Test optimizer

April 29, 2021

1 Analyse des Optimizers

Le rôle de l'optimizer est de définir comment évolue (apprend) une IA pour s'adapter aux données d'entraînement. Par défaut, on utilise souvent la Descente de Gradient Stochastique (SGD)

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Nous allons ici créer un Neuronne Linéaire sans fonction d'activation, et nous allons l'entrainer à multiplier par la valeur cible.

[2]:
```
class NeuronneLineaire:
    def __init__(self, pseudo_alea=True, lr=0.8, cible=0.5):
        if pseudo_alea:
            np.random.seed(2)
        self.w = np.random.random()
        self.lr = lr # le learning rate
        self.liste_w = [] # l'historique des poids
        self.liste_e = [] # l'historique des erreurs
        self.cible = cible # le poids ciblé par l'entrainement

    def calcul(self, x):
        return self.w * x # prédiction du résultat

    def erreur(self, x, y):
        y_ = self.calcul(x)
        e  = (y - y_) ** 2 # calcul de l'erreur mis au carré
        return y_, e

    def validation(self):
        y_, e = self.erreur(1, self.cible)
        self.liste_w.append(self.w)
        self.liste_e.append(e)

    def retropropagation(self, x, y):
        if type(x) == int:
            x, y = np.array([x]), np.array([y])
        y_, e = self.erreur(x, y)
```

```
        dw = (2*(y_-y) * 1 * x).mean() # Calcul de la mise à niveau du poids
        self.w -= self.lr * dw # Mise à jour du poids

    def plot(self):
        x = np.linspace(0.25, 1, 1_001)
        y = (x - self.cible) ** 2 # calcul de la courbe d'erreur

        plt.figure(figsize=(12, 6))
        plt.plot(x, y, label="Courbe d'erreur")
        plt.plot(self.liste_w, self.liste_e, 'o-', lw=3, label="Apprentissage")
        plt.grid()
        plt.legend()
```

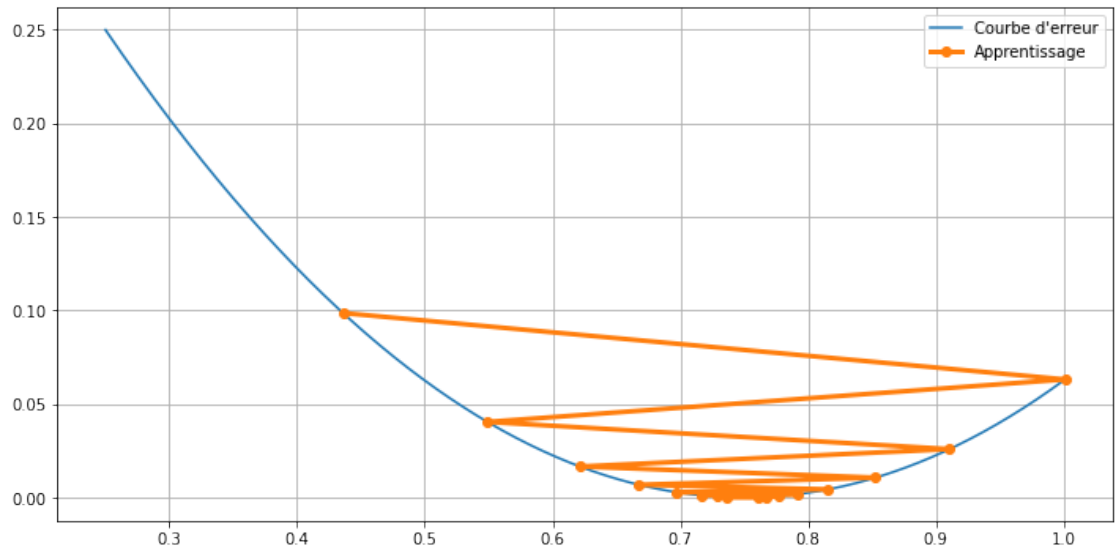
2 Descente de Gradient Stochastique (SGD)

On remarquera que pour un grand learning rate, l'apprentissage oscille énormément autour de la valeure recherchée.
Tandis que pour un petit learning rate, celui-ci descent assez lentement.

