Funciones de optimización

U-TAD

Repaso de Batch Gradient Descent

- Con este algoritmo de optimización deben procesarse todos los ejemplos del conjunto de datos de entrenamiento antes de modificar los parámetros del modelo
- Si el conjunto de datos es muy grande, que es lo normal en DL, (ej. 2.000.000 ejemplos) el proceso de optimización puede ser muy lento

$J=0, dw_1=0, dw_2=0, \ldots, dw_n=0, db=0$

for i = 1 hasta m: // m = # ejs.conj.datos y l = # capas

$\hat{y}^{(i)} = a^{(l)(i)}z^{(l)(i)}$

$J \mathrel{+}= L(\hat{y}^{(i)}, y^{(i)})$

$dw_1 \mathrel{+}= dJ/dw_1, dw_2 \mathrel{+}= dJ/dw_2, \ldots, dw_n \mathrel{+}= dJ/dw_n, db \mathrel{+}= dJ/db$

$J=J/m, dw_1=dw_1/m, \ldots, dw_n=dw_n/m, db=db/m$

$w_1=w_1-ndw_1, w_2=w_2-ndw_2, \ldots$

Mini-Batch Gradient Descent

- Una de las variaciones interesantes de Batch Gradient Descent consiste en dividir el conjunto de datos de entrenamiento en subconjuntos más pequeños denominados **mini-batches** (esta es la diferencia fundamental con GD)

$x_1 \quad x_2 \quad \dots \; x_n \quad y$

$x_{1(1)} \quad x_{2(1)} \dots x_{n(1)} \quad y_{(1)}$

...

$x_{1(m)} \quad x_{2(m)} \quad x_{n(m)} \quad y_{(m)}$

y como matriz: $X = [$ $\begin{matrix} x_{1(1)} & x_{1(2)} \dots x_{1(m)} \\ \dots \\ x_{n(1)} & x_{n(2)} \dots x_{n(m)} \end{matrix}$ $] = [\, x_{(1)} \; x_{(2)} \dots x_{(m)} \,]$

$x_{1(1)} \; x_{2(1)} \dots x_{n(1)}$

$Y = [\, y_{(1)} \; y_{(2)} \dots y_{(m)} \,]$

Mini-Batch Gradient Descent

x_1 x_2 … x_n y

$x_{1(1)}$ $x_{2(1)}$ … $x_{n(1)}$ $y_{(1)}$

…

$x_{1(m)}$ $x_{2(m)}$…$x_{n(m)}$ $y_{(m)}$

$$X = [\, x_{(1)}\ x_{(2)} \dots x_{(1000)} \dots x_{(m)} \,]$$

$$Y = [\, y_{(1)}\ y_{(2)} \dots y_{(1000)} \dots y_{(m)} \,]$$

Creamos Mini Batches de 1000 ejs. donde t=#mini-batches

$$X\{1\} = [\, x_{(1)}\ x_{(2)} \dots x_{(1000)} \,] \qquad X\{2\} = [\, x_{(1001)}\ x_{(1002)} \dots x_{(2000)} \,] \dots X\{t\}$$

$$Y\{1\} = [\, y_{(1)}\ y_{(2)} \dots y_{(1000)} \,] \qquad Y\{2\} = [\, y_{(1001)}\ y_{(1002)} \dots y_{(2000)} \,] \dots Y\{t\}$$

Mini-Batch GD opera igual que Batch GD, con la diferencia de que ahora vamos a actualizar los parámetros de cada modelo con cada mini-batch en lugar de con cada recorrido completo de nuestro conjunto de datos (tendremos un bucle más)

Mini-Batch Gradient Descent

for j=1 to t { // t = # min-batches

for i=1 to $m_{\{j\}}$ { // $m_{\{j\}}$ = # ejemplos en el mini-batch $X_{\{j\}}$

