UT5. Entrenamiento redes neuronales.

IES Abastos

CE Inteligencia Artificial y Big Data

Sistemas de Aprendizaje Automático.

Bloques de la unidad:

1. Funciones de coste.
2. Entrenamiento con gradient descent.
3. Stochastic Gradient Descent (SGD).
4. Laboratorio 1.
5. Backpropagation.
6. Ejercicios regla cadena.
7. **Unidades de activación.**
8. **Inicialización de parámetros.**
9. **Batch normalization.**
10. **Optimización avanzada.**
11. **Regularización.**
12. **Laboratorio 2.**

7. Unidades de activación

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).

7. Unidades de activación

En los siguientes puntos del tema veremos aspectos prácticos y avanzados para conseguir un entrenamiento **satisfactorio** y **eficiente** de redes neuronales.

7. Unidades de activación

Uno de los elementos más críticos en una red neuronal es la elección de unidad de activación o non-linearity.

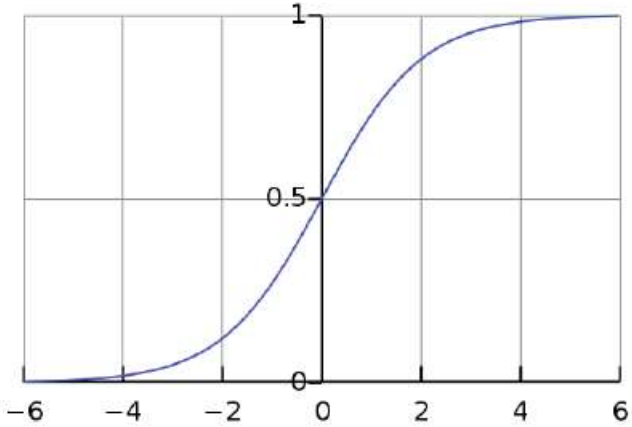
En el primer laboratorio hemos visto la utilización de la unidad más clásica, la unidad sigmoid.

La irrupción de nuevas y más efectivas unidades de activación ha sido uno de los grandes avances en el mundo del deep learning.

7. Unidades de activación

Unidad sigmoid: 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.

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



7. Unidades de activación

Problemas unidad sigmoid: Esta función han caído en desuso debido a los siguientes problemas:

1. Las unidades sigmoid saturadas “matan” los gradientes.
2. La salida de sigmoid no está centrada en 0.
3. Implica el cálculo de una función exponencial .

7. Unidades de activación

Las unidades sigmoid saturadas “matan” los gradientes.

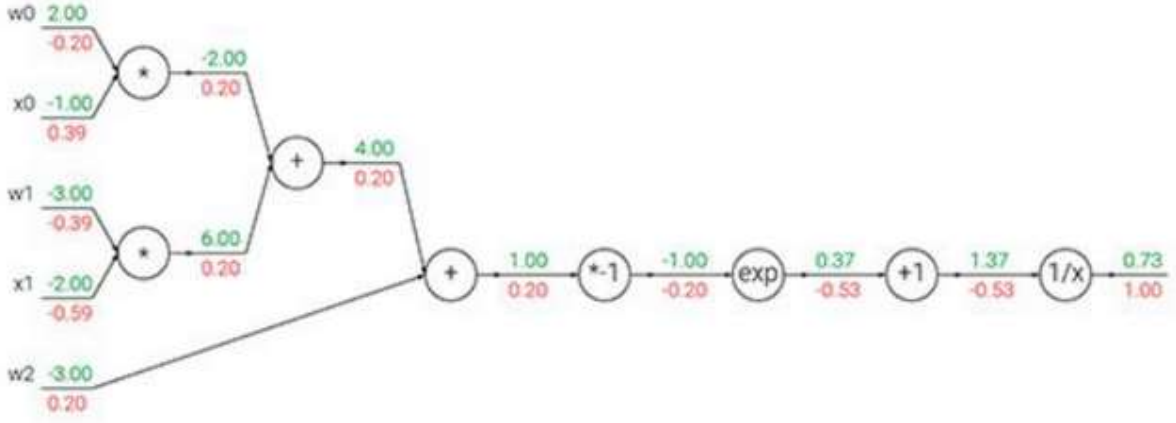
Decimos que la unidad se satura cuando sus valores se aproximan a 1 o a 0. El régimen de saturación se caracteriza porque pequeños cambios en x no implican apenas variación en y.

