Módulo 4: Aprendizaje Automático

Temario de la Clase 7
20 de septiembre del 2024
Redes neuronales artificiales: optimizadores

- Consideraciones matemáticas
- Descenso del gradiente, SGD y por lotes
- Optimizadores
- Adagrad
- RMSProp
- Adadelta
- Adam
- Comparación

Introducción

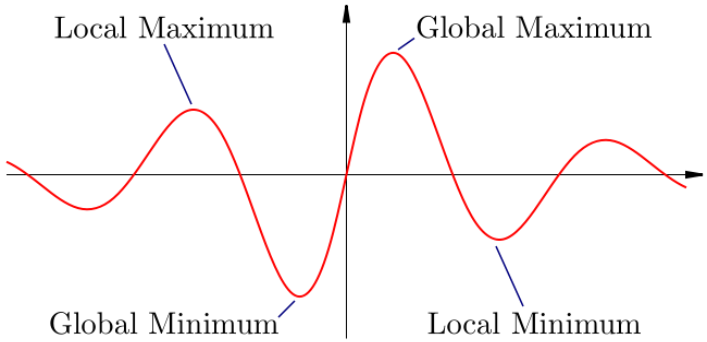
Como vimos anteriormente, los algoritmos de aprendizaje supervisado computan una función de error en base a los resultados de los modelos y los targets de los datasets. Cuando vimos el concepto de regularización ampliamos la idea de función de error a la de función objetivo.

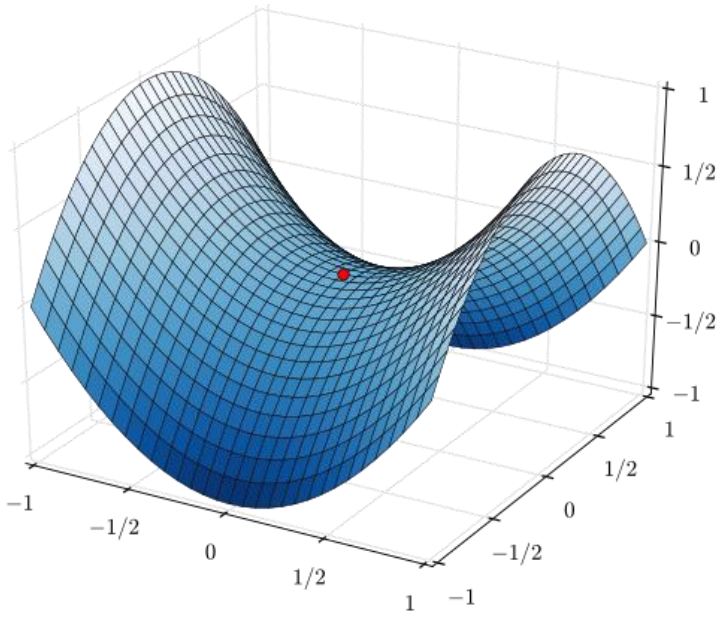
En todos los casos, la función objetivo es una función a **minimizar**. Encontrar el valor mínimo de una función multivariada es un problema muy estudiado en análisis matemático y ha inspirado distintos tipos de soluciones.

En ML buscamos encontrar métodos que, partiendo de un punto inicial de la superficie de la función objetivo, podamos acercarnos de forma sucesiva hacia un mínimo global.

Introducción

El problema radica en que, por lo general, las funciones objetivo no se comportan como funciones convexas, donde encontrar puntos mínimos se puede realizar de manera relativamente sencilla. Las funciones objetivo que se computarán a partir de nuestros algoritmos se comportan mas bien como funciones no convexas, que pueden tener distintos puntos que se comportan como mínimos locales, puntos de ensilladura (donde la derivada se anula pero no son máximos ni mínimos locales) y gradientes que desaparecen. Para este tipo de situaciones no se pueden encontrar soluciones analíticas, por lo que se recurre a algoritmos de optimización.





Introducción

Sin embargo, debemos tener en cuenta que, si bien la optimización tiene el objetivo de reducir los parámetros de error de un modelo (error de entrenamiento), el objetivo del ML o del Deep learning es el de encontrar un **modelo adecuado**, dado un conjunto finito de datos (error de generalización).

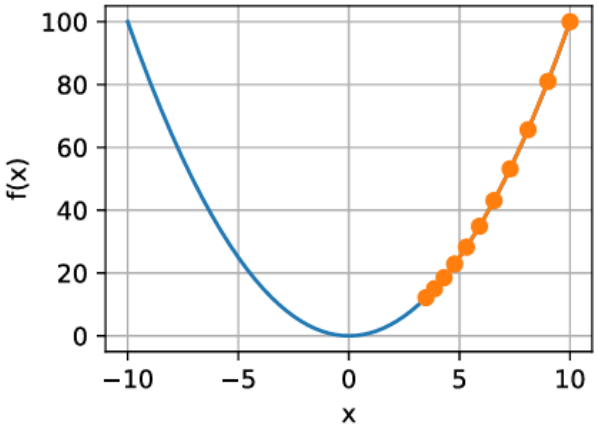
Minimizar el error de entrenamiento no nos asegura encontrar el mejor conjunto de parámetros para minimizar el error de generalización.

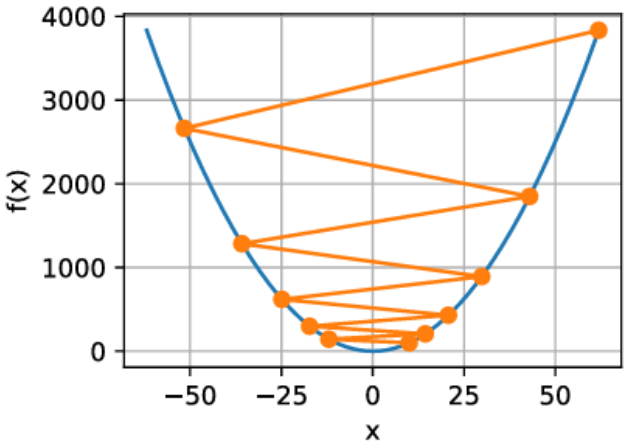
Podemos decir que no es necesario encontrar la solución exacta a la hora de optimizar, ya que puede que sea útil una optimización local o una solución aproximada.

