Métodos de descida acelerada

M. Fernanda P. Costa

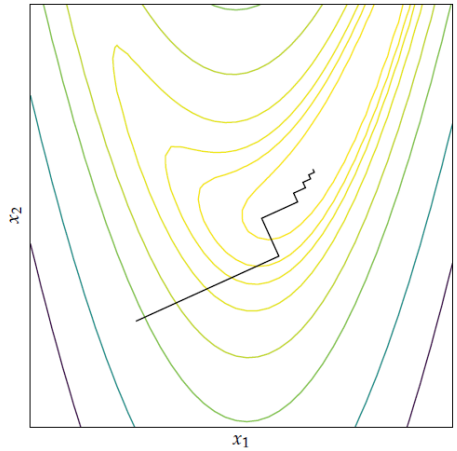
Departamento de Matemática
Universidade do Minho

Outline

- Treinar uma *deep neural network* muito grande pode ser um processo extremamente lento.
- Uma das estratégias para acelerar o treino é usar algoritmos de otimização mais rápido do que o método do gradiente batch e suas variantes estocástica e mini-batch.
- Uma grande variedade de métodos de descida acelerada utilizam técnicas especiais para acelerar a descida.
- Nesta secção apresentamos alguns dos algoritmos mais populares: *Momentum Optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, Adam, Nadam.*

Momentum Optimization

- O método do gradiente tem um desempenho muito fraco em superfícies quase planas: têm gradientes com pequena magnitude, o que leva a exibir um comportamento em ziguezague e passos muito pequenos, que se traduz num processo muito lento para alcançar o minimizante.

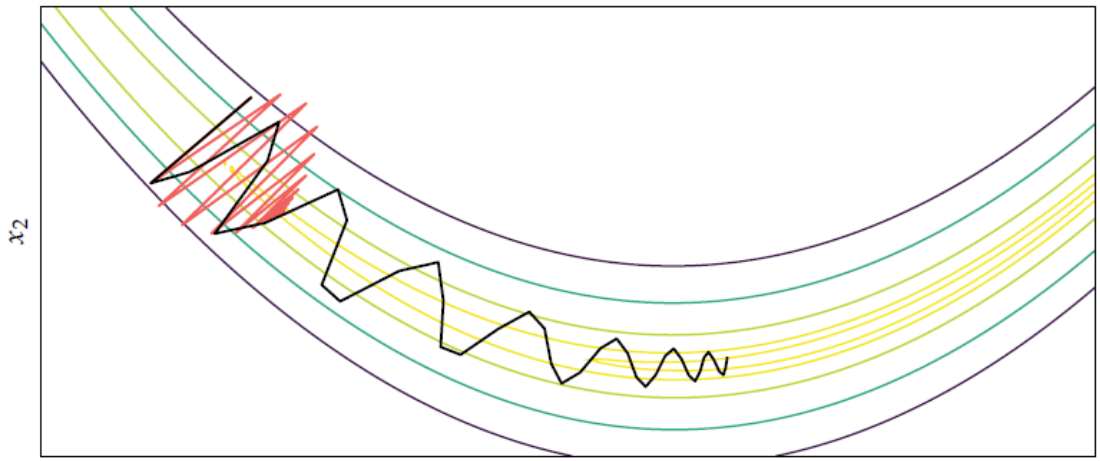


- Permitir que o *momentum*/impulso se acumule é uma forma de acelerar o progresso. Assim, o método do gradiente é modificado para incorporar o *momentum*.

As equações de atualização do método *momentum* são:

$$\begin{aligned} m^{(k+1)} &= \beta m^{(k)} - \eta \nabla F(w^{(k)}) \\ w^{(k+1)} &= w^{(k)} + m^{(k+1)} \end{aligned}$$

- $\beta \in [0, 1]$ é um novo hiperparâmetro, chamado de *momentum decay rate*, para evitar que o *momentum* cresça muito.
  Na prática, em geral o valor $\beta = 0.9$ funciona bem e quase sempre este método é mais rápido que o método gradiente.
- **Nota:** para $\beta = 0$ tem-se o método do gradiente.



