Rapport de Projet 1

SY2024106 HUANG Jingyi

Description du problème

Ce projet implique la conception d'un réseau neuronal pour classer un ensemble de données composé de tableaux et de leurs étiquettes correspondantes. Les données sont divisées en ensembles d'entraînement et de test, et la mise à jour des poids se fait à l'aide de l'algorithme de SGD. Ensuite, j'ai examiné l'impact de facteurs tels que la taille des blocs, l'effet du changement de pas, et le choix des valeurs initiales sur les résultats, notamment la vitesse de convergence et le nombre d'itérations. Enfin, j'ai tenté de visualiser la fonction définie par le réseau neuronal.

Méthode SGD

J'ai implémenté la mise à jour des poids en utilisant la méthode de la descente de gradient stochastique (SGD) dans un fichier Python appelé `NN_SGD.py`, qui contient une fonction nommée `fit`.

```
# Boucle principale d'entraînement du modèle
while min(loss) > 10 and epoch < epochs:
    # Divise les données en mini-lots en fonction de la taille du lot
(batch_size)
    for i in range(0, data_arr, batch_size):
        x_train1 = data[i:i + batch_size]
        y_train1 = y[i:i + batch_size]

        # Calcul des prédictions du modèle pour les données d'entraînement
actuelles
        y_pred = self.call(x_train1)

        # Calcul de la perte (ou erreur) du modèle par rapport aux vraies
étiquettes (y_train1)
        delta = 1e-7
        loss[epoch] = -np.sum(y_train1 * np.log(y_pred + delta))

        # Rétropropagation (backpropagation) pour mettre à jour les poids du
modèle
        dz2 = self.s - y_train1
        dw2 = self.hiden_layer1.T.dot(dz2)
        da1 = dz2.dot(self.w2.T)
        dz1 = (1 - self.hiden_layer1) * self.hiden_layer1 * da1
        dw1 = x_train1.T.dot(dz1)

        if method == 'variable':
            # Mettre à jour le taux d'apprentissage (learning rate)
            learning_rate = initial_learning_rate * (1 - epoch*10 / epochs)

        # Mise à jour des poids du modèle en utilisant le gradient calculé
        self.w1 -= learning_rate * dw1
        self.w2 -= learning_rate * dw2

        epoch += 1
```

En résumé, cette partie du code effectue les étapes suivantes :

1. Il parcourt plusieurs époques d'entraînement tant que la perte (loss) minimale sur les mini-lots est supérieure à 10 et que le nombre d'époques est inférieur au nombre maximal d'époques (epochs).
2. Pour chaque époque, il divise les données d'entraînement en mini-lots de taille `batch_size`.
3. Il calcule les prédictions du modèle pour chaque mini-lot.
4. Il calcule la perte (ou erreur) du modèle en comparant les prédictions aux vraies étiquettes (y_train1).
5. Il effectue la rétropropagation (backpropagation) pour calculer les gradients des poids du modèle par rapport à la perte.
6. Si la méthode de mise à jour du taux d'apprentissage est définie sur 'variable', il met à jour le taux d'apprentissage en fonction du nombre d'époques.
7. Enfin, il met à jour les poids du modèle en utilisant les gradients calculés et le taux d'apprentissage.

L'objectif de cette boucle est d'ajuster progressivement les poids du modèle pour minimiser la perte et améliorer les performances du réseau de neurones au fil de l'entraînement.

Learning Rate

How to change learning rate

La recherche du meilleur taux d'apprentissage (learning rate) est cruciale pour l'efficacité de l'algorithme de descente de gradient stochastique (SGD). Un taux d'apprentissage trop petit peut rendre le processus d'apprentissage très lent, tandis qu'un taux d'apprentissage trop élevé peut rendre le processus instable.

Je peux utiliser la méthode de la recherche en grille, qui consiste à tester plusieurs valeurs de taux d'apprentissage prédéfinies dans une plage donnée. Par exemple, je peux essayer 1e-1, 1e-2, 1e-3, 1e-4, etc.