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

7. Unidades de activación

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:



7. Unidades de activación

Las unidades sigmoid saturadas “matan” los gradientes.

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.

7. Unidades de activación

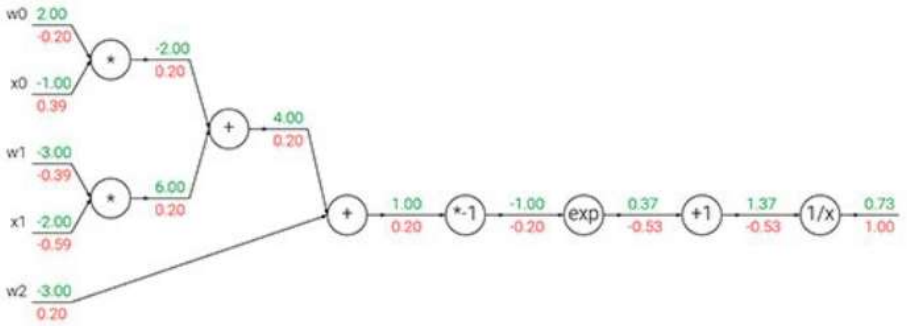
La salida de sigmoid no está centrada en 0.

- 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.

7. Unidades de activación

Implica el cálculo de una función exponencial.

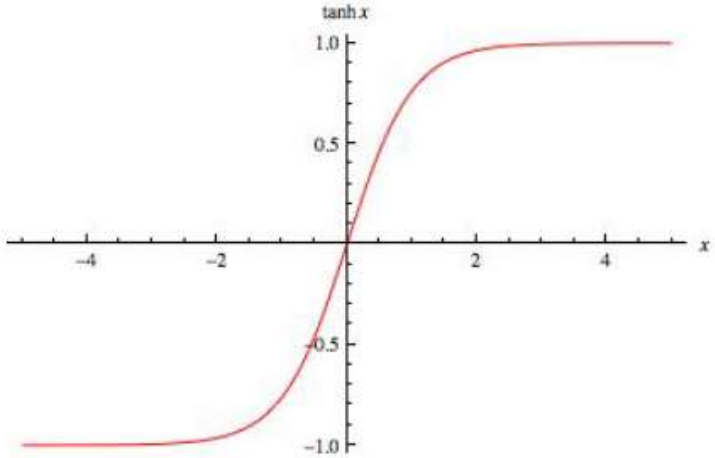
Otro pequeño inconveniente, ya que el cálculo de una función exponencial conlleva cierta complejidad en comparación con otras operaciones.



7. Unidades de activación

Tanh (tangente hiperbólica):

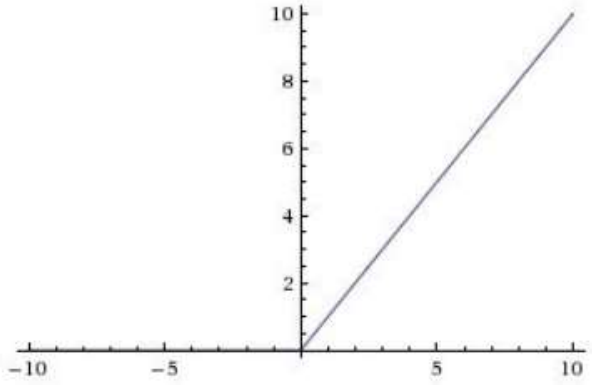
- Similar a sigmoid, pero ahora con valores entre -1 y 1.
- Sigue “matando” gradientes, pero la salida está centrada en 0.
- En la práctica, tanh es preferible a sigmoid.