$\hat{y}(i) = a^{(l)(i)}z^{(l)(i)}$

J += $L(\hat{y}^{(i)}, y^{(i)})$

dw_1+=dJ/dw_1, dw_2+=dJ/dw_2, …, dw_n+=dJ/dw_n, db+=dJ/db

} //Ahora, por cada mini-batch actualizamos los parámetros

J=J/m, dw_1=dw_1/m, …, dw_n=dw_n/m, db=db/m

w_1=w_1-ndw_1, …, w_n=w_n-ndw_n, b=b-ndb

}

Antes, computábamos todos los ejemplos, ahora lo hacemos por batches. Es decir, un epoch consiste en recorrer todos los mini batches y aplicar todas esas operaciones por cada uno de los mini-batches. La diferencia fundamental es que con Batch Gradient Descent actualizamos los parámetros una vez por cada epoch y con mini-batch los actualizamos varias veces en cada epoch.

Caso Práctico: Mini-batch Gradient Descent

Clasificación de texto con Mini-Batch Gradient Descent.ipynb

Stochastic GD (lo vemos después), nos permite setear el “learning_rate”

Epoch (1/10)

281/281 usamos mini-batch GD que actualiza los parámetros 281 veces

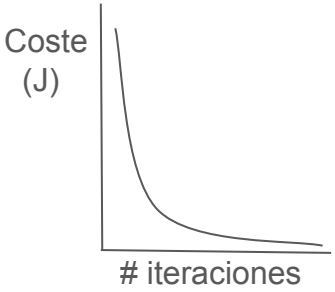
len(x_train) / 281 mini-batches = **tamaño de mini-batch de 32 (por defecto)**

para setearlo: batch_size=512 (comprueba con 32 y 512 como tarda en entrenarse, cuánto de actualiza y compara las gráficas)

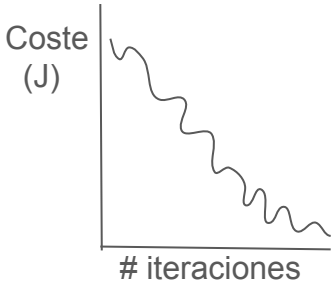
Cuanto mayor sea el tamaño de mini-batch, vamos a tener un algoritmo más lento entrenándose pero irá más dirigido hacia el valor óptimo de los parámetros que minimiza la función de error. Y cuanto menor es el tamaño del mini-batch, tendríamos menos ejemplos para modificar el valor de los parámetros, por lo que podríamos modificarlos en una dirección que no sea óptima, pero entrenará más rápido en general minimizando el valor de la función de error porque habrá más actualizaciones de los parámetros en cada uno de los epochs (lo veremos en la siguiente slide).

Stochastic Gradient Descent

Batch Gradient Descent Vs Mini-Batch Gradient Descent



Realiza una mejor predicción (menor error) por cada una de las iteraciones porque usamos todos los parámetros del modelo para realizar una sola predicción (con todos los ejemplos del conjunto de datos)

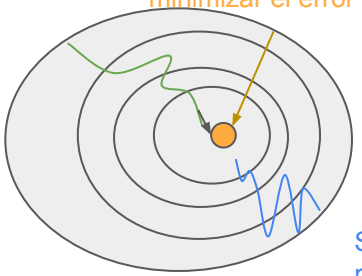


Cuando aumenta el número de iteraciones, irá reduciendo el Coste, pero no tiene por qué reducirse con cada una de las iteraciones. Porque modificamos el valor de los parámetros del modelo utilizando un subconjunto de datos

Con mini-batch, de manera general, aunque en alguna iteración puede que aumente el error, la tendencia será a minimizarse

Stochastic Gradient Descent

Tamaño de Mini-Batch

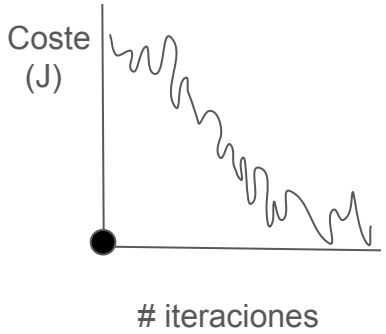


m=#ejs. datos

tam.mini-batch = m (BGD)