— método do gradiente batch; — método do *momentum*

- *Momentum* pode ser interpretado como uma bola a rolar por uma inclinação quase horizontal. A bola naturalmente ganha impulso/*momentum* à medida que a gravidade a faz acelerar, tal como o gradiente faz com que o *momentum* se acumule neste método de descida.

Algoritmo: Método do *Momentum*

1. Dar: $w^{(1)}$, η (*learning rate*), β (*decay rate*)
2. Inicializar: $m^{(1)} = [0, 0, \ldots, 0]^T$ (vetor nulo)
3. Fazer $k = 1$
4. **Enquanto** ($w^{(k)}$ não satisfaz o critério de paragem)
5. Calcular a direção de descida: $-\nabla F(w^{(k)})$
6. Calcular o *momentum*: $m^{(k+1)} = \beta m^{(k)} - \eta \nabla F(w^{(k)})$
7. Fazer $w^{(k+1)} = w^{(k)} + m^{(k+1)}$
8. Fazer $k = k + 1$
9. **Fim enquanto**

Nota: este algoritmo preocupa-se muito com as direções de procura anteriores: em cada iteração k, adiciona a direção de descida local, $-\nabla F(w^{(k)})$, ao vetor de momento $m^{(k)}$ (multiplicado pela taxa de aprendizagem η), e atualiza os parâmetros adicionando este vetor *momentum*.

Ex1: Os desempenhos do método *momentum* e do método do gradiente são comparados na figura acima.

Nesterov Accelerated Gradient (NAG)

- Uma questão problemática que surge no método *Momentum* é que os "passos"/direção não abrandam o suficiente no fundo de um vale e tendem a ultrapassar o fundo do vale.
- O método de Nesterov de Gradiente Acelerado, também conhecido como *Momentum Optimization Nesterov*, modifica-o calculando o gradiente da função objetivo não na posição atual $w^{(k)}$ mas ligeiramente à frente na direção do *momentum*, isto é, no ponto $w^{(k)} + \beta m^{(k)}$:

As equações de atualização do método NAG são:

$$\begin{aligned} m^{(k+1)} &= \beta m^{(k)} - \eta \nabla F\left(w^{(k)} + \beta m^{(k)}\right) \\ w^{(k+1)} &= w^{(k)} + m^{(k+1)} \end{aligned}$$

- Este pequeno ajuste funciona porque, em geral, o vetor de momento estará a apontar na direção certa (ou seja, em direção do ótimo), pelo que será ligeiramente mais preciso utilizar o gradiente calculado um pouco mais longe nessa direção do que o gradiente na posição atual, como se pode ver na seguinte figura:

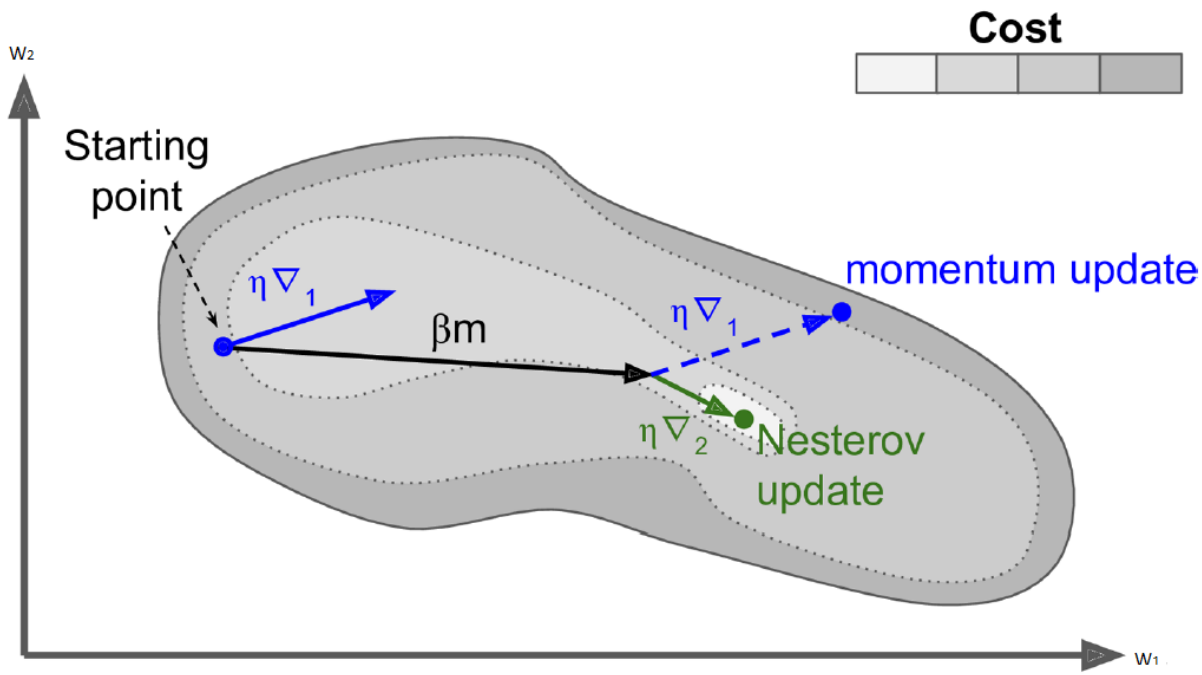


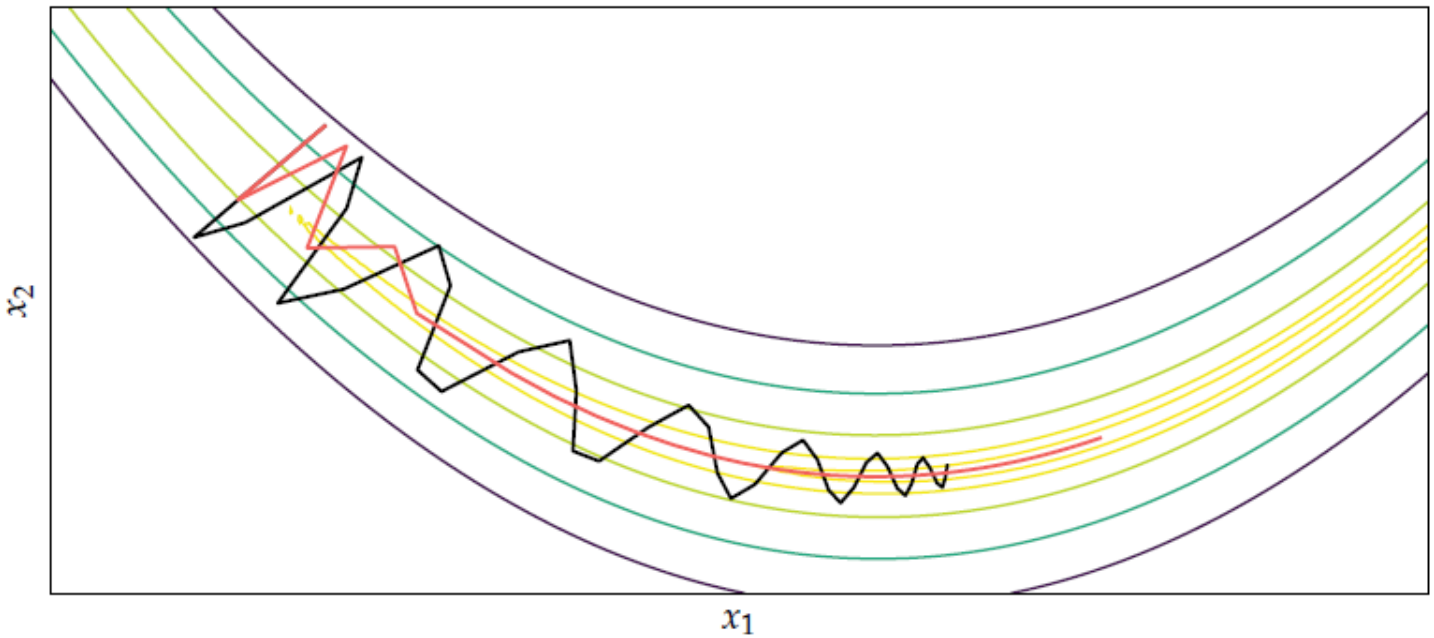
Figura: método *momentum verus* método NAG

- Na fig., ∇_1 é o gradiente da função objetivo/Cost no ponto inicial w, e ∇_2 é o gradiente no ponto $w + \beta m$.
- O métod NAG é mais rápido que o método *momentum*.