7. Unidades de activación

ReLU (rectified linear unit)

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



7. Unidades de activación

Problemas de las ReLU: Son frágiles y pueden “morir” durante el entrenamiento.

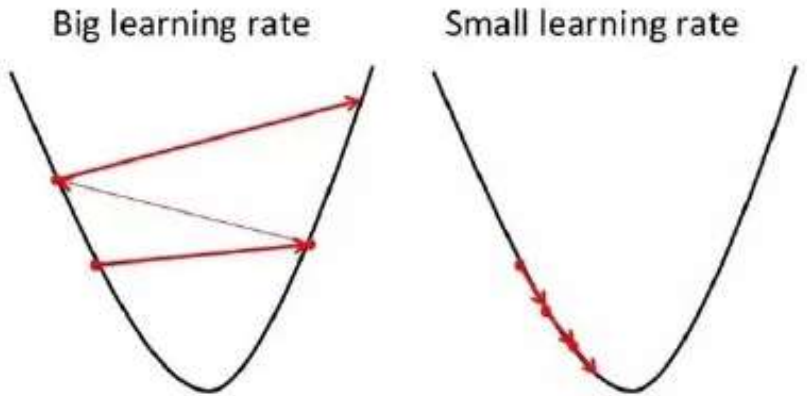
- Un cambio grande puede hacer que la unidad no vuelva a activarse nunca, esto es, que nunca vuelva a producir valores mayores que 0.
- Esto provoca que el gradiente que pasa por la neurona sea siempre 0, por lo que la neurona se considera “muerta”.
- Es normal ver redes neuronales con ReLUs muertas.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

7. Unidades de activación

RECORDATORIO LEARNING RATE

Si η es demasiado **grande**, podríamos no encontrar el mínimo o incluso divergir a valores de C mayores.

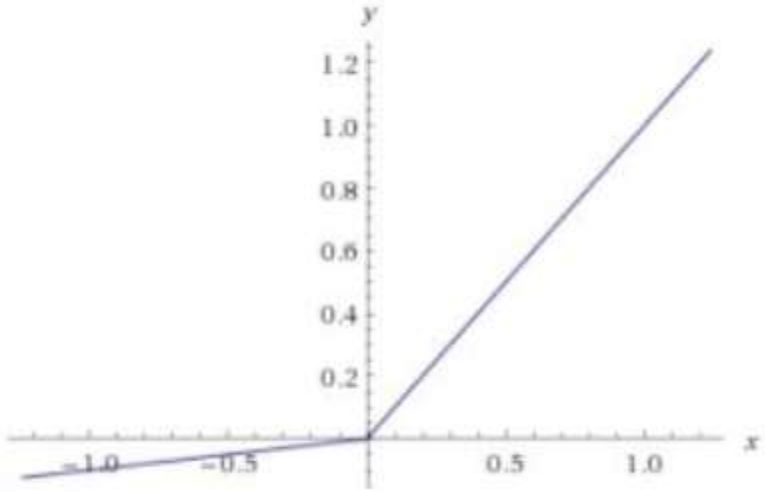
Por otro lado, un valor **demasiado pequeño** haría que el algoritmo avanzara de forma muy lenta.



7. Unidades de activación

Leaky ReLU:

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.



