optimisation

December 17, 2023

1 Idée 1

1.1 Une des idées que nous allons attaquer lors de ce TP est la deuxième : améliorer l'efficacité numérique ou la lisibilité de la fonction backPropagation donnée.

Cependant, étant donné que nous travaillons en Python, nous entreprendrons la traduction du code R fourni vers Python.

La traduction littérale est la suivante :

[60]:
```
import timeit
import numpy as np

np.random.seed(5678)

def sigmoid(x):
    return 1/(1+np.exp(-x))

def sigmoidDeriv(x):
    return x*(1-x)

def createRandomWeightsByLayer(layerSizes):
    return [np.random.normal(loc=0, scale=np.sqrt(1/(layerSizes[i]+1)),␣
 ↪size=(layerSizes[i]+1, layerSizes[i+1])) for i in range(len(layerSizes)-1)]

def square_loss_deriv(pred, y):
    return np.transpose(pred - y)

def backPropagation(dataIn, dataOut, weightsByLayer, actFunc, actFuncDeriv,␣
 ↪lossDeriv=square_loss_deriv):
    nOutLayer = dataOut.shape[0]
    weights = weightsByLayer.copy()
    weights.append(np.concatenate(
        (np.diag(np.ones(nOutLayer)), np.zeros(nOutLayer).reshape(1, -1)),␣
 ↪axis=0))
    numberOfLayers = len(weights)
    A = [None] * numberOfLayers
    Z = [None] * numberOfLayers
```

```
    # forward loop
    for i in range(numberOfLayers):
        if i == 0:
            A[i] = np.concatenate(
                (dataIn, np.ones((1, dataIn.shape[1]))), axis=0)
        else:
            A[i] = np.concatenate(
                (actFunc[i - 1](Z[i - 1]), np.ones((1, dataIn.shape[1]))),␣
↪axis=0)
        Z[i] = np.dot(weights[i].T, A[i])

    # backward loop
    delta = lossDeriv(Z[numberOfLayers - 1], dataOut)
    dloss_dweight = [None] * (numberOfLayers - 1)
    for i in range(numberOfLayers - 1, 0, -1):
        Nbneurons = Z[i - 1].shape[0]
        D = [None for _ in range(Nbneurons)]
        for j in range(Nbneurons):
            D[j] = actFuncDeriv[i - 1](Z[i - 1][j])
        delta = np.dot(np.dot(delta, weights[i].T), np.vstack(
            (np.diag([x[0] for x in D]), np.zeros(Nbneurons))))
        dloss_dweight[i - 1] = np.dot(A[i - 1], delta)
    dloss_dtheta = None
    for i in range(numberOfLayers - 1):
        dloss_dtheta = [dloss_dtheta, dloss_dweight[i]]
    return Z[numberOfLayers - 1], dloss_dtheta
```

1.1.1 Nous allons optimiser ce code en adoptant plusieurs solutions:

1) Utiliser la librairie numpy du python puisqu'elle utilise C++ en interne(under the hood).

2) Utiliser la compréhension de liste au lieu des boucles for (pour une exécution plus rapide).

3) Epargner de la mémoire en évitant de stocker des résultats intermédiaires.

4) Organiser chaque section dans une fonction pour améliorer la lisibilité.

5) Condenser plusieurs lignes de code contenant des boucles for en une seule ligne de code équivalent.(utilisation des "magical" one liners for replacing equivalent for loops).

Dans le code optimisé, j'ai inclus des commentaires fournissant une démonstration de la mise en œuvre de chaque idée d'optimisation, indiquant où et comment elles ont été appliquées dans l'algorithme.

Il est important de noter que nous avons accéléré l'exécution du code en appliquant les idées mentionnées précédemment, tout en conservant le même schéma d'exécution. En d'autres termes, nous avons optimisé les performances sans altérer le concept

fondamental et la logique sous-jacente de l'algorithme.

