”.
- Se puede probar variantes como Leaky ReLUs o Maxout.
- Se recomienda evitar el uso de sigmoid por la gran cantidad de problemas que presenta.

Roberto Casado Vara

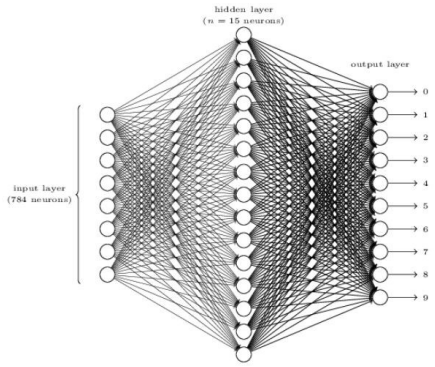
Tema 4.2: Inicialización de parámetros

Inicialización de parámetros

- Hemos visto que las redes neuronales tienen un gran número de parámetros *w* y *b* que aprender.
- La correcta inicialización de éstos es un aspecto crítico para un correcto entrenamiento.

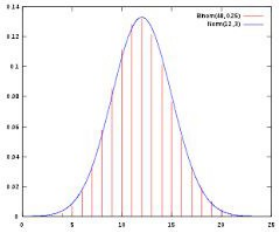
Inicialización a 0

- La inicialización de todos los parámetros a 0 es un error a evitar.
- El mismo valor en todos los parámetros hace que todas las neuronas tengan el mismo efecto en la entrada, lo cual provoca que el gradiente respecto a todos los pesos sea el mismo y, por tanto, los parámetros cambien de igual manera siempre.
- La red nunca aprenderá nada. **Es necesario romper la simetría** de los parámetros para evitar este fenómeno.



Inicialización aleatoria

- Es necesario romper la “simetría de parámetros” de la red.
- Una opción sencilla es **inicializar los pesos de manera aleatoria** utilizando una distribución normal centrada en 0 con un pequeño valor de desviación típica, por ejemplo 0.01.



- Es **importante que los números sean pequeños** (de aquí la desviación típica de 0.01) para evitar que las funciones de activación se saturen frecuentemente.
- Esto aún puede crear ciertos problemas: en redes muy profundas, tener pesos pequeños puede acabar haciendo que los valores de salida se vayan aproximando a 0 y que los gradientes mueran.

Xavier y He initialization

- Como vemos, hasta los más mínimos detalles pueden afectar negativamente al entrenamiento de una red neuronal. La inicialización de parámetros es un campo de estudio muy activo, donde nuevas estrategias surgen continuamente.

- En la actualidad, es común utilizar estrategias conocidas como **Xavier (Glorot) initialization** y **He initialization.**

- La idea 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.

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

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

He Initialization: w = np.random.randn(n) * sqrt(2/n)

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

Inicialización de los biases

- La inicialización de los biases *b* no es un elemento crítico a la hora de entrenar una red neuronal.
- Una vez los pesos *w* son inicializados aleatoriamente, la simetría de la red se rompe.
- Es común inicializarlos a 0 (default en Keras).

Roberto Casado Vara

Tema 4.3: Batch normalization

Batch normalization

- En los puntos anteriores, hemos visto cómo **la distribución de valores de entrada y salida** en las capas de una red neuronal tiene una gran repercusión durante el entrenamiento.
- Esto es similar al **preprocesamiento** de los datos de entrada en la input layer para un correcto entrenamiento.
- ¿Por qué no “preprocesar” los datos en cada capa?
- **Batch normalization** consiste en forzar que la entrada de las unidades de activación en todas las capas siga una distribución normal estándar.
- Esto evita muchos de los problemas creados por las unidades de activación y la incorrecta inicialización de parámetros, acelerando el entrenamiento de las redes neuronales.

Batch normalization

- Batch normalization se aplica, como su propio nombre indica, por **cada training batch**. Por cada batch, se calcula la media y la varianza de los inputs de las funciones de activación. Con esto, se calculan los inputs normalizados mediante la siguiente fórmula:

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

- La operación de batch normalization se ejecuta **antes de la aplicación de la función de activación**, esto es, justo después de hacer la operación lineal *Wx* + *b*, donde x es el input de la capa.

1. Llega x de entrada a la capa (salida de la capa anterior)
2. Aplicamos Wx + b
3. Aplicamos Batch normalization: BN(Wx + b)
4. Aplicamos la función de activación: f(BN(Wx + b))

Batch normalization

- Esta normalización podría reducir el poder de expresión de una red neuronal. Por tanto, es habitual reemplazar los valores ya normalizados por una nueva parametrización con dos nuevos parámetros.

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

- La idea es que la red tenga la capacidad teórica de “deshacer” los efectos de *batch normalization* si así quisiera.

Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Ventajas de batch normalization

1. **Mejora el flujo de gradientes** de la red durante el aprendizaje, aumentando por tanto la velocidad de convergencia.

2. **Permite *learning rates* mayores**.

3. **Reduce la dependencia del entrenamiento en la inicialización de parámetros**.

4. **Tiene un pequeño efecto regularizador.**

    - De manera similar a *dropout*, estamos añadiendo ruido a la red al calcular medias y varianzas por cada batch (las medias y las varianzas van cambiando por cada batch)

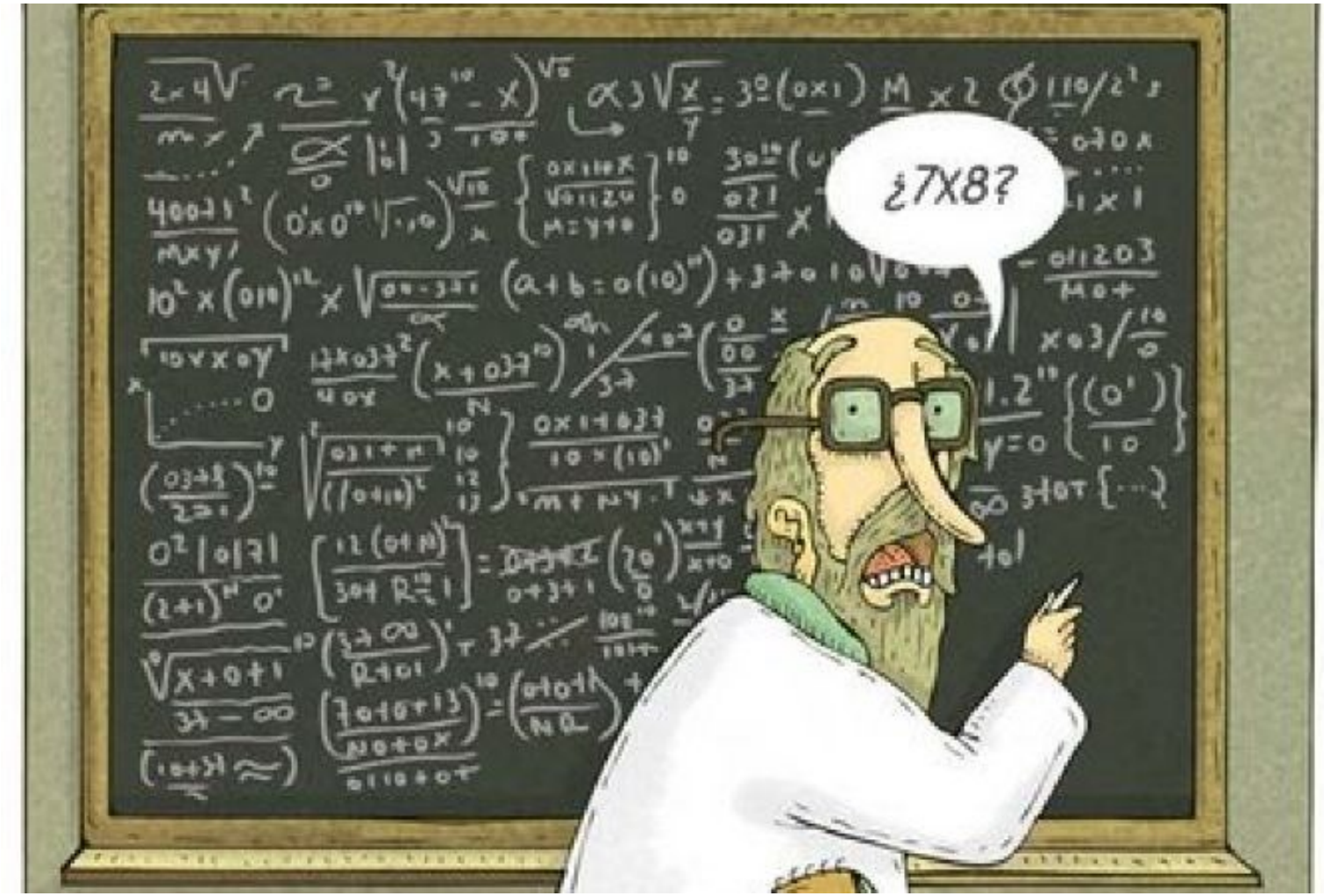
- Es muy frecuente utilizar *batch normalization*.

- Thread de Twitter (avanzado) sobre por qué funciona tan bien.

Batch normalization durante inferencia

- Una vez tenemos el modelo entrenado y estamos aplicándolo (también conocido como inferencia o *test time* vs *training time*), es muy posible que no hagamos predicciones sobre batches.
    - Podríamos tener inputs individuales que queremos evaluar.
- ¿Cómo obtener la media y la varianza que necesitamos para calcular *batch normalization* sobre un solo input?
- Normalmente, se guardan estadísticas durante el entrenamiento y luego se utilizan una vez el modelo ya está entrenado.

