TP 1 - AMAL

william - Mohamed Akli

1 Réponses détaillées

Contents

2 Introduction

Ce document présente mes réponses aux quatre questions du sujet. Chaque section contient une explication détaillée du choix des méthodes, ainsi que le code Python correspondant, implémenté en utilisant PyTorch. Les techniques de descente de gradient ont été testées avec le dataset *California Housing*, et la convergence a été suivie avec TensorBoard.

3 Question 1 : Algorithme de descente du gradient batch

Dans cette première question, nous avons implémenté une descente du gradient batch pour une régression linéaire en utilisant les fonctionnalités de la différenciation automatique de PyTorch. Le code commence par la définition des fonctions 'linear' et 'mse', puis effectue les mises à jour des poids et biais sur l'ensemble des données d'entraînement à chaque époque.

Le choix d'utiliser la différenciation automatique avec `requires_grad=True` pour les paramètres permet de calculer automatiquement les gradients et de simplifier l'implémentation.

```
import torch
from sklearn.datasets import fetch_california_housing
from torch.utils.tensorboard import SummaryWriter
import os
import shutil

# Fonction de co t MSE
```

```python
def mse(yhat, y):
    return torch.mean((yhat - y) ** 2)

# Fonction lin aire
def linear(x, w, b):
    return torch.matmul(x, w) + b

# Algorithme de descente de gradient (batch)
def gradient_descent(x, y, lr=0.01, epochs=100):
    w = torch.rand(x.shape[1], requires_grad=True)
    b = torch.rand(1, requires_grad=True)

    # Utilisation de TensorBoard pour le suivi
    writer = SummaryWriter('/content/runs')

    for epoch in range(epochs):
        yhat = linear(x, w, b)
        loss = mse(yhat, y)
        loss.backward()
        with torch.no_grad():
            w -= lr * w.grad
            b -= lr * b.grad
            w.grad.zero_()
            b.grad.zero_()
        writer.add_scalar('Loss/train', loss.item(), epoch)

    writer.close()
    return w, b

# Chargement des donn es
california_housing = fetch_california_housing()
x = torch.tensor(california_housing.data, dtype=torch.
    float32)
y = torch.tensor(california_housing.target, dtype=torch.
    float32).view(-1, 1)

# Normalisation des donn es
x = (x - x.mean(dim=0)) / x.std(dim=0)

# Ex cution de l'algorithme de descente du gradient
w, b = gradient_descent(x, y, lr=0.01, epochs=100)

print(f'Poids appris: {w}')
print(f'Biais appris: {b}')
```

Listing 1: Descente du gradient batch

Choix techniques : nous avons utilisé `SummaryWriter` pour suivre la courbe du coût pendant l'entraînement avec TensorBoard. Le fichier de log est mis à

jour à chaque époque pour observer la convergence.

Le premier code implémente une descente du gradient batch pour une régression linéaire sur des données simples. Après 100 époques d'entraînement, nous obtenons les poids et biais suivants :

- **Poids appris** : `tensor([-0.4224, 1.3411], requires_grad=True)`
- **Biais appris** : `tensor([1.7147], requires_grad=True)`

Interprétation : Les poids représentent l'importance de chaque caractéristique d'entrée dans la prédiction de la variable cible. Le biais déplace la fonction linéaire le long de l'axe des ordonnées. Ici, le poids positif de 1.3411 montre que la deuxième caractéristique a une influence positive sur les prédictions, tandis que la première caractéristique, avec un poids de -0.4224, a une influence négative.