1e-1:

```
Epoch: 249
Accuracy: 0.8600699650174912
```

1e-2:

```
Epoch: 749
Accuracy: 0.8600699650174912
```

1e-3:

```
Epoch: 5249
Accuracy: 0.8600699650174912
```

Conclude: Un taux d'apprentissage de 1e-1 offre les meilleures performances

Fixed Learning Rate VS Variable Learning Rate

Fixed Learning Rate :

```
method = 'fixed'  # 'fixed' or 'variable'
```

Dans ma code, la méthode d'apprentissage avec taux d'apprentissage fixe est définie en utilisant la valeur `'fixed'` pour la variable `method`. Cela signifie que le taux d'apprentissage reste constant tout au long de l'entraînement.

La logique de la mise à jour du poids du modèle avec un taux d'apprentissage fixe se trouve dans la boucle d'entraînement principale (la boucle `while`). Peu importe la valeur de `method`, les poids du modèle (`self.w1` et `self.w2`) sont mis à jour à l'intérieur de la boucle à l'aide de la formule suivante :

```
self.w1 -= learning_rate * dw1
self.w2 -= learning_rate * dw2
```

Cela signifie que, quel que soit le nombre d'époques ou l'avancement de l'entraînement, le taux d'apprentissage reste le même, tel que spécifié par la variable `learning_rate`.

Variable Learning Rate :

```
method = 'variable'  # 'fixed' or 'variable'
```

Si je veux utiliser un taux d'apprentissage variable, je doit définir la variable `method` sur `'variable'`. Avec cette option, le taux d'apprentissage diminue progressivement à mesure que les époques d'entraînement avancent.

La logique pour mettre à jour le taux d'apprentissage variable se trouve également dans la boucle d'entraînement principale, spécifiquement dans cette partie du code :

```
if method == 'variable':
    # Mettre à jour le taux d'apprentissage
    learning_rate = initial_learning_rate * (1 - epoch*10 / epochs)
```

Ici, le taux d'apprentissage est mis à jour à chaque époque en fonction de la formule `learning_rate = initial_learning_rate * (1 - epoch / epochs)`. Cette formule réduit progressivement le taux d'apprentissage à mesure que le nombre d'époques augmente, ce qui peut aider à converger plus efficacement vers un minimum de perte.

En résumé, je peux choisir entre un taux d'apprentissage fixe (`'fixed'`) ou un taux d'apprentissage variable (`'variable'`) en modifiant la valeur de la variable `method`. Le code gère la mise à jour du taux d'apprentissage en conséquence, en fonction de l'option sélectionnée.

Sur la condition:

```
    weight_init = 'ones'  # 可以选择 'random'、'zeros'、'ones' 或其他支持的初始值策略
    activation = 'sigmoid'  # 可以选择 'relu'、'sigmoid'、'tanh' 或其他支持的激活函数
    mymodel = NN(weight_init=weight_init, activation=activation)
    epochs = 10000
    learning_rate = 1e-3
```

fixed:

```
Epoch: 5249
Accuracy: 0.8600699650174912
```

variable:

```
Epoch: 1749
Accuracy: 0.8600699650174912
```

Conclude: L'utilisation d'un taux d'apprentissage variable peut effectivement améliorer l'efficacité de la convergence d'un réseau neuronal.

Les difficultés rencontrées: Lors de l'écriture d'un taux d'apprentissage variable, une variation mineure du taux d'apprentissage n'a pas eu d'effet significatif sur les résultats expérimentaux. Ce n'est qu'après avoir modifié la formule de mise à jour du taux d'apprentissage que les effets sont devenus visibles.