Descenso del gradiente

En el método del descenso del gradiente (GD), uno de los parámetros más difíciles de definir era el **learning rate** (η):

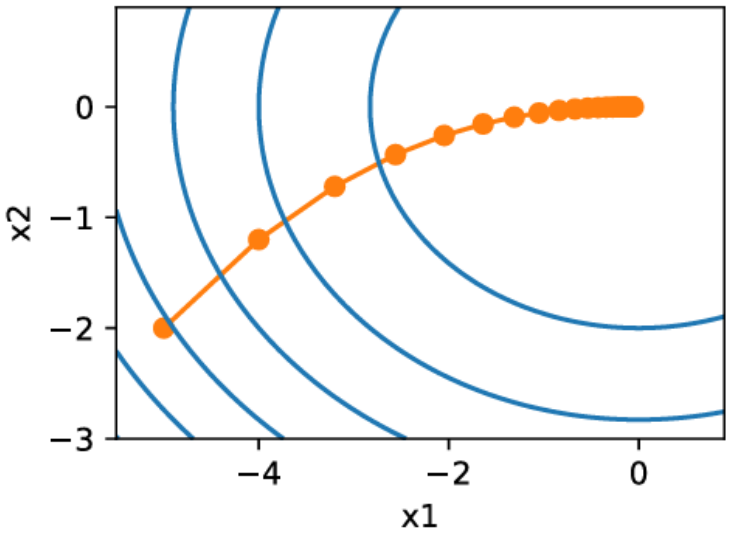
- Un valor muy bajo podía hacer que el entrenamiento se produzca de forma muy lenta y no se llegue a un mínimo deseable en un número controlado de iteraciones.
- Un valor muy alto podía hacer que se tomen pasos de actualización muy grandes y que los valores encontrados en la actualización no convergan hacia ningún lado o comúnmente, el algoritmo diverja.





Descenso del gradiente

Por lo tanto, encontrar el learning rate apropiado suele ser complejo, sobre todo en caso donde hay muchos parámetros para entrenar. En casos sencillos, en pocas epochs se puede encontrar el mínimo.



Una desventaja del GD es que para hacer una actualización, se debe analizar el dataset entero para promediar el gradiente. Esto hace que la complejidad computacional de dicho algoritmo crezca linealmente con el tamaño del dataset.

Descenso del gradiente estocástico

El descenso del gradiente estocástico (SGD) reduce el costo computacional de cada iteración al analizar solamente un ejemplo del dataset para calcular el gradiente.

Sin embargo, SGD no es eficiente ya que no explota la capacidad de las CPU y GPU de aprovechar la vectorización en los cálculos. Procesar un ejemplo a la vez es más costoso que procesar varios ejemplos de forma conjunta.

Mini batch SGD

La eficiencia computacional aumenta cuando utilizamos lotes de datos sobre los cuales realizar las operaciones necesarias.

Si tenemos un lote de tamaño B, la forma de calcular una de las componentes del gradiente es la siguiente:

$$g_t = \partial_w \frac{1}{B} \sum_{i=1}^{B} f(x_i, w)$$

Donde f denota la función objetivo y ∂_w denota la derivada parcial respecto del parámetro a optimizar. El gradiente tendrá tantas componentes como parámetros tenga el modelo.

Momentum

Una técnica propuesta para mejorar la convergencia del entrenamiento por mini-lotes consiste en actualizar los parámetros del modelo, teniendo en cuenta no solo el gradiente calculado en la iteración actual, sino también retener una cierta "memoria" de gradientes pasados.

Para ello podemos introducir una nueva variable llamada *velocidad*:

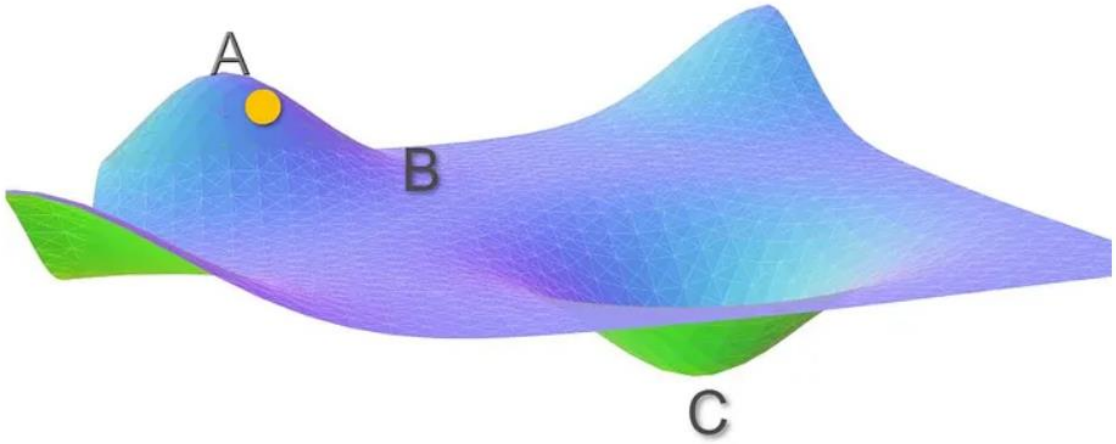
$$v_t = \beta v_{t-1} + g_t$$

Entonces la velocidad de define de forma recursiva y tiene en cuenta valores pasados del gradiente en los tiempos t-1, t-2, t-3, etc. El parámetro β se define en el rango (0,1).

$$v_t = \beta^2 v_{t-2} + \beta g_{t-1} + g_t = \sum_{\tau=0}^{t-1} \beta^\tau g_{t-\tau}$$

Momentum

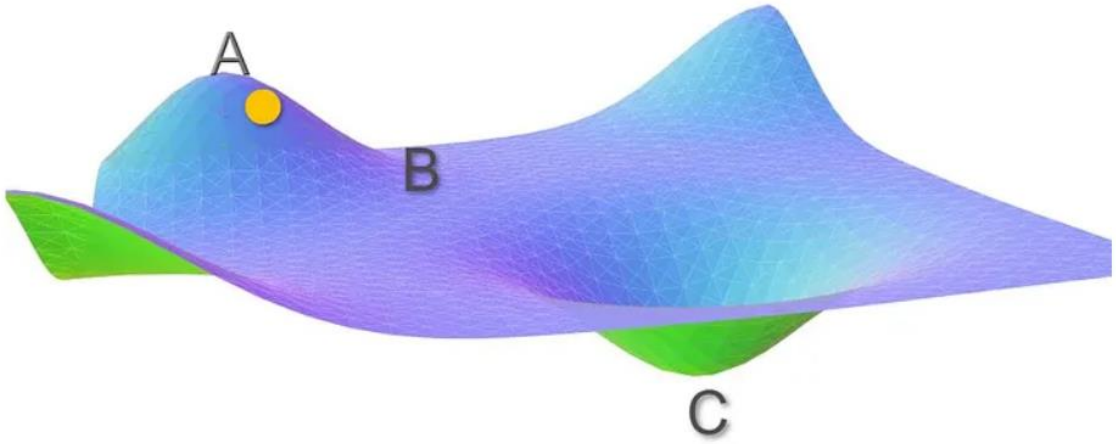
Podemos ver el concepto detrás del momentum con un ejemplo:



Si el algoritmo comienza en el punto A, puesto que la pendiente en el segmento AB es constante, el descenso del gradiente puede llegar rápidamente al punto B. Sin embargo, en el entorno de B, la pendiente disminuye rápidamente.

Momentum

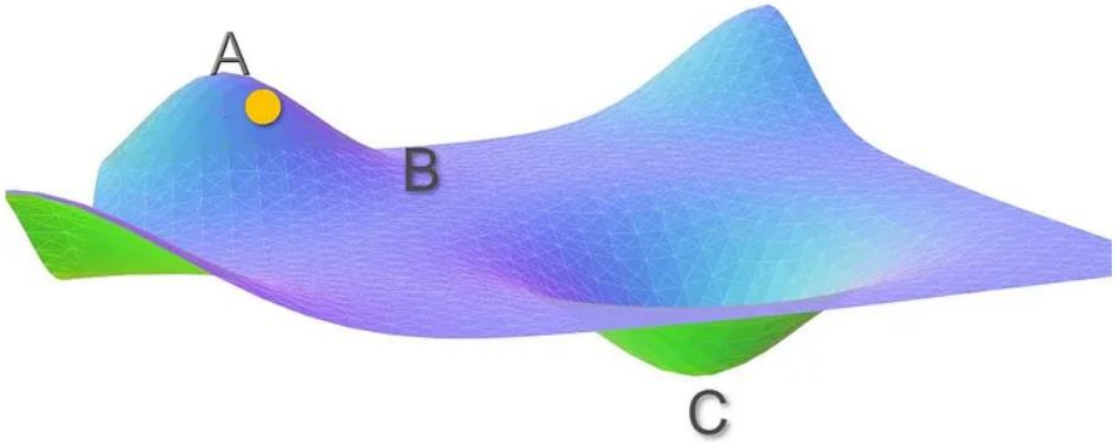
Podemos ver el concepto detrás del momentum con un ejemplo:



Al disminuir el gradiente en esa zona, las actualizaciones que se hagan por los métodos vistos hasta el momento serán muy pequeñas, por lo que será muy difícil moverse de ese punto, incluso luego de varias iteraciones.

Momentum

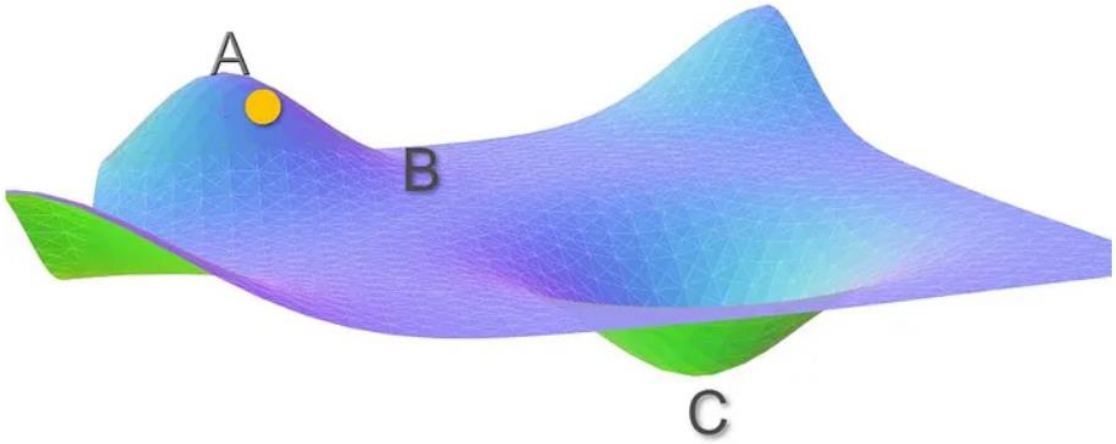
Podemos ver el concepto detrás del momentum con un ejemplo:



Idealmente, queremos que nuestro algoritmo nos lleve hasta el punto C, que es el mínimo global de la función objetivo. Puesto que el gradiente desaparece en un entorno de B, nos quedamos con un resultado poco satisfactorio.

Momentum

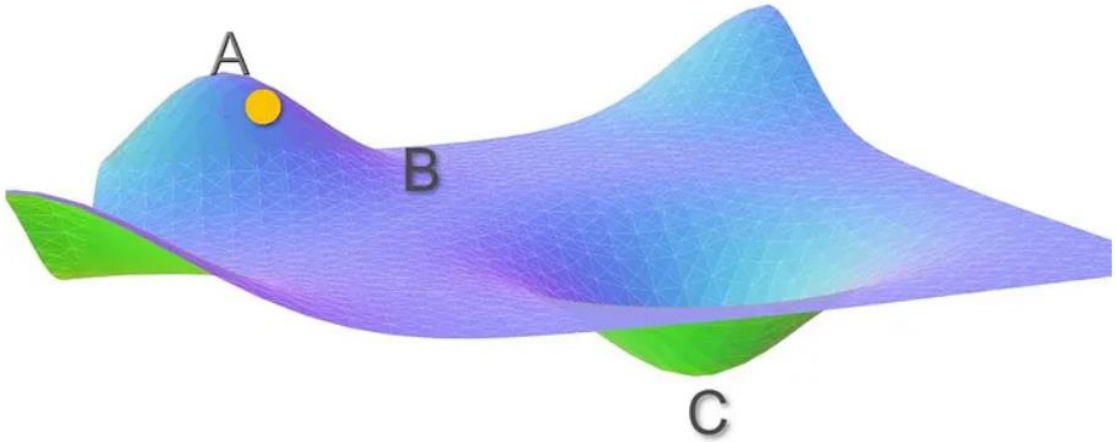
Podemos ver el concepto detrás del momentum con un ejemplo:



La idea del momentum consiste en que al desplazarnos por la pendiente AB, el algoritmo tome cierta confianza en esa dirección del gradiente y por lo tanto al llegar a B, el *momentum* que tiene acumulado sea suficiente para desplazarse por esa región y atravesarla, para luego seguir el camino más pronunciado desde B hasta C.

