LE POLYTECHNIQUE DE L'UNIVERSITE FRANÇOIS RABELAIS DE TOURS
Spécialité Informatique
64 avenue Jean Portalis,
37200 TOURS, FRANCE
Tél +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Rapport - TP1

Implémentation de l'algorithme de perceptron multicouche

FENG Jiaming 21707371

SONG Miyuan 21707400

Artificial Intelligence — Human Intelligence Exhibited by Machines

Des recherches avant 1950 ont proposé l'idée que le cerveau se compose d'un réseau d'impulsions électriques qui déclenchent et sont soigneusement organisées d'une certaine manière pour former des pensées et de la conscience. Alan Turing a montré que tout calcul peut être mis en œuvre numériquement, et à l'époque, ce n'était pas loin de l'idée de construire une machine capable d'imiter le cerveau humain.

Depuis les années 1950, l'IA moderne se concentre sur ce que l'on appelle l'IA forte. L'IA forte fait référence à l'IA qui peut généralement effectuer n'importe quelle tâche intelligente que les humains peuvent effectuer. La faible progression de l'IA forte conduit finalement à l'IA dite faible, ou à l'application de la technologie de l'IA à une portée plus petite. Jusqu'aux années 80, la recherche en IA était divisée en ces deux paradigmes. Mais vers 1980, l'apprentissage automatique est devenu un domaine de recherche de premier plan, et son objectif était de permettre aux ordinateurs d'apprendre et de construire des modèles pour pouvoir effectuer des activités telles que les prédictions dans des domaines spécifiques.

Machine Learning — Une approche pour atteindre l'intelligence artificielle

Le Machine Learning est à la base de la pratique consistant à utiliser des algorithmes pour analyser des données, en tirer des enseignements, puis faire une détermination ou une prédiction sur quelque chose dans le monde. Ainsi, plutôt que de routines logicielles de codage manuel avec un ensemble spécifique d'instructions pour accomplir une tâche particulière, la machine est «formée» à l'aide de grandes quantités de données et d'algorithmes qui lui permettent d'apprendre à exécuter la tâche.

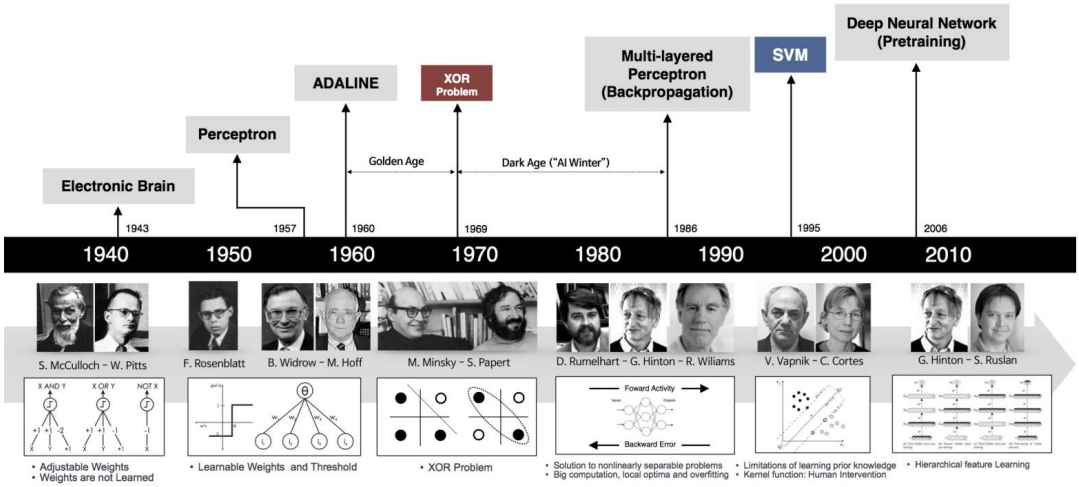
Il s'est avéré que l'un des meilleurs domaines d'application de l'apprentissage automatique pendant de nombreuses années était la vision par ordinateur, même si elle nécessitait encore beaucoup de codage manuel pour faire le travail. Les gens entraient et écrivaient des classificateurs codés à la main comme des filtres de détection de bord afin que le programme puisse identifier où un objet a commencé et s'est arrêté; détection de forme pour déterminer s'il avait huit côtés; un classificateur pour reconnaître les lettres «S-T-O-P». À partir de tous ces classificateurs codés à la main, ils développeraient des algorithmes pour donner un sens à l'image et «apprendraient» pour déterminer s'il s'agissait d'un panneau d'arrêt.

