Laboratório 4: otimização

O objetivo deste trabalho é estudar uma implementação de diferentes algoritmos de otimização, sem restrição, com uma aplicação à aproximação polinomial.

1 Dados

1.1 Notações

Consideramos o conjunto de dados, $D = (x^n, y^n)_{n=1,\cdots,N}$ onde $x^n, y^n \in \mathbb{R}$. O objetivo é de determinar o polinómio $\hat{y} = p(x)$ de grau I que passa mais perto dos pontos. A construção do polinómio é uma *machine learning* no sentido que os vamos determinar os coeficientes com os dados. Introduzimos as notações seguintes:

$$x \in \mathbb{R} \to \varphi(x) = \begin{pmatrix} (x)^0 \\ (x)^1 \\ \vdots \\ (x)^I \end{pmatrix}, \quad w = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_I \end{pmatrix}, \quad p(x; w) = \sum_{i=0}^{I} (x)^i w_i = w^T \varphi(x),$$

com w o vetor dos coeficientes para determinar. Cuidado, diferenciar $(x)^2$ (x ao quadrado) de x^2 (segundo evento).

Definimos a função custo por o conjunto de dados

$$E(w; D) = \frac{1}{2N} \sum_{n=1}^{N} \Big(p(x^n; w) - y^n \Big)^2 = \frac{1}{2|D|} \sum_{(x^n, y^n) \in D} \Big(w^T \varphi(x^n) - y^n \Big)^2$$

e procuramos

$$w^\star = \arg \min_{w \in \mathbb{R}^{I+1}} E(w; D).$$

Importante. I é um meta-parâmetro que define o número de incógnitas. O valor será determinado, experimentalmente, pelo *user*.

1.2 Algoritmo gradiente

O gradiente da funcional de custo é dado por

$$\nabla_w E(w;D) = \frac{1}{|D|} \sum_{(x^n,y^n)\in D} \Big[p(x^n;w) - y^n\Big]\varphi(x^n)$$

Vamos construir uma sequência $w^{(0)}$, $w^{(1)}$, $\cdots$, $w^{(k)}$ que converge para $w^\star$ usando o método de descida de tipo gradiente dito *batch*.

```
initialize w^(k) for k=0
do
    compute gradient G^(k)=Grad E(w^(k);D)
    compute the length alpha^(k)
    update w^(k+1)=w^(k)-alpha^(k)G^(k)
until (solution is OK)
```

Vamos definir diferentes cenarios para calcular $\alpha^{(k)}$ e tentar melhorar o algoritmo. Por outro lado, usamos como critério de paragem um dos seguintes:

- (C1) $\|G^{(k)}\| \leq \varepsilon$
- (C2) $\frac{\|w^{(k+1)}-w^{(k)}\|}{\|w^{(k)}\|} \leq \varepsilon$
- (C3) $\frac{|E(w^{(k+1)};D)-E(w^{(k)};D)|}{E(w^{(k)};D)} \leq \varepsilon$

1.3 Implementação

Usando o *header* proposto no programa `optim.py`, realizar as implementações das funções seguintes.

1. A função `polynomial` que calcula $p(x,w)$ é já implementada.
2. uma função `cost` que calcula $E(w;D)$
3. uma função `gradient` que calcula $\nabla_w E(w;D)$
4. uma função `step` que calcula $\alpha^{(k)}$ (ver secção a seguir).
5. Implementar o método do gradiente dito *batch*.

2 O passo α

2.1 Passo constante

Começamos com uma situação de passo constante onde fixamos o valor do passo independentemente de k, por exemplo $\alpha^{(k)} = 1$. Experimentar diferentes valores de $I = 1, \cdots 7$ e determinar os coeficientes associados assim que

o erro dito *in-samples errors* $E(w; D)$. Confrontar com diferentes base de dados `P2.csv` não perturbado e o ficheiro `P3.csv` com perturbações gaussianas. Os dois polinómios são $p_2(x) = \frac{1}{5} - \frac{1}{2}x + x^2$ e $p_3(x) = \frac{1}{5} - \frac{1}{2}x + x^2 - \frac{2}{3}x^3$.

2.2 Passo ótimo

Como a função E é quadrática, podemos determinar uma expressão analítica para o α ótimo

$$\alpha^{(k)} = \frac{\sum_n p(x^n; w^{(k)})\ \varphi^T(x^n) G^{(k)}}{\sum_n \varphi^T(x^n) G^{(k)}\ \varphi^T(x^n) G^{(k)}}$$

Implementar esta nova versão e comparar os tempos de execução.

2.3 *Backtracking*

Notamos por $\overline{\alpha}$ um valor definido pelo *user*. Seja $G^{(k)}$ o gradiente calculado com $w^{(k)}$. O algoritmo de *backtracking* procura o valor de $\alpha^{(k)}$ como

```
initialize alpha=alpha_bar
while(  E(w^k-alpha G^k;D)>E(w^k;D)-alpha <G^k;G^k>  )
     alpha:=alpha/2
end
alpha^k=alpha
```

Implementar a função `backtraking`.
Determinar o número de iterações necessarias para a convergência e comparar o tempo de execução com o tempo obtido na secção anterior.

3 Método *mini-batch* e estocático

O objetivo desta parte é, para um número $I + 1$ de parametros w, avaliar a convergência do método do gradiente e a sua eficiência. Dividimos o conjunto D em duas partes: 80% para criar o *training set* D_t e os 20% restantes para o *validation set* D_v.
Implementar uma função que seleciona o conjunto de dados D_t e D_v.

3.1 Método *batch*

Neste caso, a função custo E e o gradiente têm a expressão $E(w; D_t)$ e $G = \nabla_w E(w; D_t)$. Usando a implementação do script determinar, em função de I, os parâmetros $w^\star = w^\star(I)$ assim como o *in-sample error* $E(w^\star, D_t)$ e

o *out-sample error* $E(w^\star, D_v)$.
Num gráfico, visualizar os erros para $I = 2, \cdots, 7$. Deduzir o meta-parâmetro I que fornece a melhor aproximação.

3.2 Método *mini batch*

O princípio do mini-bacth é de calcular o gradiente com um conjunto $\widetilde{D}$ de dados menor que todo o dataset D_t. Para cada iteração $w^{(k)} \to w^{(k+1)}$, escolhemos um subconjunto $\widetilde{D^{(k)}} \subset D_t$ e calculamos o gradiente como $\nabla_w E(w; \widetilde{D^{(k)}} \subset D_t)$. Usamos 5% dos dados de D_t para formar $\widetilde{D^{(k)}}$.
(1) Implementar uma função que selectiona uma parte do conjunto de dados D_t. Utilizar o ficheiro `P5-large_no.csv` que corresponde a um polinómio sem perturbação. Comparar os tempos de execução, o número de iterações e as soluções respetivas entre o método *mini batch* e *batch*. Determinar os erros em função de I e identificar o meta-parâmetro ótimo.
(2) recomeçar o estudo com o ficheiro `P5-large_yes.csv` que represente o mesmo polinómio mas com uma perturbação.

3.3 Método estocástico

No método estocástico corresponde ao caso *mini batch* limite onde usamos apenas um elemento de D_t para construir $\widetilde{D^{(k)}}$, seja $|\widetilde{D^{(k)}}| = 1$.
Realizar de novo os testes numéricos com o ficheiro `P5-large_yes.csv`. Verificar o ganho de tempo e a qualidade da solução.