Algoritmo: Método NAG

1. Dar: $w^{(1)}$, η (*learning rate*), β (*decay rate*)
2. Inicializar: $m^{(1)} = [0, 0, \ldots, 0]^T$ (vetor nulo)
3. Fazer $k = 1$
4. **Enquanto** ($w^{(k)}$ não satisfaz o critério de paragem)
5. Calcular a direção de descida na posição futura: $-\nabla F(w^{(k)} + \beta m^{(k)})$
6. Calcular o *momentum*: $m^{(k+1)} = \beta m^{(k)} - \eta \nabla F(w^{(k)} + \beta m^{(k)})$
7. Fazer $w^{(k+1)} = w^{(k)} + m^{(k+1)}$
8. Fazer $k = k + 1$
9. **Fim enquanto**

Ex2: os desempenhos do método *momentum* e do método NAG são comparados na seguinte figura:



— método NAG; — método *momentum*

Adagrad - Adaptive subgradient method

- Os métodos *Momentum* e NAG atualizam todas as componentes de $w = (w_1, w_2, \ldots, w_I)^T$ com a mesma *learning rate* η.
- O método Adagrad, adapta uma *learning rate* para cada componente de w. O Adagrad atenua a influência de parâmetros com gradientes consistentemente elevados, aumentando assim a influência de parâmetros com atualizações pouco frequentes.

  As equações de atualização do método Adagrad são:

$$\begin{aligned} s^{(k+1)} &= s^{(k)} + \nabla F(w^{(k)}) .* \nabla F(w^{(k)}) \\ w^{(k+1)} &= w^{(k)} - \eta \nabla F(w^{(k)}) ./ (\epsilon + \sqrt{s^{(k+1)}}) \end{aligned}$$

- o vector s acumula o quadrado dos gradientes;
- $.*$, $./$, $\sqrt{\ }$, representam a multiplicação, divisão e raiz quadrada, elemento a elemento de um vetor.
- $\epsilon > 0$, geralmente da ordem 10^{-8}, para evitar a divisão por zero.

Algoritmo: Método de descida acelerada Adagrad

1. Dar: $w^{(1)}$, η (*learning rate*), $\epsilon > 0$
2. Inicializar: $s^{(1)} = [0, 0, \ldots, 0]^T$ (vetor nulo)
3. Fazer $k = 1$
4. **Enquanto** ($w^{(k)}$ não satisfaz o critério de paragem)
5. Calcular a direção de descida: $-\nabla F(w^{(k)})$
6. Fazer $s^{(k+1)} = s^{(k)} + \nabla F(w^{(k)}) .* \nabla F(w^{(k)})$
7. Fazer $w^{(k+1)} = w^{(k)} - \eta \nabla F(w^{(k)}) ./ (\epsilon + \sqrt{s^{(k+1)}})$
8. Fazer $k = k + 1$
9. **Fim enquanto**

Vantagem: Adagrad é muito menos sensível ao parâmetro η da *learning rate*. $\eta = 0.01$ é normalmente o valor usado por defeito.

Desvantagem: As componentes de s são estritamente não decrescentes. A soma acumulada faz com que a *learning rate* efetiva diminua ao longo do treino, tornando-se frequentemente infinitesimal que o algoritmo acaba por parar completamente antes de atingir a solução ótima.

RMSProp

- Como vimos, o AdaGrad corre o risco de desacelerar um pouco rápido demais e de nunca convergir para a solução ótimo.
- O RMSProp estende o Adagrad para corrigir isto, ou seja, para evitar o efeito de uma *learning rate* monótona decrescente.

  RMSProp mantém uma média decrescente de gradientes ao quadrado.

