Programmation différentiable
Optimisation : Implémentation et Évaluation

1 Introduction

Ce TP est une étude de plusieurs algorithmes d'optimisation, leur implémentation pratique, et leur évaluation dans le contexte de la programmation différentiable. Ces algorithmes d'optimisation sont cruciaux pour l'entraînement des modèles paramétriques, car ils ajustent les paramètres du modèle afin de minimiser la fonction de perte.

Les objectifs principaux de ce TP sont :

- Comprendre le fonctionnement de différents algorithmes d'optimisation.
- Implémenter et évaluer ces algorithmes dans des scénarios d'entraînement de réseaux de neurones.
- Explorer l'impact des schedulers de taux d'apprentissage sur la convergence des modèles.

Chaque algorithme sera évalué en minimisant des fonctions de perte convexe et non convexe, et en entraînant un réseau de neurones simple. En outre, nous explorerons l'utilisation de `LRScheduler` et de `LRSchedulerOnPlateau` pour ajuster dynamiquement le taux d'apprentissage afin d'optimiser la convergence du modèle. Les résultats obtenus permettront de comparer l'efficacité de ces algorithmes d'optimisation dans différents scénarios d'entraînement et de déterminer leur impact sur les performances du modèle.

2 Modalités de Rendu

Pour ce travail pratique, chaque étudiant devra créer un fichier `Jupyter Notebook` au format `.ipynb`. Ce notebook devra contenir l'implémentation de chaque algorithme d'optimisation et des fonctions associées dans des blocs de code distincts.

Il est attendu que le notebook soit bien structuré, avec une séparation claire entre les différentes parties du travail. Chaque bloc de code doit être précédé de commentaires explicatifs détaillant la logique et le fonctionnement de l'algorithme ou de la fonction implémentée. De plus, des blocs de texte explicatif, en utilisant la syntaxe Markdown, devront accompagner chaque section pour introduire les concepts théoriques et les décisions de conception prises lors de l'implémentation.

L'accent sera mis sur la clarté et la lisibilité du code, ainsi que sur la pertinence et la qualité des explications fournies. Les étudiants devront également inclure une conclusion résumant les observations et résultats obtenus à la suite de l'évaluation des différents algorithmes avec les representations numériques et graphiques adaptées.

3 Jeux de Données Synthétiques

Pour ce TP, nous utiliserons deux jeux de données synthétiques : un jeu linéaire et un jeu non linéaire. Ces jeux de données permettront d'évaluer les performances des algorithmes d'optimisation et des schedulers de taux d'apprentissage dans des contextes différents.

3.1 Jeu de Données Linéaire

Le jeu de données linéaire est généré à partir d'une relation linéaire entre les variables x et y avec l'ajout d'un bruit gaussien. Ce jeu de données est utilisé pour tester les algorithmes dans des conditions où la relation entre les entrées et les sorties est simple et bien définie.

```python
import numpy as np
import matplotlib.pyplot as plt

# Génération du jeu de données linéaire
np.random.seed(0)
n_samples = 100
x_linear = np.linspace(-10, 10, n_samples)
y_linear = 3 * x_linear + 5 + np.random.normal(0, 2,
    n_samples)
```

Listing 1: Génération du jeu de données linéaire

3.2 Jeu de Données Non Linéaire

Le jeu de données non linéaire est basé sur une relation quadratique entre x et y avec un bruit ajouté. Ce jeu de données permet d'évaluer les algorithmes dans des situations où la relation entre les variables est plus complexe.

```
# Génération du jeu de données non linéaire
y_nonlinear = 0.5 * x_linear**2 - 4 * x_linear + np.random.
    normal(0, 5, n_samples)
```

Listing 2: Génération du jeu de données non linéaire

Ces deux jeux de données offrent un cadre idéal pour explorer la performance des algorithmes d'optimisation dans des contextes variés.

4 Présentation des Algorithmes

4.1 Stochastic Gradient Descent (SGD)

SGD est une méthode d'optimisation qui met à jour les paramètres du modèle en fonction du gradient de la fonction de perte par rapport aux paramètres. Cette méthode est simple et efficace pour de nombreux problèmes d'optimisation.

Formule :

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t) \tag{1}$$

```
class SGD(Optimizer):
    def __init__(self, params, learning_rate=0.01):
        # Implémentation ici...
```

Listing 3: Implémentation de SGD

4.2 RMSProp

RMSProp adapte le taux d'apprentissage pour chaque paramètre en fonction de la moyenne des carrés des gradients passés. Cela aide à stabiliser les mises à jour des paramètres en réduisant l'impact des grands gradients.

Formule :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla_\theta L(\theta_t) \tag{2}$$

```
class RMSProp(Optimizer):
    def __init__(self, params, learning_rate=0.01, decay
        =0.9):
        # Implémentation ici...
```

Listing 4: Implémentation de RMSProp

4.3 Adagrad

Adagrad ajuste le taux d'apprentissage pour chaque paramètre en fonction des gradients passés, favorisant des mises à jour plus importantes pour les paramètres moins fréquemment mis à jour.

Formule :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_\theta L(\theta_t) \tag{3}$$

```
class Adagrad(Optimizer):
    def __init__(self, params, learning_rate=0.01):
        # Implémentation ici...
```

Listing 5: Implémentation de Adagrad

4.4 Adam

Adam combine les avantages de RMSProp et de Momentum en utilisant des moments exponentiels pour estimer les premiers et seconds moments des gradients.

Formules :

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\theta L(\theta_t) \tag{4}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla_\theta L(\theta_t))^2 \tag{5}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{6}$$

```
class Adam(Optimizer):
    def __init__(self, params, learning_rate=0.001, beta1
        =0.9, beta2=0.999, eps=1e-8):
        # Implémentation ici...
```

Listing 6: Implémentation de Adam

4.5 AdamW

AdamW est une variante d'Adam qui ajoute une régularisation L2 pour améliorer la généralisation.

Formule :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t - \eta \lambda \theta_t \tag{7}$$

```
class AdamW(Optimizer):
    def __init__(self, params, learning_rate=0.001, beta1
        =0.9, beta2=0.999, eps=1e-8, weight_decay=0.01):
        # Implémentation ici...
```

Listing 7: Implémentation de AdamW

5 Évaluation des Optimiseurs

5.1 Fonctions de Perte

Les fonctions de perte incluent une fonction convexe $(x-2)^2$ et une fonction non convexe $3x^2 - 2x$.

```
def f(x):
    return (x - 2)**2

def f_nonconvexe(x):
    return 3*x**2 - 2*x
```

Listing 8: Définition des fonctions de perte

5.2 Expérimentation

L'évaluation des optimiseurs est réalisée en minimisant ces fonctions de perte. Les résultats sont imprimés à chaque itération pour comparer les performances des différents algorithmes.

```
def eval_optim():
    # Implémentation ici...
```

Listing 9: Évaluation des optimiseurs

6 Réseau de Neurones

6.1 Définition du Modèle

Un réseau de neurones simple avec une couche cachée est défini, ainsi que la fonction de perte MSE (Mean Squared Error).

```
def func_nn(x, W1, b1, W2, b2):
    h1 = W1 * x + b1
    y = W2 * h1 + b2
    return y

def mse(y, y_hat):
    return (y - y_hat) ** 2
```

Listing 10: Définition du modèle de réseau de neurones

6.2 Entraînement du Réseau

L'entraînement du réseau de neurones est effectué en utilisant des données générées, et les performances sont évaluées pour différents optimiseurs.

```python
def eval_nn_optim():
    # Implémentation ici...
```

Listing 11: Entraînement du réseau de neurones

7 Scheduler de Taux d'Apprentissage

7.1 LRScheduler

Le `LRScheduler` ajuste le taux d'apprentissage en fonction des métriques observées pendant l'entraînement.

```python
class LRScheduler:
    def __init__(self, optimizer, initial_lr):
        # Implémentation ici...
```

Listing 12: Implémentation de LRScheduler

7.2 LRSchedulerOnPlateau

Cette classe ajuste le taux d'apprentissage en fonction de l'absence d'amélioration sur un certain nombre d'époques.

```python
class LRSchedulerOnPlateau(LRScheduler):
    def __init__(self, optimizer, initial_lr, patience=10,
        factor=0.1, min_lr=1e-6, mode='min', threshold=1e-4):
        # Implémentation ici...
```

Listing 13: Implémentation de LRSchedulerOnPlateau

Nous évaluerons les schedulers de taux d'apprentissage, `LRScheduler` et `LRSchedulerOnPlateau`, dans le cadre des deux fonctions mentionnées, à savoir la minimisation des fonctions de perte et l'entraînement du réseau de neurones. Cette évaluation permettra d'analyser leur impact sur la convergence et les performances du modèle.