7. Unidades de activación

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

Maxout

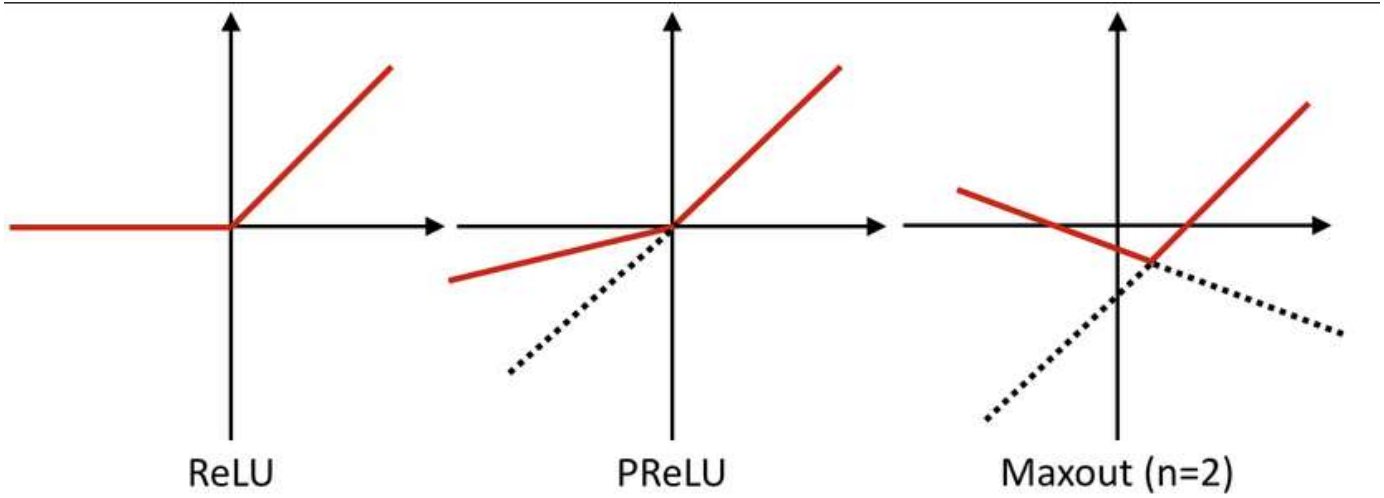
Otra unidad que intenta solucionar el problema de las ReLUs “muertas”. Es una generalización de las ReLU y Leaky ReLU.

La unidad tiene dos sets de parámetros w y b en vez de uno.

Si la red aprende w1 = 0 y b1 = 0, tenemos ReLU.

Es una unidad más versátil pero más compleja. Requiere el doble de parámetros.

7. Unidades de activación



7. Unidades de activación

- 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.

7. Unidades de activación

Actividad por grupos: En aules tenéis un paper actual sobre funciones de activación, por equipos resolver las siguientes preguntas.

1. ¿Cómo está estructurado el documento?
2. ¿Cuantas funciones de activación aparecen? Enumerarlas.
3. ¿Cuántos autores aparecen citados? ¿En qué páginas aparecen las referencias?
4. Enumera los dataset que utiliza.
5. Enumera las arquitecturas que utiliza para el experimento.
6. ¿Qué métrica de desempeño utilizan?
7. ¿Podemos acceder al código del experimento?
8. ¿Qué conclusiones podemos sacar de este experimento?

8. 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.

Recordar: Nuestro objetivo al entrenar la red neuronal es conseguir que nuestra aproximación al objetivo real sea lo mejor posible, esto es, minimizar en la medida de lo posible **C(w, b)**.

8. Inicialización de parámetros

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.

8. Inicialización de parámetros

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.

8. Inicialización de parámetros

Inicialización aleatoria:

- 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.

8. Inicialización de parámetros

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.

8. Inicialización de parámetros

Xavier 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

8. Inicialización de parámetros

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 que los pesos w son inicializados aleatoriamente, la simetría de la red se rompe.
- Es común inicializarlos a 0 (default en Keras).

8. Inicialización de parámetros

Recurso muy interesante, vamos a aplicar los conceptos aprendidos.

http://www.deeplearning.ai/ai-notes/initialization/

9. 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?

9. Batch normalization

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.

9. 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:

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

9. Batch normalization

“se calcula la media y la varianza de los inputs de las funciones de activación”

```
[12] batch_size= 64
     epochs = 20
     history = model.fit(x_train, y_train,
                         batch_size = batch_size,
                         epochs = epochs,
                         verbose=1,
                         validation_data=(x_test, y_test))
     score = model.evaluate(x_test, y_test, verbose=0)
```

```
Epoch 1/20
938/938 [==============================] - 16s 15ms/step - loss: 2.1860 - accuracy: 0.4158 - val_loss: 2.0265 - val_accuracy: 0.5128
Epoch 2/20
938/938 [==============================] - 4s 5ms/step - loss: 1.7984 - accuracy: 0.5721 - val_loss: 1.5722 - val_accuracy: 0.6243
Epoch 3/20
938/938 [==============================] - 5s 5ms/step - loss: 1.4006 - accuracy: 0.6170 - val_loss: 1.2662 - val_accuracy: 0.6489
```

9. Batch normalization

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))

9. 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.