¿Dudas?



Roberto Casado Vara

Tema 4.4: Optimización avanzada

Recordatorio

- Como sabemos, el entrenamiento de una red neuronal consiste en la solución de un problema de optimización.
- Recordatorio: **Stochastic Gradient Descent**

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

- En nuestro caso, para una red neuronal con una batch de *m* elementos:

$$w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}$$

Recordatorio

- El entrenamiento ocurre *batch* a *batch* hasta completar todos los elementos del dataset. Cuando hemos utilizado todos los valores del dataset en batches, decimos que hemos entrenado una **epoch**.

- **Proceso**:

```
for epoch=1 to num_epochs:
    mientras queden training examples por ver en la epoch:
        1.  elegir una batch de m elementos no utilizados en la epoch
        2.  calcular gradientes de los parámetros y aplicar SGD
```

$$w'_k = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

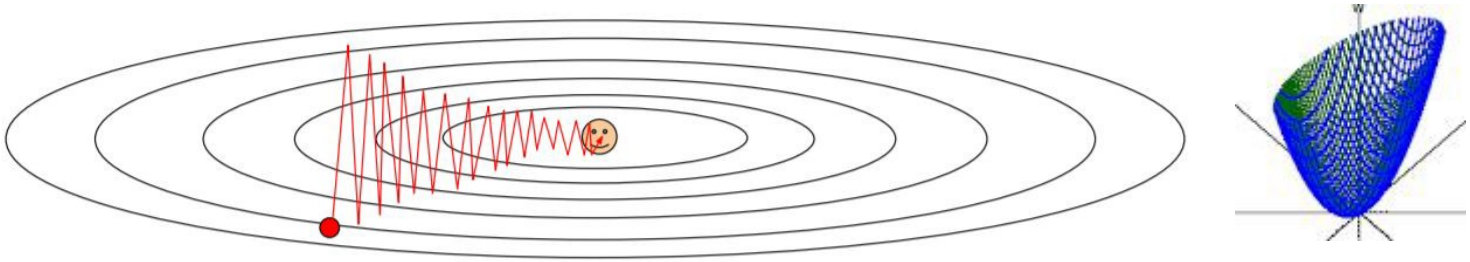
$$b'_l = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l}$$

Problemas de SGD

- SGD tiene en ocasiones algunos problemas que dificultan en ocasiones el entrenamiento de redes neuronales.

1. Zig-zags ineficientes

- Ocurren cuando la *loss* disminuye rápidamente en una dirección y lentamente en otra.



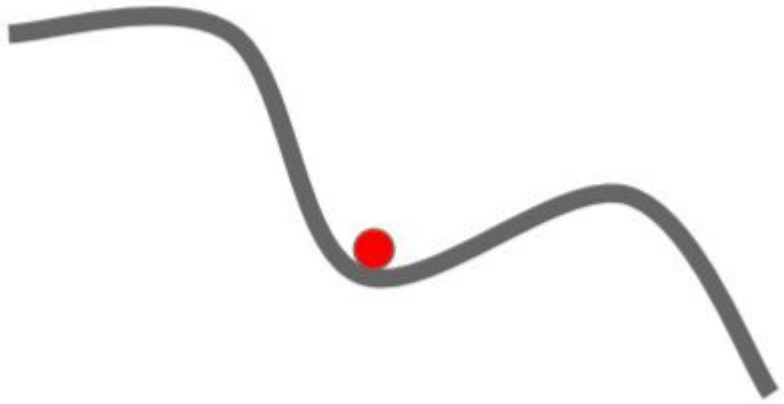
Fuente: http://cs231n.github.io/

- Mucho peor en una red neuronal, donde tenemos miles o millones de posibles direcciones.

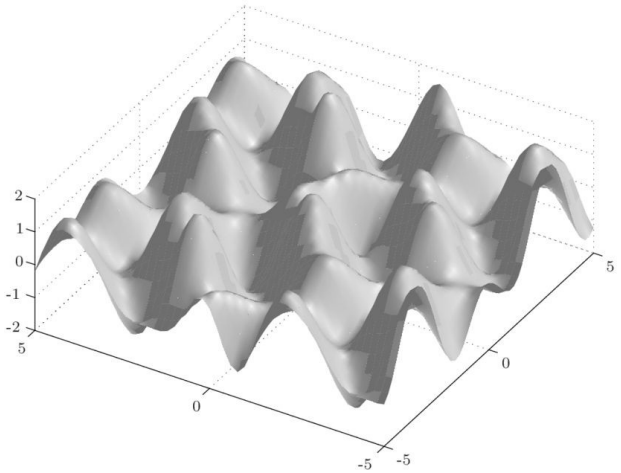
Problemas de SGD

2. Mínimos locales

- La derivada es 0 en ellos y podemos quedarnos atrapados.
- Podría haber soluciones mejores.