  As equações de atualização do RMSProp são:

$$\begin{aligned} s^{(k+1)} &= \beta s^{(k)} + (1-\beta)\nabla F(w^{(k)}) .* \nabla F(w^{(k)}) \\ w^{(k+1)} &= w^{(k)} - \eta \nabla F(w^{(k)}) ./ (\epsilon + \sqrt{s^{(k+1)}}) \end{aligned}$$

Algoritmo: Método de descida acelerada RMSProp

1. Dar: $w^{(1)}$, η (*learning rate*), β (*decay rate*), $\epsilon > 0$
2. Inicializar: $s^{(1)} = [0, 0, \ldots, 0]^T$ (vetor nulo)
3. Fazer $k = 1$
4. **Enquanto** ($w^{(k)}$ não satisfaz o critério de paragem)
5. Calcular a direção de descida: $-\nabla F(w^{(k)})$
6. Fazer $s^{(k+1)} = \beta s^{(k)} + (1 - \beta)\nabla F(w^{(k)}) .* \nabla F(w^{(k)})$
7. Fazer $w^{(k+1)} = w^{(k)} - \eta \nabla F(w^{(k)}) ./ (\epsilon + \sqrt{s^{(k+1)}})$
8. Fazer $k = k + 1$
9. **Fim enquanto**

- **Nota:** β é um novo hiperparâmetro, geralmente o valor $\beta = 0.9$ funciona bem, não havendo necessidade de o ajustar.

Adam - Adaptive moment estimation method

- O método Adam também adapta uma *learning rate* para cada parâmetro $w = (w_1, w_2, \ldots, w_l)^T$. O Adam combina as ideias dos métodos *momentum* e RMSProp. Armazena uma média decrescente de gradientes como o método *momentum* e uma média decrescente de gradientes ao quadrado como o RMSProp.

- Inicializar os vetores m e s como vetores nulos introduz um viés. Para solucionar este problema, é introduzido um passo de correção de enviesamento.

As equações de atualização do Adam são:

$$
\begin{aligned}
m^{(k+1)} &= \beta_1 m^{(k)} + (1-\beta_1)\nabla F(w^{(k)}) \\
s^{(k+1)} &= \beta_2 s^{(k)} + (1-\beta_2)\nabla F(w^{(k)}) .* \nabla F(w^{(k)}) \\
\hat{m}^{(k+1)} &= \frac{m^{(k+1)}}{1-\beta_1^k} \\
\hat{s}^{(k+1)} &= \frac{s^{(k+1)}}{1-\beta_2^k} \\
w^{(k+1)} &= w^{(k)} - \eta \hat{m}^{(k+1)} ./ (\epsilon + \sqrt{\hat{s}^{(k+1)}})
\end{aligned}
$$

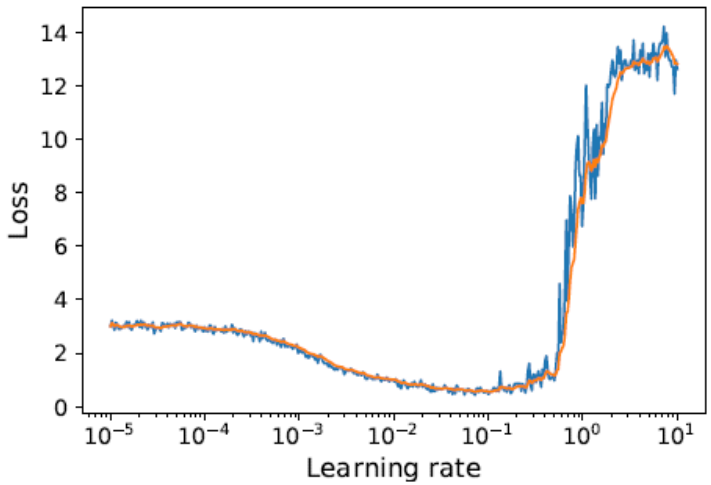
- k representa o número da iteração (começa em 1).
- por defeito os valores dos hiperparâmetros são: $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