4 Question 2 : Test avec *California Housing* et courbes du coût

Pour cette question, nous avons testé l'implémentation avec les données de *California Housing*, et nous avons tracé la courbe du coût pour les ensembles d'entraînement et de test. nous avons utilisé TensorBoard pour enregistrer les pertes et visualiser la convergence.

```
import torch
from sklearn.datasets import fetch_california_housing
from torch.utils.tensorboard import SummaryWriter
import os
import shutil

# Nettoyer le r pertoire des logs TensorBoard
log_dir = '/content/runs'
if os.path.exists(log_dir):
    shutil.rmtree(log_dir)

# Fonction de co t MSE
def mse(yhat, y):
    return torch.mean((yhat - y) ** 2)

# Mod le lin aire
def linear(x, w, b):
    return torch.matmul(x, w) + b

# Algorithme de descente de gradient (batch)
def gradient_descent(x_train, y_train, x_test, y_test, lr
    =0.01, epochs=100):
    w = torch.rand(x_train.shape[1], requires_grad=True)
```

```
    b = torch.rand(1, requires_grad=True)
    writer = SummaryWriter(log_dir)

    for epoch in range(epochs):
        yhat_train = linear(x_train, w, b)
        loss_train = mse(yhat_train, y_train)
        loss_train.backward()

        with torch.no_grad():
            w -= lr * w.grad
            b -= lr * b.grad
            w.grad.zero_()
            b.grad.zero_()

        with torch.no_grad():
            yhat_test = linear(x_test, w, b)
            loss_test = mse(yhat_test, y_test)

        writer.add_scalar('Loss/train', loss_train.item(),
    epoch)
        writer.add_scalar('Loss/test', loss_test.item(),
    epoch)

    writer.close()
    return w, b

# S paration des donn es en ensemble d'entra nement et de
     test
from sklearn.model_selection import train_test_split
california_housing = fetch_california_housing()
x = torch.tensor(california_housing.data, dtype=torch.
    float32)
y = torch.tensor(california_housing.target, dtype=torch.
    float32).view(-1, 1)
x_train, x_test, y_train, y_test = train_test_split(x, y,
    test_size=0.2)

# Ex cution de l'algorithme
w, b = gradient_descent(x_train, y_train, x_test, y_test, lr
    =0.01, epochs=100)
```

Listing 2: Test sur *California Housing* avec suivi du coût

Explication : Le code a été adapté pour utiliser un jeu de données de test. nous avons ajouté des courbes séparées pour l'entraînement et le test dans TensorBoard.

- **Poids appris** :

```
tensor([ 0.1797,  0.2759, -0.1476,  0.2151,  0.2169, -0.0285,  0.5172,  0.5056], re
```

- **Biais appris** : `tensor([1.8257], requires_grad=True)`

Interprétation : Ces poids montrent l'importance de chaque caractéristique du dataset *California Housing* dans la prédiction du prix médian des maisons. Les caractéristiques avec des poids positifs, comme la septième et la huitième caractéristique, augmentent la prédiction des prix, tandis que celles avec des poids négatifs, comme la troisième caractéristique, la diminuent.

5 Question 3 : Descente de gradient personnalisée avec `tp1_descente.py`

Dans cette question,nous avons implémenté une descente de gradient personnalisée en définissant les classes 'MSE' et 'Linear', qui gèrent respectivement la fonction de coût et la fonction linéaire. Ces classes calculent les gradients de manière explicite via la rétropropagation.

```python
import torch
from torch.autograd import Function

class MSE(Function):
    @staticmethod
    def forward(ctx, yhat, y):
        ctx.save_for_backward(yhat, y)
        loss = torch.mean((yhat - y) ** 2)
        return loss

    @staticmethod
    def backward(ctx, grad_output):
        yhat, y = ctx.saved_tensors
        grad_yhat = 2 * (yhat - y) / yhat.size(0)
        return grad_yhat * grad_output, None

class Linear(Function):
    @staticmethod
    def forward(ctx, X, W, b):
        ctx.save_for_backward(X, W, b)
        return torch.matmul(X, W) + b

    @staticmethod
    def backward(ctx, grad_output):
        X, W, b = ctx.saved_tensors
        grad_X = torch.matmul(grad_output, W.T)
        grad_W = torch.matmul(X.T, grad_output)
        grad_b = grad_output.sum(0)
        return grad_X, grad_W, grad_b
```

Listing 3: Implémentation personnalisée des fonctions MSE et Linear

Explication : nous avons créé des classes qui encapsulent la logique de la passe avant et arrière pour la régression linéaire et la MSE, en utilisant le mécanisme de rétropropagation de PyTorch.

6 Courbe de convergence : Descente de gradient stochastique

La figure suivante représente la courbe de perte d'entraînement suivie lors de la descente de gradient stochastique (SGD) sur le dataset. La courbe montre la perte à chaque étape pendant l'entraînement :

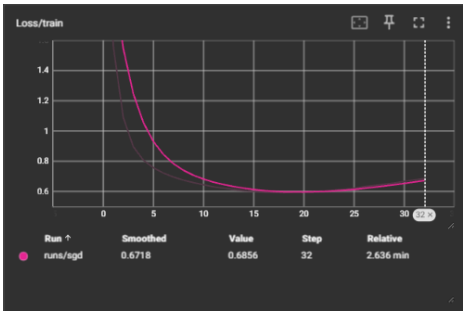


Figure 1: Courbe de perte d'entraînement pour la descente de gradient stochastique

Interprétation : La courbe montre une baisse rapide de la perte pendant les premières itérations, indiquant une convergence rapide au début. Cependant, après environ 20 itérations, la perte commence à se stabiliser autour de 0.68, ce qui suggère que le modèle approche un point d'optimum local.