Deep Learning — Une technique pour mettre en œuvre Machine Learning

Le deep learning a vu le jour vers 2000 et est basé sur les résultats de la recherche sur l'IA et le machine learning. Les informaticiens utilisent les réseaux de neurones dans de nombreuses couches grâce à de nouvelles topologies et méthodes d'apprentissage. Cette évolution des réseaux de neurones a réussi à résoudre des problèmes complexes dans divers domaines. Les réseaux de neurones sont inspirés par notre compréhension de la biologie de notre cerveau - toutes ces interconnexions entre les neurones. Mais, contrairement à un cerveau biologique où n'importe quel neurone peut se connecter à n'importe quel autre neurone à une certaine distance physique, ces réseaux de neurones artificiels ont des couches discrètes, connexions et directions de propagation des données.

Vous pouvez, par exemple, prendre une image, la découper en un tas de tuiles qui sont entrées dans la première couche du réseau neuronal. Dans la première couche, les neurones individuels, puis transmettent les données à une deuxième couche. La deuxième couche de les neurones font leur tâche, et ainsi de suite, jusqu'à ce que la couche finale et la sortie finale soient produites.

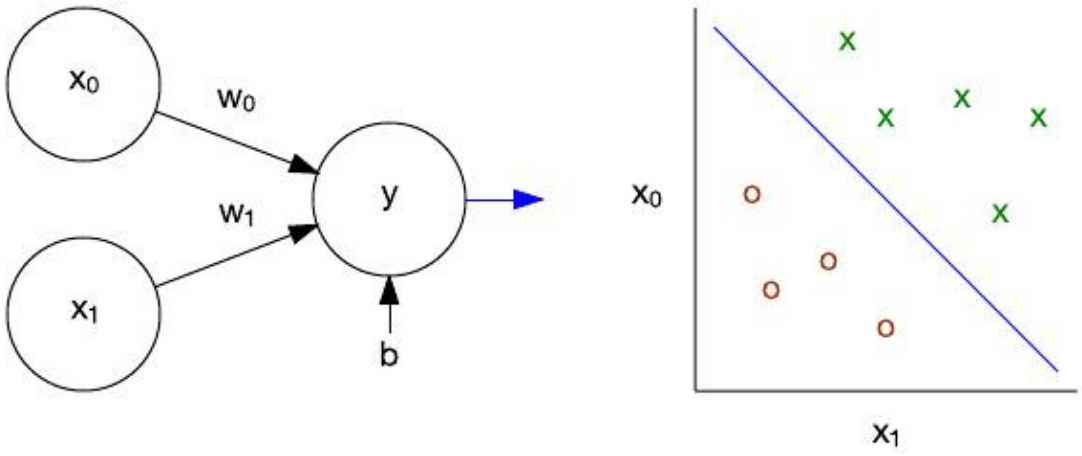
Chaque neurone attribue une pondération à son entrée - à quel point elle est correcte ou incorrecte par rapport à la tâche exécutée. La sortie finale est ensuite déterminée par le total de ces pondérations. Alors, pensez à notre exemple de panneau d'arrêt. Les attributs d'une image de panneau d'arrêt sont découpés et «examinés» par les neurones - sa forme octogonale, sa couleur rouge de pompier, ses lettres distinctives, sa taille de panneau de signalisation et son mouvement ou son absence. La tâche du réseau neuronal est de déterminer s'il s'agit d'un arrêt signe ou non. Il arrive avec un «vecteur de probabilité», vraiment une supposition très instruite, basée sur la pondération. Dans notre exemple, le système pourrait être à 86% confiant que l'image est un panneau d'arrêt, 7% confiant qu'il s'agit d'un signe de limite de vitesse , et 5% c'est un cerf-volant coincé dans un arbre, et ainsi de suite - et l'architecture du réseau indique ensuite au réseau de neurones s'il a raison ou non.



