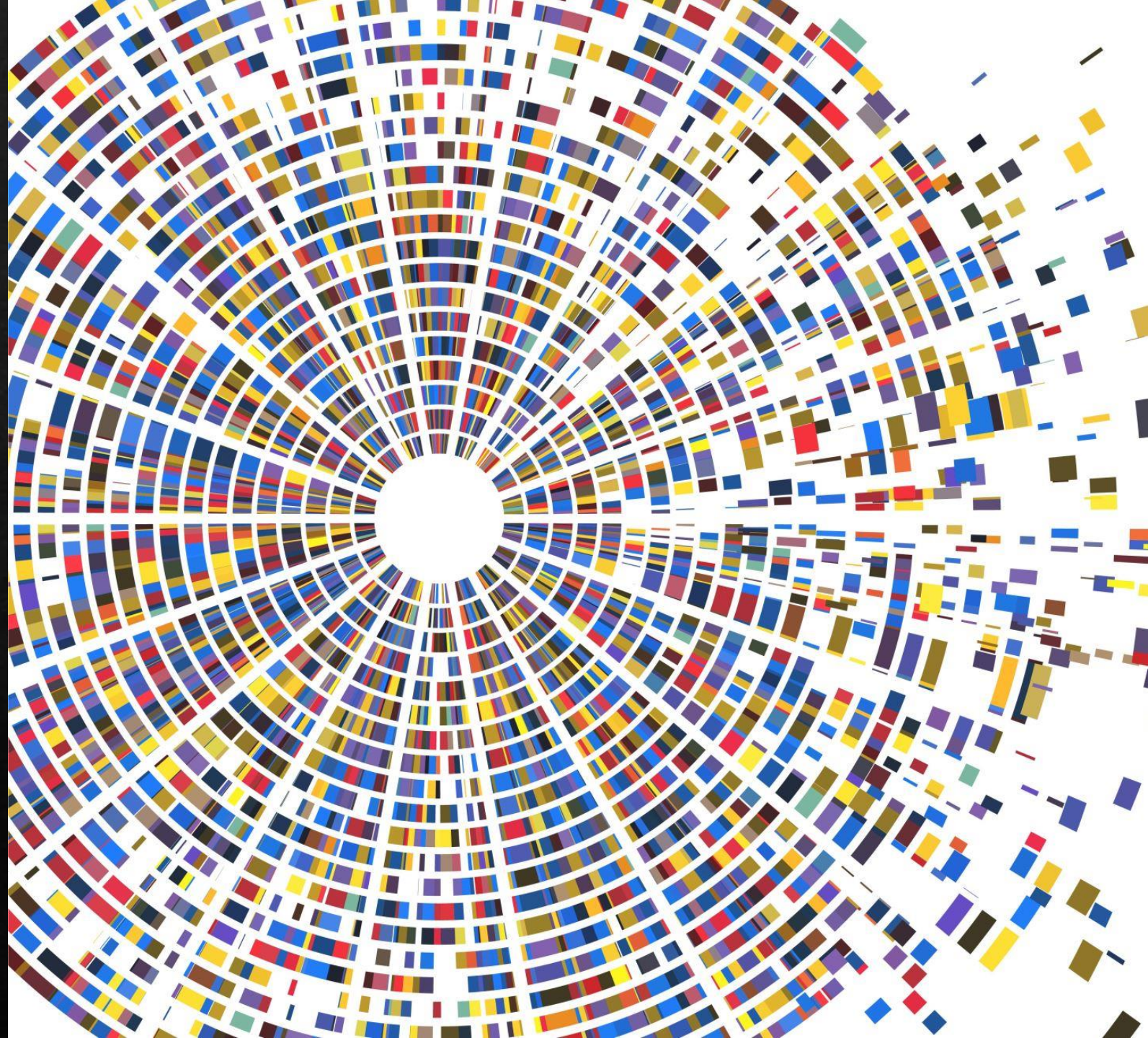
INT 21

Optimisation d'un réseau de neurones artificiel



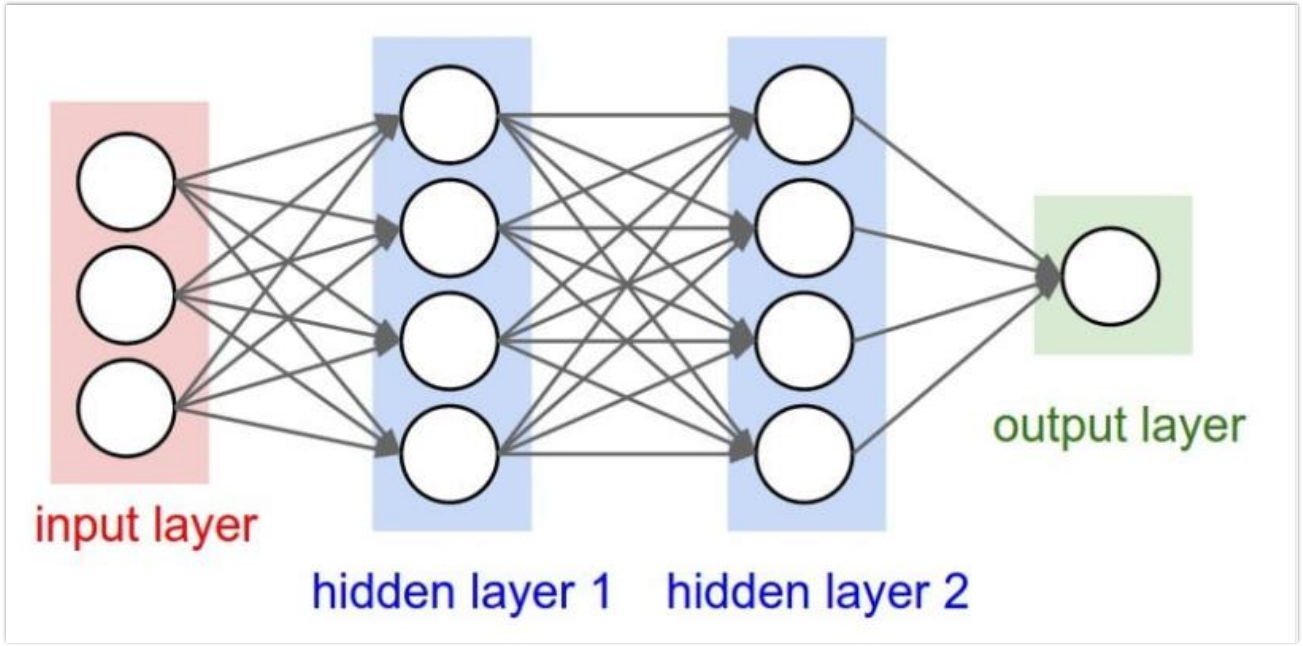
Introduction : comment fonctionne un réseau neuronal ?

Des couches et des neurones

Comment fonctionne l'entrainement :