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

9. Batch normalization

Ventajas:

- Mejora el flujo de gradientes de la red durante el aprendizaje, aumentando por tanto la velocidad de convergencia.
- Permite learning rates mayores.
- Reduce la dependencia del entrenamiento en la inicialización de parámetros.
- Tiene un pequeño efecto regularizador, se puede decir que añadimos un pequeño ruido, viene bien para que no memorice.

10. Optimización avanzada.

Recordatorio:

Como sabemos, el entrenamiento de una red neuronal consiste en la solución de un problema de optimización.

Stochastic Gradient Descent: 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

10. Optimización avanzada.

```
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
```

```
[12] batch_size= 64
     epochs = 20
     history = model.fit(x_train, y_train,
                         batch_size = batch_size,
                         epochs = epochs,
                         verbose=1,
                         validation_data=(x_test, y_test))
     score = model.evaluate(x_test, y_test, verbose=0)
```

```
Epoch 1/20
938/938 [==============================] - 16s 15ms/step - loss: 2.1860 - accuracy: 0.4158 - val_loss: 2.0265 - val_accuracy: 0.5128
Epoch 2/20
938/938 [==============================] - 4s 5ms/step - loss: 1.7984 - accuracy: 0.5721 - val_loss: 1.5722 - val_accuracy: 0.6243
Epoch 3/20
938/938 [==============================] - 5s 5ms/step - loss: 1.4006 - accuracy: 0.6170 - val_loss: 1.2662 - val_accuracy: 0.6489
```

10. Optimización avanzada.

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

1. Zig-zags ineficientes.
2. Mínimos locales.
3. Puntos de silla.

10. Optimización avanzada.

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.

10. Optimización avanzada.

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 implica normalmente overfitting.

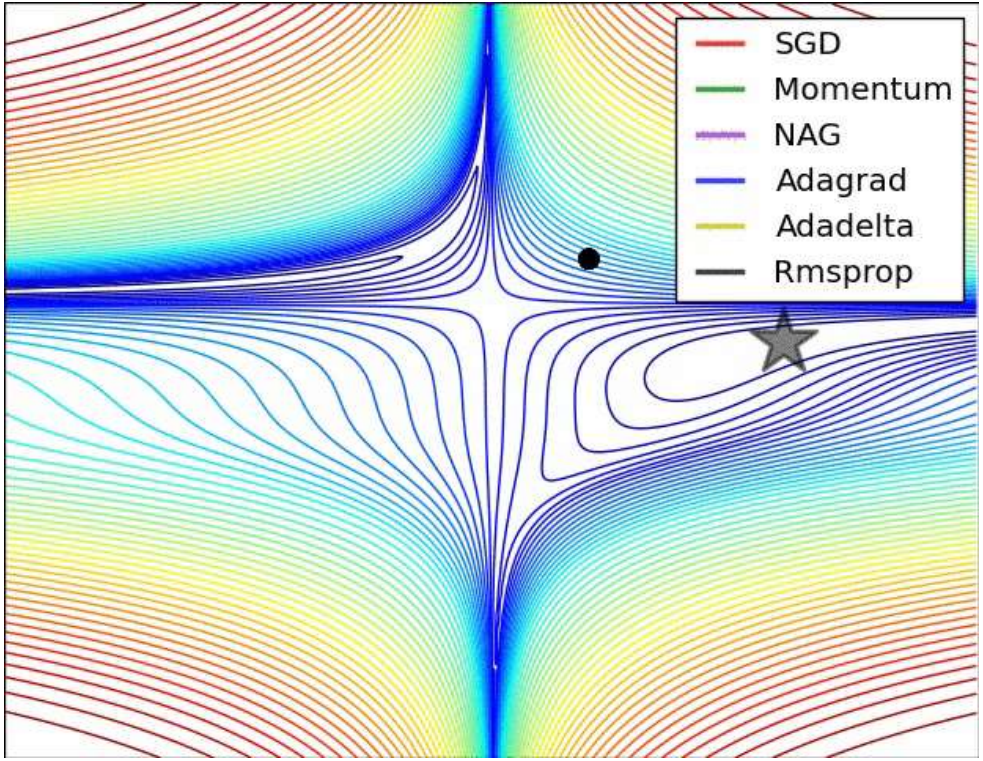
Una variante muy popular de SGD para resolver los problemas que hemos visto en el apartado anterior es **SGD con Momentum.**