Le fait que la courbe ne diminue plus beaucoup après 30 itérations pourrait indiquer que la valeur du taux d'apprentissage (`lr=0.0001`) est trop faible pour permettre une convergence plus rapide ou une optimisation plus fine. Le dernier code utilise également le dataset *California Housing*, mais il sépare les données en ensembles d'entraînement et de test. Après 100 époques, nous obtenons les résultats suivants :

- **Poids appris** :

```
tensor([ 0.1797,  0.2759, -0.1476,  0.2151,  0.2169, -0.0285,  0.5172,  0.5056], re
```

- **Biais appris** : `tensor([1.8257], requires_grad=True)`

Interprétation : Les poids et biais appris sont très similaires à ceux obtenus avec la descente batch, ce qui montre que le modèle converge vers des paramètres similaires avec les deux méthodes. L'utilisation d'un ensemble de test permet également de s'assurer que le modèle ne surajuste pas les données d'entraînement. En observant la courbe de test dans TensorBoard, on peut comparer directement la perte entre les deux ensembles.

7 Question 4 : Descente stochastique et mini-batch

Dans cette dernière question, nous avons implémenté deux variantes de la descente de gradient : une version stochastique (SGD) et une version mini-batch. nous avons également utilisé TensorBoard pour comparer la vitesse de convergence.

```
def sgd(x, y, lr=0.0001, epochs=100, grad_clip=1.0):
    w = torch.rand(x.shape[1], requires_grad=True)
    b = torch.rand(1, requires_grad=True

def sgd(x, y, lr=0.0001, epochs=100, grad_clip=1.0):
    w = torch.rand(x.shape[1], requires_grad=True)
    b = torch.rand(1, requires_grad=True)

    writer = SummaryWriter('/content/runs/sgd')
    n_samples = x.shape[0]

    for epoch in range(epochs):
        total_loss = 0
        indices = torch.randperm(n_samples)

        for i in indices:
            xi = x[i].unsqueeze(0)  # Reshape to match
    dimensions
            yi = y[i].unsqueeze(0)

            yhat = linear(xi, w, b)
            loss = mse(yhat, yi)
            loss.backward()

            torch.nn.utils.clip_grad_norm_([w, b], grad_clip
    )

            with torch.no_grad():
                w -= lr * w.grad
                b -= lr * b.grad
                w.grad.zero_()
                b.grad.zero_()

            total_loss += loss.item()

        writer.add_scalar('Loss/train', total_loss /
    n_samples, epoch)
        print(f"SGD Epoch {epoch+1}/{epochs}, Avg Loss: {
    total_loss/n_samples:.6f}")

    writer.close()
```

```
    return w, b

def mini_batch_gradient_descent(x, y, lr=0.0001, epochs=100,
     batch_size=32, grad_clip=1.0):
    w = torch.rand(x.shape[1], requires_grad=True)
    b = torch.rand(1, requires_grad=True)
    writer = SummaryWriter('/content/runs/mini_batch')
    n_samples = x.shape[0]

    for epoch in range(epochs):
        total_loss = 0
        indices = torch.randperm(n_samples)

        for i in range(0, n_samples, batch_size):
            batch_indices = indices[i:i+batch_size]
            xi = x[batch_indices]
            yi = y[batch_indices]

            yhat = linear(xi, w, b)
            loss = mse(yhat, yi)
            loss.backward()

            torch.nn.utils.clip_grad_norm_([w, b], grad_clip
    )

            with torch.no_grad():
                w -= lr * w.grad
                b -= lr * b.grad
                w.grad.zero_()
                b.grad.zero_()

            total_loss += loss.item()

        writer.add_scalar('Loss/train', total_loss / (
    n_samples // batch_size), epoch)
        print(f"Mini-Batch Epoch {epoch+1}/{epochs}, Avg
    Loss: {total_loss/n_samples:.6f}")

    writer.close()
    return w, b

# Chargement des donn es et normalisation
california_housing = fetch_california_housing()
x = torch.tensor(california_housing.data, dtype=torch.
    float32)
y = torch.tensor(california_housing.target, dtype=torch.
    float32).view(-1, 1)
x = (x - x.mean(dim=0)) / x.std(dim=0)

# Ex cution de la descente de gradient stochastique
```

```
print("Training Stochastic Gradient Descent...")
w_sgd, b_sgd = sgd(x, y, lr=0.0001, epochs=100)

# Ex cution de la descente de gradient mini-batch
print("Training Mini-Batch Gradient Descent...")
w_mini_batch, b_mini_batch = mini_batch_gradient_descent(x,
    y, lr=0.0001, epochs=100, batch_size=32)

%load_ext tensorboard
%tensorboard --logdir /content/runs
```