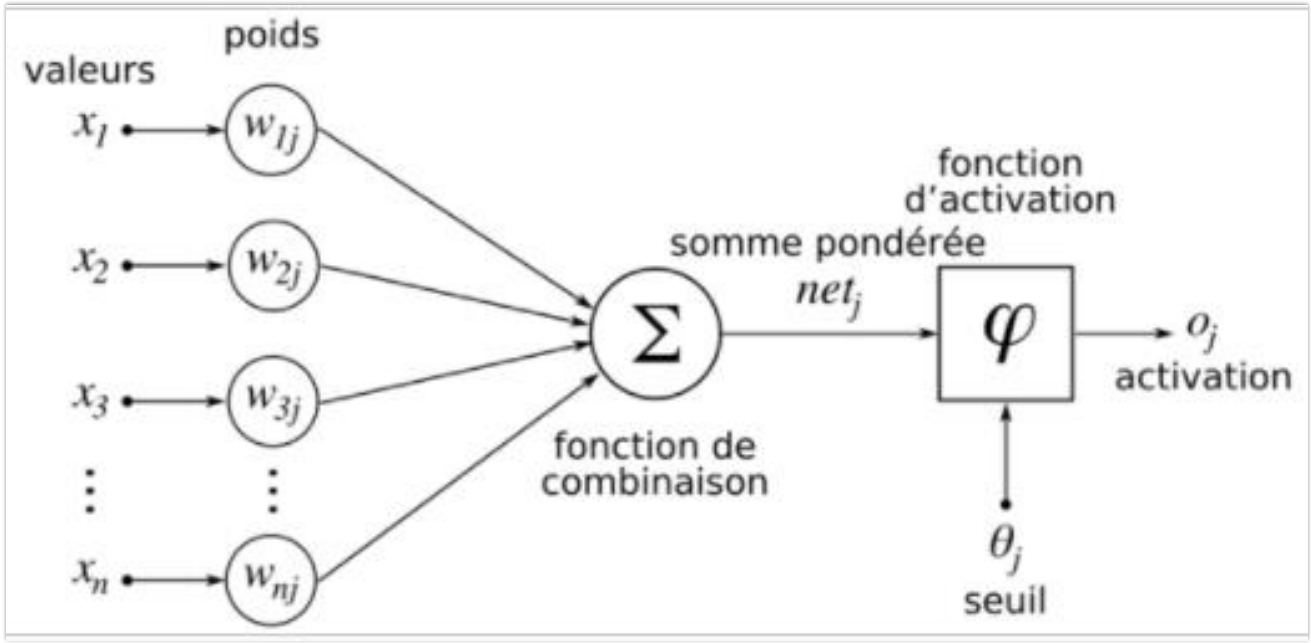
- On passe une donnée dans l'input layer,
- On regarde le résultat qu'on compare avec la sortie voulue
- On calcule l'erreur et on va modifier les poids des branches et les biais des neurones par la méthode de rétropropagation du gradient