10. Optimización avanzada.

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.

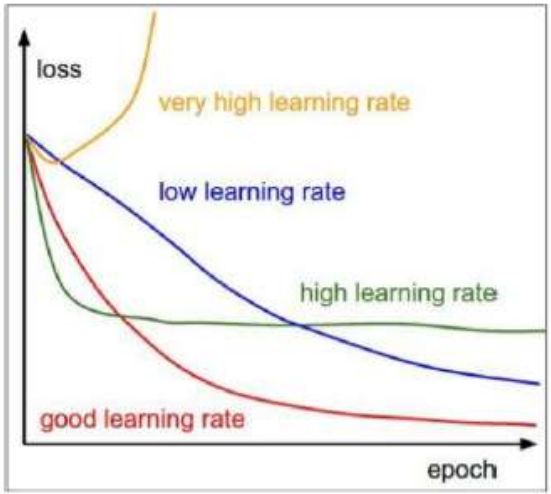
Los métodos más utilizados son: Adagrad, RMSProp, Adam, NAG, Adadelta.

10. Optimización avanzada.



10. Optimización avanzada.

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



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

10. Optimización avanzada.

Learning rate decay: 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.

10. Optimización avanzada.

Learning rate decay:

```
# Añadimos nuestro clasificador
model.add(Flatten())
model.add(Dense(1024, activation='relu', kernel_regularizer=l2(0.01)))
model.add(Dense(10, activation='softmax'))

# Compilamos el modelo
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=0.0001, decay=1e-6),
              metrics=['accuracy'])
```

10. Optimización avanzada.

¿ 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.

10. Optimización avanzada.

La discusión científica continúa. Parte de la comunidad incluso defiende la utilización de SGD clásico frente a estos métodos más modernos.

Nos vamos a la documentación de Keras:

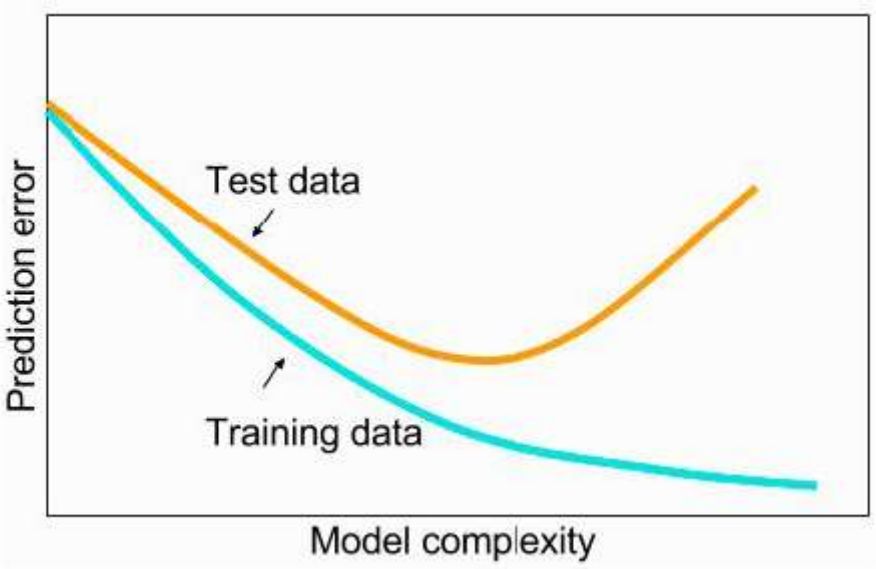
https://keras.io/api/optimizers/

11. 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.