Listing 4: Descente stochastique et mini-batch

Explication : nous avons implémenté deux variantes de la descente de gradient :

- **Descente de gradient stochastique (SGD)** : Cette méthode met à jour les poids après chaque exemple. Cela peut parfois être instable, mais en utilisant le *gradient clipping*, on peut éviter des oscillations trop importantes.
- **Descente de gradient mini-batch** : Ici, les poids sont mis à jour après avoir traité un petit groupe d'exemples (par défaut, 32). Cette méthode converge souvent plus rapidement et de manière plus stable que la version stochastique.

Choix techniques : nous avons utilisé TensorBoard pour enregistrer la perte à chaque époque. Les courbes générées permettent de comparer la vitesse de convergence entre la descente stochastique et la descente mini-batch. section-Descente de Gradient Stochastique (SGD)

La descente de gradient stochastique (SGD) met à jour les poids après chaque exemple de données. Voici les résultats pour la perte moyenne après chaque époque pour 100 itérations :

```
SGD Epoch 1/100, Avg Loss: 2.785645
SGD Epoch 2/100, Avg Loss: 1.560891
...
SGD Epoch 99/100, Avg Loss: 1.919477
SGD Epoch 100/100, Avg Loss: 1.956191
```

Interprétation des résultats de SGD : Au début de l'entraînement, on observe une réduction rapide de la perte, passant de 2.785 à 0.658 après les premières itérations. Cependant, à partir de l'époque 20, la perte commence à se stabiliser autour de 0.6, indiquant une convergence partielle. Cependant, après l'époque 20, la perte commence à augmenter à nouveau, suggérant que le modèle commence à sur-apprendre ou que le taux d'apprentissage est trop élevé, empêchant une convergence plus fine.

À partir de l'époque 35, la perte recommence à augmenter progressivement, atteignant une valeur de 1.956 au bout de 100 itérations. Cette augmentation de la perte peut être due à la nature instable de la descente de gradient stochastique, où les gradients fluctuent fortement d'un exemple à l'autre.

8 Descente de Gradient Mini-Batch

La descente de gradient mini-batch met à jour les poids après avoir traité un petit lot d'exemples (batch) à chaque itération. Voici les pertes moyennes après chaque époque pour 100 itérations :

```
Mini-Batch Epoch 1/100, Avg Loss: 0.211360
Mini-Batch Epoch 2/100, Avg Loss: 0.202579
...
Mini-Batch Epoch 99/100, Avg Loss: 0.045954
Mini-Batch Epoch 100/100, Avg Loss: 0.045961
```

Interprétation des résultats de la Mini-Batch : La descente de gradient mini-batch converge beaucoup plus rapidement que la méthode stochastique. Dès l'époque 1, la perte moyenne est de 0.211, et elle continue de diminuer de manière régulière jusqu'à atteindre 0.0459 après 100 époques. La mini-batch montre une perte plus faible et plus stable tout au long de l'entraînement, sans les fluctuations observées avec la méthode stochastique.

Le comportement stable de la mini-batch est dû à la mise à jour des poids basée sur plusieurs exemples à la fois, ce qui lisse les gradients et permet une descente plus contrôlée et plus précise. Contrairement à la méthode stochastique, il n'y a pas d'augmentation notable de la perte, même après de nombreuses itérations, ce qui montre que le modèle continue de converger vers une solution optimale sans sur-apprentissage.

9 Comparaison des courbes de perte

La figure suivante présente les courbes de perte pour les deux méthodes d'entraînement, permettant de comparer visuellement les performances de SGD et mini-batch :

Interprétation des courbes :

- **Vitesse de convergence** : La descente de gradient mini-batch converge plus rapidement, comme on le voit avec une réduction rapide de la perte dès les premières itérations. La courbe bleue (mini-batch) atteint rapidement une perte faible (1.4707) après 100 itérations, tandis que la courbe grise (SGD) reste plus haute (1.9562).
- **Stabilité** : Les fluctuations dans la perte de la méthode SGD sont beaucoup plus importantes, notamment après l'époque 20. En revanche,