Momentum

Podemos ver el concepto detrás del momentum con un ejemplo:

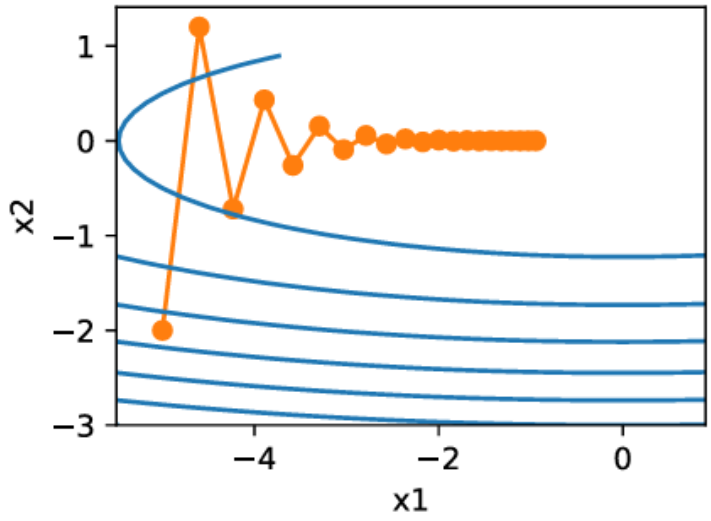


Claramente, para ello, el algoritmo debe almacenar cierta memoria de los valores anteriores del gradiente. Esto va a aumentar la complejidad computacional del algoritmo pero con el beneficio de asegurar una mejor convergencia de la solución. Se dice que hay una *aceleración* de la convergencia.

Momentum

Veamos ahora un ejemplo de una función objetivo con una característica muy plana en una dimensión. Sea la función: $f(x) = 0.1x_1^2 + 2x_2^2$ con un mínimo en el punto (0,0).

El gradiente de x_2 tendrá una variación más pronunciada en comparación con x_1. Por lo tanto la convergencia del algoritmo se mueve más rápido en la dirección vertical que en la dirección horizontal. Con un learning rate de 0.4, luego de 20 epochs se llega a la solución:



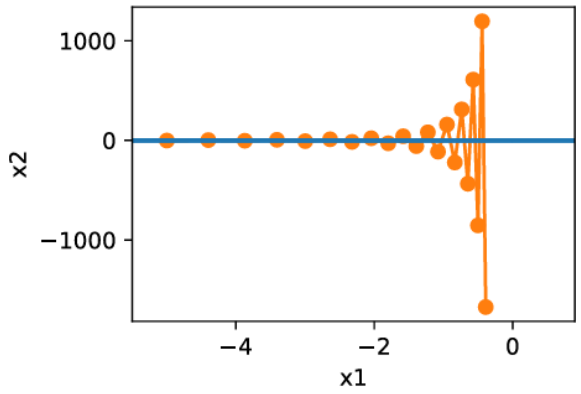

```
epoch 20, x1: -0.943467, x2: -0.000073
```

Momentum

Para el learning rate establecido, la convergencia del parámetro x_2 trae aparejado el hecho de que el parámetro x_1 tenga una convergencia lenta y no llegue a su valor óptimo.

Por otra parte, si se aumenta el learning rate de 0.4 a 0.6, tendremos que x_1 converge a 0 más rápidamente pero hay una marcada divergencia para x_2 .

```
epoch 20, x1: -0.387814, x2: -1673.365109
```



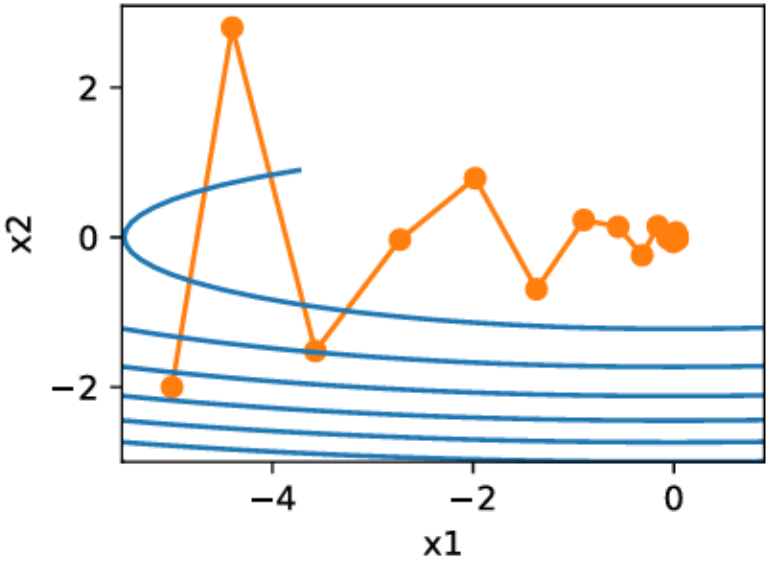
Momentum

El introducir la noción de momentum en las ecuaciones puede resolver el problema descripto anteriormente. Si ahora hacemos la actualización de los parámetros en términos del vector de velocidad en lugar de hacerlo directamente en términos del gradiente tenemos:

$$v_t = \beta v_{t-1} + g_t$$
$$x_t = x_{t-1} - \eta v_t$$

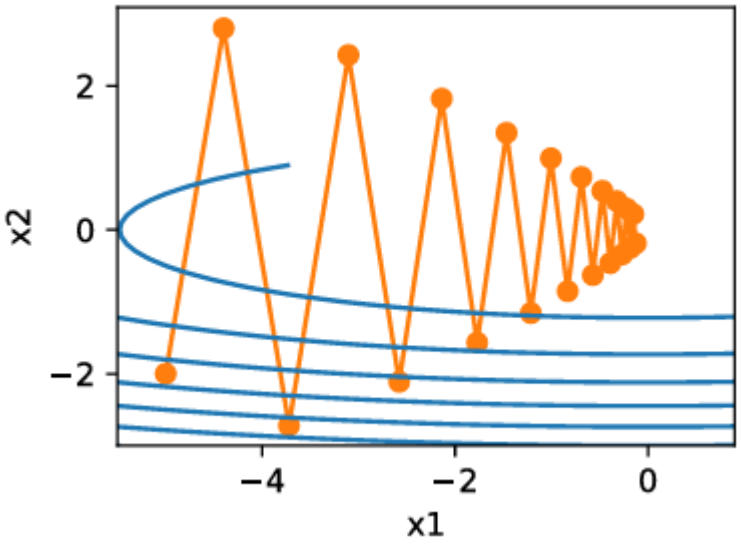
Con β=0.5 y η=0.6 nuestro problema converge de una forma más satisfactoria.

```
epoch 20, x1: 0.007188, x2: 0.002553
```



Momentum

Si β=0, las ecuaciones se reducen al descenso del gradiente clásico. Valores más grandes para β indican un promediador móvil de mayor rango. Un valor más pequeño descarta más rápidamente los valores anteriores del gradiente. Incluso con β=0.25, el algoritmo parece actuar de forma más errática pero de igual manera converge a la solución buscada. Esto contrasta mucho con las soluciones halladas sin momentum.

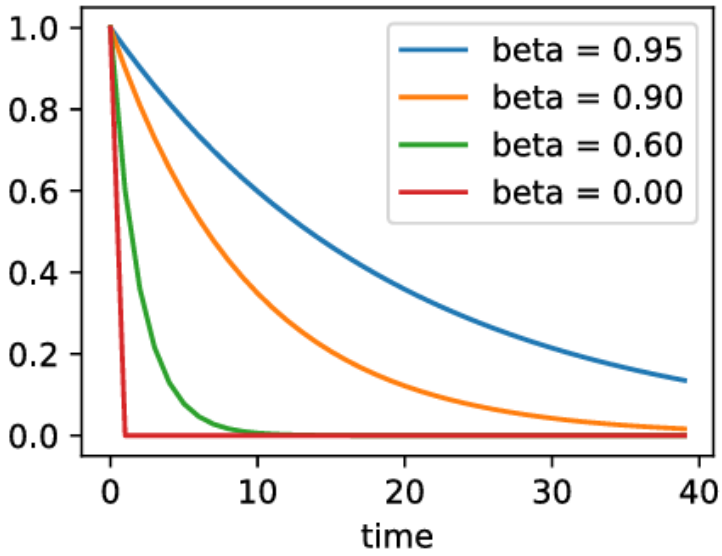



```
epoch 20, x1: -0.126340, x2: -0.186632
```

Momentum

El momentum puede incorporarse a las implementaciones de GD y SGD definiendo el valor inicial v=0.

En el siguiente gráfico puede verse, para distintos valores de β, de qué manera los valores anteriores del gradiente se ven afectados por el factor β^{τ}:



Momentum

De esta forma, si β = 0.1, en la iteración n=3; el gradiente en t =3 contribuirá el 100% del valor, el gradiente en t=2 contribuirá el 10% del valor, y el gradiente en t=1 contribuirá el 1% de su valor.

Si β =0.9, en la iteración n=3; el gradiente en t =3 contribuirá el 100% del valor, el gradiente en t=2 contribuirá el 90% del valor, y el gradiente en t=1 contribuirá el 81% de su valor.

Entonces un mayor valor de β retendrá mayor información de los gradientes anteriores.

$$v(0) = 0$$

$$v(1) = \beta * v(0) + (1-\beta) * \delta(1)$$
$$v(1) = (1-\beta) * \delta(1)$$

$$v(2) = \beta * v(1) + (1-\beta) * \delta(2)$$
$$v(2) = \beta * \{(1-\beta) * \delta(1)\} + (1-\beta) * \delta(2)$$
$$v(2) = (1-\beta)\{\beta * \delta(1) + \delta(2)\}$$

$$v(3) = \beta * v(2) + (1-\beta) * \delta(3)$$
$$v(3) = \beta * \{(1-\beta)\{\beta * \delta(1) + \delta(2)\}\} + (1-\beta) * \delta(3)$$
$$v(3) = (1-\beta)\{\beta^2 * \delta(1) + \beta * \delta(2) + \delta(3)\}$$

$$v(n) = (1-\beta) \sum_{t=1}^{n} \beta^{n-t} \delta(t)$$

Optimizadores

Así como introdujimos el concepto de momentum para mejorar la convergencia de los algoritmos, en presencia de comportamientos no convexos, podemos introducir distintas variaciones del modelo básico del **descenso del gradiente** para hacer que nuestros modelos de ML aprendan de forma más organizada.

A estas versiones modificadas se las denomina **optimizadores** y son el resultado de añadir consideraciones y nuevos parámetros y variables al proceso de entrenamiento de los modelos.

En la actualidad, existe un gran número de optimizadores y la mayoría de ellos están implementados en las librerías de uso libre disponibles, como Pytorch y TensorFlow.

Optimizadores

Los algoritmos optimizadores fueron introducidos por científicos de ciencia de datos en papers de divulgación para resolver problemas específicos.

Por lo general, uno de los hiperparámetros más difícil de ajustar es el learning rate.

La solución demostrada por el momentum funcionó bien en los ejemplos mostrados ya que, en cierta forma, adaptó el learning rate para cada parámetro de manera independiente.

Optimizadores

Vamos a discutir acerca de 4 de los optimizadores más conocidos:

- Adagrad
- RMSProp
- Adadelta
- Adam

Adagrad

Supongamos el hecho de que tenemos un dataset con características dispersas. Es decir, aquellas que representan variables que no suelen aparecer comúnmente o lo hacen de forma ocasional. Estas características tomarán valores iguales a 0 la mayoría de las veces.

Por ejemplo, en un dataset para modelos de lenguaje, la palabra *Adagrad* aparecerá menos veces que la palabra *entrenamiento*; o en un modelo recomendador de películas, un documental surcoreano será menos probable de encontrar que una comedia estadounidense .

Por lo general, los parámetros asociados a features poco frecuentes solo se actualizan significativamente cuando el algoritmo se topa con dichos ejemplos.

Adagrad

Si se estableciere un learning rate variable que vaya decayendo en el tiempo (como es común en ciertos optimizadores), puede que los parámetros asociados a features frecuentes se optimizen rápidamente mientras que los asociados a features poco frecuentes no se lleguen a ajustar porque no se alcanzó a observarlos lo suficiente.

