Optimización numérica y su aplicación en Machine Learning

Alcance:

- Conocer el problema de la optimización y su terminología.
- Reconocer las principales variantes del método de Gradiente Descendente (Normal, Batch, Mini, Stochastic).
- Implementar y analizar sus ventajas con `keras`.

Hasta el momento, el curso ha sido una exposición exhaustiva a distintos tipos de algoritmo y cómo éstos resuelven los problemas canónicos de regresión y clasificación. Sin embargo, hemos omitido un aspecto esencial sobre el aprendizaje estadístico: *¿Cómo las máquinas son capaces de aprender los parámetros de un modelo?* Este tema resulta ser vital para comprender cómo funciona una red neuronal. Dependiendo de cómo construyamos una, su desempeño puede variar de manera substancial dependiendo del método de optimización que implementemos.

En esta sección estudiaremos los principales mecanismos de optimización que se utilizan para entrenar redes neuronales.

¿Qué son y por qué necesitamos algoritmos de optimización?

Como lo indica su nombre, un algoritmo de optimización permite encontrar o aproximarnos al óptimo de una función objetivo de un algoritmo.

Un algoritmo de minimización permite encontrar o aproximar el óptimo de una función, llamada por lo general **función objetivo del algoritmo**. El *'óptimo'* de la función objetivo depende de lo que estemos haciendo. Si estamos minimizando una función de pérdida o costo, el óptimo que buscará el algoritmo serán los parámetros del modelo que hacen que esa función de costo tome el menor valor posible. El procedimiento es análogo cuando queremos máximizar una función de ganancia (por ejemplo la verosimilitud).

Existen distintas formas para encontrar el óptimo de una función, comenzaremos con el más intuitivo y en el que se basan la mayoría de los algoritmos de optimización actuales: Los métodos basados en gradiente

Métodos basados en gradiente

Gradiente Descendente clásico (*Gradient Descent*)

Imaginemos que estamos en un parque el cual tiene un terreno bastante irregular, con colinas y llanos. Nosotros estamos parados en un punto cualquiera y nuestro objetivo es llegar al punto más bajo del parque (vamos a minimizar). Existe un pequeño detalle, tenemos los ojos vendados, por lo que no podemos ver hacia adonde nos conviene movernos. Lo único que podemos hacer es sentir la inclinación del terreno bajo nuestros pies.

Una aproximación lógica a este problema es irnos siempre por la dirección de mayor descenso desde el punto en el que estamos parados. La aproximación que utilizan los métodos basados en gradiente es la misma: Calculan, en un cierto punto, el **gradiente** de la función objetivo en ese punto y eligen la dirección que tiene *"mayor inclinación"*.

Consideremos el clásico problema de ajustar una regresión lineal a un conjunto de datos. Recordando la primera unidad del curso, nuestro modelo de regresión tiene la siguiente forma:

$$\hat{y}(\gamma, \beta) = \gamma + \beta \cdot x$$

Donde:

γ

es el intercepto.

β

es la pendiente.

Los parámetros de este modelo (y que definen la curva) son el intercepto

γ

y la pendiente

β

Si trabajamos bajo la supuesto que los datos en efecto siguen un comportamiento lineal, nuestro problema es encontrar los parámetros análogos a

γ

y

β

que hagan que la curva se ajuste a los datos. Para simplificar el ejemplo, asumiremos que los datos están centrados en **0** y nuestro problema solo está en encontrar el valor correcto de

β

Ahora debemos definir una función de costo, es decir, el **cómo** vamos a penalizar al modelo cuando proponga una combinación de parámetros que no se ajuste a los datos. Utilizaremos la suma de los cuadrados de la regresión como **función de costo**:

$$\text{RSS} = J(\beta) = \sum_{i=0}^{N} (y_i - \hat{y}(\beta)_i)^2$$

El algoritmo de gradiente descendente lo que hará es una actualización iterativa del parámetro que estamos tratando de estimar

$$\beta$$

De esta forma se encontrarán valores que minimicen la función de costo del modelo. En cada iteración **j**, el modelo evaluará la función de costo en el parámetro actual estimado y analizará, basándose en la gradiente de la función de costo, hacia qué valores del parámetro la función de costo parece decrecer más.

Formalmente, si

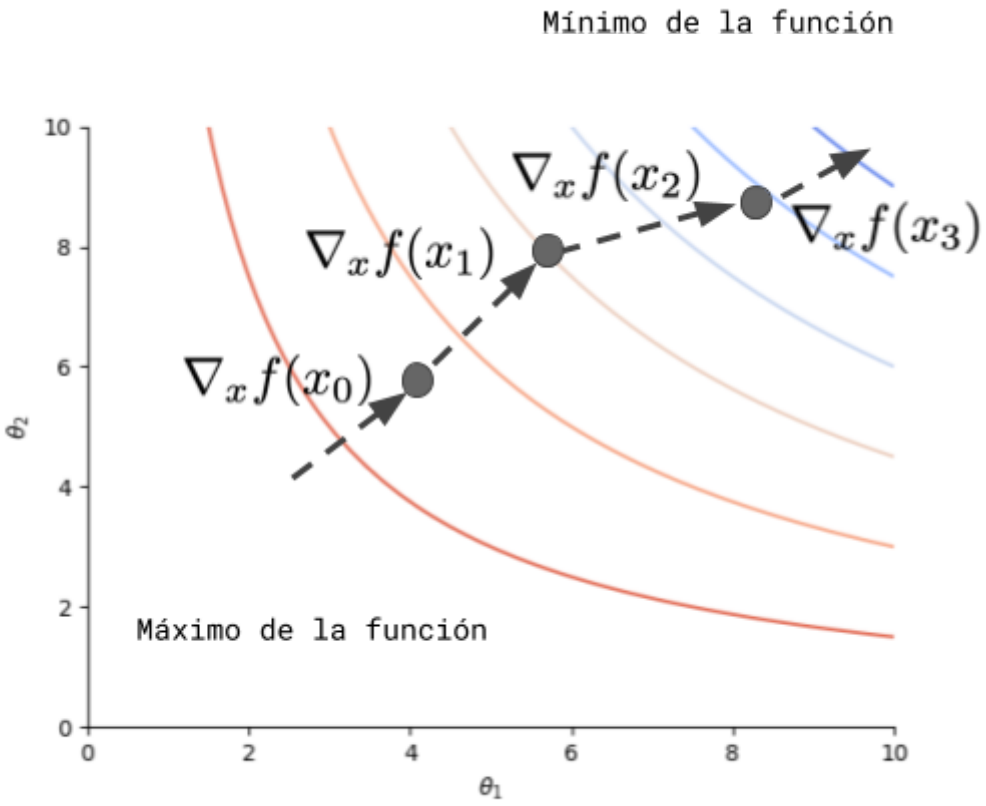
$$\theta$$

son los parámetros del modelo que estemos entrenando y

$$E[J(\theta)]$$

el valor esperado de la función de costo elegida evaluada en los parámetros actuales, la actualización de parámetros que implementa gradiente descendente es de la siguiente manera:

$$\hat{\theta}_i \leftarrow \hat{\theta}_i - \alpha \nabla_\theta \mathbb{E}[J(\theta)]$$



El coeficiente del gradiente

$$\alpha$$

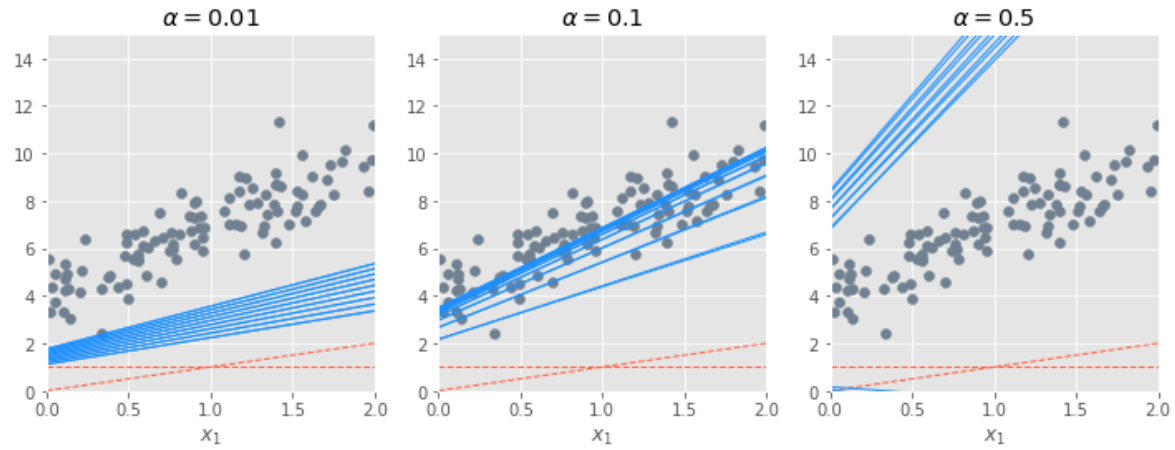
se llama ***learning rate*** (tasa de aprendizaje, abreviado también como `lr` en `keras`) y es un aspecto clave de este método, pues dicta *qué tanto* va a cambiar el parámetro en cada iteración. Un learning rate alto provoca que el método se mueva a saltos grandes en el espacio de búsqueda, por otro lado, un learning rate bajo provoca que este avance a saltos pequeños.

Efecto del learning rate

Para ejemplificar el efecto del Learning Rate, tomemos el siguiente ejemplo original de *Geron, A. 2017. Hands On Machine Learning*:

```
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
from opti_graphs import *
import matplotlib.pyplot as plt
from ipywidgets import interactive
import numpy as np
plt.style.use('ggplot')
```

```
# Generamos un conjunto de datos artificial
X, y = artificial_points()

plt.figure(figsize=(10, 4))
# compararemos el efecto del descenso de gradiente batch en tres niveles
for i, n in enumerate([0.01,0.1, 0.5]):
    plt.subplot(1, 3, i + 1)
    batch_gd_plot(X, y, theta=1, alpha=n)
    plt.tight_layout()
```



En la simulación generada, evaluamos el comportamiento del algoritmo en función a la tasa de aprendizaje. Cuando es muy bajo

$$\alpha = 0.01$$

el algoritmo podrá alcanzar la solución eventualmente, pero se demorará. Cuando el valor es relativamente alto

$$\alpha = 0.5$$

el algoritmo diverge respecto a los datos y su comportamiento se torna errático, fallando en la solución. Al implementar un valor mediano, en este caso el algoritmo alcanza una representación más cierta de los datos por cada iteración

Dificultades de GD clásico

- El *learning rate* es un hiperparámetro, lo que significa que probablemente nuestra mejor apuesta a encontrar valores óptimos de este sea mediante heurísticas (e.g. La intuición nos dice que quizás no sea conveniente usar un