Initial weights

Il existe trois principales méthodes d'initialisation des matrices de poids :

1. **Initialisation à zéro** : Dans cette méthode, tous les éléments des matrices de poids sont initialisés à zéro. Cela signifie que les poids initiaux sont tous identiques, ce qui conduit généralement à ce que le réseau ne parvienne pas à briser la symétrie, il n'est donc généralement pas recommandé de l'utiliser, sauf dans des cas spécifiques.
2. **Initialisation aléatoire** : Dans cette méthode, les éléments des matrices de poids sont initialisés à de petites valeurs aléatoires, souvent tirées d'une distribution uniforme ou gaussienne. Cela aide à briser la symétrie en introduisant de la variabilité, permettant ainsi aux différents neurones d'apprendre des caractéristiques différentes. Il s'agit d'une méthode courante d'initialisation, en particulier dans les réseaux de neurones profonds.
3. **Initialisation à un** : Dans cette méthode, tous les éléments des matrices de poids sont initialisés à un. Tout comme l'initialisation à zéro, cela conduit à ce que les poids initiaux soient tous identiques, ce qui n'introduit pas de variabilité et peut donc rendre le réseau incapable d'apprendre efficacement. Cette méthode est généralement moins utilisée en raison de son manque de variabilité.

En général, l'initialisation aléatoire est la méthode la plus couramment utilisée car elle introduit de la variabilité et permet au réseau de briser la symétrie, ce qui facilite l'apprentissage des caractéristiques des données. L'initialisation à zéro et l'initialisation à un sont moins courantes car elles ne sont généralement pas bénéfiques pour l'apprentissage.

Sur la condition:

```
    weight_init = 'ones'  # 可以选择 'random'、'zeros'、'ones' 或其他支持的初始值策略
    activation = 'sigmoid'  # 可以选择 'relu'、'sigmoid'、'tanh' 或其他支持的激活函数
    mymodel = NN(weight_init=weight_init, activation=activation)
    epochs = 10000
    learning_rate = 1e-3

    method = 'fixed'  # 'fixed' or 'variable'
    batch_size = 32

    loss = mymodel.fit(x_train, y_train, learning_rate, epochs, method=method,
batch_size=batch_size)
```

Les résultats sont les suivants :

ones:

```
Epoch: 5249
Accuracy: 0.8600699650174912
```

zeros:

```
Epoch: 249
Accuracy: 0.8600699650174912
```

random:

```
Epoch: 249
Accuracy: 0.8600699650174912
```

Conclude: Les résultats de l'expérience sont conformes à la théorie. L'initialisation aléatoire donne les meilleurs résultats.

Les difficultés rencontrées: Il n'est pas facile de distinguer la différence d'efficacité entre l'initialisation aléatoire et l'initialisation à un sur ce jeu de données.

Batch size

"La taille des blocs" (batch size) est un hyperparamètre important en apprentissage profond, utilisé pour contrôler la quantité d'échantillons utilisée à chaque itération lors de l'entraînement d'un modèle. Il a un impact significatif sur le processus d'entraînement et les performances du modèle, et voici quelques-uns de ses rôles :

1. **Contrôle de la demande en mémoire** : Une taille des blocs plus petite nécessite moins de mémoire, ce qui est essentiel pour l'entraînement sur des matériels ayant des ressources limitées. Les réseaux neuronaux profonds nécessitent une grande quantité de mémoire pour stocker les poids, les gradients et les valeurs d'activation, donc réduire la taille des blocs peut réduire la demande en mémoire.
2. **Accélération de l'entraînement** : Une taille des blocs plus grande a tendance à accélérer le processus d'entraînement, car davantage d'échantillons sont traités à chaque itération. Cela permet d'exploiter la parallélisation matérielle pour augmenter l'efficacité des calculs, en particulier sur les GPU.
3. **Amélioration de la stabilité des mises à jour de poids** : Une taille des blocs plus grande tend à entraîner des mises à jour de poids plus stables, car elle réduit l'impact du bruit des échantillons individuels. Cela peut contribuer à une convergence plus rapide et à une réduction des oscillations lors de l'entraînement.
4. **Impact sur la généralisation** : Une taille des blocs plus petite peut entraîner un surapprentissage plus rapide, car chaque mise à jour est basée sur moins d'exemples d'apprentissage. Une taille des blocs plus grande peut favoriser de meilleures performances en généralisation, car elle utilise plus d'informations d'échantillonnage. Cependant, le choix de la taille des blocs dépend également des méthodes de régularisation et de la taille du jeu de données.
5. **Impact sur les courbes d'apprentissage** : Le choix de la taille des blocs influence les courbes d'apprentissage, y compris la vitesse de convergence de la perte et sa stabilité. Une taille des blocs plus petite peut nécessiter davantage d'itérations pour converger, tandis qu'une taille des blocs plus grande peut permettre une convergence plus rapide.