Es por ello que vamos a introducir la noción de un learning rate "variable" asociado a las veces que una determinada feature se observó. Puesto que determinar si una feature se observó o no es una definición bastante ambigua, se establece un criterio de memoria asociado al segundo momento del gradiente.

En otras palabras, buscamos implementar un algoritmo que establezca un learning rate que decaiga rápidamente para features frecuentes y más lentamente para features poco frecuentes.

Adagrad

Puesto que importa la ocurrencia del gradiente más que el signo del mismo, tomamos el cuadrado del mismo como parámetro.

Definimos una nueva variable de estado de forma similar a la velocidad:

$$s_t = s_{t-1} + g_t^2$$

Dicha variable tendrá un valor mayor para features densas y menor valor para features dispersas. Podemos definir entonces un nuevo learning rate:

$$\eta_t = \frac{\eta}{\sqrt{s_t + \varepsilon}}$$

Donde ε es un valor pequeño para asegurar que no dividamos por cero.

Adagrad

El optimizador **Adagrad** (Adaptive Gradient) introducido en 2011, se caracteriza entonces por tener la siguiente regla de actualización de parámetros:

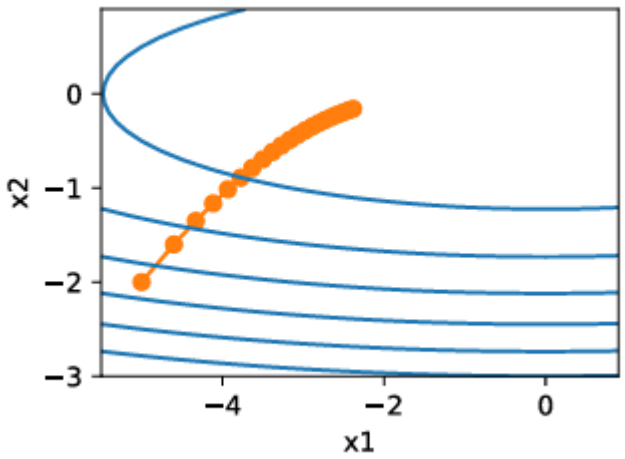
$$s_t = s_{t-1} + g_t^2$$

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{s_t + \varepsilon}} g_t$$

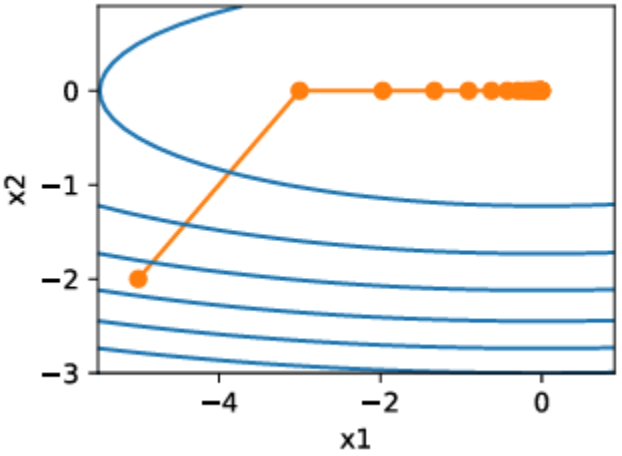
Donde η es el learning rate y ε suele ser un valor tomado como 10^{-6} o 10^{-7} para no perder estabilidad computacional. Se caracteriza por tener un learning rate adaptado de forma dinámica a cada feature (coordenada), de modo que gradientes grandes se compensan con learning rates mas pequeños.

Adagrad

Para el problema de optimización de la función anterior podemos ver los resultados con Adagrad para η=0.4 y un η=2.




```
epoch 20, x1: -2.382563, x2: -0.158591
```




```
epoch 20, x1: -0.002295, x2: -0.000000
```

RMSProp

El algoritmo **RMSProp** (Root Mean Square Propagation) fue introducido en 2012 para resolver uno de los problemas de Adagrad. Puesto que la suma de gradientes pasados se hace sobre el valor cuadrático de los mismos, la suma es estrictamente positiva y por lo tanto tiende a hacer decaer rápidamente el learning rate para características comunes.

Una solución es proponer una variable de estado basada en el momentum para que los gradientes mas recientes tengan mayor influencia que gradientes anteriores, y de esa forma limitar el decaimiento del learning rate.

$$s_t = \gamma s_{t-1} + (1-\gamma) g_t^2$$

RMSProp

$$s_t = \gamma s_{t-1} + (1-\gamma) g_t^2$$

$$s_t = (1-\gamma)(g_t^2 + \gamma g_{t-1}^2 + \gamma^2 g_{t-2}^2 + \cdots)$$

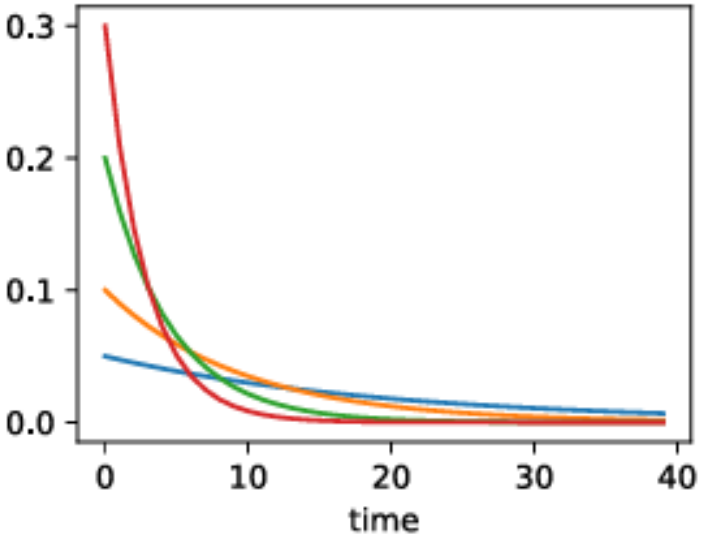
Dicha expresión se denomina *promediador móvil exponencial* debido a que establece una suma ponderada de los factores (en este caso gradientes) afectados por un parámetro de decaimiento (γ >0).