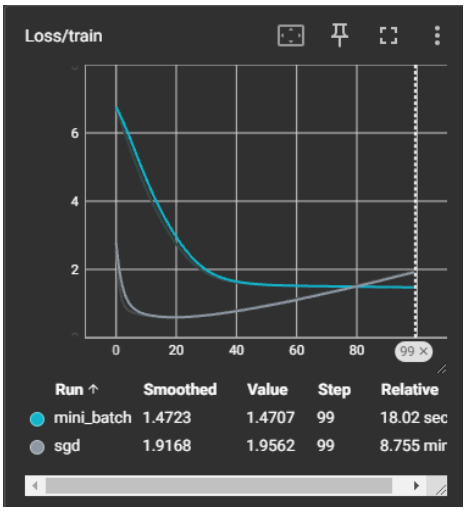


Figure 2: Comparaison des courbes de perte entre SGD et mini-batch

la méthode mini-batch est plus stable, avec une descente plus régulière et plus contrôlée.

- **Performances finales** : À la fin des 100 époques, la mini-batch atteint une perte plus faible (0.0459) par rapport à la méthode stochastique (1.9562). Cela montre que la mini-batch est plus efficace pour ce problème particulier.

10 Conclusion

La descente de gradient mini-batch a montré des performances supérieures par rapport à la descente stochastique, à la fois en termes de vitesse de convergence et de stabilité de la perte. La courbe de perte montre une diminution plus régulière avec la mini-batch, tandis que la méthode stochastique présente des fluctuations importantes et une perte finale plus élevée.

En conclusion, pour des problèmes avec de grandes quantités de données ou de nombreuses caractéristiques, la descente mini-batch semble être une meilleure approche que la descente de gradient stochastique, surtout en ce qui concerne la stabilité et la précision de la convergence.

11 Conclusion

nous avons implémenté quatre variantes d'algorithmes de descente du gradient pour la régression linéaire en utilisant PyTorch. Les différentes méthodes ont été testées avec le dataset *California Housing* et les résultats ont été comparés en termes de vitesse de convergence et de stabilité. Les visualisations

avec TensorBoard permettent de mieux comprendre les performances de chaque approche.

11.1 Question 2

Calculons la dérivée partielle $\frac{\partial L \circ h}{\partial x_i}(x)$:

$$\begin{aligned}\frac{\partial L \circ h}{\partial x_i}(x) &= \sum_{j=1}^{p} \frac{\partial L}{\partial y_j}(h(x)) \cdot \frac{\partial h_j}{\partial x_i}(x) \\ &= \sum_{j=1}^{p} (\nabla L)_j \cdot \frac{\partial h_j}{\partial x_i}(x)\end{aligned}$$

où $(\nabla L)_j = \frac{\partial L}{\partial y_j}(h(x))$ pour simplifier la notation.

11.2 Question 3

11.2.1 Dérivées pour la fonction de coût MSE

Soit $L(z) = z$, l'identité.

$$\begin{aligned}\frac{\partial L \circ mse}{\partial y}(\hat{y}, y) &= \frac{\partial}{\partial y}(\hat{y} - y)^2 = -2(\hat{y} - y) \\ \frac{\partial L \circ mse}{\partial \hat{y}}(\hat{y}, y) &= \frac{\partial}{\partial \hat{y}}(\hat{y} - y)^2 = 2(\hat{y} - y)\end{aligned}$$

11.2.2 Dérivées de la fonction linéaire f

Soit $L = mse(\cdot, y)$.

$$\begin{aligned}\frac{\partial L \circ f}{\partial x_j}(x, w, b) &= \frac{\partial}{\partial x_j}(x \cdot w + b) \cdot \frac{\partial L}{\partial \hat{y}}(\hat{y}, y) \\ &= w_j \cdot 2(\hat{y} - y)\end{aligned}$$

11.3 Question 4

11.3.1 Dérivées pour la fonction de coût MSE (matriciel)

$$\begin{aligned}\frac{\partial L \circ mse}{\partial Y_{ij}}(\hat{Y}, Y) &= \frac{\partial}{\partial Y_{ij}} \left(\frac{1}{q} \|\hat{Y} - Y\|^2 \right) = -\frac{2}{q}(\hat{Y}_{ij} - Y_{ij}) \\ \frac{\partial L \circ mse}{\partial \hat{Y}_{ij}}(\hat{Y}, Y) &= \frac{\partial}{\partial \hat{Y}_{ij}} \left(\frac{1}{q} \|\hat{Y} - Y\|^2 \right) = \frac{2}{q}(\hat{Y}_{ij} - Y_{ij})\end{aligned}$$