Mon principal intérêt ici réside dans **l'impact sur les courbes d'apprentissage**.

Sur la condition:

```
    weight_init = 'random'  # 可以选择 'random'、'zeros'、'ones' 或其他支持的初始值策略
    activation = 'sigmoid'  # 可以选择 'relu'、'sigmoid'、'tanh' 或其他支持的激活函数
    mymodel = NN(weight_init=weight_init, activation=activation)
    epochs = 10000
    learning_rate = 1e-3

    method = 'fixed'  # 'fixed' or 'variable'
    batch_size = 32

    loss = mymodel.fit(x_train, y_train, learning_rate, epochs, method=method,
batch_size=batch_size)
```

Les résultats sont les suivants :

batch_size = 32:

```
Epoch: 249
Accuracy: 0.8600699650174912
```

batch_size = 64:

```
Epoch: 2249
Accuracy: 0.8750624687656172
```

batch_size = 128:

```
Epoch: 5669
Accuracy: 0.9480259870064968
```

Conclude: Plus la taille des blocs, moin la vitesse de convergence, et plus de accuracy.

Les difficultés rencontrées: Je ne comprends pas complètement le mécanisme derrière ce phénomène.

Activation function

Voici une comparaison et une analyse des fonctions d'activation courantes (ReLU, Sigmoid et Tanh) :

1. **ReLU (Rectified Linear Unit)** :
    - Formule : f(x) = max(0, x)
    - Avantages :
        - Très simple et rapide à calculer.
        - Résout le problème de la disparition du gradient, ce qui accélère l'entraînement.
        - Fonctionne généralement bien dans l'apprentissage en profondeur.
    - Inconvénients :
        - Risque de "neurones morts", où un neurone cesse d'apprendre si sa sortie est toujours négative.
        - Ne convient pas à tous les types de problèmes, en particulier ceux avec une plage de sortie entre 0 et 1.
2. **Sigmoid** :
    - Formule : f(x) = 1 / (1 + exp(-x))
    - Avantages :
        - Plage de sortie entre 0 et 1, adaptée aux problèmes de classification binaire.
        - Facilement interprétable.
    - Inconvénients :
        - Risque de disparition du gradient, en particulier dans les réseaux de neurones profonds.
        - Calcul relativement lent.
        - La sortie n'est pas centrée autour de zéro, ce qui peut entraîner des problèmes de décalage lors de l'entraînement.
3. **Tanh (Tangente hyperbolique)** :
    - Formule : f(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x))

- Avantages :
  - Plage de sortie entre -1 et 1, centrée autour de zéro.
  - Peut être meilleur que Sigmoid pour certains problèmes, notamment ceux qui requièrent une symétrie.
- Inconvénients :
  - Risque de disparition du gradient.
  - Calcul relativement lent.