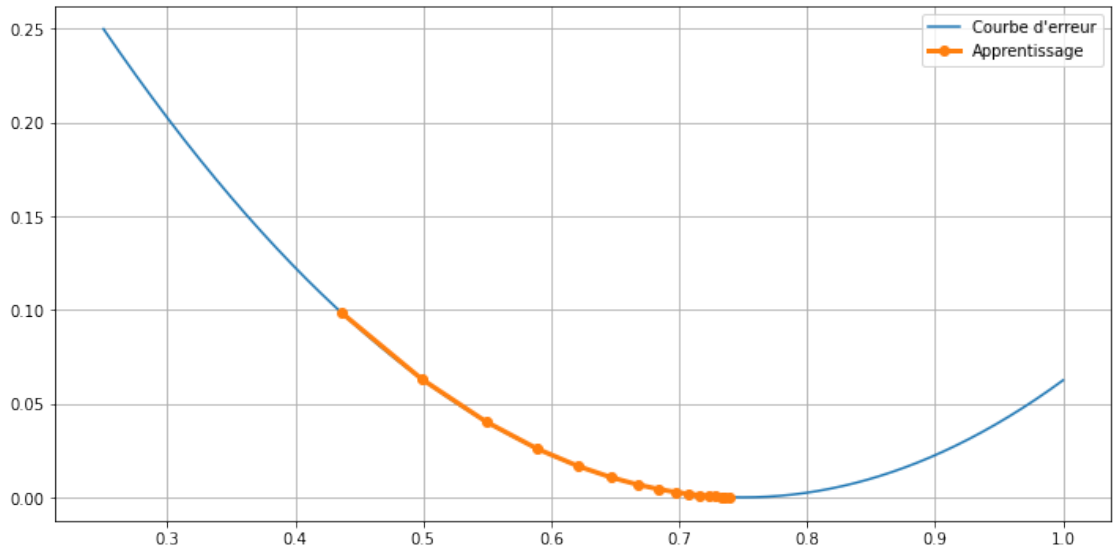
[3]:
```
# Grand learning rate
reseau = NeuronneLineaire(lr=0.9, cible=0.75)
for i in range(16):
    reseau.validation()
    reseau.retropropagation(1, 0.75)
reseau.calcul(1)
```

[3]:
```
0.7411615422393394
```

[4]:
```
reseau.plot()
```



[5]:
```
# Petit learning rate
reseau = NeuronneLineaire(lr=0.1, cible=0.75)
for i in range(16):
    reseau.validation()
    reseau.retropropagation(1, 0.75)
reseau.plot()
```