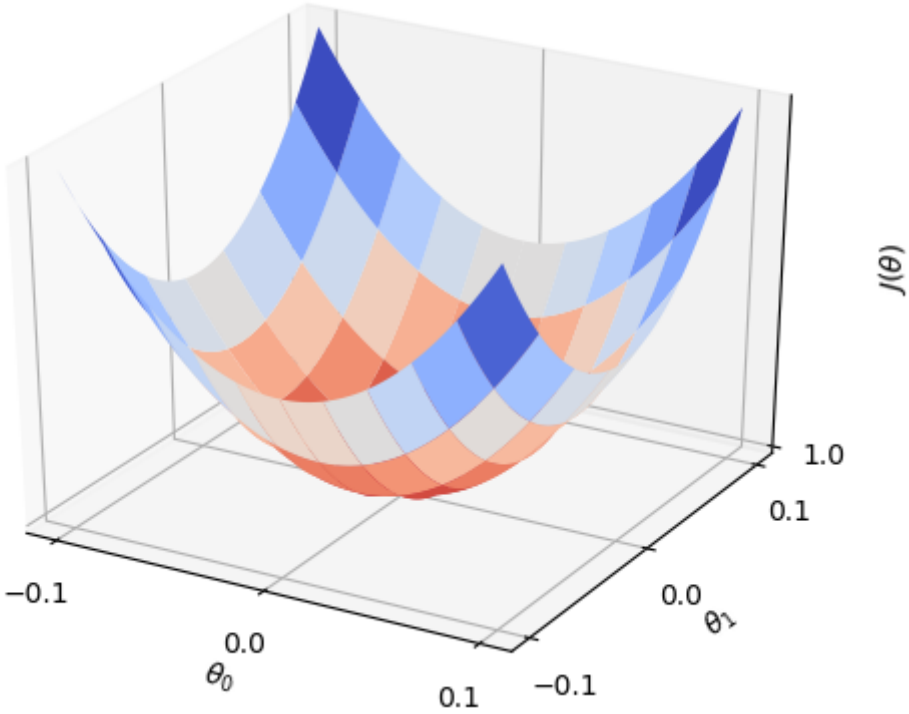
$$lr = 10^{20}$$

- Aún cuando encontremos un valor razonable para el learning rate, la superficie del espacio de búsqueda puede tener una topología complicada, haciendo que nuestro **lr** se desempeñe bien en ciertos lugares y pésimo en otros. Este problema está asociado con la posibilidad de encontrar óptimos locales.
- Una característica especialmente complicada de los espacios de búsqueda son las llamadas *plateau*, las cuales representan valles, rodeados por máximos o mínimos locales, para un algoritmo basado en gradiente este tipo de geografías son mortales, pues no le dan información de hacia adonde ir en ningún punto.
- Aplicar el mismo learning rate a todos los atributos es poco flexible, lo ideal sería que el optimizador pudiese elegir cuanta importancia darle a los movimientos realizados en cada atributo para buscar la solución.
- En su forma clásica, gradiente descendente utiliza todo el conjunto de entrenamiento para efectuar una sola actulización de los parámetros del modelo, lo que lo hace poco eficiente cuando se quiere trabajar con las cantidades de datos que manejamos usualmente.

Digresión: Optimización Convexa

Los algoritmos de descenso de gradiente forman parte del estudio de la **optimización convexa**. Todo espacio o superficie de un problema que tenga forma de pozo se conoce como un espacio convexo. El objetivo es encontrar un mínimo/máximo en toda la superficie de la función. Partamos por un ejemplo donde existe un mínimo **global**, generado con la función `global_surface`.

```
plt.style.use('default'); global_surface()
```



En esta situación, existe un punto mínimo donde los valores de

$$\theta_0$$

y

$$\theta_1$$

minimizan la función objetivo

$$J(\theta)$$

Ésta zona está caracterizada con un color rojo. En este ejemplo, nuestro algoritmo considerará los valores de

$$\theta_0$$

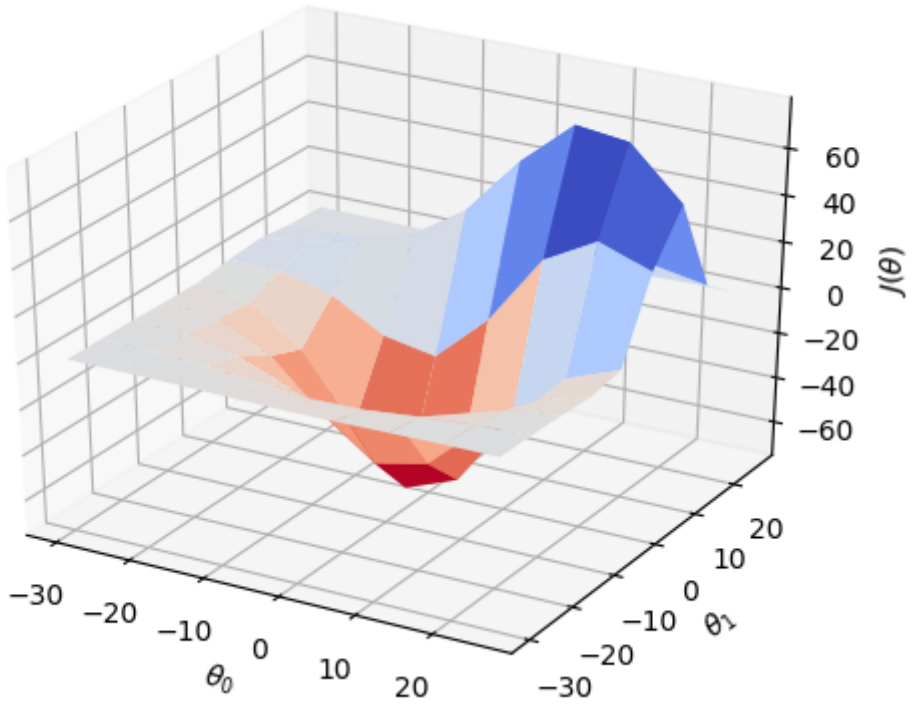
y

θ_1

cuando estén cercanos a cero, dado que es en ésta combinación de parámetros donde se encuentra el óptimo.

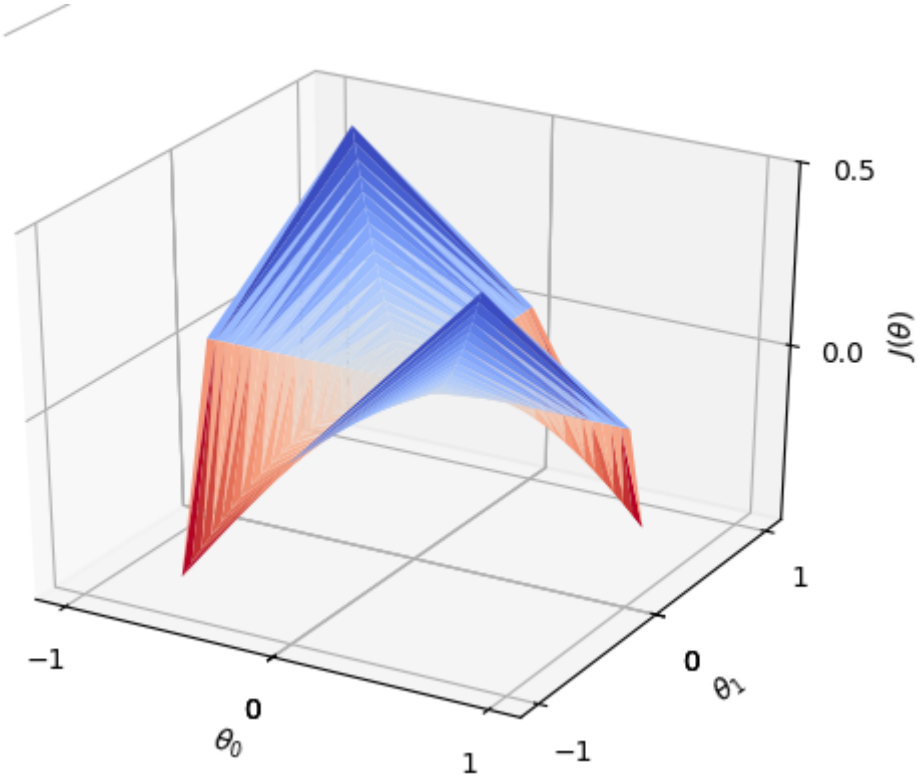
Lamentablemente ésta es una situación ideal que no se encuentra mucho en el mundo real. Por lo general las funciones objetivo tienen superficies mucho más complejas, relacionadas con la dimensionalidad del problema. Una visualización más real se ofrece con la figura generada con `max_min_surface()`.

```
plt.style.use('default'); max_min_surface()
```