Pour étudier spécifiquement le rôle des fonctions d'activation, j'ai mené des expériences dans le fichier `NN_no_batch.py` afin d'éliminer complètement l'influence des batch sur les expériences. Voici le code :

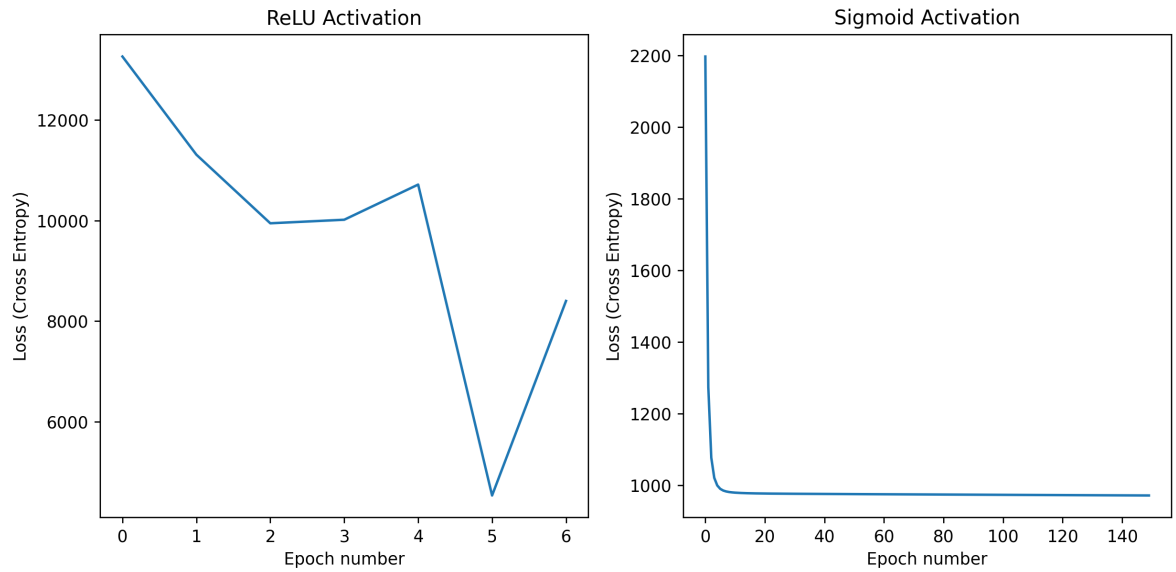
```
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def relu(self, x):
        return np.maximum(0, x)

    def activate(self, x):
        if self.activation == 'sigmoid':
            return self.sigmoid(x)
        elif self.activation == 'relu':
            return self.relu(x)
```

Après avoir exécuté le programme, les phénomènes suivants:

```
D:\1.学校\研究生毕业\研究生课程学习\3研二下\SO\SO_2\NN_no_batch.py:35: RuntimeWarning: overflow encountered in exp
  self.s = np.exp(self.output)
D:\1.学校\研究生毕业\研究生课程学习\3研二下\SO\SO_2\NN_no_batch.py:38: RuntimeWarning: invalid value encountered in divide
  self.s = np.divide(self.s, self.sm2)
ReLU Activation Accuracy: 0.052993375828021494
Sigmoid Activation Accuracy: 0.8586426696662918
```



Analyser: Les données d'entrée du jeu de données contiennent des valeurs négatives, et l'utilisation de ReLU en tant que fonction d'activation peut entraîner la suppression de ces valeurs négatives. Cela peut entraîner une perte de certaines caractéristiques dans les calculs du réseau neuronal, ce qui finit par réduire la précision des résultats. (Ce qui précède est une hypothèse, qui nécessite une étude approfondie pour être clarifiée.)

Revenons à `NN_SGD.py`, j'ai testé les performances de la fonction d'activation sigmoid et tanh, sur la condition:

```
    weight_init = 'random'  # 可以选择 'random'、'zeros'、'ones' 或其他支持的初始值策略
    activation = 'sigmoid'  # 可以选择 'relu'、'sigmoid'、'tanh' 或其他支持的激活函数
    mymodel = NN(weight_init=weight_init, activation=activation)
    epochs = 10000
    learning_rate = 1e-1

    method = 'fixed'  # 'fixed' or 'variable'
    batch_size = 32
```

Sigmoid:

```
Epoch: 249
Accuracy: 0.8765617191404298
```

Tanh:

```
Epoch: 9999
Accuracy: 0.09745127436281859
```

Conclude: Sigmoid est meilleure que Tanh ici.

fonctions gi(x)