= 1<x<m (MBGD)

Se queda cerca,
pero no converge = 1 (SGD)



Batch vs Mini-Batch vs Stochastic

- **Batch** Gradient Descent (ideal para conjuntos de datos pequeños)
  - Procesa el conjunto de datos completo en cada iteración (un solo batch que cabe en la RAM)
  - Función de optimización más lenta
  - Suele converger y alcanzar el mínimo de la función de error

- **Mini-Batch** Gradient Descent (ideal para conjuntos de datos grandes >2M ejs.)
  - Procesa un subconjunto del conjunto de datos en cada iteración (tam mini-bath? 64, 128, 256, 512)
  - Función de optimización más rápida que Batch Gradient Descent
  - Suele converger y alcanzar el mínimo de la función de error de manera menos directa

- **Stochastic** Gradient Descent (para conjuntos de datos muy grandes y pocos recursos computacionales para entrenar)
  - Procesa un ejemplo del conjunto de datos en cada iteración
  - Función de optimización más rápida de las tres
  - No suele converger y alcanzar el mínimo de la función de error, aunque alcanza un valor aceptable

Caso práctico: Stochastic Gradient Descent

Clasificación de texto con Stochastic Gradient Descent.ipynb

Importante: Revisar en history = model.fit(**batch_size = 1**, …)

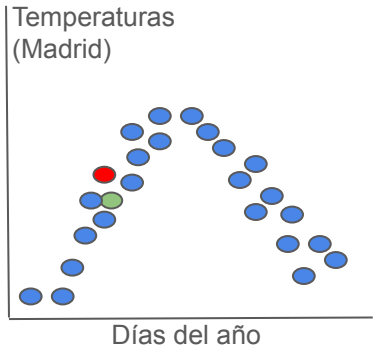
Por cada uno de los epochs haremos 8982 actualizaciones de los parámetros

(es mucho más lento debido a esas actualizaciones en un mismo epoch)

Con sgd, en un mismo número de epochs conseguiremos mejores resultados que con bgd y mgd

Exponentially Weighted Moving Average (Part I)

Técnica estadística que debemos conocer: Media móvil ponderada exponencialmente



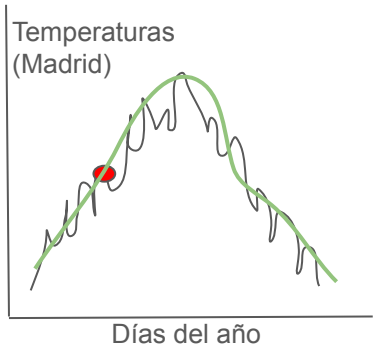
O_1 = 3º, O_2 = 3º, O_3 = 4º, …, O_{365} = 10º

En base a las temperaturas medias por día del año pasado, quiero predecir la temperatura para un día concreto (mañana, pasado o un día del mes que viene)

Por ejemplo, si el 01-04-2015 la temperatura fue de 17º, y supongamos que fue un día con una temperatura algo mayor a la media de finales de mayo y principios de junio. Por lo que calcularíamos la media de los puntos cercanos y nos quedamos con esa previsión. No tiene porqué coincidir con la temperatura de ese mismo día.

Exponentially Weighted Moving Average (Part I)

El día que queremos predecir, se correspondía con un pico en la gráfica, por lo que es más probable que acertemos si cogemos ese valor medio entre las temperaturas cercanas a ese día.



Para esta gráfica que tiene muchos picos, necesitamos suavizarla. Es decir, que nos mantenga la tendencia con nuestra serie temporal y que suavice los picos.

Si aplicamos la técnica estadística para la gráfica de temperaturas, obtendremos la función representada en verde, teniendo en cuenta algunas temperaturas de días pasados, para suavizar los picos con la influencia de las otras temperaturas.