Algoritmo: Método de descida acelerada Adam

1. Dar: $w^{(1)}$, η (*learning rate*), β_1, β_2 (*decay rates*), $\epsilon > 0$
2. Inicializar: $m^{(1)} = [0, 0, \ldots, 0]^T$, $s^{(1)} = [0, 0, \ldots, 0]^T$
3. Fazer $k = 1$
4. **Enquanto** ($w^{(k)}$ não satisfaz o critério de paragem)
5. Calcular a direção de descida: $-\nabla F(w^{(k)})$
6. Fazer $m^{(k+1)} = \beta_1 m^{(k)} + (1 - \beta_1)\nabla F(w^{(k)})$
7. Fazer $s^{(k+1)} = \beta_2 s^{(k)} + (1 - \beta_2)\nabla F(w^{(k)}) .* \nabla F(w^{(k)})$
8. Fazer $\hat{m}^{(k+1)} = \dfrac{m^{(k+1)}}{1 - \beta_1^k}$
9. Fazer $\hat{s}^{(k+1)} = \dfrac{s^{(k+1)}}{1 - \beta_2^k}$
10. Fazer $w^{(k+1)} = w^{(k)} - \eta \hat{m}^{(k+1)} ./ (\epsilon + \sqrt{\hat{s}^{(k+1)}})$
11. Fazer $k = k + 1$
12. **Fim enquanto**

Learning rate (step size)

- O método mais simples é usar uma única *learning rate* constante, η. Mas encontrar um bom η é muito importante pois
  - se η for demasiado grande, o método pode não convergir;
  - se η for demasiado pequeno, o método poderá convergir para o ótimo mas levará muito tempo.
- Uma estratégia para encontrar uma bom η, é começar com um valor muito pequeno e aumentá-lo gradualmente. e vai-se avaliando a função Loss.



Fazer o gráfico Loss *verus* *learning rate*, e escolher o η com o menor valor de *Loss*. (Na prática, escolher η ligeiramente menor ao que tem o menor valor de *Loss*, para assegurar a estabilidade).

- Mas podemos fazer muito melhor do que usar uma *learning rate* constante:
  - se começarmos com um η grande e reduzi-lo assim que a Loss parar de diminuir rapidamente, podemos chegar a uma solução ótima mais rapidamente do que com uma *learning rate* constante;
- Existem muitas estratégias diferentes para reduzir *learning rate* durante o treino. Estas estratégias são chamadas *learning rate shedules*, sendo aqui apresentadas algumas das mais usadas.

- power sheduling: $\eta_k = \dfrac{\eta_0}{(1 + k/s)^c}$, ($k$ é a iteração)
  - hiperparâmetros - η_0 (valor inicial da *learning rate*), potência c (tipicamente 1), steps s;

  A *learning rate* reduz a cada step. Após s steps, reduz para $\frac{\eta_0}{2}$, após mais s steps, reduz para $\frac{\eta_0}{3}$, e a seguir para $\frac{\eta_0}{4}$, e assim sucessivamente.

- exponential sheduling: $\eta_k = \eta_0 \times 0.1^{k/s}$, ($k$ é a iteração)
  - hiperparâmetros - η_0, steps s;

  A *learning rate* reduz gradualmente por um fator de 10 a cada s steps. Enquanto na power sheduling a *learning rate* vai reduzindo cada vez mais lentamente, na exponential sheduling continua reduzindo-a em um fator de 10 a cada s steps.

- piecewise constant sheduling: $\eta_k = \eta_i$
  - usa uma *learning rate* constante para um numero de épocas (ex., $\eta_0 = 0.1$ para 5 épocas), a seguir usa uma *learning rate* mais pequena para outro número de épocas (ex. $\eta_0 = 0.01$ para 50 épocas), e assim sucessivamente;
  - ou, podemos fazer $\eta_i = \eta_0 \times \gamma^i$ que reduz a *learning rate* inicial por um fator γ a cada número de épocas, onde $0 < \gamma < 1$;

Embora esta estratégia possa funcionar bem, ela requer alguns ajustes para descobrir a sucessão correta dos η_i e por quanto tempo usar cada um deles.

Bibliografia

- Aurélien Géron. **Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow**, Second Edition, O'Reilly, 2019.