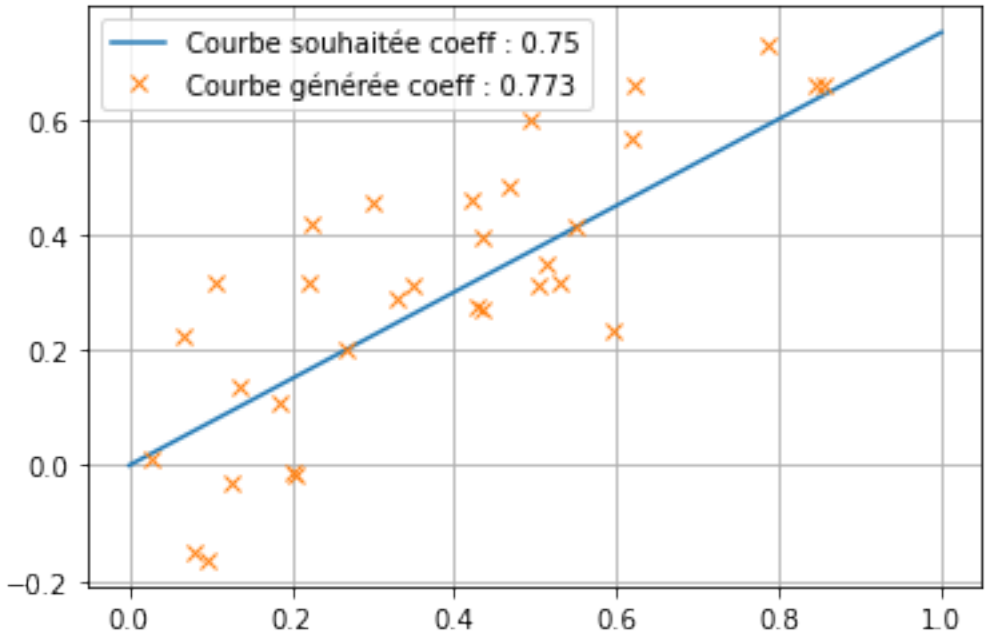


3 SGD avec plusieurs entrées

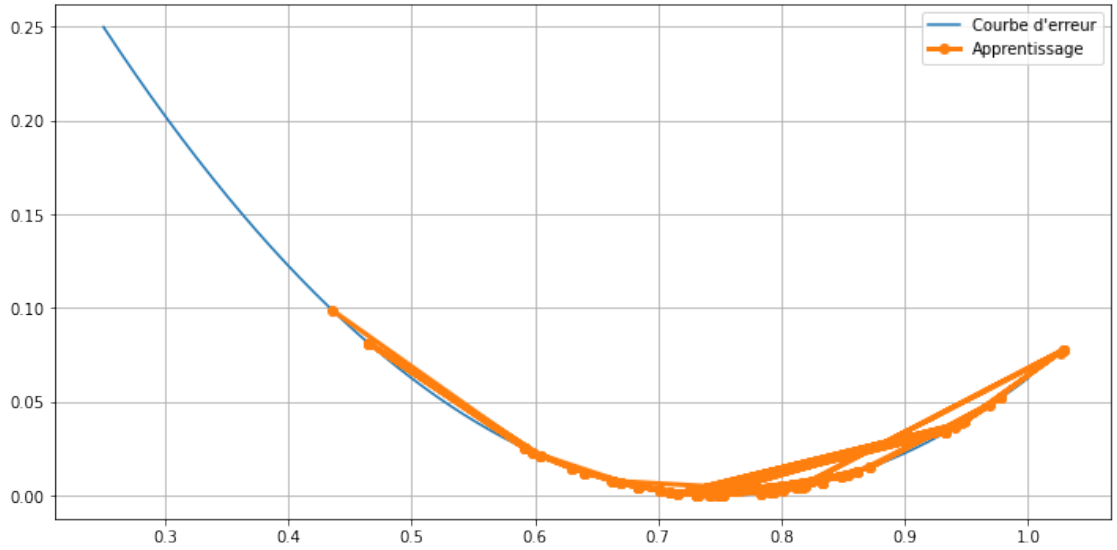
En réalité, lorsque que l'on entraine un réseau de neuronnes, les données d'entrainement ne sont jamais exactement les données rencontrées lors de l'utilisation. De plus les valeurs peuvent être sujet à de petites variations (bruit) aléatoire.
Nous avons essayer de recréer ici cette environnemnet d'entrainement avec la fonction générer :
$y = x \times cible + \delta \times bruit$
avec $\delta \in [|-0.5, 0.5|]$ choisi aléatoirement.

[6]:
```
# on génère les entrées et sorties voulu
def generer(n, pseudo_alea=True, cible=0.5, bruit=0):
    if pseudo_alea:
        np.random.seed(2)
    x = np.random.random(n) # on génère l'entrée
    y = x * cible # on calcul la sortie pour chaque entrée
    y += (np.random.random(n) - 0.5) * bruit # on rajoute du bruit sur les␣
 ↪sorties
    # la variable bruit étant l'amplitude maximal du bruit appliquée
```