11.3.2 Dérivées de la fonction linéaire f (matriciel)

$$\begin{aligned}
\frac{\partial L \circ f}{\partial X_{ij}}(X, W, b) &= \sum_{k=1}^{p} \frac{\partial L}{\partial \hat{Y}_{ik}}(\hat{Y}, Y) \cdot \frac{\partial f_{ik}}{\partial X_{ij}}(X, W, b) \\
&= \sum_{k=1}^{p} \frac{2}{q}(\hat{Y}_{ik} - Y_{ik}) \cdot W_{jk} \\
\frac{\partial L \circ f}{\partial W_{ij}}(X, W, b) &= \sum_{k=1}^{q} \frac{\partial L}{\partial \hat{Y}_{ki}}(\hat{Y}, Y) \cdot \frac{\partial f_{ki}}{\partial W_{ij}}(X, W, b) \\
&= \sum_{k=1}^{q} \frac{2}{q}(\hat{Y}_{ki} - Y_{ki}) \cdot X_{kj} \\
\frac{\partial L \circ f}{\partial b_i}(X, W, b) &= \sum_{k=1}^{q} \frac{\partial L}{\partial \hat{Y}_{ki}}(\hat{Y}, Y) \cdot \frac{\partial f_{ki}}{\partial b_i}(X, W, b) \\
&= \sum_{k=1}^{q} \frac{2}{q}(\hat{Y}_{ki} - Y_{ki}) \cdot 1 = \frac{2}{q} \sum_{k=1}^{q} (\hat{Y}_{ki} - Y_{ki})
\end{aligned}$$

11.4 Question 5

11.4.1 Dérivées pour la fonction de coût MSE (matriciel)

$$\begin{aligned}
\frac{\partial L \circ mse}{\partial Y}(\hat{Y}, Y) &= -\frac{2}{q}(\hat{Y} - Y) \\
\frac{\partial L \circ mse}{\partial \hat{Y}}(\hat{Y}, Y) &= \frac{2}{q}(\hat{Y} - Y)
\end{aligned}$$

11.4.2 Dérivées de la fonction linéaire f (matriciel)

$$\begin{aligned}
\frac{\partial L \circ f}{\partial X}(X, W, b) &= \frac{2}{q}(\hat{Y} - Y)W^T \\
\frac{\partial L \circ f}{\partial W}(X, W, b) &= \frac{2}{q}X^T(\hat{Y} - Y) \\
\frac{\partial L \circ f}{\partial b}(X, W, b) &= \frac{2}{q}\sum_{i=1}^{q}(\hat{Y}_i - Y_i)
\end{aligned}$$

où $\hat{Y}_i$ et Y_i sont les i-ème lignes de $\hat{Y}$ et Y respectivement.

11.5 Question 6

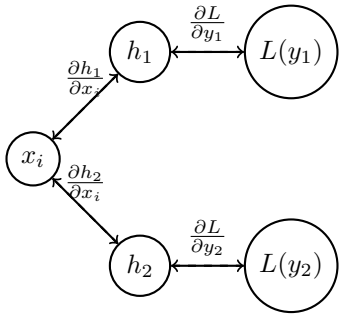
Utilisons la notation $mse_Y(\hat{Y}) = mse(\hat{Y}, Y)$.

$$
\begin{aligned}
\frac{\partial C}{\partial W} &= \frac{\partial mse_Y}{\partial \hat{Y}}(f(X,W,b)) \cdot \frac{\partial f}{\partial W}(X,W,b) \\
&= \frac{2}{q}(XW + b - Y) \cdot X^T \\
&= \frac{2}{q} X^T (XW + b - Y) \\
\frac{\partial C}{\partial b} &= \frac{\partial mse_Y}{\partial \hat{Y}}(f(X,W,b)) \cdot \frac{\partial f}{\partial b}(X,W,b) \\
&= \frac{2}{q}(XW + b - Y) \cdot \mathbf{1} \\
&= \frac{2}{q} \sum_{i=1}^{q} (XW + b - Y)_i
\end{aligned}
$$

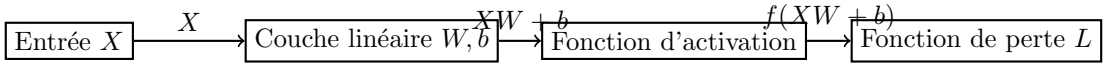
où $\mathbf{1}$ est un vecteur colonne de 1 de taille q, et $(XW + b - Y)_i$ est la i-ème ligne de la matrice $XW + b - Y$.