Exponentially Weighted Moving Average (Part I)

La función que vamos a utilizar para recalcular cada uno de los valores de temperatura de cada uno de los días va a ser la siguiente:

$V_0 = 0, \quad B = 0.9 \ (0 < B < 1)$

$V_t = BV_{t-1} + (1-B)O_t$

t = número de día (1-365), B = hiperparámetro Beta, Ot = la temp.para ese día

$V_1 = 0.9*0 + (1-0.9)*3 = 0.3\ ºC$

$V_2 = 0.9*0.3 + (1-0.9)*3 = 0.57\ ºC$

…

$V_{365} = 0.9*V_{364} + (1-0.9)*10 = ?\ ºC$

V_t -> la temperatura media aproximada de los últimos 1/(1-B) días

1/(1-0.9) = 10 días
1/(1-0.95) = 20 días (la línea se suaviza más)

Exponentially Weighted Moving Average (Part II)

$V_0 = 0$ $B = 0.9$

$V_t = BV_{t-1} + (1-B)O_t$

$V_{100} = 0.9*V_{99} + 0.1*O_{100} = 0.1O_{100} + 0.9V_{99} =$

$= 0.1O_{100} + 0.9(0.9\mathbf{V_{98}} + 0.1O_{99}) =$

$= 0.1O_{100} + 0.9(0.1O_{99} + \underline{0.9}\mathbf{(\underline{0.1}O_{98} + \underline{0.9}V_{97})})$

Podemos observar que para calcular V100 tenemos en cuenta todas las temperaturas anteriores, pero en menor medida según se aleja.

$V_{100} = 0.1O_{100} + 0.9V_{99} = 0.1O_{100} + \underline{0.09}O_{99} + \underline{0.81}V_{98} =$

$= 0.1O_{100} + 0.09O_{99} + 0.081O_{98} + 0.729V_{97}$

A partir de los 10 días (por el B=0.9), multiplicará a las temperaturas un valor tan pequeño que, prácticamente, no tendrá nada de influencia.

Exponentially Weighted Moving Average (Part II)

Esta técnica funciona muy bien para valores medios, sin embargo, lo valores que se encuentran al comienzo, aquellos que ni siquiera tienen 10 días anteriores para calcular la media, están algo desajustados.

Por ejemplo, para 1 de enero (O_1=3ºC) y V_1 = 0.9*0 + 0.1*3 = 0.3 ºC

para 2 de enero (O_2=3ºC) y V_2 = 0.9*0.3 + 0.1*3 = 0.57 ºC

Por tanto, se produce un error y tendremos que aplicar un *bias correction*:

V_t / (1 - B^t) = (BV_{t-1} + (1 - B)O_t) / (1 - B^t)

V_1 = 0.3 / (1 - 0.9^1) = 0.3 / 0.1 = 3

V_2 = 0.57 / (1 - 0.9^2) = 0.57 / 0.19 = 3

Para los valores medios no cambia el resultado, porque al elevar 0.9 por un número t alto, queda un número tan pequeño que prácticamente es como si estuviéramos dividiendo el V_t entre 1 (así que solo aplica a los primeros t)

Caso Práctico: Exponentially Weighted Moving Average

Exponentially+weighted+moving+average.ipynb

Probamos con distintos betas y vemos cómo se suaviza la curva según aumenta.

El ideal es 0.9 porque valores mayores de beta se desajusta de la tendencia.

Para las primeras temperaturas, la media es bastante mala, por lo que le aplicamos bias correction.

Momentum Gradient Descent (mejora a las anteriores)

Muy similar a mini-batch Gradient Descent, con la diferencia de que vamos a aplicar la técnica estadística anterior (ewma), y será más rápido.