En esta situación la búsqueda de parámetros en la superficie irregular dependerá en parte por el punto de inicio del algoritmo. Puede existir situaciones donde el algoritmo se puede quedar estancado en mínimos locales, lejos de la minimización global de la superficie de la función. Otro ejemplo de éste comportamiento se ofrece con la figura `saddle_surface()`.

```
plt.style.use('default'); saddle_surface()
```



Uno de los principales problemas que se enfrenta la Inteligencia Artificial es encontrar nuevas formas de optimizar en superficies complejas.

Gradiente Descendente Estocástico (SGD)

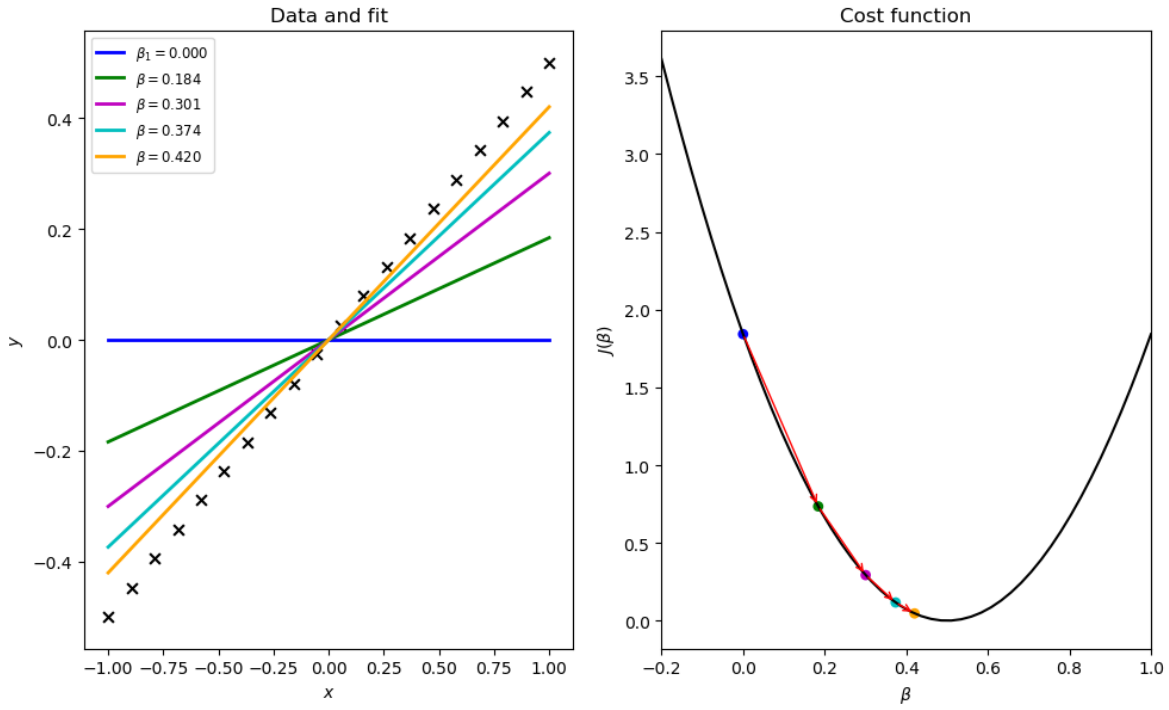
Es el método por defecto para aprender los parámetros de la mayoría de los modelos de machine learning, a diferencia de GD, SGD no utiliza todo el conjunto de entrenamiento para hacer una actualización de los parámetros, en lugar de eso, **actualiza los parámetros una vez por cada elemento del conjunto de entrenamiento**.

$$\hat{\theta}_i^j \leftarrow \hat{\theta}_i^{j-1} - \alpha \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

Como SGD tiene una tasa de actualización mucho más alta que GD, suele ser más eficiente para encontrar mínimos/máximos, el problema es que esa misma tasa alta de actualización de parámetros hace que el método tenga una alta varianza en cuanto a performance.

Veamos como se comporta gráficamente SGD:

```
interactive(gradient_1d, alpha = (1,6))
```



En el widget anterior se muestran las distintas curvas que ajusta el modelo en cada iteración, partiendo de un coeficiente

$$\beta_0 = 0.000$$

ajusta el valor del

$$\beta$$

cada vez más, provocando que la curva se ajuste mejor en cada iteración. En el gráfico de la derecha se muestran los valores de la función de costo para el valor del parámetro en cada iteración.

Podemos jugar con el valor de

$$\alpha$$

para ver como varía la forma en la que se recorre el espacio de búsqueda. Un learning rate alto hace que los saltos en los costos de la función de costo sean más bruscos.