11. Regularización.



11. 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.
- Regularización L1.
- Dropout.
- Early stopping

11. Regularización.

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. A mayor valor, más penalización a los parámetros y más regularización.

11. Regularización.

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.

Función de coste

Penalización

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

11. Regularización.

Regularización L2.

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.

Función de coste

Penalización

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

11. Regularización.

Regularización L2. Es bastante usada en la práctica.

```
# Añadimos nuestro clasificador
model.add(Flatten())
model.add(Dense(1024, activation='relu', kernel_regularizer=l2(0.01)))
model.add(Dense(10, activation='softmax'))

# Compilamos el modelo
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=0.0001, decay=1e-6),
              metrics=['accuracy'])
```

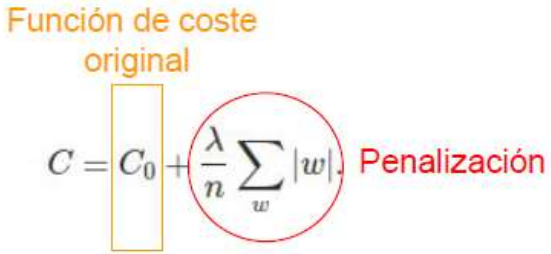
11. Regularización.

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.



11. Regularización.

Regularización L1.

```
# Prueba con regularización L1

# importamos la capa regularización
from keras.regularizers import l1

# Inizializamos el modelo
model = Sequential()

# Definimos una capa convolucional
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu',
input_shape=(32, 32, 3)))

# Definimos una segunda capa convolucional
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))

# Definimos una tercera capa convolucional
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))

# Añadimos nuestro clasificador
model.add(Flatten())
model.add(Dense(1024, activation='relu', kernel_regularizer=l1(0.01)))
model.add(Dense(10, activation='softmax'))

# Compilamos el modelo
model.compile(loss='categorical_crossentropy',
        optimizer=Adam(lr=0.0001, decay=1e-6),
```

11. Regularización.

Dropout.

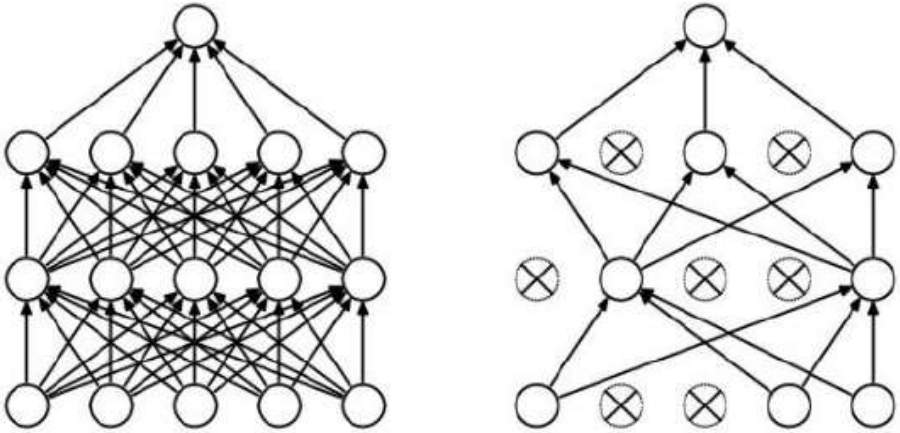
Probablemente la técnica de regularización más utilizada en la actualidad.

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.

11. Regularización.

Dropout.



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

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

11. Regularización.

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.

11. Regularización.

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.

11. Regularización.

Dropout.

```
# Definimos la tercera capa convolucional
model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(Dropout(0.25))

# Añadimos nuestro clasificador
model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# Compilamos el modelo
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=0.0001, decay=1e-6),
              metrics=['accuracy'])
```

11. Regularización.

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.

11. Regularización.

Early stopping.

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.