forward propagation

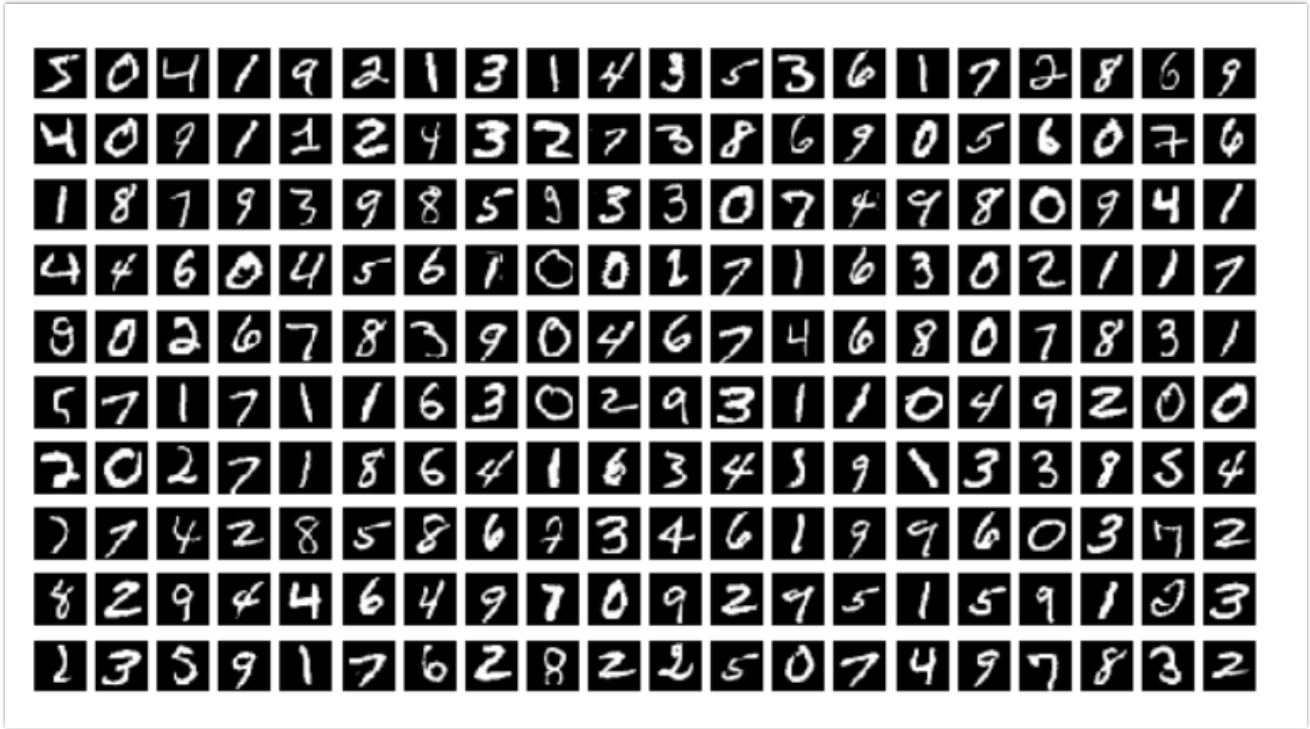


Backward propagation

Fonctionnement d'un neurone



Cas d'application : reconnaitre des chiffres sur une image



Entrée et sortie

- En entrée : une image 28*28 avec des niveaux de gris transformée en vecteur

- En sortie : un vecteur de taille 10 contenant la ressemblance entre l'entrée et chaque chiffre

Exemple : pour un 3 en entrée :

[0.0, 0.1, 0.1, 0.9, 0.1, 0.0, 0.1, 0.1, 0.2, 0.1]

Optimisation génétique des parametres du réseau :

Description de la population

```python