Como referencia, el valor fijado para el parámetro es de 0.5, por lo tanto el modelo debiese buscar valores lo mas cercanos posibles a este valor. Si fijamos un

$$\alpha = 3$$

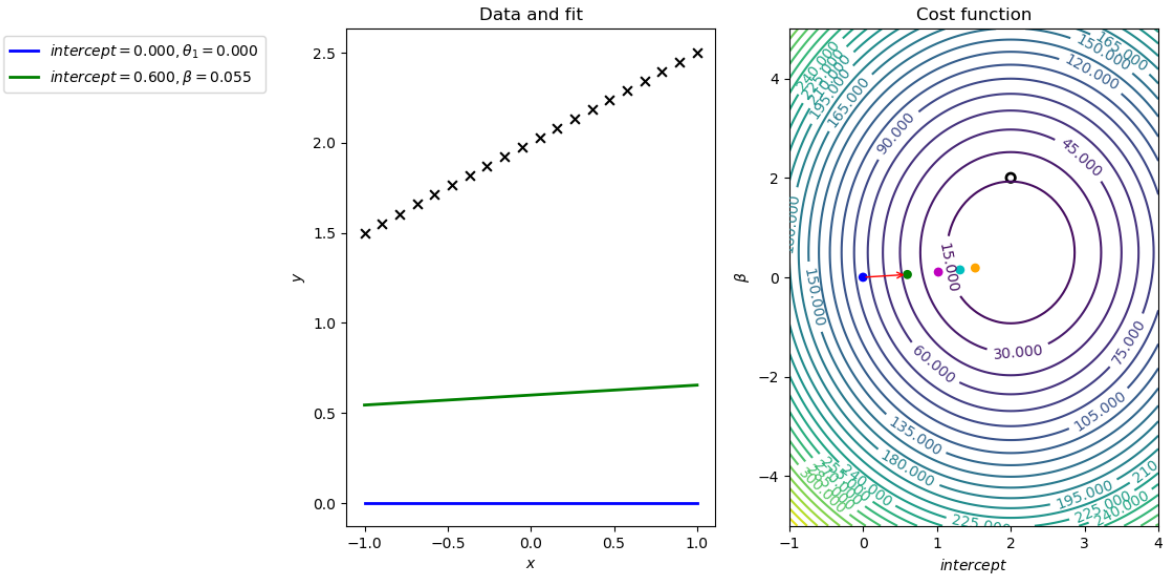
vemos una clara convergencia a este valor de forma casi inmediata, sin embargo, si fijamos un

$$\alpha$$

demasiado pequeño o demasiado caemos en sub-óptimos.

Veamos ahora como se comporta el mismo problema pero ahora considerando el intercepto como parte de los parámetros de búsqueda:

```
interactive(gradient_2d, alpha = (0.10,.3))
```



Al introducir una segunda dimensión al problema, el espacio de búsqueda crece enormemente. En el ejemplo, éste pasó de ser una curva a ser un plano. De forma análoga, si tuviéramos que ajustar un modelo con tres parámetros o más parámetros tendríamos que estar buscando en un hiperespacio. El tamaño del espacio de búsqueda en modelos complejos se convierte en algo imposible de explorar por completo. Por esta razón es importante utilizar un método de minimización que permita recorrer este espacio de forma inteligente.

Existe un fenómeno que ocurre al elegir un learning rate demasiado alto para las características del espacio de búsqueda, el cual se conoce como **overshooting**. Si en el widget aumentamos el valor de

al máximo veremos que las curvas logradas son bastante inexactas y que el recorrido del espacio de búsqueda son saltos de un lado hacia otro sin poder llegar al centro, básicamente el learning rate es tan alto que el optimizador no puede acertar al mínimo, aunque sabe hacia adonde está.

Antes de seguir, experimente con el valor del *learning rate* y observe el comportamiento del optimizador bajo distintos valores. ¿Qué pasa cuando elegimos un `lr` demasiado bajo o uno demasiado alto?.

Batch Training

En este caso, se calcula el error para cada ejemplo, pero **solo actualiza los parámetros luego de que todos los ejemplos de entrenamiento han sido evaluados**. En el caso de gradiente descendente, esto es el cálculo del gradiente para todos los ejemplos, antes de actualizar los parámetros.

$$\hat{\theta} \leftarrow \hat{\theta} - \alpha \nabla_{\theta} J(\theta)$$

"Batch" se refiere conceptualmente a un *"trozo"*, por lo que Batch Training trabaja con un "*trozo*" de los datos igual al conjunto de datos completo.

Una de las principales ventajas del batch training es que tiene un estimador insesgado de los gradientes: mientras mayor sea el número de ejemplos, menor será el error estándar.

Mini-batch training

Tanto Gradiente Descendente como Batch Training requieren que se vea por completo el conjunto de datos antes de actualizar los pesos. Mientras que en Gradiente Descendente lo hace luego de cada ejemplo de entrenamiento, Batch requiere de ingresar todo el conjunto de entrenamiento para actualizar los pesos. Una alternativa intermedia es utilizar una cantidad de ejemplos de entrenamiento mayor que un ejemplo individual y menor que el total. Siguiendo la analogía de "batch" como un trozo, mini-batches se refieren a dividir el dataset de entrenamiento en trozos pequeños o particiones (Al tamaño de cada partición se le llama *"batch size"*), de forma que podamos actualizar los parámetros con más información que solo un ejemplo, es decir, la regla de actualización será:

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} J(x^{j:j+b}, y^{j:j+b})$$

Hay que notar que es importante que el particionamiento no sea hecho con una máscara estática, es decir, los elementos que componen una partición deben ser elegidos aleatoriamente del conjunto de entrenamiento y las particiones deben ser disjuntas.

Comportamiento de los optimizadores

A continuación generaremos un experimento donde evaluaremos el comportamiento en la estimación de parámetros de un modelo de clasificación binaria. Nuestro objetivo será observar cómo cambian los valores del parámetro en la medida que aumentamos la cantidad de iteraciones a disposición con las estrategias de optimización SGD, Batch y Mini Batch.

Por ahora, no es necesario prestar mucha atención al código de este experimento. Posteriormente estudiaremos con más detalle cómo desarrollar redes neuronales. Lo importante es fijarse en los gráficos desarrollados.