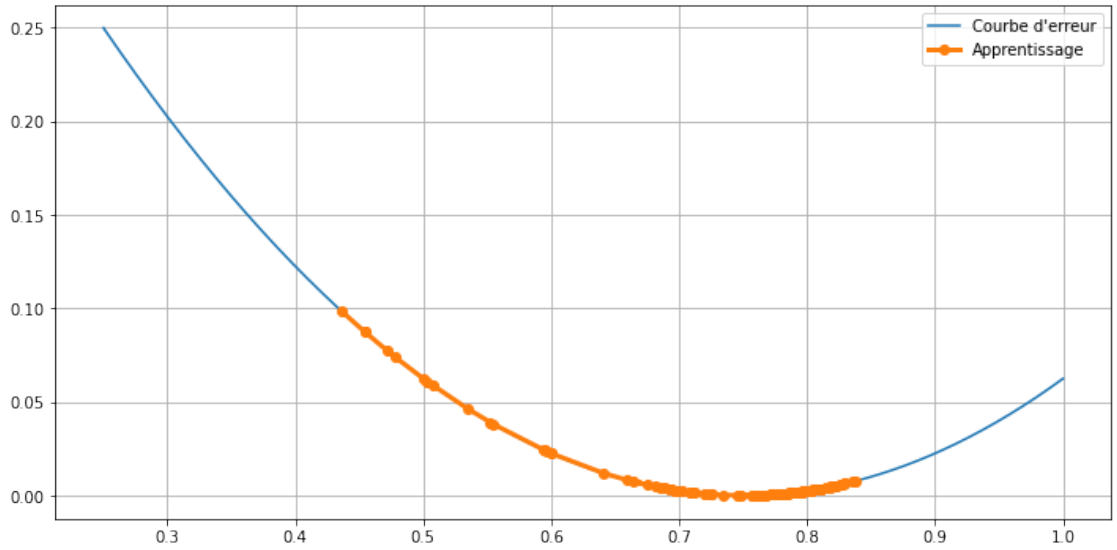
```
    return x, y
liste_x, liste_y = generer(32, pseudo_alea=True, cible=0.75, bruit=0.5) #␣
 ↪courbe générée
liste_x_ = np.linspace(0, 1, 101)
liste_y_ = liste_x_ * 0.75 # Courbe souhaitée

plt.plot(liste_x_, liste_y_, label="Courbe souhaitée coeff : 0.75")
plt.plot(liste_x, liste_y, "x",
         label=f"Courbe générée coeff : {round((liste_y/liste_x).mean(), 3)}")
plt.legend()
plt.grid()
```



[7]:
```
# Grand learning rate
reseau = NeuronneLineaire(lr=0.9, cible=0.75)
for i in range(4):
    for x, y in zip(liste_x, liste_y):
        reseau.validation()
        reseau.retropropagation(x, y)
reseau.plot()
```



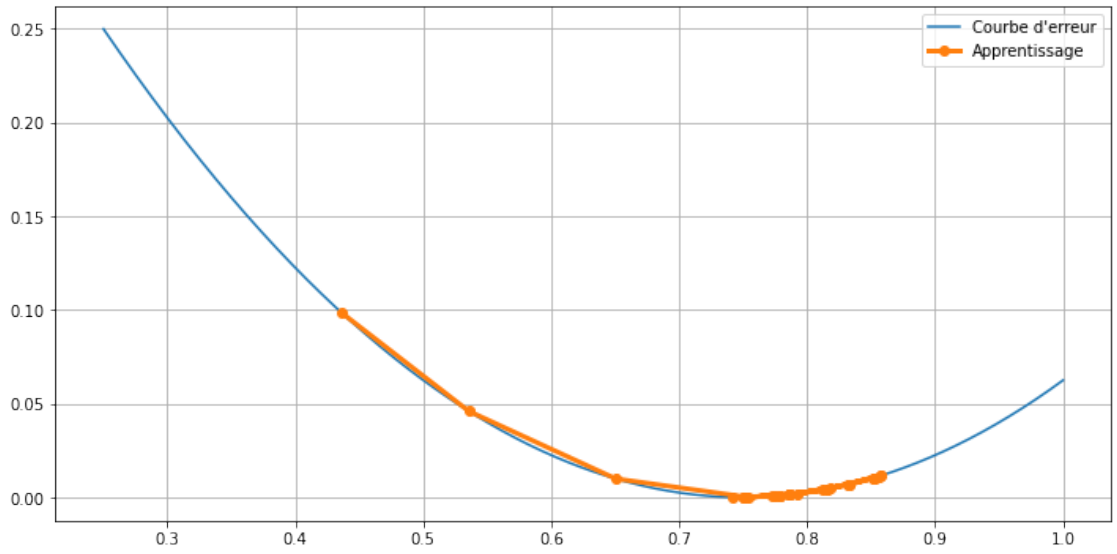
[8]:
```
# Petit learning rate
reseau = NeuronneLineaire(lr=0.1, cible=0.75)
for i in range(4):
    for x, y in zip(liste_x, liste_y):
        reseau.validation()
        reseau.retropropagation(x, y)
reseau.plot()
```