class Population :

    def __init__(self) :
        self.nb_layers = randint(2,5)
        self.nb_neurones_couches = [randint(20,100) for i in range(self.nb_layers-1)]
        self.f_acti = l_fonctions[randint(0,1)]
        self.learning_rate = [0.05 ,0.1, 0.15 ,0.2][randint(0,3)]
```

```python
l_fonctions = [[tanh,tanh_prime],[atan,atan_prime]]
```

Nombre de couche

Nombre de neurones par couches

Fonction d'activation à utiliser

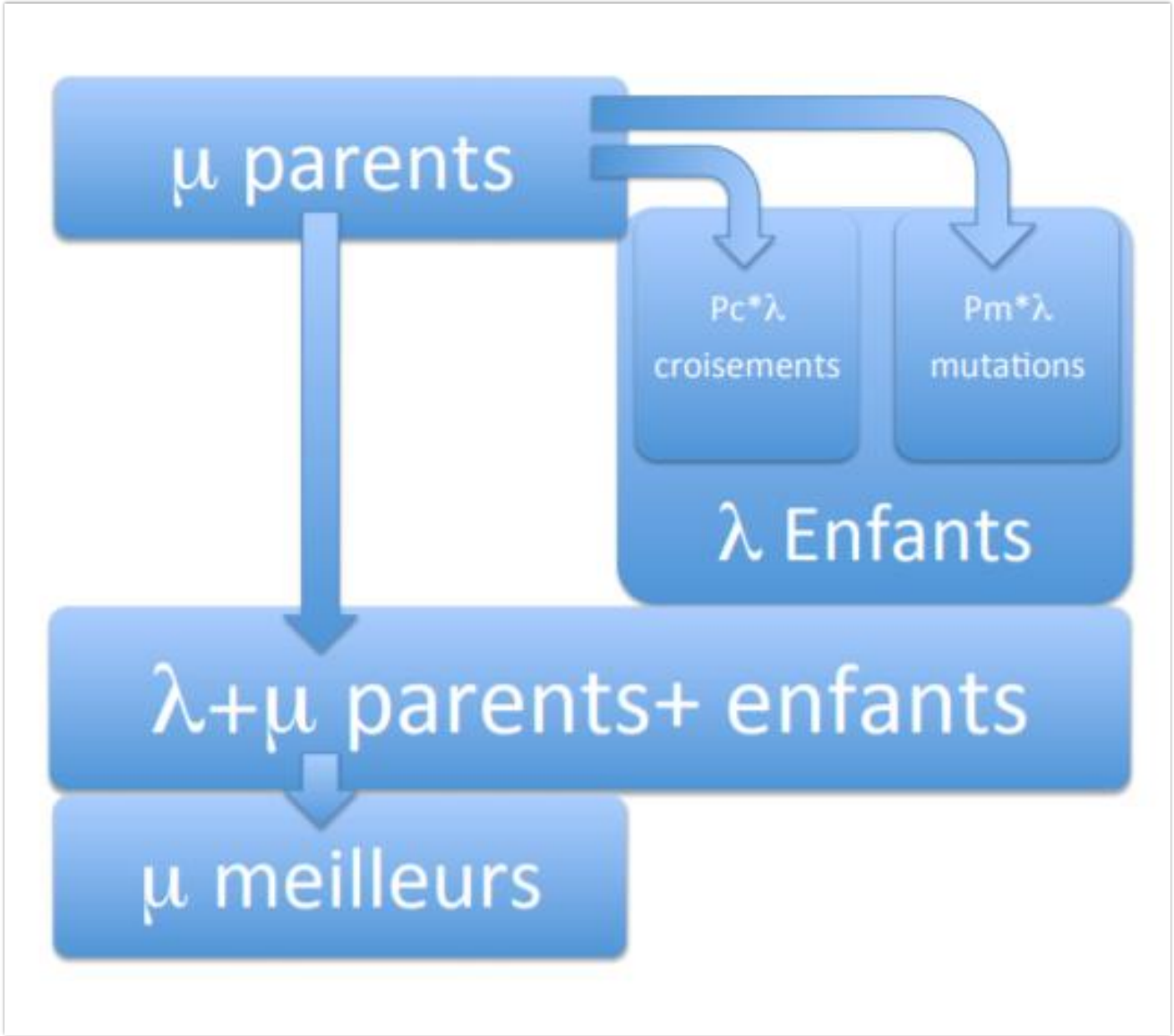
Pas de la descente de gradient pour optimisation du réseau à chaque entrainement