1.1.2 Le code optimisé sera le suivant:

[61]:
```
def _forward_propagation(dataIn, weightsByLayer, actFunc, A, Z):
    for i in range(1, len(weightsByLayer)):
        A.append(np.concatenate(
            (actFunc[i - 1](Z[i - 1]), np.ones((1, dataIn.shape[1]))), axis=0))
        Z.append(np.dot(weightsByLayer[i].T, A[i]))


def _back_propagation(dataOut, weightsByLayer, actFuncDeriv, lossDeriv, A, Z):
    numberOfLayers = len(weightsByLayer)
    # minimisation de l'appel du variable numberOfLayers pour l'indexage: par␣
 ↪exepmle Z[-1] au lieu de Z[numberOfLayers-1]
    delta = lossDeriv(Z[ - 1], dataOut)
    dloss_dweight = [None for _ in range(numberOfLayers - 1)]
    #exemple de condensation de plusieurs boucles for en une seule ligne de code
    delta = np.dot(np.dot(delta, weightsByLayer[ - 1].T), np.vstack(
        (np.diag([actFuncDeriv[ - 2](Z[ - 2][j])[0] for j in range(Z[ - 2].
 ↪shape[0])]), np.zeros(Z[ - 2].shape[0]))))
    dloss_dweight[ - 2] = np.dot(A[ - 2], delta)
    # exemple de l'utilisation de la comprehension des listes + élimination du␣
 ↪variable intermédiaire dloss_dtheta
    return [dloss_dweight[i] for i in range(numberOfLayers - 1)]


def back_propagation(dataIn, dataOut, weightsByLayer, actFunc, actFuncDeriv,␣
 ↪lossDeriv=square_loss_deriv):
    nOutLayer = dataOut.shape[0]
    weights = weightsByLayer.copy()
    weights.append(np.concatenate(
        (np.diag(np.ones(nOutLayer)), np.zeros(nOutLayer).reshape(1, -1)),␣
 ↪axis=0))

    # exemple d'utilisation de la comprehension des listes + élimination de la␣
 ↪declaration non nécessaire des variables A et Z
    A = [np.concatenate((dataIn, np.ones((1, dataIn.shape[1]))), axis=0)]
    Z = [np.dot(weights[0].T, A[0])]
    _forward_propagation(dataIn, weights, actFunc, A, Z)

    # Backward propagation avec matrix multiplication et list comprehension
    dloss_dtheta = _back_propagation(
        dataOut, weights, actFuncDeriv, lossDeriv, A, Z)

    return Z[-1], dloss_dtheta
```

La lisibilité s'est amélioré en organisant le code dans des fonctions.

1.1.3 Maintenant, nous allons tester et comparer la rapidité des deux codes:

Ce test implique le benchmarking, où nous procédons de tester chacun des algorithmes sur un réseau de neurone 1000 fois,
Ensuite, nous calculons le temps moyen de ces 1000 essais.
Le temps moyen représentera le temps d'execution de chacun des algorithms

[70]:
```
layerSizes = [2, 3, 4, 2]
actFuncByLayer = [sigmoid, sigmoid, np.tanh, np.tanh]
DerivActFunc = [sigmoidDeriv, sigmoidDeriv, np.tanh, np.tanh]
Weights = createRandomWeightsByLayer(layerSizes)
dataInputs = np.random.rand(layerSizes[0], 1)
dataOutputs = np.random.rand(layerSizes[-1], 1)

# Performance comparison
print("Benchmarking...")
print("Running the functions 1000 times...")
times_backPropagation = []
for _ in range(1000):
    start_time = timeit.default_timer()
    BP = backPropagation(dataIn=dataInputs, dataOut=dataOutputs,
                         weightsByLayer=Weights, actFunc=actFuncByLayer,␣
 ↪actFuncDeriv=DerivActFunc)
    elapsed = timeit.default_timer() - start_time
    times_backPropagation.append(elapsed)

times_back_propagation = []
for _ in range(1000):
    start_time = timeit.default_timer()
    BP = back_propagation(dataIn=dataInputs, dataOut=dataOutputs,
                          weightsByLayer=Weights, actFunc=actFuncByLayer,␣
 ↪actFuncDeriv=DerivActFunc)
    elapsed = timeit.default_timer() - start_time
    times_back_propagation.append(elapsed)

print("Pour back propagation original: ", np.mean(times_backPropagation))
print("Pour back propagation optimisé: ", np.mean(times_back_propagation))
```

```
Benchmarking...
Running the functions 1000 times...
Pour back propagation original:  8.204289996501757e-05
Pour back propagation optimisé:  4.0525399965190446e-05
```