Podremos aplicarla a batch, mini-batch y stochastic gradient descent.

for j = 1 to # mini-batches

for i = 1 to # ejemplos mini-batch j //Normalmente, se calcula vectorizadamente

$\hat{y}^{(i)}$ = Forward Propagation

J += L($\hat{y}^{(i)}$, $y^{(i)}$)

dw_1 += dJ/dw_1, …, dw_n, db

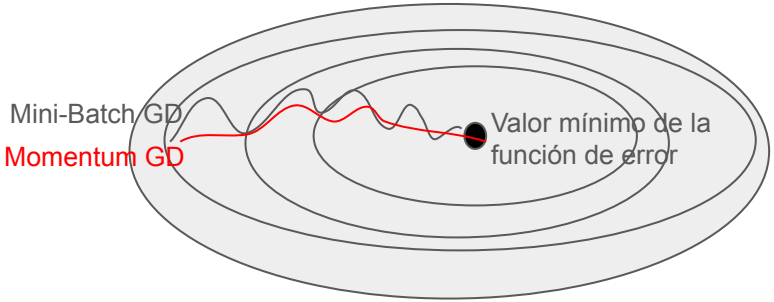
J = J/$m_{\{j\}}$, dw_1 = dw_1/$m_{\{j\}}$, …, dw_n = dw_n/$m_{\{j\}}$, db = db/$m_{\{j\}}$ //Ahora aplicamos ewma:

$Vdw_1 = B*Vdw_1 + (1-B)*dw_1$, ..., $Vdw_n = B*Vdw_n + (1-B)*dw_n$, $Vdb = B*Vdb + (1-B)*db$

w_1 = w_1 - nVdw_1, …, w_n = w_n - nVdw_n, b = b -nVdb

Momentum Gradient Descent (mejora a las anteriores)

Vamos a tratar de ir más directo al mínimo error, teniendo en cuenta valores anteriores:



Hacemos que todo el proceso de aprendizaje y optimización de los valores de los parámetros de nuestra red neuronal profunda sea mucho más rápido.

No se aplica el bias correction por términos de eficiencia, porque cuando entrenamos una red neuronal lo hacemos para un conjunto de mini-batches y epochs relativamente grandes y, por tanto, no importa que al comienzo empiece con mayor error porque siempre acabará estabilizándose (por lo que no suele implementarse en DL).

Solo tiene una pequeña desventaja respecto a mini-batch GD, y es que tiene dos hiperparámetros que será necesario seleccionar manualmente (learning rate y beta-0.9)

Caso práctico: Momentum Gradient Descent

Clasificación de texto con Momentum Gradient Descent.ipynb

Se selecciona modificando los parámetros de SGD, como momentum (beta):

opt_func = optimizers.SGD(learning_rate=0.01, **momentum=0.9**)

Compara las gráficas de error y accuracy de mini-batch con y sin **momentum**:

Minimizamos el error más rápidamente (con B <= 0.9, prueba con distintas Betas).

RMSprop

Vamos a ver variaciones de momentum Gradient Descent, que se corresponden con dos algoritmos de optimización muy utilizados a día de hoy (serán los que usemos en los ejemplos a partir de ahora). El primero de ellos es **RMSprop**:

for j=1 to # mini-batches {

for i = 1 to # ejs. mini-batch {

$\hat{y}^{(i)}$ = *Forward Propagation*

$J \mathrel{+}= L(\hat{y}^{(i)}, y^{(i)})$

$dw_1 \mathrel{+}= dJ/dw_1, \ldots, dw_n \mathrel{+}= dJ/dw_n, db \mathrel{+}= dJ/b$

$J = J/m_{\{j\}}, dw_1 = dw_1/m_{\{j\}}, \ldots, dw_n = dw_n/m_{\{j\}}, db = db/m_{\{j\}}$

$Sdw_1 = BSdw_1 + (1-B)*dw_1^{\mathbf{2}}, \ldots, Sdb = BSdb + (1-B)*db^{\mathbf{2}}$

epsilon (cte -> 10^-8)

$w_1 = w_1 - n*dw_1/sqrt(Sdw_1 + E), \ldots, w_n = w_n - n*dw_n/sqrt(Sdw_n + E),$