Idée initiale de la fonction de fitness :

- la précision après le dernier entrainement du réseau donnée par :

```
def mse(y_true, y_pred):
    return np.mean(np.power(y_true-y_pred, 2))
```

Renouvellement de la population



1er essai avec un simple operateur de mutation

```python
def mutation(self,ngeneration) :
    mut_f = randint(1,int(50/ngeneration))
    mut_layers = randint(1,int(20*ngeneration/self.accuracy))
    mut_iteration = randint(1,20)
    if mut_f == 1 :
        self.f_acti = l_fonctions[randint(0,1)]
    if mut_layers == 1 and self.nb_layers < 10 :
        self.nb_layers += 1
        self.nb_neurones_couches.append(50)
    if mut_layers == 2 and self.nb_layers > 2 :
        self.nb_layers -= 1
        self.nb_neurones_couches = self.nb_neurones_couches[:-1]
    if mut_iteration == 1 :
        self.nb_trans = randint(5,20)
    for i in range(len(self.nb_neurones_couches)) :
        mut_nb_neurones = randint(1,30)
        if mut_nb_neurones == 1 :
            self.nb_neurones_couches[i] = randint(20,200)
    return self
```

```
un individu est :  nb layers : 6
nb neurones/couches[34, 25, 73, 54, 47]
fonction : [<function tanh at 0x7fb550b95700>, <function tanh_prime at 0x7fb550b95790>]

un individu est :  nb layers : 8
nb neurones/couches[193, 129, 170, 27, 190, 22, 159]
fonction : [<function atan at 0x7fb550b95820>, <function atan_prime at 0x7fb550b958b0>]

un individu est :  nb layers : 5
nb neurones/couches[128, 188, 127, 31]
fonction : [<function atan at 0x7fb550b95820>, <function atan_prime at 0x7fb550b958b0>]
```

Chances de mutations choisies arbitrairement

On a beaucoup de couches et de neurones ce qui n'est pas forcement optimal même si la précision est au rendez vous.

Pb : nombre d'entrainement aleatoire

2eme essai : on fixe le nombre d'étapes et on prend en compte le temps d'exécution

```
un individu est :  nb layers : 3
nb neurones/couches[25, 64]
fonction : [<function tanh at 0x7f6607a60790>, <function tanh_prime at 0x7f6607a60820>]
sa fitness est :  0.002702752248660032

un individu est :  nb layers : 5
nb neurones/couches[62, 92, 29, 27]
fonction : [<function tanh at 0x7f6607a60790>, <function tanh_prime at 0x7f6607a60820>]
sa fitness est :  0.002616340445157202

un individu est :  nb layers : 7
nb neurones/couches[174, 173, 187, 55, 26, 39]
fonction : [<function atan at 0x7f6607a608b0>, <function atan_prime at 0x7f6607a60940>]
sa fitness est :  0.002472783202047333
```

On trouve déjà un nombre de couche moins important pour une des solutions

On implémente ensuite l'operateur de croisement

```
def crossover(self,ind2) :

    #choose nb layers/learning rate
    new = copy.copy(Population())
    chose_layer = randint(1,2)
    chose_learning_rate = randint(1,2)
    if chose_layer == 1 :
        new.nb_layers = self.nb_layers
    else :
        new.nb_layers = ind2.nb_layers

    new.nb_neurones_couches = [0 for i in range(new.nb_layers-1)]
    if chose_learning_rate == 1 :
        new.learning_rate = self.learning_rate
    else :
        new.learning_rate = ind2.learning_rate

    #choose the nb of neurones from the layer
    chose_neurones = [randint(0,1) for i in range(new.nb_layers-1)]
    for k,elem in enumerate(chose_neurones) :
        if elem == 0 :
            try :
                new.nb_neurones_couches[k] = self.nb_neurones_couches[k]
            except :
                new.nb_neurones_couches[k] = ind2.nb_neurones_couches[k]
        else :
            try :
                new.nb_neurones_couches[k] = ind2.nb_neurones_couches[k]
            except :
                new.nb_neurones_couches[k] = self.nb_neurones_couches[k]
    return new
```

Choix du nombre de couches

Choix du coefficient d'apprentissage

Choix du nombre de neurones par couches

Pour des questions de rapidité on limite aussi le nombre de layers à 5 et le nombre de neurones par couches à 60

Résultats :

- On gagne presque un facteur 10 sur la precision !!

- Il y a toujours beaucoup de couches (le maximum)

=> Comment les limiter ?

```
un individu est :
 nb layers : 5
nb neurones/couches[33, 58, 31, 30]
fonction : [<function tanh at 0x7f83ebe26820>, <function tanh_prime at 0x7f83ebe268b0>]
0.2
sa fitness est :  0.0005333148464980189

un individu est :
 nb layers : 5
nb neurones/couches[30, 23, 55, 30]
fonction : [<function atan at 0x7f83ebe26940>, <function atan_prime at 0x7f83ebe269d0>]
0.2
sa fitness est :  0.0009061223938593 66

un individu est :
 nb layers : 5
nb neurones/couches[54, 23, 38, 30]
fonction : [<function atan at 0x7f83ebe26940>, <function atan_prime at 0x7f83ebe269d0>]
0.2
sa fitness est :  0.0007312170667173812

un individu est :
 nb layers : 5
nb neurones/couches[30, 34, 36, 33]
fonction : [<function tanh at 0x7f83ebe26820>, <function tanh_prime at 0x7f83ebe268b0>]
0.2
sa fitness est :  0.0005360165772643754

un individu est :
 nb layers : 5
nb neurones/couches[33, 20, 31, 21]
fonction : [<function tanh at 0x7f83ebe26820>, <function tanh_prime at 0x7f83ebe268b0>]
0.2
sa fitness est :  0.0007465465138747 74

un individu est :
 nb layers : 5
nb neurones/couches[30, 29, 22, 33]
fonction : [<function atan at 0x7f83ebe26940>, <function atan_prime at 0x7f83ebe269d0>]
0.2
sa fitness est :  0.0008030250538448703
```

Modification de la fonction de fitness pour prendre en compte le nombre de nœuds :

- On veut pouvoir jouer plus sur le nombre de neurones/couches pour les limiter

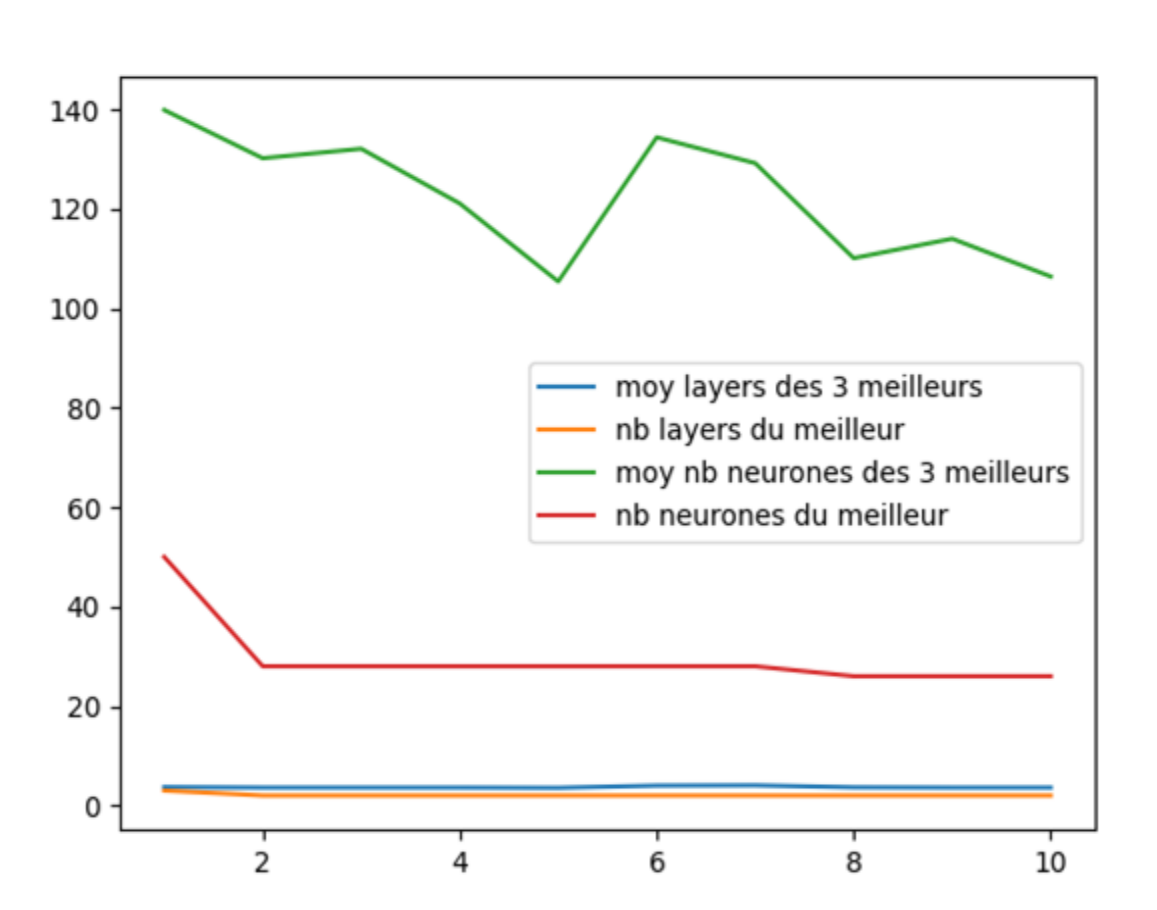
⇒ Il faut donc les prendre en compte dans l'erreur :

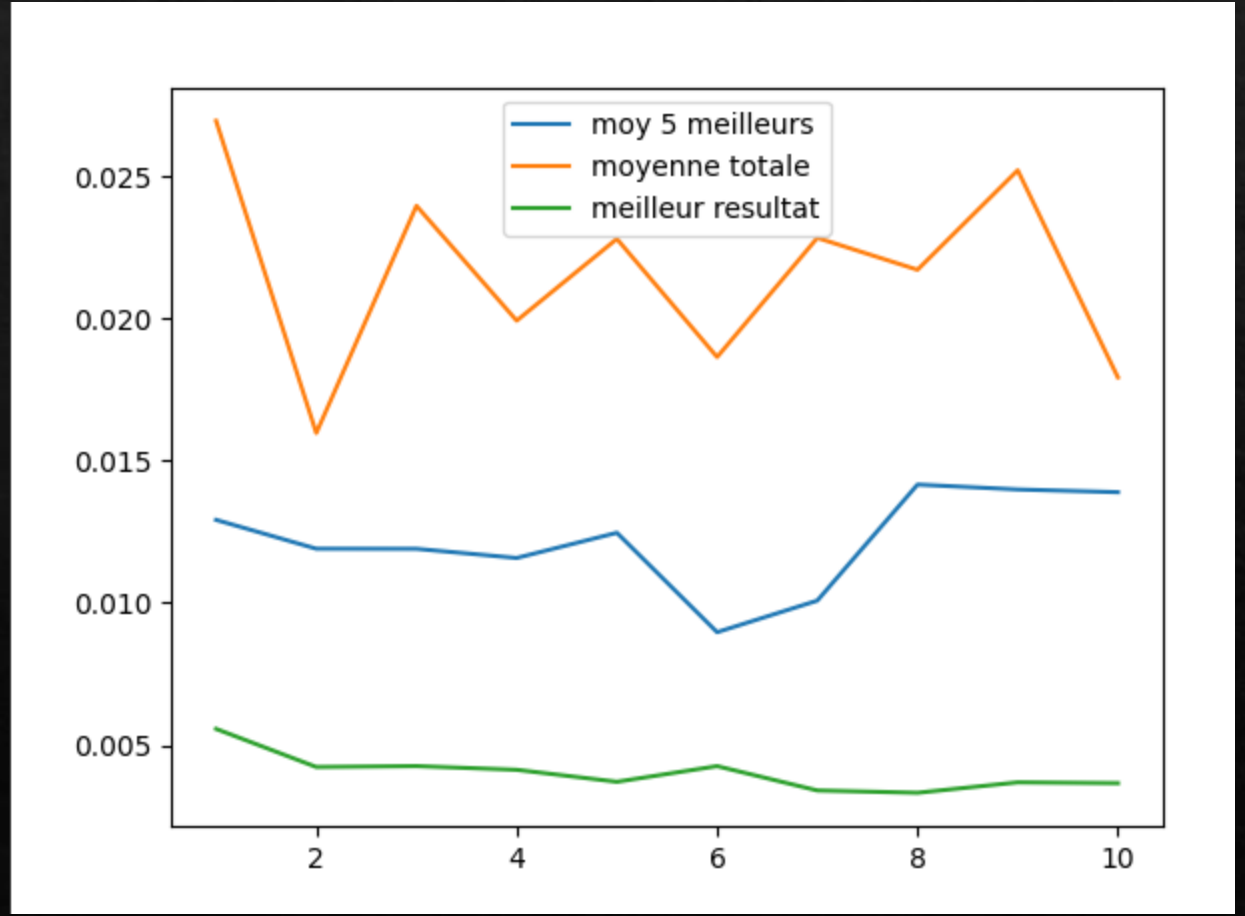
```
return err*(sum(self.nb_neurones_couches))
```

On diminue aussi le nombre d'entrainement de chaque réseau pour gagner du temps (passe de 35 à 15) et pouvoir augmenter la population

Résultats :

```
un individu est :
 nb layers : 2
nb neurones/couches[26]
fonction : [<function atan at 0x7f79a8a034c0>, <function atan_prime at 0x7f79a8a
03550>]
0.2
sa fitness est :  0.8750847721524396
```





Améliorations :

- Changer de structure de réseau de neurones (ici fully connected)
- Ajouter un plus gros panel de fonctions d'activations
- Prendre en compte la décroissance de la l'erreur pour la fonction de fitness (car moins d'iterations