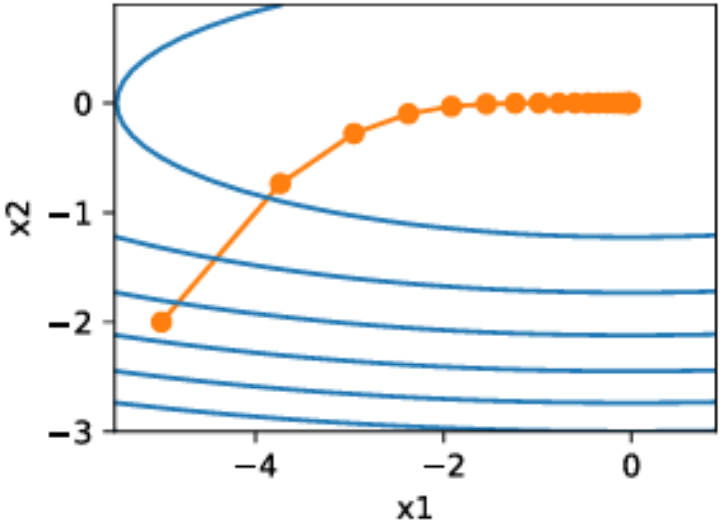
Y los parámetros se actualizan de forma similar a Adadelta según:

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{s_t + \varepsilon}} g_t$$

RMSProp

Para distintos valores de γ (0.95, 0.9, 0.8 y 0.7) la influencia de los últimos 40 gradientes será:






```
epoch 20, x1: -0.010599, x2: 0.000000
```

Además podemos ver la convergencia con RMSProp para el problema de optimización antes descripto.

Adadelta

Es otra variante de Adagrad, propuesta en 2012. Utiliza dos variables de estado: una para calcular un promediador móvil exponencial del segundo momento del gradiente (igual que RMSProp), y otra para calcular un promediador móvil de la variación misma de los parámetros del modelo.

$$s_t = \gamma s_{t-1} + (1-\gamma) g_t^2$$
$$\Delta x_t = \gamma \Delta x_{t-1} + (1-\gamma) g_t'^2$$

Los parámetros se actualizan según:

$$w_t = w_{t-1} - \frac{\sqrt{\Delta x_{t-1} + \varepsilon}}{\sqrt{s_t + \varepsilon}} g_t$$

Y g_t' es la actualización realizada sobre el parámetro.

Adadelta

Como puede verse, a diferencia de los métodos anteriores, Adadelta no utiliza un learning rate, sino que utiiza los valores anteriores del gradiente y de las actualizaciones de los parámetros para modular la tasa de aprendizaje.

Debido a que se define una nueva variable de estado, aumenta la complejidad computacional.

Adam

El optimizador Adam (Adaptive Moment Estimation), introducido en 2014 busca combinar las bondades de los optimizadores anteriores. Utiliza el promediador móvil exponencial para calcular tanto el *momentum* como el segundo momento del gradiente. Las variables de estado serán:

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t$$

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) g_t^2$$

Valores comunes son 0.9 para β_1 y 0.999 para β_2. Los valores iniciales de estas variables están sesgados hacia 0, por lo que se aplica un factor de normalización:

$$v'_t = \frac{v_t}{1 - \beta^t}$$

$$s'_t = \frac{s_t}{1 - \beta^t}$$

Adam

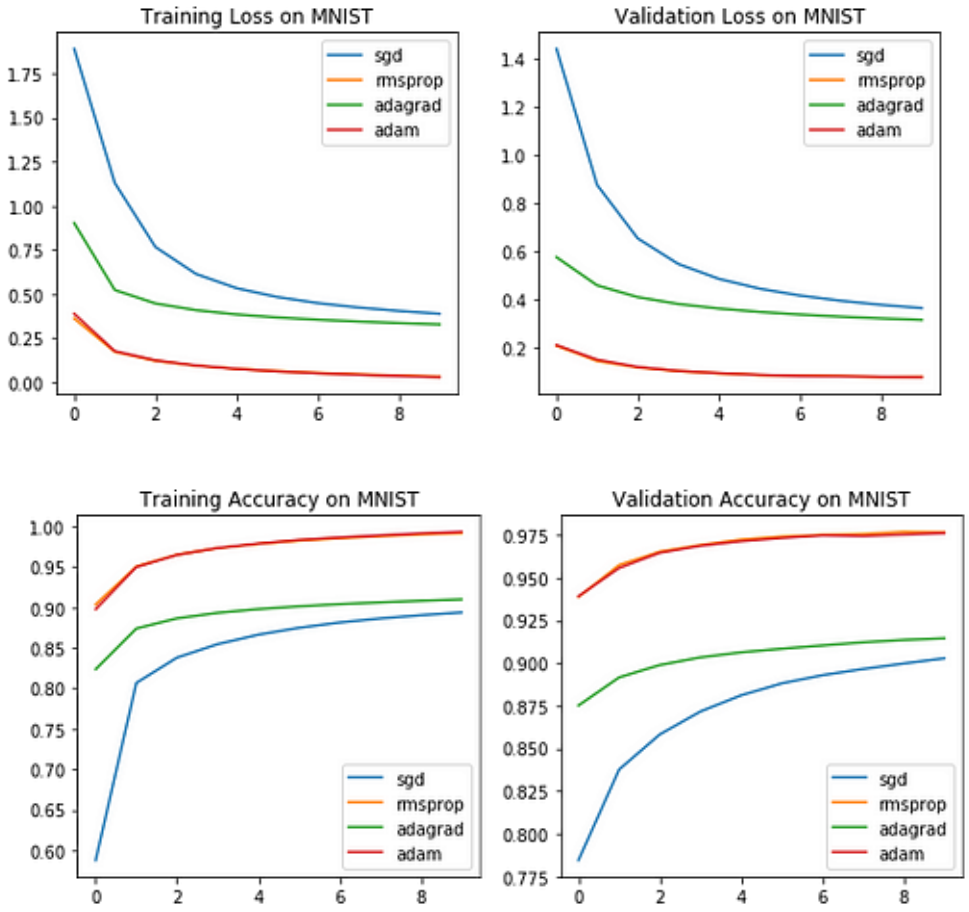
Ahora la actualización de parámetros se realiza de forma:

$$w_t = w_{t-1} - \frac{\eta v'_t}{\sqrt{s'_t + \varepsilon}}$$

Como se ve, a diferencia del gradiente, se utiliza la velocidad para actualizar los parámetros. Adam combina lo mejor de Adagrad y RMSprop, ya que adapta la tasa de aprendizaje para cada parámetro y es menos sensible a la elección del valor inicial de la misma. En general converge más rápido que muchos otros optimizadores y funciona bien en una variedad de problemas de aprendizaje profundo.