`plot_y(x_test, out)` est une fonction qui crée un graphique de dispersion en 3D pour visualiser la sortie du réseau neuronal sur les points de données de test. Voici ce qu'elle fait :

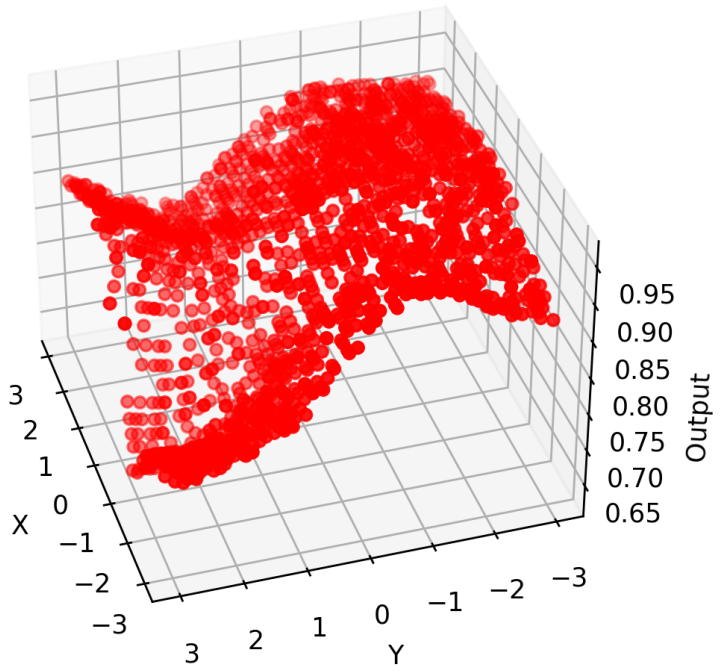
1. `x_test` contient les points de données de test (caractéristiques) que vous souhaitez visualiser.
2. `out` contient la sortie de votre réseau neuronal pour les points de données de test. Dans votre code, `out` est le résultat de l'appel à `mymodel.call(x_test)`, qui représente les prédictions du réseau pour les données de test.

La fonction `plot_y(x_test, out)` effectue les opérations suivantes :

- Elle extrait les valeurs de `x_test` pour les coordonnées X et Y.
- Elle extrait les trois valeurs de sortie de `out`, qui correspondent aux prédictions du réseau neuronal pour trois classes ou catégories différentes.

- Elle crée un graphique de dispersion en 3D où les coordonnées X et Y sont déterminées par `x_test`, et la coordonnée Z (axe vertical) est déterminée par la troisième valeur de sortie de `out`, qui représente la prédiction du réseau pour la troisième classe.

Essentiellement, ce graphique vous permet de visualiser comment les prédictions du réseau neuronal (axe Z) varient en fonction des différents points de données d'entrée (axes X et Y) pour la troisième classe. Cela peut vous aider à comprendre comment le réseau neuronal classe ou sépare efficacement les points de données en différentes catégories.



Méthode de Newton

Voici les codes dans `NN_Newton.py`:

```
    def fit(self, data, y, epochs=1, batch_size=32):
        loss = np.ones(epochs) * 100
        epoch = 0
        data_arr = data.shape[0]

        while min(loss) > 10 and epoch < epochs:
            # Split data into mini-batches based on batch_size
            for i in range(0, data_arr, batch_size):
                x_train1 = data[i:i + batch_size]
                y_train1 = y[i:i + batch_size]

                y_pred = self.call(x_train1)
                delta = 1e-7
                loss[epoch] = -np.sum(y_train1 * np.log(y_pred + delta))
```

```
            # Backpropagation
            dz2 = self.s - y_train1
            dw2 = self.hiden_layer1.T.dot(dz2)
            da1 = dz2.dot(self.w2.T)
            dz1 = (1 - self.hiden_layer1) * self.hiden_layer1 * da1
            dw1 = x_train1.T.dot(dz1)

            # 使用牛顿法更新权重
            hessian1 = np.dot(x_train1.T, x_train1)
            hessian_inv1 = np.linalg.pinv(hessian1)
            gradient1 = np.dot(x_train1.T, dz1)
            dw1 = np.dot(hessian_inv1, gradient1)

            hessian2 = np.dot(self.hiden_layer1.T, self.hiden_layer1)
            hessian_inv2 = np.linalg.pinv(hessian2)
            gradient2 = np.dot(self.hiden_layer1.T, dz2)
            dw2 = np.dot(hessian_inv2, gradient2)

            self.w1 -= dw1
            self.w2 -= dw2

            epoch += 1

        loss = loss[0:epoch]
        print("Epoch:", epoch - 1)
        return loss
```

Ce code utilise la méthode de Newton pour mettre à jour les poids d'un réseau neuronal. La méthode de Newton est une méthode itérative utilisée pour trouver les racines (ou les points d'extrémité) d'une fonction, et elle peut accélérer la convergence lors de la mise à jour des poids.

Dans un réseau neuronal, la mise à jour des poids est généralement effectuée à l'aide de la descente de gradient, mais dans ce code, la méthode de Newton est utilisée pour mettre à jour les poids. Voici les parties clés de la méthode de Newton dans le code :

1. Mise à jour des poids pour la première matrice de poids `self.w1` :
    - Calcul du gradient de la fonction de perte par rapport à la première matrice de poids, soit `dw1`.
    - Calcul de la matrice hessienne de la fonction de perte par rapport à la première matrice de poids, soit `hessian1`.
    - Calcul de l'inverse de la matrice hessienne, soit `hessian_inv1`.
    - Calcul du produit entre le gradient et l'inverse de la matrice hessienne pour obtenir la longueur du pas de mise à jour `dw1`.
    - Utilisation du pas de mise à jour `dw1` pour mettre à jour la première matrice de poids `self.w1`.
2. Mise à jour des poids pour la deuxième matrice de poids `self.w2`, qui suit un processus similaire à celui de la première matrice de poids :
    - Calcul du gradient de la fonction de perte par rapport à la deuxième matrice de poids, soit `dw2`.

- Calcul de la matrice hessienne de la fonction de perte par rapport à la deuxième matrice de poids, soit `hessian2`.
- Calcul de l'inverse de la matrice hessienne, soit `hessian_inv2`.
- Calcul du produit entre le gradient et l'inverse de la matrice hessienne pour obtenir la longueur du pas de mise à jour `dw2`.
- Utilisation du pas de mise à jour `dw2` pour mettre à jour la deuxième matrice de poids `self.w2`.

Question Classique

La variation des paramètres du réseau

J'ai créé un nouveau fichier appelé `NN_add_layer.py` dans lequel j'ai ajouté plusieurs couches de neurones à un réseau neuronal. Cela permet de le comparer avec le programme initial `NN_SGD.py`.

Ce réseau neuronal a six couches cachées :

1. La première couche cachée : elle comprend 2 caractéristiques d'entrée et 30 neurones.
2. La deuxième couche cachée : elle comprend 30 neurones.
3. La troisième couche cachée : elle comprend 30 neurones.
4. La quatrième couche cachée : elle comprend 30 neurones.
5. La cinquième couche cachée : elle comprend 30 neurones.
6. La sixième couche cachée : elle comprend 3 neurones, utilisés pour une tâche de classification.