Batch Size

```
# generamos los imports básicos de Keras
from keras import Sequential
from keras.layers import Dense, Activation
from keras.callbacks import LambdaCallback
from keras.optimizers import SGD
from sklearn import datasets
from sklearn.model_selection import train_test_split

# Realizamos un dataset con 500 muestras,
# 20 atributos y un vector objetivo con 2 clases
X, y = datasets.make_classification(n_samples = 500)
# generamos la división en conjuntos de entrenamiento/validación
X_train, X_test, y_train, y_test = train_test_split(X,y)
```

En esta parte definiremos el método de optimización generado con Keras. Por efectos prácticos, implementaremos un Descenso de Gradiente Estocástico clásico, para lo cual vamos a obviar los elementos de `momentum`, `decay` y `nesterov`. Posteriormente hablaremos más sobre estas opciones en el contexto de optimizadores modernos.

```
sgd = SGD(lr=0.01, # Tasa de aprendizaje
          momentum=0, #Obviamos momento
          decay=0, # obviamos decaimiento de los pesos
          nesterov=False) #Obviamos momento de nesterov
```

Generamos un modelo con `Sequential` que nos permitirá añadir capas de manera secuencial. Tendremos una capa con 20 neuronas y una activación softmax. Este modelo será compilado, y en este paso definiremos la función de pérdida `'sparse_categorical_crossentropy'` e incluiremos nuestro optimizador definido arriba.

```
# generamos un objeto que nos permita acoplar capas
model = Sequential()
model.add(Dense(20, # agregamos una capa densa con 20 neuronas
                input_dim=20, # 20 atributos de entrada
                activation='softmax')) # y una activación softmax

# compilamos el modelo
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=sgd)
```

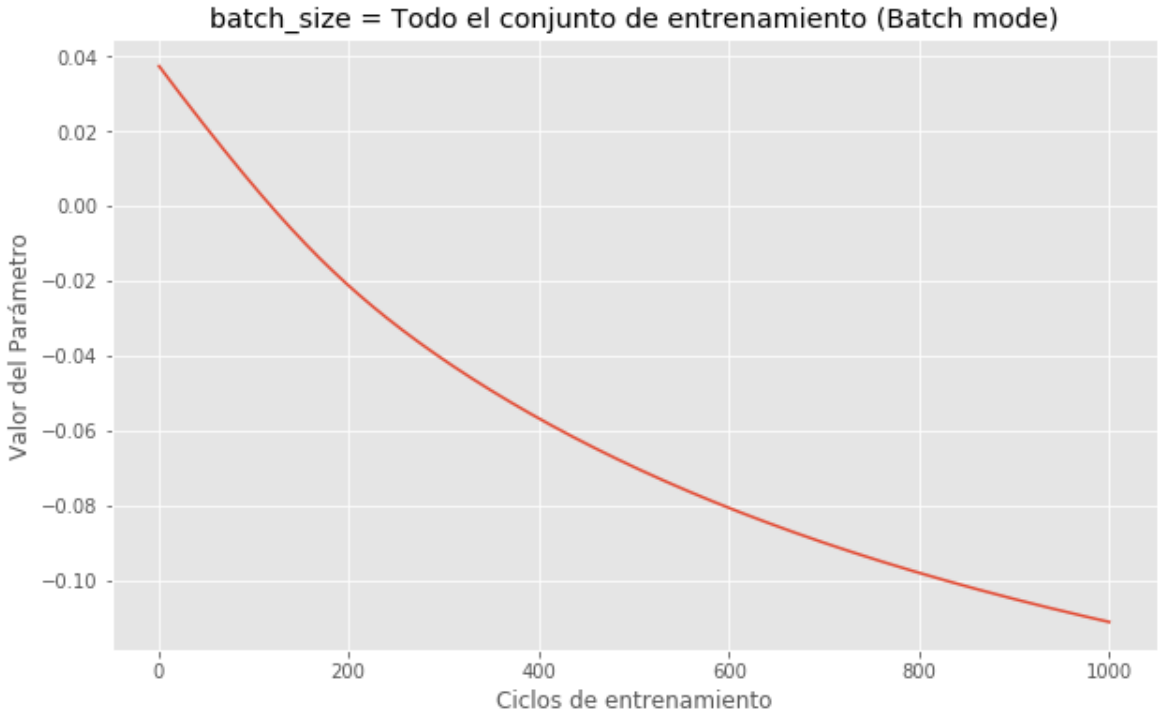
Para evaluar el comportamiento del parámetro, necesitamos preservarlos con el método `LambdaCallback`. Para definir el entrenamiento en modo **batch**, hay que considerar dos elementos:

1. El argumento `batch_size` en el método `model.fit` **deber ser igual** a la cantidad de casos en el conjunto de entrenamiento.
2. El argument `epochs` en el método `model.fit` **debe declarar la cantidad de veces que el modelo se entrena**. Una época hace referencia a una iteración donde se entrena y actualizan los parámetros del modelo con todos los datos.

```
# vamos a preservar los parámetros
weights = []
# Preservaremos todos los parámetros al finalizar una época de entrenamiento
print_weights = LambdaCallback(on_epoch_end=lambda epoch, logs:
weights.append(model.layers[0].get_weights()))

# entrenamos el modelo


model.fit(X_train, y_train, # conjunto de entrenamiento
          batch_size=X_train.shape[0], # definimos la cantidad de datos a
ingresar
         epochs=1000, # la cantidad de iteraciones de entrenamiento.
          verbose=0, # no deseamos implementar un reporte insitu
         validation_data=(X_test, y_test),
          # incluimos nuestro callback
         callbacks=[print_weights])

# extraemos los coeficientes
w_all_data = [item[0] for item in [item[0] for item in [item[0] for item in
weights]]]
```

Generamos el gráfico

```
plt.style.use('ggplot')
plt.figure(figsize=(10, 6))
plt.plot(np.linspace(1, 1000, 1000), w_all_data)
plt.xlabel('Ciclos de entrenamiento');
plt.ylabel('Valor del Parámetro');
plt.title('batch_size = Todo el conjunto de entrenamiento (Batch mode)');
```



Al entrenar con un batch size igual al tamaño del conjunto de entrenamiento podemos ver como el parámetro disminuye casi como una linea recta hacia un valor. Sin embargo, a pesar de haberlo entrenado con 100 ciclos de entrenamiento no parece converger a algún valor específico. Ante este escenario, una alternativa es aumentar la cantidad de épocas de entrenamiento.