Fuente: http://cs231n.github.io/

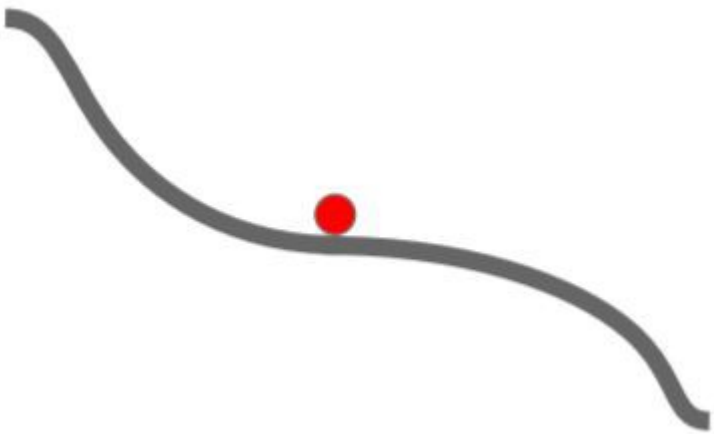


Fuente: Research Gate

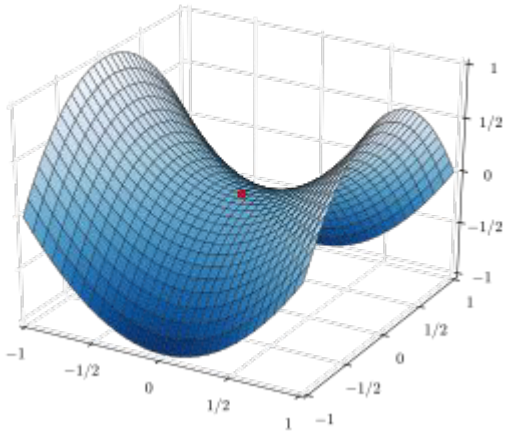
Problemas de SGD

3. Puntos de silla

- La derivada es también 0, podemos quedarnos atrapados.
- Muy comunes en redes neuronales.



Fuente: http://cs231n.github.io/



Fuente: Wikipedia

Apuntes sobre optimización

- Es conveniente recordar lo complejo que es el proceso de minimización de una función con miles o millones de parámetros, así como su “visualización”.
- Como empezamos desde parámetros aleatorios, las soluciones o modelos encontrados al finalizar el entrenamiento serán distintos.
- En la práctica, por suerte, el entrenamiento de redes neuronales suele converger a soluciones similares en cuanto a rendimiento.
- Encontrar el mínimo global para una red neuronal es computacionalmente intratable. Tenemos que conformarnos con que el mínimo encontrado sea lo suficientemente bueno.
- Por suerte, normalmente no queremos encontrar el mínimo global. Un mínimo global podría corresponderse con **overfitting**.

SGD con Momentum

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

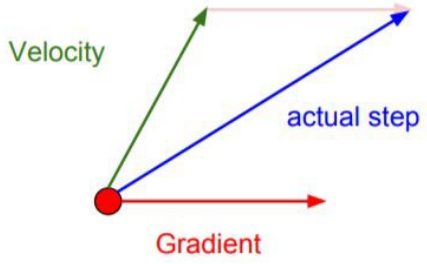
SGD clásico

- Variante muy popular de SGD para resolver los problemas que hemos visto en el apartado anterior.
- Se añade un vector de velocidad que produce un "momentum" en la minimización.

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

- ρ se corresponde con la *fricción*. Su valor suele estar alrededor de 0.9.
- La idea es que el descenso va adquiriendo "velocidad" o cierta inercia en las direcciones correctas de bajada a través de todos los updates, acelerando el proceso de convergencia.



SGD con Momentum

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

SGD clásico

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

- La velocidad acumulada nos ayuda a pasar los puntos con gradiente 0, por lo que solucionamos el problema de los mínimos locales y los puntos de silla.