L'histoire générale de Deep Learning

Perceptron

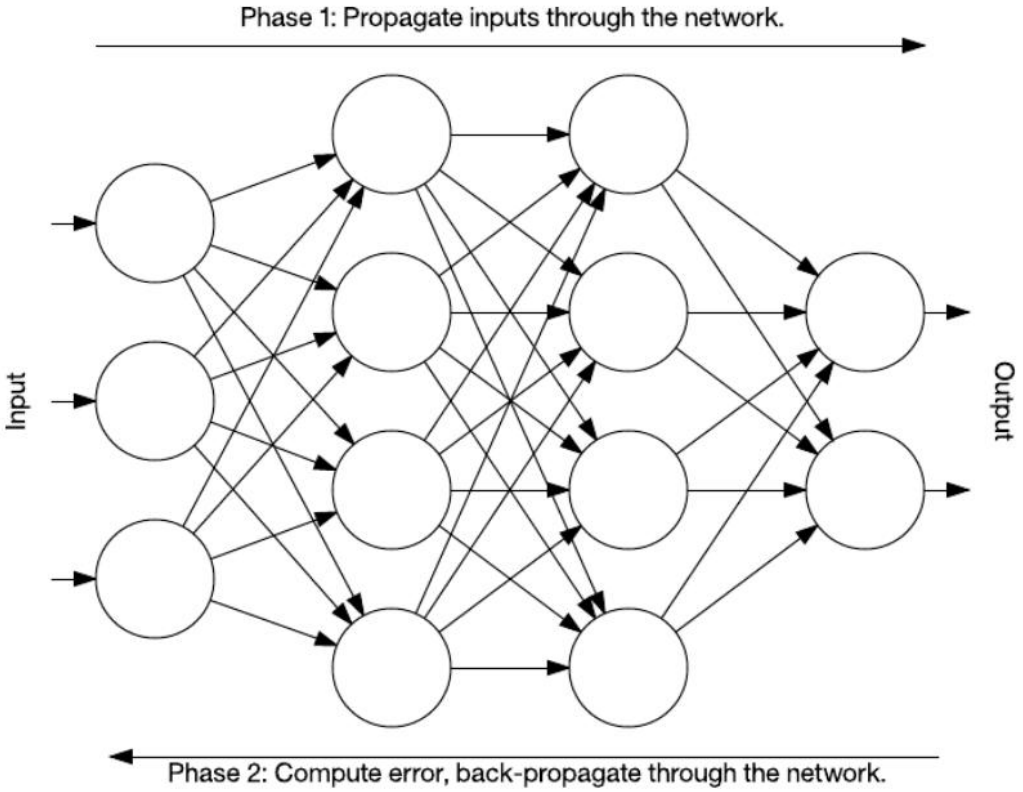
Avant de parler de perceptrons multicouches, parlons de perceptrons purs. Les perceptrons sont un algorithme d'apprentissage supervisé précoce pour les réseaux de neurones monocouche. Étant donné un vecteur de caractéristique d'entrée, l'algorithme perceptron peut apprendre à classer l'entrée en catégories spécifiques. En utilisant l'ensemble d'apprentissage, les poids et les seuils du réseau classifié linéairement peuvent être mis à jour. Le perceptron a été initialement implémenté pour l'IBM 704 et a ensuite été utilisé sur du matériel personnalisé pour la reconnaissance d'image. En fait, un simple perceptron est une structure de réseau composée de deux couches de neurones, à savoir la couche d'entrée et la couche de sortie. La couche d'entrée accepte les données externes, transforme la structure en couche de sortie grâce au calcul des fonctions d'activation et des seuils. Le perceptron est devenu plus tard la base structurelle de nombreux réseaux de neurones, et sa fondation théorique a également été construite sur le "modèle de neurone M-P" par Petz et al. Tout d'abord, il est souligné ici que le perceptron est un apprentissage supervisé, donc c'est un algorithme de classification (mais il ne peut pas résoudre le problème XOR ...)



Perceptron Multicouche(Backpropagation)

La véritable puissance des réseaux de neurones réside dans leurs multiples couches de déformation. La formation d'un perceptron monocouche est simple, mais le réseau résultant n'est pas très puissant. La question devient donc comment former un réseau à plusieurs couches? Des algorithmes de rétropropagation sont utilisés.

La rétropropagation est un algorithme qui forme un réseau neuronal à plusieurs couches. Elle est réalisée en deux phases. La première étape consiste à propager l'entrée à la dernière couche (connue sous le nom de feedforward) via un réseau de neurones. Dans la deuxième étape, l'algorithme calcule une erreur, puis la propage en arrière de la dernière couche (ajustement des poids) à la première couche.



Nous savons que les réseaux de neurones ont tous une fonction de perte. Cette fonction a des définitions différentes selon les différentes tâches, mais le but de cette fonction de perte est de calculer la distance entre les données de sortie modélisées par le réseau neuronal actuel et les données idéales. Après avoir calculé la perte, selon l'algorithme de rétropropagation, divers paramètres du réseau peuvent être mis à jour afin que la perte diminue continuellement, ce qui peut rendre les données de sortie plus idéales.