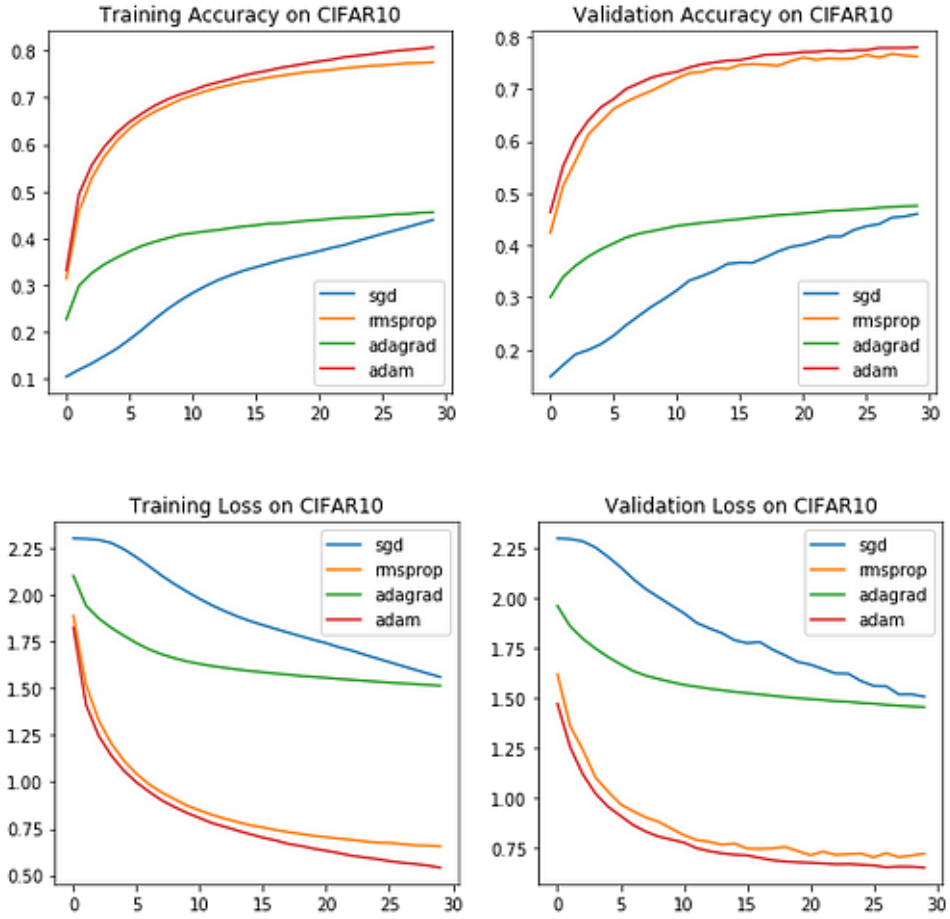
Comparación de optimizadores

Sobre el dataset MNIST (dataset pequeño):



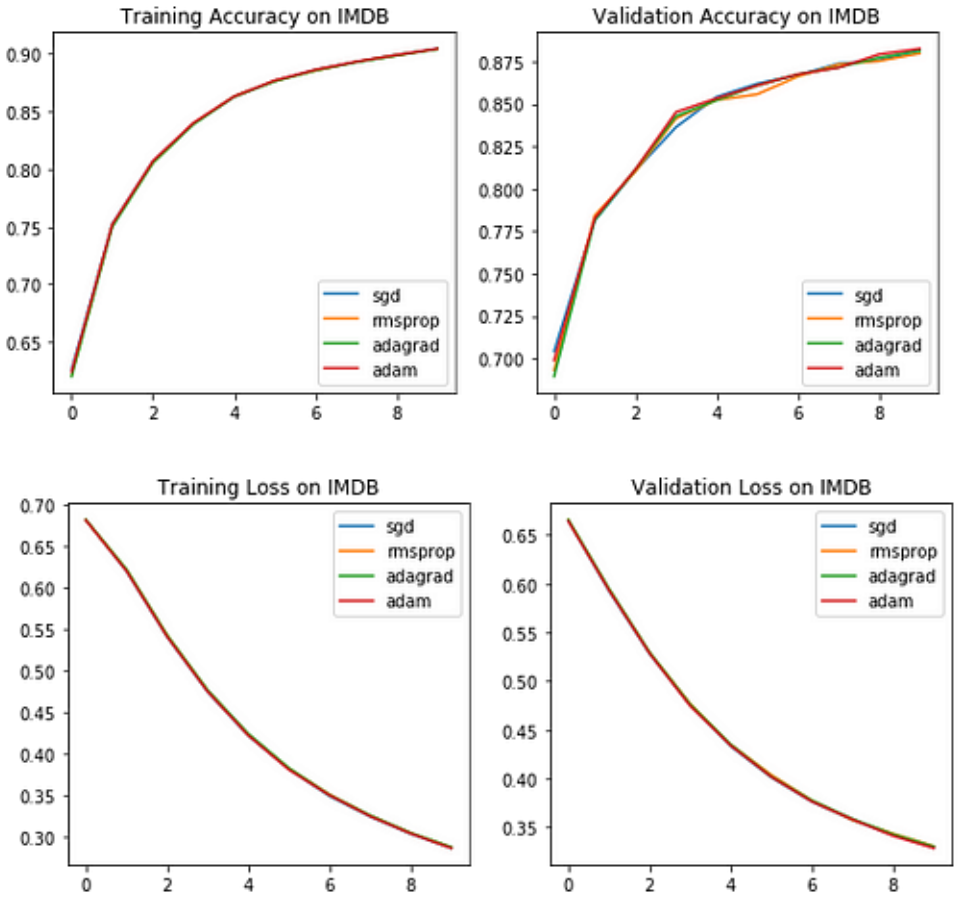
Comparación de optimizadores

Sobre el dataset CIFAR10 (dataset más grande):



Comparación de optimizadores

Sobre el dataset IMDB (lenguaje natural, 8000 palabras):



Comparación de optimizadores

- Comparación 1
- Comparación 2
- Comparación 3