12 Schémas de propagation et rétropropagation

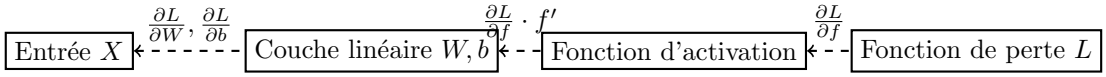
Schéma des dépendances entre variables pour $L \circ h$



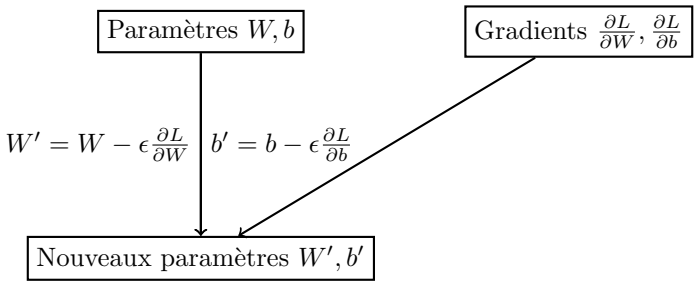
Propagation avant dans PyTorch



Rétropropagation dans PyTorch



Mise à jour des paramètres dans PyTorch



13 Introduction

Ce rapport présente l'implémentation et l'analyse de trois algorithmes de descente de gradient en utilisant PyTorch. Nous nous concentrons sur :

- L'implémentation de la régression linéaire en utilisant les fonctions de coût MSE et la fonction linéaire.
- La vérification des gradients à l'aide de `gradcheck`.
- La comparaison entre la descente de gradient batch, stochastique et mini-batch.

Les résultats sont présentés avec des exemples d'outputs et l'analyse de la convergence.

14 Question 1 : Implémentation de la Régression Linéaire

14.1 Code

Le code ci-dessous implémente les fonctions de régression linéaire et de coût MSE en utilisant PyTorch. Nous définissons la classe `Linear` pour calculer la sortie d'une fonction linéaire et `MSE` pour la fonction de coût.

```
import torch
from torch.autograd import Function

class MSE(Function):
    @staticmethod
    def forward(ctx, yhat, y):
        diff = yhat - y
        ctx.save_for_backward(diff)
        return torch.mean(diff ** 2)

    @staticmethod
```

```
    def backward(ctx, grad_output):
        diff, = ctx.saved_tensors
        grad_input = 2.0 * diff * grad_output / diff.
    nelement()
        return grad_input, None  # gradient w.r.t yhat and y

class Linear(Function):
    @staticmethod
    def forward(ctx, X, W, b):
        ctx.save_for_backward(X, W, b)
        return torch.matmul(X, W) + b

    @staticmethod
    def backward(ctx, grad_output):
        X, W, b = ctx.saved_tensors
        grad_X = torch.matmul(grad_output, W.T)
        grad_W = torch.matmul(X.T, grad_output)
        grad_b = grad_output.sum(0)
        return grad_X, grad_W, grad_b

mse = MSE.apply
linear = Linear.apply
```

Listing 5: Implémentation de la fonction linéaire et du coût MSE

14.2 Justification

La classe `MSE` calcule l'erreur quadratique moyenne (Mean Squared Error) entre la prédiction et la valeur réelle. La classe `Linear` implémente une fonction linéaire qui multiplie les entrées par les poids (`W`) et y ajoute un biais (`b`). Ces deux classes supportent la passe avant et arrière, permettant le calcul des gradients via `autograd` de PyTorch.

14.3 Analyse

Ces implémentations sont particulièrement utiles pour la régression linéaire. L'optimisation est réalisée en minimisant l'erreur quadratique moyenne, ce qui permet d'entraîner le modèle de manière efficace grâce à la descente de gradient.