Algorithme de descente de gradient pour l'optimisation

Après avoir donné une fonction de perte, elle est exprimée en C. Pour le dire clairement, la rétropropagation consiste à trouver $\partial C / \partial w$ et $\partial C / \partial b$, puis à multiplier cette valeur par les w et b correspondants pour la soustraire. Les paramètres ont été

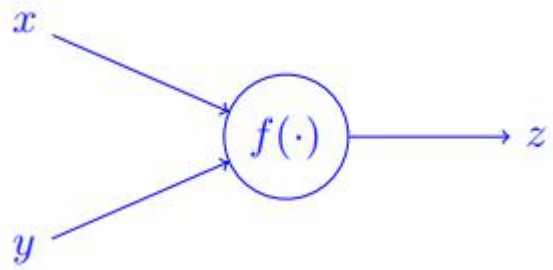
mis à jour. Pourquoi de telles opérations peuvent-elles optimiser le réseau et réduire la valeur des pertes?

Le concept dérivé liés à la vitesse. ∂C / ∂w et ∂C / ∂b correspondent à la vitesse à laquelle la valeur de perte C change par rapport à w et v. Si ∂C / ∂w est positif, augmentez w et C. augmentera. Si vous voulez que C diminue, vous devez diminuer w; et plus la valeur absolue de ∂C / ∂w est grande, la valeur de w par rapport à C. Plus l'impact est important, un léger changement de w et un grand changement de C. Si vous souhaitez optimiser le C pour qu'il devienne plus petit, combien devrait w diminuer en conséquence? Il n'y a pas non plus de réponse définitive. Ici, nous multiplions la vitesse de changement et le taux d'apprentissage comme une valeur réduite. Grâce à plusieurs cycles d'itération. Au final, nous voulons que c atteigne un minimum.

Mise en oeuvre

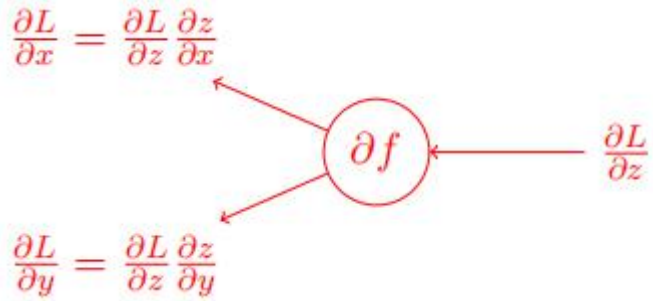
Dans notre TP, nous divisons notre algorithme en deux parties, propagation avant(forward) et propagation arrière(backward).

Propagation avant : fournir le réseau un échantillon des entrées, calculer l'activation et la sortie pour chaque neurone dans l'ordre croissant.




```
# propager l'entrée à la dernière couche
def forward(self):
    self.Z=[]  # Sortie de nœuds neuronaux
    self.A=[]  # Valeur d'entrée
    a=self.data
    self.A.append(a)
    # Calculez la valeur de sortie de chaque nœud
    for i in range(self.num_layer-1):
        #zl = self.weights[l] * self.data[l-1] + self.biases[l]
        #
        if self.is_bias:
            z=np.dot(self.weights[i].T,a)+self.biases[i]
        else:
            z=np.dot(self.weights[i].T,a)
        a=np.maximum(z,0)  # relu Fonction d'activation
        self.Z.append(z)
        self.A.append(a)
    out = np.dot(self.weights[self.num_layer-1].T,a) # Résultats de la couche de sortie
    self.out_softmax = self.softmax(out)
```

Propagation arrière : évaluer la dérivé de la fonction de pertes pour les paramètres et les mettre à jour pour chaque neurones dans l'ordre décroissant.



Pour appliquer les méthodes basées des algorithmes de gradients, il faut calculer : ∂J / ∂Wl , ∂J / ∂bl , l = 1, 2, ..., L

```
def backward(self):
    self.E = []
    self.dW = []
    self.db = []
    # eL =∂J / ∂zL  le gradient de l'erreur selon bL = eL
    e = (self.out_softmax - self.labels)/(self.labels.shape[1])
    dw = np.dot(self.A[-1],e.T) # Calcul le gradient de l'erreur selon WL
    self.dW.append(dw)
```

Pour chaque l = L L 1, L L 2, ...2, 1, calcul d'erreur de sortie de la couche l :

$$e^l = (\mathbf{W}^{l+1}e^{l+1}) \odot f'(z^l)$$

Où ⊙ est la multiplication par élément entre 2 tenseurs de dimension égale.