Fuente: http://cs231n.github.io/

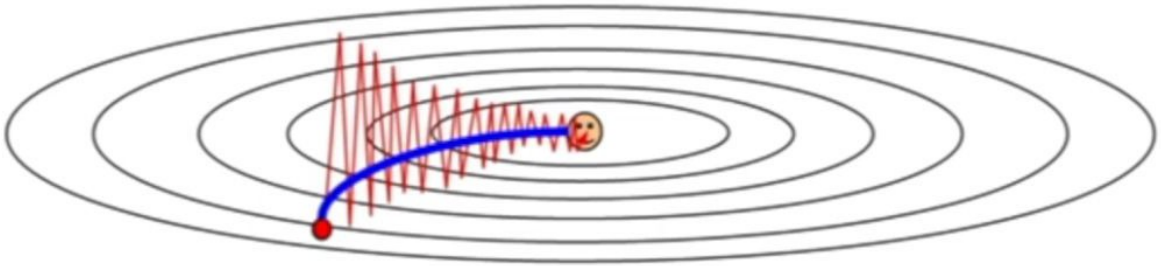
SGD con Momentum

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

SGD clásico

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

- El problema de los “zig-zags” también se soluciona. Estos se compensan con el vector velocidad anterior. Además ganamos velocidad “horizontal”.



Fuente: http://cs231n.github.io/

Nesterov Momentum

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

SGD clásico

- Otra estrategia para aplicar momentum.

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

SGD con momentum

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

Fuente: http://cs231n.github.io/

Métodos de *adaptive learning rate*

- Los métodos de ***adaptive learning rate*** intentan modificar la learning rate para cada parámetro. La idea es que parámetros que se actualizan más frecuentemente tengan cambios más pequeños, mientras que los parámetros con menos actualizaciones de gradientes vean "acelerada" su learning rate.

- **Adagrad**: divide la learning rate en cada parámetro por una suma acumulando los cuadrados de los gradientes. De este modo, parámetros con valores grandes de gradiente van obteniendo una learning rate menor.

- Implica tener que guardar, por cada parámetro, una suma de gradientes al cuadrado.

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

Métodos de *Adaptive learning rate*

- Otras variantes son **RMSProp** y **Adam**.

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)
```

Adam

```
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

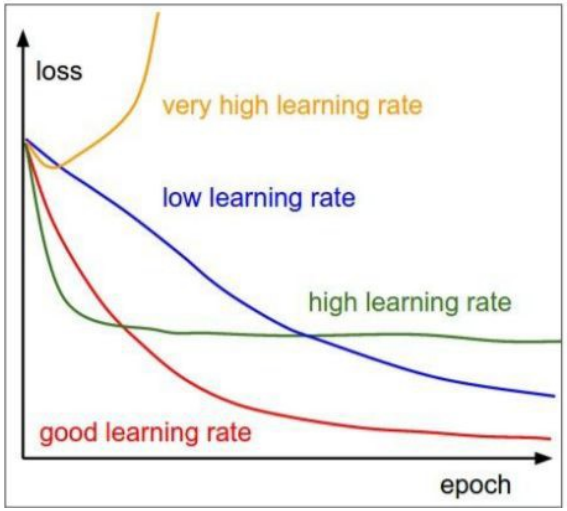
Bias correction

AdaGrad / RMSProp

Fuente: http://cs231n.github.io/

Learning rate decay

- Todos los algoritmos que hemos visto utilizan learning rate, que es probablemente el hiperparámetro más crítico a elegir.



Fuente: http://cs231n.github.io/

- Una idea para entrenar de manera más eficiente es ***learning rate decay***, en el que la learning rate se va 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 la rate se va reduciendo permitiendo ser más finos en la búsqueda de una solución.

¿Qué utilizar?

- Normalmente, los **métodos adaptativos** (AdaGrad, RMSProp, Adam) consiguen mayores velocidades de convergencia y, por tanto, entrenamientos más rápidos de redes neuronales.
- **Adam** suele dar muy buenos resultados y sus parámetros por defecto funcionan muy bien, lo que ayuda a no tener que pensar qué learning rate utilizamos.
- La discusión científica continúa. Parte de la comunidad incluso defiende la utilización de SGD clásico frente a estos métodos más modernos.

Sistemas Cognitivos Artificiales

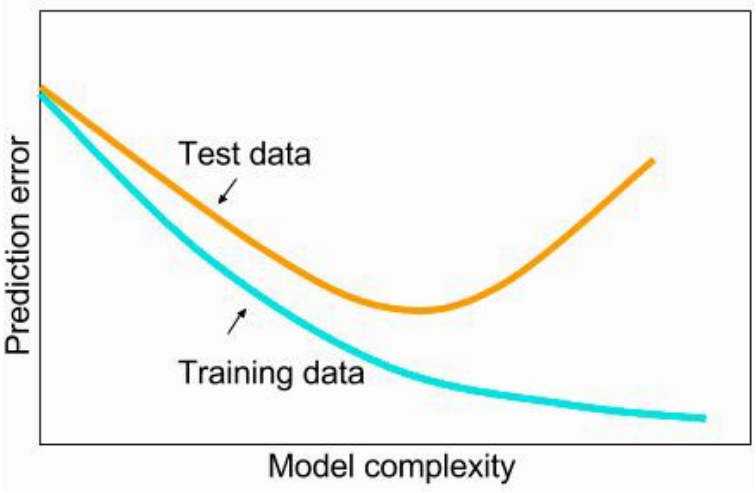
Roberto Casado Vara

Tema 4.5: Regularización

Overfitting y regularización

- Las redes neuronales son modelos con un gran poder de representación. Puede pasar que el modelo memorice los datos de entrenamiento pero pierda capacidad de generalización al ser evaluado sobre datos no vistos durante el entrenamiento.

- Este fenómeno se conoce como **overfitting**. El modelo está aprendiendo detalles particulares de los datos de entrenamiento en vez de las características generales de un dataset.



Fuente:
http://gluon.mxnet.io/chapter02_supervised-learning/regularization-scratch.html

Overfitting y regularización

- Las técnicas de **regularización** intentan solucionar el problema del overfitting.
- Hay muchas formas de aplicar regularización. Es común aplicar varias estrategias a la vez.

Regularización L2

- Una de las formas más comunes de aplicar regularización en *Machine Learning*.
- Añade una **penalización** en la función de coste. Esta penalización suma los valores al cuadrado de los *weights* de una red neuronal.
- El hiperparámetro ƛ controla la fuerza de la regularización. Amayor valor, más penalización a los parámetros y más regularización.

Función de coste

Penalización

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

Regularización L2

- Al *penalizar* el cuadrado de los parámetros, estamos favoreciendo que estos tengan valores absolutos pequeños. Durante el entrenamiento, la red sólo asignará valores grandes a los parámetros si al hacer esto consigue reducir la *loss* en una cantidad mayor que la penalización.

- Intuitivamente, al penalizar valores de parámetros grandes, estamos *reduciendo el espacio de posibles soluciones*. La red neuronal es menos libre de asignar valores cualesquiera, y por tanto pierde capacidad de expresividad. Esto hace que sea más difícil fijarse en los detalles particulares del training data, por lo que reducimos el problema de overfitting.

- La regularización L2 es bastante usada en la práctica.

Función de coste original

Penalización

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

Regularización L1

- Similar a la regularización L2 pero sumando valores absolutos en vez de cuadrados.
- Tiende a hacer que los parámetros sean *sparse* (dispersos). Muchos parámetros se quedan con valores 0 o cercanos a 0.
- No es tan utilizada como la regularización L2 para redes neuronales.

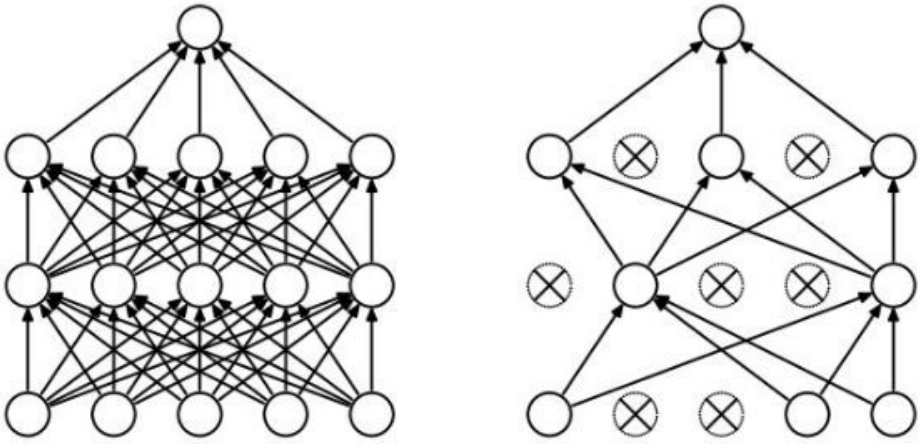
Función de coste original

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

Penalización

Dropout

- Probablemente la técnica de regularización más utilizada en la actualidad.
- Consiste en aplicar salida 0 a un porcentaje de neuronas de la red en cada batch de manera aleatoria. Es como hacer que un porcentaje de neuronas no estuvieran presentes.
- Cada neurona se desactiva con una probabilidad *p* (otro hiperparámetro). Valores comunes de *p* van desde 0.1 a 0.5.



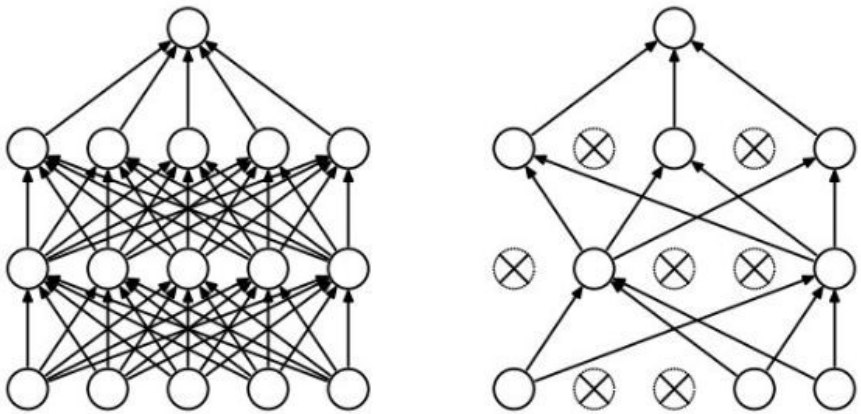
(a) Standard Neural Net (b) After applying dropout.

Fuente: http://cs231n.stanford.edu/

Dropout

- Dropout impide la co-adaptación de las features. Al hacer desaparecer neuronas al azar, la red tiene que aprender nuevas formas de correlacionar los valores de salida de las neuronas.

- Podemos ver esto como que la eliminación de ciertas neuronas al azar *dificulta a la red memorizar resultados*, por lo que obtenemos un efecto regularizador.

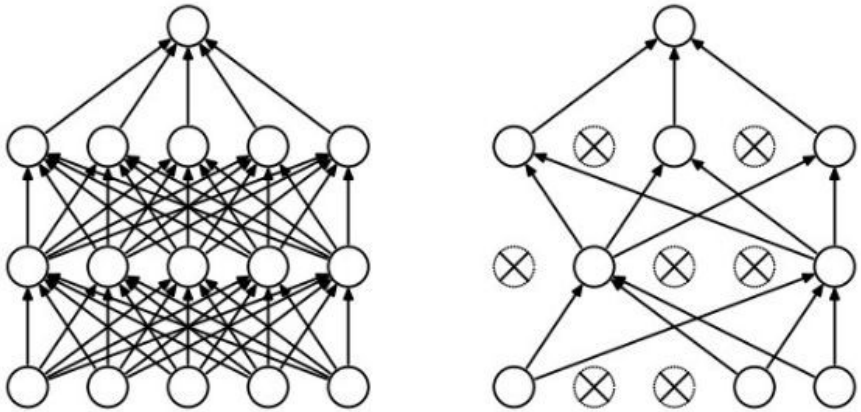


(a) Standard Neural Net (b) After applying dropout.

Fuente: http://cs231n.stanford.edu/

Dropout

- Otra forma de interpretar dropout es que actúa como un **ensemble** de modelos.
- Por cada batch, estamos obteniendo una “nueva” red neuronal según las neuronas que quedan activas.
- Podemos interpretar dropout como un proceso por el que estamos entrenando a la vez un gran número de redes neuronales ligeramente distintas.

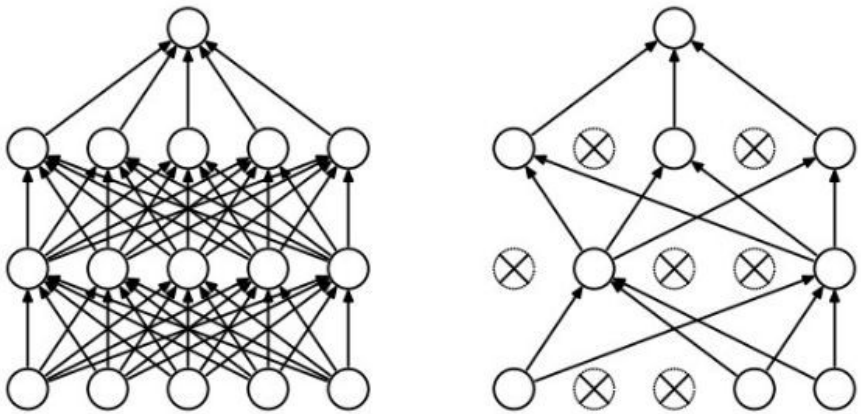


(a) Standard Neural Net (b) After applying dropout.

Fuente: http://cs231n.stanford.edu/

Dropout

- Durante *test time*, al utilizar el modelo ya entrenado, dropout no se aplica. Todas las neuronas están activas.
- Durante *backpropagation*, los valores de los gradientes transmitidos hacia atrás en las neuronas que desaparecen son 0.

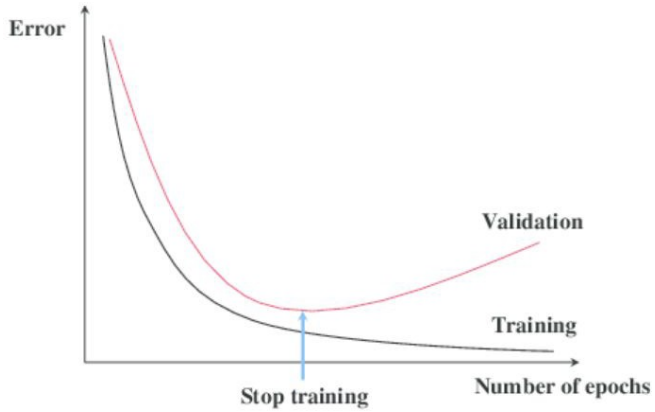


(a) Standard Neural Net (b) After applying dropout.

Fuente: http://cs231n.stanford.edu/

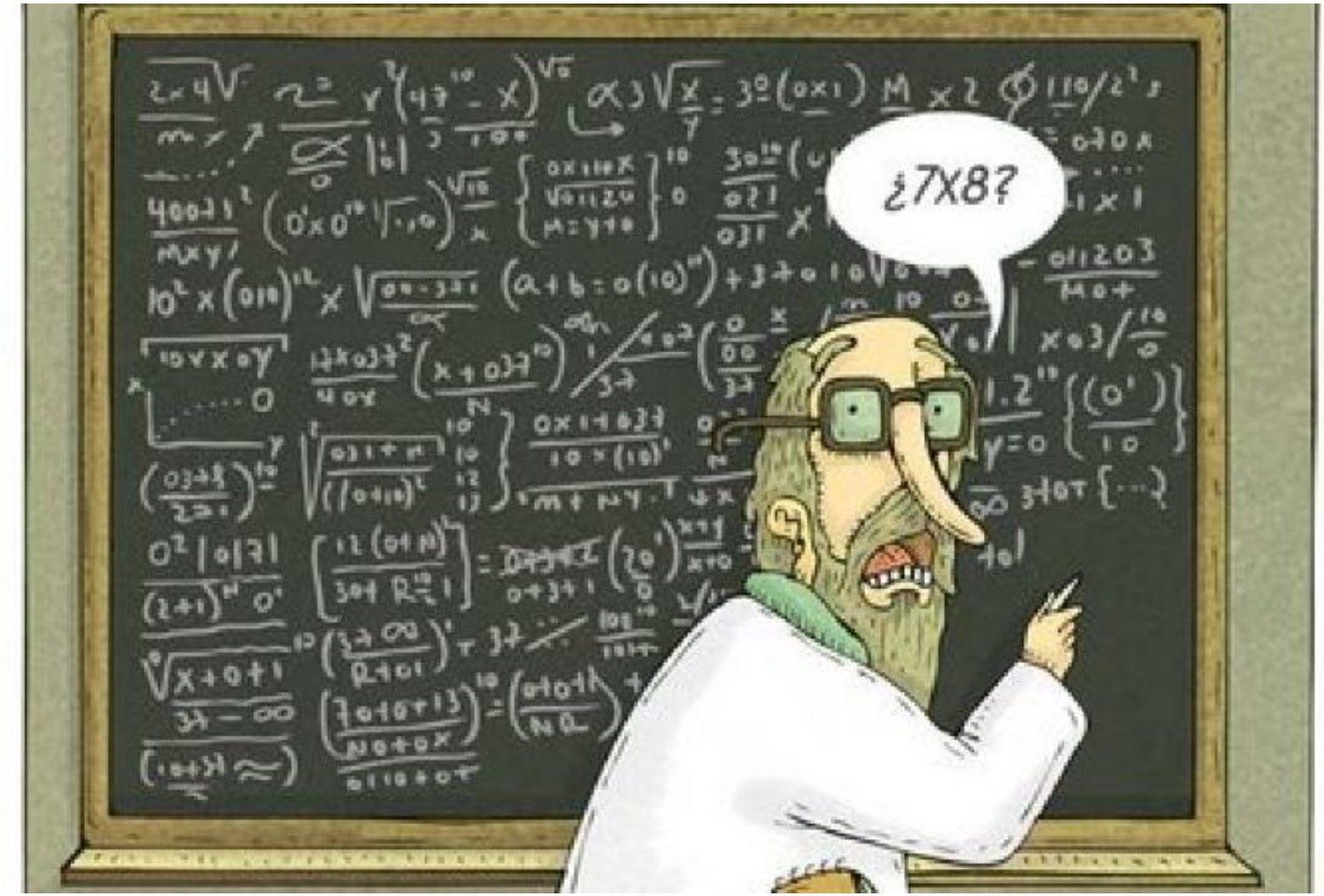
Early stopping

- Otra técnica muy utilizada en la práctica, que además ayuda a no perder tiempo entrenando una red en overfitting.
- Consiste en utilizar un *validation set* para ver cuándo la red empieza a caer en overfitting. En ese momento, dejamos de entrenar.
- Una estrategia común es medir el momento en que la *validation loss* deja de mejorar, esperar unas cuantas épocas para asegurarnos y quedarnos con el último modelo guardado que presentaba mejoras en la *validation loss*.



Fuente: https://www.researchgate.net/figure/Early-stopping-method_fig3_283697186

¿Dudas?



www.unir.net