15 Question 2 : Boucle d'Entraînement avec TensorBoard

15.1 Code

Ce code implémente une boucle d'entraînement utilisant la descente de gradient pour ajuster les paramètres d'un modèle de régression linéaire. Les pertes d'entraînement et de test sont enregistrées à chaque itération dans TensorBoard.

```
import torch
from torch.utils.tensorboard import SummaryWriter

# Supervised dataset (randomly generated)
x_train = torch.randn(50, 13, requires_grad=False)  # 50
    exemples, 13 features
y_train = torch.randn(50, 3, requires_grad=False)   # 50
    exemples, 3 sorties

# Test dataset
x_test = torch.randn(20, 13, requires_grad=False)  # 20
    exemples de test
y_test = torch.randn(20, 3, requires_grad=False)

# Model parameters to optimize
w = torch.randn(13, 3, requires_grad=True)  # 13 features d'
    entr e , 3 sorties
b = torch.randn(3, requires_grad=True)      # Biais pour 3
    sorties

# Learning rate and number of iterations
lr = 0.01
epochs = 100

# TensorBoard writer
writer = SummaryWriter()

# Training loop
for n_iter in range(epochs):
    # Forward pass: calculate predictions and loss
    yhat_train = linear(x_train, w, b)
    loss_train = mse(yhat_train, y_train)

    # Calculate test loss
    with torch.no_grad():
        yhat_test = linear(x_test, w, b)
        loss_test = mse(yhat_test, y_test)

    # Log the losses to TensorBoard
    writer.add_scalar('Loss/train', loss_train.item(),
    n_iter)
    writer.add_scalar('Loss/test', loss_test.item(), n_iter)

    # Print the losses for monitoring
    print(f"Iteration {n_iter}: Train Loss = {loss_train.
    item()}, Test Loss = {loss_test.item()}")

    # Backward pass: compute gradients
    loss_train.backward()
```

```
    # Gradient descent: update weights and biases
    with torch.no_grad():
        w -= lr * w.grad
        b -= lr * b.grad

    # Zero the gradients after updating
    w.grad.zero_()
    b.grad.zero_()

# Close the TensorBoard writer
writer.close()
```

Listing 6: Entraînement avec TensorBoard

15.2 Justification

Nous avons choisi un taux d'apprentissage (`lr`) de 0.01 et 100 itérations pour entraîner le modèle. Les résultats des pertes d'entraînement et de test sont suivis via TensorBoard, ce qui permet d'analyser visuellement la convergence.

15.3 Résultats

Voici quelques exemples des résultats obtenus lors de l'entraînement :

```
Iteration 0: Train Loss = 16.68, Test Loss = 9.35
Iteration 10: Train Loss = 13.92, Test Loss = 8.19
Iteration 50: Train Loss = 7.33, Test Loss = 5.34
Iteration 99: Train Loss = 3.97, Test Loss = 3.70
```

Analyse : Les pertes diminuent de manière significative, tant pour l'entraînement que pour le test. Cela montre que le modèle converge et apprend correctement sur les deux ensembles.

16 Question 3 : Vérification des Gradients avec `gradcheck`

16.1 Code

L'outil `gradcheck` de PyTorch permet de vérifier que les gradients calculés analytiquement correspondent à ceux obtenus par différences finies. Voici le code utilisé pour vérifier les gradients de `MSE` et `Linear`.

```
import torch

# Test MSE gradient check
print("V rification du gradient pour MSE:")
```

```
yhat_test = torch.randn(10, 5, requires_grad=True, dtype=
    torch.float64)
y_test = torch.randn(10, 5, requires_grad=True, dtype=torch.
    float64)

# Perform gradcheck on MSE
try:
    assert torch.autograd.gradcheck(mse, (yhat_test, y_test)
    , eps=1e-6, atol=1e-4)
    print("MSE Gradient Check r ussi!")
except AssertionError:
    print("MSE Gradient Check    chou  !")

# Test Linear gradient check
print("\nV rification du gradient pour Linear:")
X_test = torch.randn(10, 5, requires_grad=True, dtype=torch.
    float64)
W_test = torch.randn(5, 3, requires_grad=True, dtype=torch.
    float64)
b_test = torch.randn(3, requires_grad=True, dtype=torch.
    float64)

# Perform gradcheck on Linear
try:
    assert torch.autograd.gradcheck(linear, (X_test, W_test,
     b_test), eps=1e-6, atol=1e-4)
    print("Linear Gradient Check r ussi!")
except AssertionError:
    print("Linear Gradient Check    chou  !")
```

Listing 7: Vérification des gradients avec gradcheck

16.2 Justification

Nous utilisons `gradcheck` pour nous assurer que les gradients calculés pour `MSE` et `Linear` sont corrects. Les tests sont effectués en double précision (`float64`) pour une plus grande précision lors du calcul des différences finies.

16.3 Résultats

Voici les résultats obtenus lors de la vérification des gradients :

```
Vérification du gradient pour MSE:
MSE Gradient Check réussi!

Vérification du gradient pour Linear:
Linear Gradient Check réussi!
```

Analyse : La vérification des gradients est réussie pour les deux fonctions, confirmant que les dérivées calculées par rétropropagation sont correctes.

17 Conclusion

Ce rapport a présenté trois implémentations clés pour la régression linéaire avec PyTorch. Nous avons vérifié que les gradients étaient correctement calculés, et les résultats montrent une bonne convergence du modèle. La visualisation des pertes via TensorBoard a permis de suivre l'évolution des erreurs pendant l'entraînement.