Mini Batch

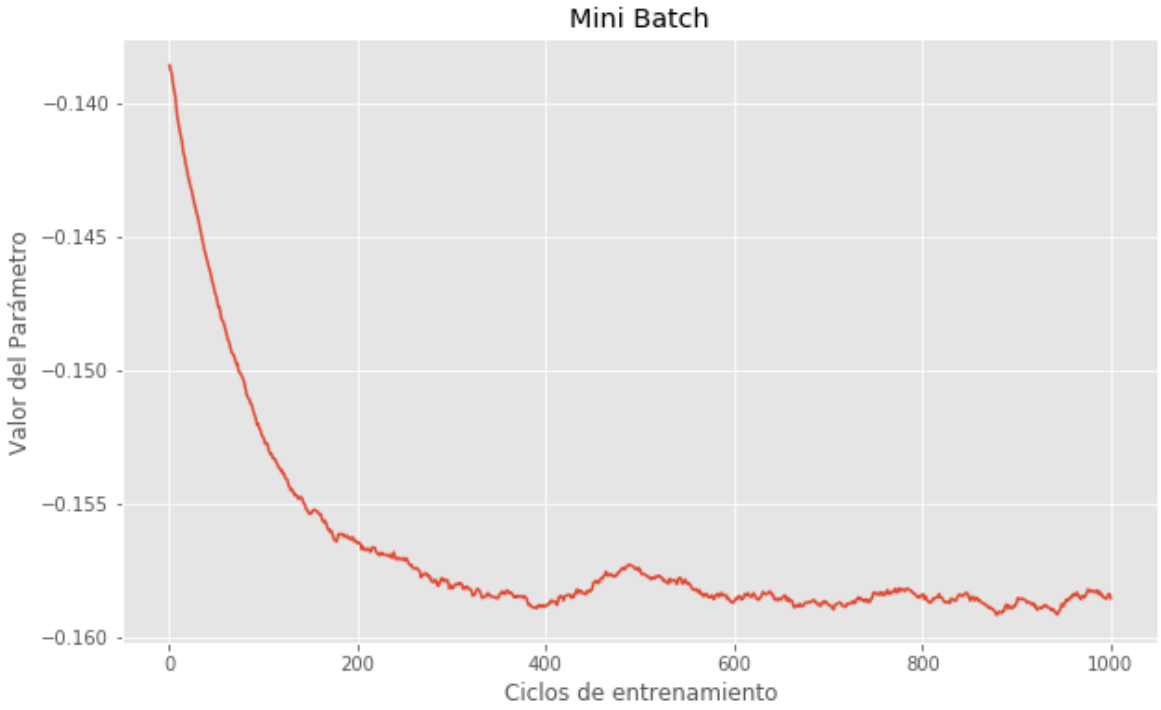
Mini batch se presenta como un punto intermedio entre ingresar todos ejemplos de entrenamiento (modo batch) e ingresar sólo un ejemplo a la vez (modo estocástico). Para decidir el tamaño de los batches, hay que tomar en consideración que por lo general se implementan números que surjan de la potencia de dos, tales como 16, 32, 64, 128, 256,512...

Hay que considerar que cuando implementamos batches pequeños, el proceso de entrenamiento tendrá una convergencia más rápida, a expensas de aumentar marginalmente el ruido asociado al entrenamiento. De similar manera, batches grandes generarán entrenamientos más lentos, con una mayor exactitud general respecto al gradiente de la función de costo.

Para este ejemplo, fijamos el tamaño en 64 ejemplos de entrenamiento por época, y aumentamos el número de épocas a 1000. Dado que ya tenemos definido el modelo anteriormente, tan solo con cambiar la configuración en `fit` estamos bien.

```
weights = []
print_weights = LambdaCallback(on_epoch_end=lambda epoch, logs:
weights.append(model.layers[0].get_weights()))

model.fit(X_train, y_train,
          batch_size=64, # fijamos la cantidad de datos en cada batch
          epochs=1000, #aumentamos la cantidad de ciclos de entrenamiento.
          verbose=0,
          validation_data=(X_test, y_test),
         callbacks=[print_weights])

w_all_data = [item[0] for item in [item[0] for item in [item[0] for item in
weights]]]
```

```
plt.figure(figsize=(10, 6))
plt.plot(np.linspace(1, 1000, 1000), w_all_data)
plt.xlabel('Ciclos de entrenamiento')
plt.ylabel('Valor del Parámetro')
plt.title('Mini Batch');
```



Utilizando un batch size de 64 registros vemos una mejor convergencia, la curva comienza a tender rápidamente durante los primeros ciclos hacia los valores cercanos a la convergencia y poco a poco se estanca.

Gradiente Estocástico Descendente

Por último, en el modo de Gradiente Estocástico, la única modificación a implementar es el hecho que `batch_size` en `model.fit` debe ser igual a 1.

