Programmation différentiable
Elements d'Optimization

Julien Perez

January 15, 2025

Fondamentaux de l'Optimisation

- **Définition de l'optimisation dans le contexte du deep learning :**
  - L'optimisation dans le domaine du deep learning se réfère au processus d'ajustement des paramètres d'un modèle de réseau de neurones afin de minimiser une fonction de perte spécifique.
  - Il s'agit essentiellement de trouver les valeurs optimales des paramètres du modèle pour obtenir les performances souhaitées, telles que la précision élevée dans la classification ou la génération de résultats précis dans le cas de modèles génératifs.

Fondamentaux de l'Optimisation

- **Objectif : Minimisation d'une fonction de perte :**
  - La fonction de perte mesure la différence entre les prédictions du modèle et les valeurs réelles des données.
  - L'objectif de l'optimisation est de trouver les paramètres du modèle qui minimisent cette fonction de perte, ce qui entraîne une meilleure performance du modèle sur les tâches spécifiques pour lesquelles il est conçu.

Fondamentaux de l'Optimisation

- **Notions de gradients et de descente de gradient :**
  - Les gradients représentent la direction et le taux de changement maximal d'une fonction.
  - Dans la descente de gradient, les paramètres du modèle sont ajustés itérativement dans la direction opposée du gradient de la fonction de perte, afin de minimiser la perte.
  - Cette technique est largement utilisée dans l'apprentissage profond pour optimiser les modèles de réseau de neurones.

Besoins en Optimisation

- **Taille des données : Gigantesque.**
  - Les ensembles de données utilisés en deep learning sont souvent massifs, comprenant des millions voire des milliards de points de données.
  - Traiter de telles quantités de données nécessite des algorithmes d'optimisation efficaces et des architectures de calcul parallèle pour une formation rapide et efficace des modèles.

Besoins en Optimisation

- **Complexité des modèles : Profondeur, paramètres.**
  - Les modèles de deep learning peuvent être très complexes, avec des dizaines, voire des centaines, de couches et des millions de paramètres.
  - Optimiser de tels modèles exige des algorithmes capables de naviguer efficacement dans l'espace de recherche des paramètres pour trouver les valeurs optimales, tout en évitant les pièges tels que le surapprentissage.

Besoins en Optimisation

- **Sensibilité aux hyperparamètres.**
  - Les performances des modèles de deep learning sont fortement influencées par les choix des hyperparamètres, tels que le taux d'apprentissage, la taille du lot, etc.
  - Trouver les bons hyperparamètres peut nécessiter des efforts d'optimisation supplémentaires, tels que la recherche par grille ou la recherche aléatoire, ce qui ajoute une complexité supplémentaire au processus d'optimisation.

Besoins en Optimisation

- Les performances des modèles de deep learning sont fortement influencées par les choix des hyperparamètres, tels que :
  - **Taux d'apprentissage :** Contrôle la taille des pas de mise à jour des paramètres pendant l'optimisation. Exemple : 0.001, 0.01, etc.
  - **Taille du lot (batch size) :** Nombre d'exemples d'entraînement utilisés pour calculer le gradient moyen. Exemple : 32, 64, 128, etc.
  - **Architecture du réseau :** Nombre de couches, taille des couches, choix des fonctions d'activation, etc.
  - **Régularisation :** Utilisation de techniques comme la pénalisation L1/L2, le dropout, etc.
  - **Stratégie d'initialisation des poids :** Choix de la distribution initiale pour l'initialisation des poids du réseau.
  - **Stratégie d'optimisation :** Choix de l'optimiseur (SGD, Adam, RMSProp, etc.) et de ses hyperparamètres associés.
- Trouver les bons hyperparamètres peut nécessiter des efforts d'optimisation supplémentaires, tels que la recherche par grille ou la recherche aléatoire, ce qui ajoute une complexité supplémentaire au processus d'optimisation.

Difficultés de l'Optimisation

- **Problème de la convergence :**
  - L'optimisation peut parfois ne pas converger vers une solution optimale, soit en raison de la mauvaise initialisation des paramètres, soit en raison de la nature non convexe de la fonction de perte.
  - Des méthodes d'optimisation efficaces doivent être capables de surmonter ces obstacles pour garantir la convergence vers un minimum global ou un minimum local satisfaisant.

Difficultés de l'Optimisation I

- L'optimisation peut parfois ne pas converger vers une solution optimale, soit en raison de la mauvaise initialisation des paramètres, soit en raison de la nature non convexe de la fonction de perte.
- **Mauvaise initialisation des paramètres :** Si les paramètres du modèle sont initialisés de manière inappropriée, l'optimisation peut être bloquée dans des régions sous-optimales de l'espace des paramètres.

Difficultés de l'Optimisation II

- **Nature non convexe de la fonction de perte :** Les fonctions de perte des modèles de deep learning sont souvent non convexes, ce qui signifie qu'elles peuvent contenir de multiples minima locaux et des points selles. Cela rend la convergence difficile, car les méthodes d'optimisation peuvent être piégées dans des minima locaux ou des points selles et ne pas atteindre le minimum global souhaité.
- Des méthodes d'optimisation efficaces doivent être capables de surmonter ces obstacles pour garantir la convergence vers un minimum global ou un minimum local satisfaisant.

Difficultés de l'Optimisation

- **Risque de surapprentissage (overfitting) :**
  - Le surapprentissage se produit lorsque le modèle s'adapte trop bien aux données d'entraînement, capturant le bruit et les variations aléatoires au lieu des véritables motifs sous-jacents.
  - Les algorithmes d'optimisation doivent être accompagnés de techniques de régularisation, telles que la régularisation L1/L2, le dropout, etc., pour atténuer le risque de surapprentissage.

Difficultés de l'Optimisation

- **Sensibilité aux conditions initiales :**
  - Les performances finales du modèle peuvent être fortement influencées par les conditions initiales des paramètres du modèle.
  - Des initialisations inappropriées peuvent conduire à une convergence vers des optima locaux sous-optimaux ou à une divergence du processus d'optimisation.

Difficultés de l'Optimisation

- **Plateaux, minima locaux :**
  - Les plateaux correspondent à des zones relativement plates de la fonction de perte où le gradient devient proche de zéro, ce qui ralentit ou arrête le processus d'optimisation.
  - Les minima locaux sont des points où le gradient de la fonction de perte est nul, mais qui ne sont pas nécessairement optimaux au sens global.
  - Les algorithmes d'optimisation doivent être robustes face à ces défis pour éviter de rester piégés dans de mauvais optima locaux ou de stagner sur des plateaux.

Descente de Gradient Simple

La descente de gradient simple est l'un des algorithmes d'optimisation les plus fondamentaux utilisés en apprentissage automatique et en apprentissage profond. Il consiste à ajuster itérativement les paramètres du modèle dans la direction opposée du gradient de la fonction de perte.

- **Formule de mise à jour du paramètre θ:**

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla L(\theta_t)$$

où θ_t est le vecteur de paramètres à l'itération t, η est le taux d'apprentissage, et $\nabla L(\theta_t)$ est le gradient de la fonction de perte L par rapport aux paramètres θ_t.

Descente de Gradient Simple

Limitations de la descente de gradient simple:

- Sensible au choix du taux d'apprentissage : Un taux d'apprentissage trop grand peut entraîner une divergence, tandis qu'un taux trop petit peut entraîner une convergence lente.
- Peut rester piégé dans des minima locaux : La descente de gradient simple n'est pas garantie de converger vers un minimum global et peut rester piégée dans des minima locaux ou des points selles.
- Sensible aux conditions initiales : Les performances peuvent varier en fonction des conditions initiales des paramètres du modèle.

Formulation Mathématique de l'Optimisation

- **Problème d'optimisation :**

$$\min_{\theta} L(\theta) \quad \text{où} \quad L(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(f_\theta(x_i), y_i)$$

où :

- $L(\theta)$ est la fonction de perte globale.
- ℓ est la perte individuelle pour chaque échantillon.
- f_θ est le modèle paramétré par θ.
- (x_i, y_i) sont les données d'entraînement.

Calcul du Gradient

- **Gradient de la fonction de perte :**

$$\nabla_\theta L(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell(f_\theta(x_i), y_i)$$

- **Interprétation :**
  - Le gradient indique la direction dans laquelle la fonction de perte augmente le plus rapidement.
  - En suivant la direction opposée au gradient, on peut réduire la fonction de perte.

Descente de Gradient Stochastique (SGD)

- **Algorithme :**

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \ell(f_\theta(x_i), y_i)$$

où η est le taux d'apprentissage, et (x_i, y_i) est un échantillon aléatoire.

- **Avantages :**
  - Moins coûteux que le calcul du gradient global.
  - Permet une mise à jour fréquente des paramètres.
- **Inconvénients :**
  - Variance élevée des mises à jour.
  - Peut osciller autour du minimum.

Moment et Accélération de Nesterov

- **Moment :**

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

  où β est le facteur de moment.

- **Accélération de Nesterov :**

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

- **Avantages :**
  - Accélère la convergence en réduisant les oscillations.

Méthode de Newton

- **Formule de mise à jour :**

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

  où H est la matrice Hessienne de $L(\theta)$.
- **Avantages :**
  - Convergence rapide vers le minimum local.
- **Inconvénients :**
  - Coût computationnel élevé pour calculer et inverser H.

Algorithme de Quasi-Newton (BFGS)

- **Mise à jour de la Hessienne approximative :**

$$H_{t+1} = H_t + \frac{y_t y_t^T}{y_t^T s_t} - \frac{H_t s_t s_t^T H_t}{s_t^T H_t s_t}$$

où $y_t = \nabla_\theta L(\theta_{t+1}) - \nabla_\theta L(\theta_t)$ et $s_t = \theta_{t+1} - \theta_t$.

- **Avantages :**
  - Moins coûteux que la méthode de Newton.
  - Convergence rapide sans nécessiter le calcul exact de H.

Algorithmes de Proximal

- **Problème :**

$$\min_{\theta} \left(L(\theta) + \lambda R(\theta) \right)$$

où $R(\theta)$ est une fonction de régularisation.

- **Mise à jour Proximal :**

$$\theta_{t+1} = \text{prox}_{\lambda R}(\theta_t - \eta \nabla_\theta L(\theta_t))$$

- **Avantages :**
  - Convient pour les problèmes avec des régularisations non différentiables comme L1.

Principaux Algorithmes

1. RMSProp (Root Mean Square Propagation)

- Adaptation du taux d'apprentissage basé sur les gradients récents.
- Réduit la sensibilité aux variations abruptes.

2. Adagrad (Adaptive Gradient Algorithm)

- Adapte le taux d'apprentissage pour chaque paramètre.
- Performant pour les données rares.

3. Adam (Adaptive Moment Estimation)

- Combinaison de RMSProp et d'Adagrad avec moments de gradients.
- Ajustement dynamique du taux d'apprentissage.

RMSProp (Root Mean Square Propagation)

- RMSProp est une technique d'optimisation qui ajuste le taux d'apprentissage de manière adaptative en tenant compte des gradients récents.
- L'idée principale derrière RMSProp est de diviser le taux d'apprentissage pour chaque paramètre par une moyenne mobile de l'élément quadratique du gradient.
- Cette technique réduit la sensibilité aux variations abruptes des gradients et permet une convergence plus stable.
- Formule de mise à jour du paramètre θ :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \cdot g_t$$

où g_t est le gradient, v_t est la moyenne mobile de l'élément quadratique du gradient et ϵ est une petite constante pour éviter la division par zéro.

Adagrad (Adaptive Gradient Algorithm)

- Adagrad est un algorithme d'optimisation qui adapte le taux d'apprentissage de manière individuelle pour chaque paramètre.
- Il donne une grande mise à jour pour les paramètres rares et une petite mise à jour pour les paramètres fréquents.
- Adagrad accumule les carrés des gradients précédents pour chaque paramètre et utilise cette information pour ajuster le taux d'apprentissage de manière adaptative.

La formule de mise à jour du paramètre θ est :

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$$

où η est le taux d'apprentissage, g_t est le gradient, G_t est une matrice diagonale contenant la somme des carrés des gradients jusqu'à l'itération t, et ϵ est une petite constante pour éviter la division par zéro.

Adam (Adaptive Moment Estimation)

- Adam est un algorithme d'optimisation qui combine les idées de RMSProp et d'Adagrad avec les moments des gradients.
- Il utilise une estimation du premier moment (moyenne) et du deuxième moment (variance non centrée) des gradients pour adapter dynamiquement le taux d'apprentissage.
- Adam est robuste et efficace pour un large éventail de problèmes d'optimisation.
- Formule de mise à jour du paramètre θ :

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

où $\hat{m}_t$ et $\hat{v}_t$ sont des estimations non centrées du premier moment et du deuxième moment des gradients, respectivement.

Algorithmes d'Optimisation Avancés

Nadam (Nesterov-accelerated Adaptive Moment Estimation)

- Amélioration de l'algorithme Adam en incorporant la méthode Nesterov pour accélérer la convergence.
- Combine les avantages de Nesterov Momentum et Adam.

Adadelta

- Variation d'Adagrad qui adapte dynamiquement le taux d'apprentissage sans nécessiter de paramètre global.
- Efficace pour les modèles à grande échelle et pour les données rares.

Learning Rate Schedulers

- **Modification du taux d'apprentissage au fil du temps :**
  - Le taux d'apprentissage est un hyperparamètre crucial qui contrôle la taille des pas effectués lors de la mise à jour des poids du modèle.
  - Les Learning Rate Schedulers sont des techniques qui ajustent dynamiquement le taux d'apprentissage au cours de l'entraînement pour améliorer la convergence et la performance du modèle.
- **Stratégies :**
  - **Decay (décroissance) exponentielle :** Le taux d'apprentissage diminue exponentiellement à chaque époque ou après un certain nombre d'itérations. Cette stratégie favorise une convergence plus lente mais plus stable. Exemple : $lr_{\text{new}} = lr_{\text{initial}} \times e^{-kt}$.
- **Step (par palier) décroissant :** Le taux d'apprentissage diminue à des moments prédéfinis pendant l'entraînement, généralement après un certain nombre d'époques. Cette stratégie peut être plus agressive et est souvent utilisée pour stabiliser l'apprentissage lorsque la convergence ralentit. Exemple : $lr_{\text{new}} = lr_{\text{initial}} \times \gamma^{\lfloor \frac{t}{} }$

Méthodes de Régularisation

- **Objectif : Réduire le surapprentissage**
  - Le surapprentissage se produit lorsque le modèle s'adapte trop bien aux données d'entraînement mais ne généralise pas bien aux nouvelles données.
  - Techniques de régularisation :
    - **Dropout :** Désactive aléatoirement des neurones pendant l'entraînement pour forcer le réseau à apprendre de manière plus robuste.
    - **Pénalisation L1/L2 :** Ajoute des termes de régularisation pour limiter la complexité du modèle en réduisant les poids des paramètres.
    - **Augmentation des données :** Utilisation de techniques pour générer des variations artificielles des données d'entraînement, ce qui aide à généraliser le modèle.

Graphe Dynamique et Mise à Jour des Paramètres

- **Les frameworks de deep learning (comme TensorFlow, PyTorch) utilisent des graphes dynamiques :**
  - Les graphes dynamiques sont des structures de données utilisées par les bibliothèques de deep learning pour représenter les opérations effectuées lors de la construction et de l'exécution des modèles.
  - Contrairement aux graphes statiques où les opérations sont définies à l'avance, les graphes dynamiques permettent une construction et une modification flexibles du graphe pendant l'exécution.

Graphe Dynamique et Mise à Jour des Paramètres

- **Le graphe dynamique trace les opérations effectuées sur les tenseurs :**
  - Chaque nœud du graphe représente une opération mathématique, comme l'addition ou la multiplication de tenseurs, ou des opérations plus complexes telles que les convolutions ou les activations.
  - Les arêtes du graphe représentent le flux de données entre les opérations, indiquant ainsi les dépendances et les relations entre les tenseurs.

Graphe Dynamique et Mise à Jour des Paramètres

- **Lors de la rétropropagation, le gradient est calculé et utilisé pour mettre à jour les paramètres via les optimiseurs :**
  - Une fois que la perte est calculée par rapport à la sortie du modèle, la rétropropagation est utilisée pour calculer les gradients de la perte par rapport à chaque paramètre du modèle.
  - Les optimiseurs utilisent ces gradients pour ajuster les valeurs des paramètres dans la direction qui minimise la perte, ce qui met à jour les paramètres du modèle lors de l'entraînement.

Le sous-package `optim` de PyTorch

- **Motivation :**
  - Le sous-package `optim` de PyTorch est conçu pour fournir une interface conviviale pour divers algorithmes d'optimisation utilisés dans l'entraînement de modèles de deep learning.
  - Il simplifie la mise en œuvre et l'expérimentation avec différents algorithmes d'optimisation en offrant une API cohérente et flexible.

Le sous-package `optim` de PyTorch

- **Implémentation d'algorithmes d'optimisation populaires :** Le sous-package `optim` propose une large gamme d'algorithmes d'optimisation tels que SGD, Adam, RMSProp, Adagrad, etc.
- **Personnalisation du comportement d'optimisation :** Il permet de spécifier divers paramètres d'optimisation tels que le taux d'apprentissage, le poids de la décroissance, les moments, etc.
- **Intégration transparente avec les modèles PyTorch :** Les optimiseurs de ce sous-package peuvent être directement utilisés avec les modèles PyTorch, simplifiant ainsi le processus d'entraînement.
- **Suivi et journalisation des métriques :** Certains optimiseurs intègrent des fonctionnalités de suivi et de journalisation des métriques d'entraînement, facilitant l'évaluation et la comparaison des performances des modèles.

Le sous-package optim de PyTorch

Exemple de code Python :

```
import torch
import torch.optim as optim

# Création du modèle
model = ...

# Définition de la fonction de perte
criterion = ...

# Définition de l'optimiseur (par exemple, Adam)
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Définition du scheduler
scheduler = StepLR(optimizer, step_size=30, gamma=0.1)

# Boucle d'apprentissage
for epoch in range(num_epochs):
    ...
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    scheduler.step()
    ...
```

Optimisation Distribuée

- Utilisation de plusieurs processeurs ou GPUs pour accélérer l'optimisation.
- Exemple : La descente de gradient distribuée (Distributed Gradient Descent).
- Nécessaire pour traiter de grands ensembles de données ou des architectures complexes.

AutoML et Optimisation Automatisée

- Utilisation de techniques d'apprentissage automatique pour automatiser le processus de conception de modèles.
- Exemple : L'optimisation de l'architecture du réseau neuronal, la sélection d'hyperparamètres.
- Permet de réduire la dépendance à l'expertise humaine et d'améliorer l'efficacité du processus d'optimisation.

Conclusion

- L'optimisation est essentielle pour entraîner efficacement des modèles de deep learning.
- Les algorithmes d'optimisation adaptatifs sont cruciaux pour gérer les défis spécifiques au deep learning.
- Les schedulers de taux d'apprentissage permettent d'ajuster efficacement l'apprentissage.
- Comprendre le lien entre le graphe dynamique et la mise à jour des paramètres est fondamental pour concevoir des architectures efficaces.