Pour le code original, le temps moyen du l'execution de l'algorithme d'environ 8.2×10^{-5} secondes, tandis que pour le code optimisé, le temps moyen du l'execution de l'alogrithm est 4×10^{-5} secondes,
Le rapport du temps d'exécution optimisé sur le temps moyen original est d'environ 2
Ainsi, l'algorithme optimisé est deux fois plus rapide que l'original.

2 Idée 2

2.0.1 Maintenant, nous aborderons à ameliorer davantage l'algorithm en modifiant la logique sous-jacente du code original, tout en attaquant la 1ere idée de ce TP: brancher la backpropagation sur l'apprentissage du réseau.

2.0.2 En d'autres termes, dans la suite, nous allons améliorer le code original de la backPrpagation , en utilisant les points mentionnés précédemment(comprehension des listes, etx...) tout en modifiant sa logique, puis nous appliquerons cet algorithm optimisé à l'apprentissage d'un reseau de neurones.

Dans cette version, nous avons effectué une refonte majeure de l'algorithme de rétropropagation.

Nous avons apporté les modifications suivantes :

1. Nous avons choisi de renvoyer uniquement les poids du réseau neuronal, ce qui nous a permis de supprimer de nombreuses variables intermédiaires.
2. Nous avons nettoyé considérablement l'algorithme en éliminant de longues instructions d'une seule ligne tout en préservant les performances. Nous avons également déplacé chaque étape de l'algorithme dans sa propre fonction.
3. Nous avons appliqué une boucle d'entraînement complète (epochs, taux d'apprentissage, etc.) à l'algorithme.

2.0.3 Analysons le code:

Tout d'abord, Nous chargeons les librairies nécessaires.

1) Librairie numpy: Une puissante bibliothèque de calcul numérique.
2) Librarie timeit: permet de mesurer le temps d'exécution du code.

[99]:
```
import timeit
import numpy as np
```

Nous définissons la graine(seed): Cela garantit la reproductibilité des nombres aléatoires.

[100]:
```
np.random.seed(5678)
```

Nous définissons les fonctions d'activations et leurs derivées.

[108]:
```
def sigmoid(x):
    return 1/(1+np.exp(-x))

def identite(x):
    return x

def identiteDeriv(x):
    return 1

def sigmoidDeriv(x):
```

```
    return x*(1-x)

def tanhDeriv(x):
    return 1 - np.tanh(x)**2
```

Nous définissons une fonction d'initialisation aléatoire des poids.

[102]:
```
def createRandomWeightsByLayer(layerSizes):
    return [np.random.rand(layerSizes[i], layerSizes[i+1]) for i in␣
 ↪range(len(layerSizes)-1)]
```

Nous définissons la dérivée de la fonction de coût, qui est l'erreur quadratique moyenne (EQM).

[103]:
```
def square_loss_deriv(pred, y):
    return 2 * (pred - y) / len(y)
```

Nous définissons la fonction de Forward Propagation:

1) La fonction prend comme paramètres:
    1. dataIn: Les données d'entrée pour le réseau de neurone concernant un exemple d'entraînement unique.
    2. weightsByLayer : Une liste contenant les matrices de poids pour chaque couche du réseau de neurones.
    3. actFunc : Une liste de fonctions d'activation correspondant à chaque couche.

2) La fonction travaille de la manière suivante:
    1. La fonction initialise layers_output avec les données d'entrée.

    2. La boucle for itère ensuite à travers chaque couche du réseau de neurones, et calcul la sortie pour chaque couche :
        1. layer_input : Représente la sortie de la couche précédente (les données d'entrée pour la première couche).

        2. weights : Représente la matrice de poids pour la couche.