$b = b - n*db/sqrt(Sdb + E)$

Caso práctico: RMSprop

Clasificación de texto con RMSprop.ipynb

Usaremos otra clase diferente:

opt_func = optimizers.RMSprop() // learning_rate=0.001, rho=Beta(0.9), epsilon=1e-07

Compararlo con mini batch:

- es más lento, y el **val_loss** aumenta con las epochs y **loss** disminuye (**overfitting**). Desde el primer momento tiene un error bajo y después produce overfitting. Para mejorarlo podemos aumentar el tamaño del mini-batch (de 32 a 1024, por ejemplo), así podríamos bajar las iteraciones (epochs), por ejemplo a 10.

Adam Optimization

Algoritmo reciente que combina Momentum Gradient Descent y RMSprop:

$Vdw_1 = 0, Vdw_n = 0, Vdb = 0, Sdw_1 = 0, Sdw_n = 0, Sdb = 0$

*for **j**=1 to # mini-batches {*

for i = 1 to # ejs. mini-batch {

$\hat{y}_{(i)}$ *= Forward Propagation*

J *+= L(*$\hat{y}_{(i)}$*,* $y_{(i)}$*)*

dw_1 *+= dJ/*dw_1*, …,* dw_n *+= dJ/*dw_n*, db += dJ/b* *//Backward propagation*

$J = J/m_{\{j\}}, dw_1 = dw_1/m_{\{j\}}, \ldots, dw_n = dw_n/m_{\{j\}}, db = db/m_{\{j\}}$

$Vdw_1 = B_1*Vdw_1 + (1-B_1)*dw_1, \ldots, Vdw_n = B_1*Vdw_n + (1-B_1)*dw_n, Vdb = B_1*Vdb + (1-B_1)*db$ // Momentum GD

$Sdw_1 = B_2Sdw_1 + (1-B_2)*dw_1^2, \ldots, Sdb = B_2Sdb + (1-B_2)*db^2$ // RMSProp

$\mathbf{Vdw_{1(corr)} = Vdw_1 / (1-B_1^j), \ldots, Vdw_{n(corr)} = Vdw_n / (1-B_1^j), Vdb_{(corr)} = Vdb / (1-B_1^j)}$

$\mathbf{Sdw_{1(corr)} = Sdw_1 / (1-B_2^j), \ldots, Sdw_{n(corr)} = Sdw_n / (1-B_2^j), Sdb_{(corr)} = Sdb / (1-B_2^j)}$

$w_1 = w_1 - n*Vdw_{1(corr)} / sqrt(Sdw_{1(corr)} + E), \ldots, b = b - n*Vdb_{(corr)} / sqrt(Sdb_{(corr)} + E)$

Hiperparámetros (recomendaciones):
n -> learning rate (no hay recomendación, probad)
B_1 -> Beta1: 0.9
B_2 -> Beta2: 0.999
E -> Epsilon: 10^-8

Caso Práctico: Adam Optimization

Clasificación de texto con Adam.ipynb

*opt_func = optimizers.**Adam**()*

Comenzar con: batch_size = 32 y epochs = 10

Desde el comienzo loss (train) baja desde el principio y val_loss sube, lo que significa que se produce overfitting (compáralo con Mini-Batch Gradient Descent)

Como hacemos muchas actualizaciones por epochs y es un algoritmo que rápidamente encuentra el valor óptimo, empieza a provocar overfitting desde los primeros epochs.

Podemos aumentar el batch_size = 512 (es más rápido, al realizar muchas menos actualizaciones de los parámetros).

Comprueba la función de error en gráfica (val_loss), los valores están por debajo de 1 a partir de las primeras epochs, por lo que funciona mejor y más rápido que mini-batch.

Aunque los resultados finales (test y validación) no son muy buenos, sobre 0.80, es porque tenemos pocos datos de entrenamiento y muchas clases (46), por lo que lo consideramos aceptable.