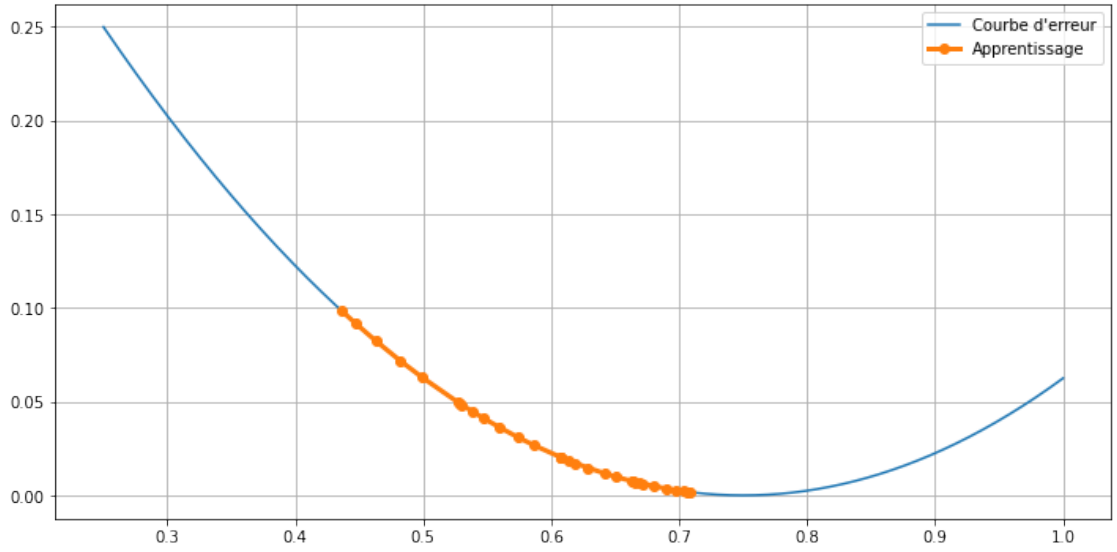
4 SGD Mini Batch

On remarquera qu'avec l'utilisation du Batch, en entrainant sur plusieurs données en même temps (ici des groupes de 4 données) avant de mettre à jour le poids, cela permet d'être moins sujet au bruit de chaque donnée individuelle.

[9]:
```
# Grand Learning rate
reseau = NeuronneLineaire(lr=0.9, cible=0.75)
for i in range(4):
    for i in range(0, 32, 4):
        x, y = liste_x[i:i+4], liste_y[i:i+4]
        reseau.validation()
        reseau.retropropagation(x, y)
reseau.plot()
```



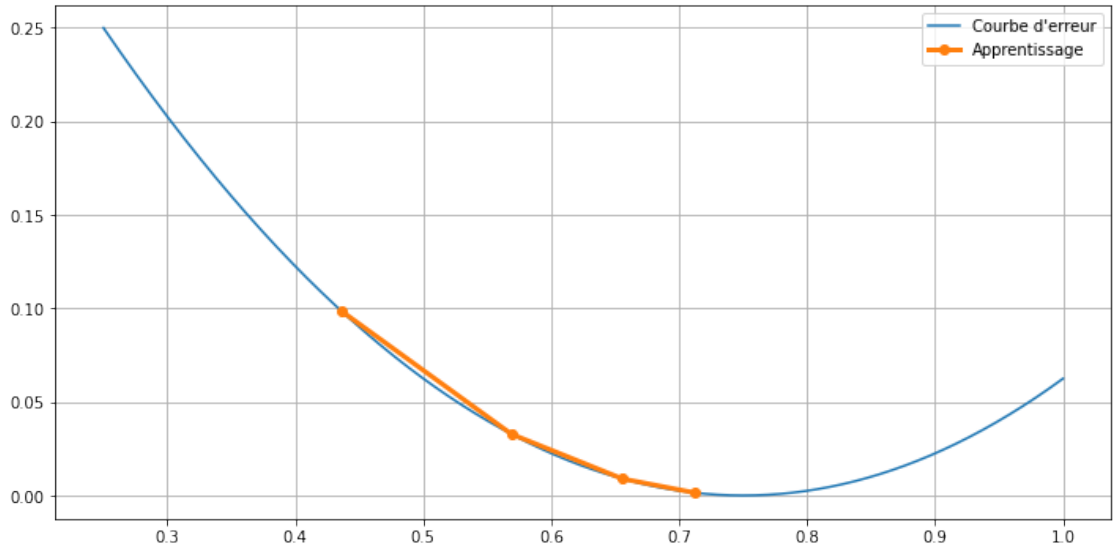
[10]:
```
# Petit Learning rate
reseau = NeuronneLineaire(lr=0.1, cible=0.75)
for i in range(4):
    for i in range(0, 32, 4):
        x, y = liste_x[i:i+4], liste_y[i:i+4]
        reseau.validation()
        reseau.retropropagation(x, y)
reseau.plot()
```