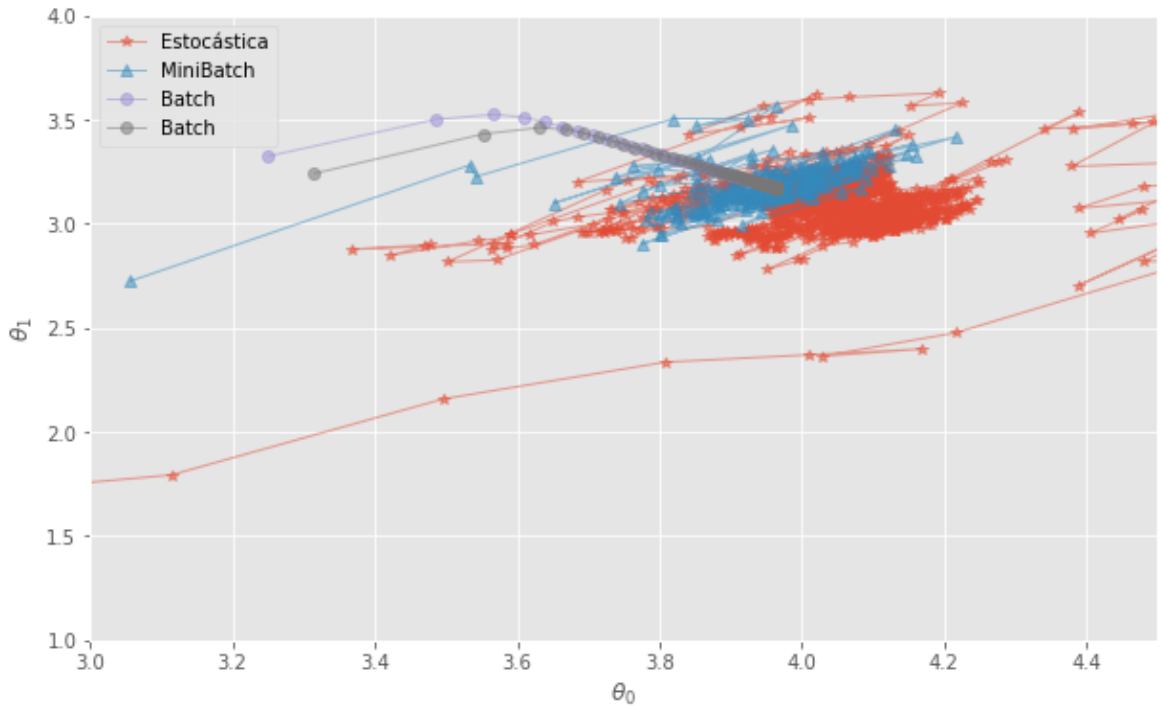
```
weights = []
print_weights = LambdaCallback(on_epoch_end=lambda epoch, logs:
weights.append(model.layers[0].get_weights()))

model.fit(X_train, y_train,
          batch_size=1, # fijamos la cantidad de datos a entrenar.
          epochs=1000, # mantenemos la cantidad de épocas a entrenar en 1000.
          verbose=0,
          validation_data=(X_test, y_test),
         callbacks=[print_weights])
w_all_data = [item[0] for item in [item[0] for item in [item[0] for item in
weights]]]
```

```
plt.figure(figsize=(10, 6))
plt.plot(np.linspace(1, 1000, 1000), w_all_data)
plt.xlabel('Ciclos de entrenamiento')
plt.ylabel('Valor del Parámetro')
plt.title('SGD');
```

Podemos ver que la curva de convergencia de un batch size de 1 es bastante más accidentada sin embargo la tendencia general converge.

Otra forma de visualizar el comportamiento de los métodos de gradientes es ver cómo se van actualizando los parámetros. Para ello el código generado con `compare_gd_strategies` toma simulaciones anteriormente realizadas. Se aprecia que el principal comportamiento del método Batch es mantener una tendencia lineal en la actualización de parámetros y **para** una vez encontrada la optimización de la superficie de respuesta. El método estocástico presenta un comportamiento mucho más errático en cuanto al camino, pero evita el problema de caer en mínimos locales. El comportamiento de Mini Batch se sitúa entre Batch y un método estocástico, dado que calcula las gradientes en pequeñas submuestras de casos. Una de las principales ventajas de Mini Batch por sobre la gradiente estocástica es que está orientada a realizar computaciones más eficientes.

```
plt.style.use('ggplot')
plt.figure(figsize=(10, 6))
X, y = artificial_points()
batch_gd = batch_gd_plot(X, y, theta=1, alpha=.2, theta_path=True)
stochastic_gd = stochastic_gd_plot(X, y, theta_path=True)
minibatch_gd = mini_batch_gd_plot(X, y, theta=1, alpha=.2, theta_path=True)
compare_gd_strategies(batch_gd, stochastic_gd, minibatch_gd)
```



Otros algoritmos de optimización usados actualmente

Resulta que ahora tenemos conocimiento sobre la variante más conocida de optimización. Lamentablemente, las aplicaciones más modernas de redes neuronales tienen un tiempo de entrenamiento substancialmente más alto. Una de las alternativas más adecuadas para solucionar este contratiempo es la implementación de variantes modernas de optimización. A continuación estudiaremos las variantes más conocidas en la actualidad:

- **Momentum**: Similar al Descenso de Gradiente Estocástico, con la diferencia que el optimizador considera la cantidad de movimiento generada por el descenso de SGD. De esta manera, es una variante robusta de Gradiente Estocástico que busca reducir la inercia excesiva del optimizador, y su sensibilidad respecto al espacio de búsqueda. Para ello inyecta cierto nivel de aleatoridad en el optimizador para reducir la inestabilidad.
- **Momentum de Nesterov (Gradiente Acelerado de Nesterov)**: Uno de los problemas que surgen con el método de optimizador por momentum es que la regularización de los momentos genere overshooting (dificultades de estacionarse en un mínimo). La solución propuesta por Yurii Nesterov es generar una predicción de los futuros pasos de actualización en la búsqueda, para evitar el exceso de inercia en el momentum.
- **Adagrad**: El problema con la optimización Momentum de Nesterov es el hecho que no diferencia la frecuencia de actualización de los parámetros inferidos por nuestro modelo candidato. Adagrad facilita la especificación del Learning Rate a nivel de parámetros. Esta es una situación deseable cuando trabajamos con matrices dispersas y necesitamos ignorar algunos pesos. A grandes rasgos, es una variante adaptativa de Gradiente Estocástico, donde regularizamos la tasa de aprendizaje del gradiente. Otra de las virtudes es que no requiere de calibración detallada de la tasa de aprendizaje, dado que se va adaptando.
- **Adadelta**: El problema con Adagrad es que adapta el learning rate en función a la suma de todos los gradientes anteriomente evaluados. De esta manera, es que se sobrerregulariza la búsqueda en la superficie de la superficie de pérdida. Adadelta busca adaptar el learning rate en función a un criterio arbitrario sobre la cantidad de gradientes anteriores a considerar en su paso adaptativo. Cuando Adadelta presenta una combinación específica de hiperparámetros, se conoce como **RMSprop**.

Referencias

- Hardt, M. 2018. EE227C: Convex Optimization and Approximation. Course notes. Department of Electrical Engineering and Computer Science, University of California, Berkeley.
- Dabbura, I. 2017. Gradient Descent algorithm and its variants. *Towards DataScience*. https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3 (visitado el 31 de Octubre, 2018)
- Patterson, J; Gibson, A. 2017. Deep Learning: a Practitioner's Approach. Ch1: A Review of Machine Learning. Stochastic Gradient Descent. O'Reilly