Les codes:

```
def fit(self, data, y, learning_rate=1e-3, epochs=1, method='fixed',
batch_size=32):
    loss = np.ones(epochs) * 100
    epoch = 0
    initial_learning_rate = learning_rate
    data_arr = data.shape[0]

    while min(loss) > 10 and epoch < epochs:
        # Split data into mini-batches based on batch_size
        for i in range(0, data_arr, batch_size):
            x_train1 = data[i:i + batch_size]
            y_train1 = y[i:i + batch_size]

            y_pred = self.call(x_train1)
            delta = 1e-7
            loss[epoch] = -np.sum(y_train1 * np.log(y_pred + delta))

            # Backpropagation
            dz6 = self.s - y_train1
            dw6 = self.hidden_layer5.T.dot(dz6)
```

```
            da5 = dz6.dot(self.w6.T)
            dz5 = (1 - self.hidden_layer5) * self.hidden_layer5 * da5
            dw5 = self.hidden_layer4.T.dot(dz5)
            da4 = dz5.dot(self.w5.T)
            dz4 = (1 - self.hidden_layer4) * self.hidden_layer4 * da4
            dw4 = self.hidden_layer3.T.dot(dz4)
            da3 = dz4.dot(self.w4.T)
            dz3 = (1 - self.hidden_layer3) * self.hidden_layer3 * da3
            dw3 = self.hidden_layer2.T.dot(dz3)
            da2 = dz3.dot(self.w3.T)
            dz2 = (1 - self.hidden_layer2) * self.hidden_layer2 * da2
            dw2 = self.hidden_layer1.T.dot(dz2)
            da1 = dz2.dot(self.w2.T)
            dz1 = (1 - self.hidden_layer1) * self.hidden_layer1 * da1
            dw1 = x_train1.T.dot(dz1)

            if method == 'variable':
                # Update learning rate
                learning_rate = initial_learning_rate * (1 - epoch / epochs)

            self.w1 -= learning_rate * dw1
            self.w2 -= learning_rate * dw2
            self.w3 -= learning_rate * dw3
            self.w4 -= learning_rate * dw4
            self.w5 -= learning_rate * dw5
            self.w6 -= learning_rate * dw6

            epoch += 1

    loss = loss[0:epoch]
    print("Epoch:", epoch - 1)
    return loss
```

Sur la condition:

```
weight_init = 'random'  # 可以选择 'random'、'zeros'、'ones' 或其他支持的初始值策略
activation = 'sigmoid'  # 可以选择 'relu'、'sigmoid'、'tanh' 或其他支持的激活函数
mymodel = NN(weight_init=weight_init, activation=activation)
epochs = 10000
learning_rate = 1e-4

method = 'fixed'  # 'fixed' or 'variable'
batch_size = 32
```

Si on ne change pas les neurones:

NN_SGD.py:

```
Epoch: 499
Accuracy: 0.8600699650174912
```

NN_add_layer.py:

```
Epoch: 249
Accuracy: 0.8600699650174912
```

Si on change les neurones a 5:

NN_SGD.py:

```
Epoch: 3999
Accuracy: 0.8600699650174912
```

NN_add_layer.py:

```
Epoch: 1249
Accuracy: 0.8600699650174912
```

Conclude: L'augmentation du nombre de couches dans le réseau neuronal a clairement eu un impact positif dans le cadre de ce projet, ce qui suggère qu'ajouter des couches peut améliorer efficacement la capacité du réseau à traiter des tâches complexes. Cependant, il est important de noter que cette constatation ne signifie pas que l'ajout de couches est toujours une décision appropriée, car une augmentation excessive de la profondeur du réseau peut entraîner des problèmes de surapprentissage et ralentir la vitesse de convergence de l'entraînement.

En revanche, la réduction du nombre de neurones dans le réseau a eu un impact négatif dans le cadre de ce projet, ce qui indique que la diminution de la largeur du réseau peut rendre plus difficile la capture de modèles complexes, ce qui entraîne une baisse des performances. Cependant, il est important de noter que l'augmentation du nombre de neurones n'est pas toujours la solution, car un excès de neurones peut entraîner une augmentation des coûts de calcul et nécessiter un ensemble de données plus important pour éviter le surapprentissage.

Par conséquent, dans la pratique, il est essentiel de choisir judicieusement la profondeur et la largeur du réseau en fonction des exigences spécifiques de la tâche. Il s'agit d'un problème crucial d'ajustement des hyperparamètres qui nécessite une exploration et une optimisation approfondies dans le domaine de l'apprentissage automatique.