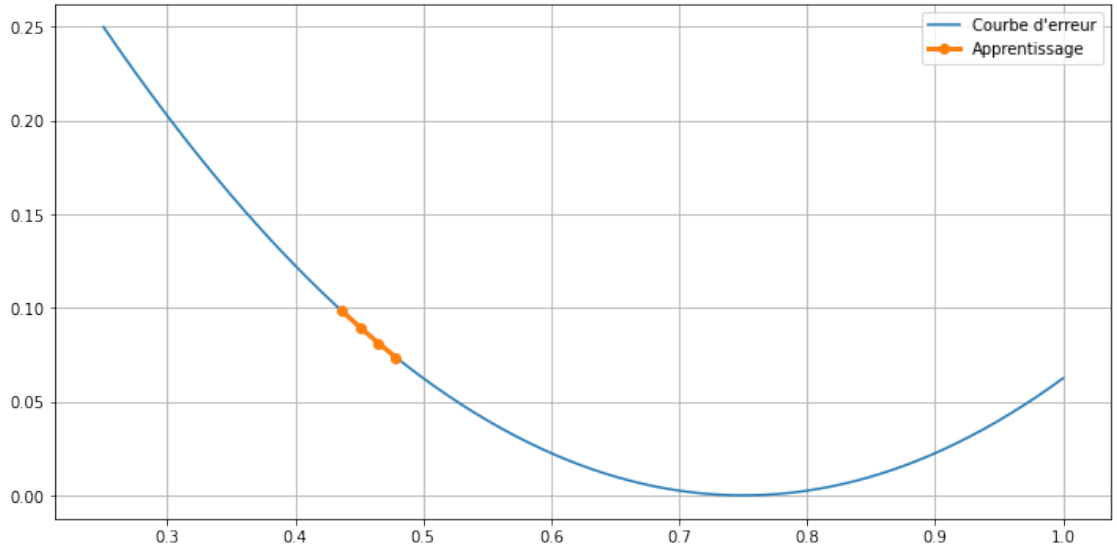
5 Descente de Gradient

L'idée de mettre toutes les données dans le batch et d'entrainer tout d'un coup, est cependant mauvaise, car le réseau de neuronne ne se met pas à jour assez régulièrement. Ce qui entraine une convergence vers le minimum très lent.

[11]:
```
# Grand Learning rate
reseau = NeuronneLineaire(lr=0.9, cible=0.75)
for i in range(4):
    reseau.validation()
    reseau.retropropagation(liste_x, liste_y)
reseau.plot()
```



[12]:
```
# Grand Learning rate
reseau = NeuronneLineaire(lr=0.1, cible=0.75)
for i in range(4):
    reseau.validation()
    reseau.retropropagation(liste_x, liste_y)
reseau.plot()
```



6 Poursuite des Recherches

D'autre paremètres peuvent encore être pris en compte :
- l'accélération (Si l'on se trouve encore loin du minimum, on accélère)
- le moment (Pour ne pas rester bloquer dans un minimum local)

D'autres Optimizer basé sur le SGD :
- Momentum
- Nesterov accelerated gradient
- Adagrad
- Adadelta
- RMSprop
- Adam (couramment utilisé)