Répéter la procédure de la couche l à $l-1$:

$$\frac{\partial J}{\partial \mathbf{W}^l} = \frac{\partial J}{\partial z^l}\frac{\partial z^l}{\partial \mathbf{W}^l} = e^l(a^{l-1})^T$$

$$\frac{\partial J}{\partial b^l} = \frac{\partial J}{\partial z^l}\frac{\partial z^l}{\partial b^l} = e^l$$

```
# calcul d'erreur de sortie de la couche l
for i in range(self.num_layer-1):
    a = self.A[-i-2]
    z = self.Z[-i-1]
    e = np.dot(self.weights[-i-1], e)
    e[z<=0] = 0
    dw = np.dot(a,e.T)
    if self.is_bias:
        db = np.sum(e, axis=1, keepdims=True)
        self.db.append(db)
    self.dW.append(dw)
```

Enfin , nous utilisons les résultats calculés pour ajuster les poids et les biases.

```
# Ajuster les poids et les biais
for i in range(self.num_layer):
    self.weights[i] += -self.lr * self.dW[-i-1]
if self.is_bias:
    for i in range(self.num_layer-1):
        self.biases[i] += -self.lr * self.db[-i-1]
```

Quelques points dans TP

Voici la sortie idéale pour la comparaison.

	0	1	2	3	4	5	6	7	8	9	10
0	0	2	2	0	0	1	2	0	2	1	0

Il est converti sous la forme suivante par la fonction convert_labels.

	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	1	0	0	0	1	0
1	0	0	0	1	1	0	0	1	1	0	0
2	0	1	1	0	0	0	1	0	0	0	1

La fonction de coût:

$$J(\hat{\mathbf{Y}}, \mathbf{Y}) = -\frac{1}{M}\sum_{i=1}^{M}\sum_{j}^{C} y_{ji} \log \hat{y}_{ji}$$

```
# cost or loss function
def cost(self,Y,Yhat):
    return -np.sum(Y*np.log(Yhat))/Y.shape[1]
```

Softmax

Dans le machine learning, en particulier le deep learning, softmax est une fonction très courante et importante, en particulier dans les scénarios multi-classes Il mappe certaines entrées à des nombres réels entre 0-1, et la somme garantie normalisée est 1, donc la somme des probabilités de classifications multiples est exactement 1.

```
# Softmax score
def softmax(self,V):
    e_V = np.exp(V - np.max(V, axis = 0, keepdims = True))
    Z = e_V / e_V.sum(axis = 0)
    return Z
```

Résultats après softmax:

	0	1	2	3	4	5	6	7	8	9	10
0	0.32798	0.31426	0.32399	0.31896	0.31789	0.32429	0.33573	0.32373	0.32706	0.32347	0.31102
1	0.31710	0.28862	0.30990	0.29856	0.29625	0.31060	0.31663	0.30931	0.31723	0.30870	0.28205
2	0.35492	0.39712	0.36610	0.38248	0.38587	0.36511	0.34764	0.36696	0.35571	0.36783	0.40693

Résultat du test

```
Number of neurones in layer1?10
Number of neurones in layer2?10
Number of neurones in layer3?10
Number of neurones in layer4?10
28.407445340652746
23.986417930906576
20.189519288000973
16.590172207447914
13.066780729854637
9.582412081153727
```

```
0.17237599421150065
0.17232351127048812
0.17227107211478812
0.17221868296159892
0.17216628820723695
0.17211384749713643
---> 0.1720614848048155
```

	0	1	2	3	4	5
0	1	1	1	1	0	0
1	0	0	0	0	1	0
2	0	0	0	0	0	1

	0	1	2	3	4	5
0	0.98656	0.94638	0.98361	0.97907	0.00694	0.00001
1	0.01341	0.05351	0.01635	0.02087	0.88926	0.01495
2	0.00004	0.00012	0.00003	0.00006	0.10381	0.98504

Les résultats des tests sont très réussis, on constate que la réduction des pertes est très évidente.

Après les tests, nous avons constaté que plus il y a de neurones dans chaque couche, plus les résultats de prédiction sont précis après un nombre limité d'itérations