        3. np.dot(layer_input, weights) : Calcule le produit scalaire de la sortie de la couche precedente et de la matrice de poids, représentant la somme pondérée des entrées de chaque neurone dans la couche actuelle.

        4. actFunci : Applique la fonction d'activation correspondant à la couche actuelle à la somme pondérée, produisant la sortie de la couche actuelle.
        5. layer_output : Représente la sortie de la couche actuelle.

        6. layers_output.append(layer_output) : Ajoute la sortie de la couche à la liste layers_output.

3) La fonction retourne: layers_output: Une liste qui contient la sortie de chaque couche lors de la passe forward. Le dernier élément de cette liste est la sortie finale du réseau de neurones pour l'entrée donnée.

[104]:
```
def forward_propagation(dataIn, weightsByLayer, actFunc):
    layers_output = [dataIn]

    for i in range(1, len(weightsByLayer)+1):
        layer_input = layers_output[-1]
        weights = weightsByLayer[i-1]
        layer_output = actFunc[i](np.dot(layer_input, weights))
        layers_output.append(layer_output)

    return layers_output
```

Nous définissons la fonction de Backward Propagation:

1) La fonction prend comme paramètres:
    1. dataIn: Les données d'entrée pour le réseau de neurone concernant un exemple d'entraînement unique.
    2. dataOut: Les trues Labels de notre donnée d'apprentissage.
    3. weightsByLayer : Une liste contenant les matrices de poids pour chaque couche du réseau de neurones.
    4. actFunc : Une liste de fonctions d'activation correspondant à chaque couche.
    5. actFuncDeriv: Une liste des dérivées des fonctions d'activation de chauqe couche.
    La fonction de cout dans ce cas est fixée et correspond à l'erreur quadratique moyenne.
2) La fonction travaille de la manière suivante:
    1. num_layers : Représente le nombre total de couches dans le réseau de neurones.
    2. layers_output : Cette variable stocke la sortie de chaque couche lors de la passe forward. Elle est obtenue en appelant la fonction forward_propagation.
    3. error : Calcule l'erreur en prenant la dérivée de la fonction de perte par rapport à la sortie de la dernière couche.
    4. deltas : Une liste stockera les dérivées de l'erreur par rapport à la sortie de chaque couche. Le premier élément.
    5. La boucle for itère en sens inverse à travers les couches, commençant par la deuxième couche cachée(l'avant-dernière couche) et se déplaçant vers la couche d'entrée:
        1. À l'intérieur de la boucle, elle calcule l'erreur pour chaque couche en prenant le produit scalaire du delta de la couche précédente avec la transposée des poids connectant la couche actuelle à la couche suivante.
        2. L'erreur nouvellement calculée est ensuite ajoutée à la liste deltas après avoir été multipliée par la dérivée de la fonction d'activation pour la couche actuelle.
    6. Après la boucle, la liste deltas est inversée pour garantir qu'elle est ordonnée de la couche d'entrée à la couche de sortie.
3) La fonction retourne: deltas: Une liste dont chaque élément représente la dérivée de l'erreur par rapport à la sortie de la couche correspondante, pour la mise à jour des poids lors de l'étape d'optimisation de la descente de gradient.

[105]:
```
def backward_propagation(dataIn, dataOut, weightsByLayer, actFunc,␣
↪actFuncDeriv, lossDeriv=square_loss_deriv):
    num_layers = len(weightsByLayer)
    layers_output = forward_propagation(dataIn, weightsByLayer, actFunc)
    error = lossDeriv(layers_output[-1], dataOut)
    deltas = [error * actFuncDeriv[-1](layers_output[-1])]

    for i in range(num_layers - 2, -1, -1):
        error = deltas[-1].dot(weightsByLayer[i + 1].T)
        deltas.append(error * actFuncDeriv[i](layers_output[i + 1]))

    deltas.reverse()

    return deltas
```

Nous définissons la fonction de ise à jour des poids:

1) La fonction prend comme paramètres:
   1. weightsByLayer : Une liste contenant les matrices de poids pour chaque couche du réseau de neurones.
   2. deltas : Une liste contenant les gradients calculés pour chaque couche lors de la rétro-propagation(back propagation).
   3. layers_output: La liste des sorties de chaque couche lors de la passe forward.
   4. learning_rate: Le taux d'apprentissage, un hyperparamètre contrôlant la taille de pas des mises à jour de poids.
      learning_rate dans ce cas est fixée et correspond à 0,1.
2) La fonction travaille de la manière suivante:
   La fonction itère sur chaque couche avec poids du réseau de neurones.
   Pour chaque couche, elle récupère l'entrée de cette couche (layer_input) depuis layers_output.
   Ensuite, elle met à jour les poids de cette couche en utilisant la formule de la descente de gradient.

[106]:
```
def update_weights(weightsByLayer, deltas, layers_output, learning_rate):
    for i in range(len(weightsByLayer)):
        layer_input = layers_output[i]
        weightsByLayer[i] -= learning_rate * layer_input.T.dot(deltas[i])
```

Maintenant, nous sommes capable de commencer l'apprentissage du réseau de neurones Le reseau de neurone à lequel on applique l'apprentissage a les specifications suivantes: 1. Nombres de couches: le reseau de neurones est formé de 4 couches:
1) une couche d'entrée composée de 2 neurones. 2) 2 couches cachées avec 3 neurones et 4 neurones respectivement. 3) une couche de sortie composée de 2 neurones. 2. Fonction de perte(fonction de cout) :
la fonction de cout utilisée est l'erreur quadratique moyenne(MSE) 3. Fonctions d'activation: 1) pour la couche d'entrée, la fonction d'activation utilisée est la fonction sigmoid. 2) pour la 1ère couche cachée, la fonction d'activation utilisée est la fonction sigmoid. 3) pour la 2ème couche cachée, la fonction d'activation utilisée est la fonction tangente hyperbolique(tanh) 4) pour la

couche de sortie, la fonction d'activation utilisée est la fonction tangente hyperbolique(tanh) 4. Algorithm d'optimisation:
l'algorithm d'optimisation utilisé est la descente de gradient, avec un taux d'apprentissage constant = 0.1 (alpha = 0.1).
les poids initiaux (avant l'application de l'algorithm du descente de gradient) sont initialisés aléatoirement (np.normal.rand)

[111]:
```
def train_neural_network(dataIn, dataOut, weightsByLayer, actFunc,␣
 ↪actFuncDeriv, lossDeriv=square_loss_deriv, epochs=1, learning_rate=0.1):
    for _ in range(epochs):
        for i in range(len(dataIn)):
            input_data = dataIn[i:i+1]
            target_output = dataOut[i:i+1]

            deltas = backward_propagation(input_data, target_output,
                                          weightsByLayer, actFunc,␣
 ↪actFuncDeriv, lossDeriv)
            update_weights(weightsByLayer, deltas, forward_propagation(
                input_data, weightsByLayer, actFunc), learning_rate)

    return weightsByLayer


layerSizes = [2, 3, 4, 2]
actFuncByLayer = [identite, sigmoid, np.tanh, np.tanh]
DerivActFunc = [identiteDeriv, sigmoidDeriv, tanhDeriv, tanhDeriv]
Weights = createRandomWeightsByLayer(layerSizes)
dataInputs = np.random.rand(1, layerSizes[0])
dataOutputs = np.random.rand(1, layerSizes[-1])

times_back_propagation = []
for _ in range(1000):
    start_time = timeit.default_timer()
    BP = train_neural_network(dataIn=dataInputs, dataOut=dataOutputs,
                              weightsByLayer=Weights, actFunc=actFuncByLayer,␣
 ↪actFuncDeriv=DerivActFunc)
    elapsed = timeit.default_timer() - start_time
    times_back_propagation.append(elapsed)

print("Average training time: ", np.mean(times_back_propagation))
```

```
Average training time:  4.570959992997814e-05
```

NOTE:

Dans notre code: bien que le nombre d'epochs et le taux d'apprentissage sont fixés, le fonctionnement du code est identique indépendamment de ces valeurs.

[ ]: