19/11/2023

Computation Graph

Advanced Neural Net

Ramiqi Andi

1. Table des matières

2. Introduction

Dans ce rapport, nous plongeons dans le monde fascinant des graphes computationnels, une composante essentielle de l'informatique moderne et de l'analyse de données. Les graphes computationnels ne sont pas seulement des outils pour représenter des données ; ils sont aussi des moyens puissants pour comprendre et manipuler les relations complexes qui existent au sein de ces données.

L'objectif principal de ce travail pratique (TP) est de se familiariser avec les concepts de base des graphes computationnels et d'apprendre à les utiliser efficacement à l'aide d'une implémentation se rapprochant de PyTorch, une bibliothèque de machine learning très populaire. PyTorch offre une approche intuitive et flexible pour construire et entraîner des réseaux de neurones, et il est particulièrement adapté pour travailler avec des graphes computationnels grâce à sa capacité à gérer des calculs dynamiques et sa facilité d'utilisation.

Pour cela, plusieurs classes tel que « Variable » et « Functions » seront mis en place et expliquées avant d'entrainer un model précis.

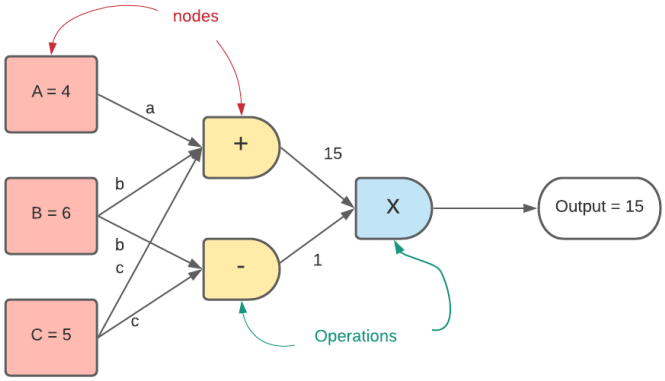
3. Les graphes computationnels

Un graphe est une structure de données composée de nœuds (ou sommets) et d'arêtes (ou liens). Les nœuds représentent des entités tandis que les arêtes représentent les relations entre ces entités.

Dans un graphe computationnel, chaque nœud peut effectuer des calculs simples. Ces calculs dépendent souvent des entrées provenant d'autres nœuds. Par exemple, dans un réseau de neurones (un type de graphe computationnel), chaque neurone (nœud) calcule une sortie en fonction des entrées qu'il reçoit des neurones précédents.

Propagation Avant (Forward Propagation)

La propagation avant est le processus par lequel les valeurs d'entrée sont transmises à travers le graphe, de nœud en nœud, jusqu'à ce qu'une sortie soit obtenue. Chaque nœud traite ses entrées à l'aide d'une fonction (par exemple, une fonction d'activation dans les réseaux de neurones) et transmet le résultat au(x) nœud(s) suivant(s).



Simple exemple d'un graph computationnel

Rétropropagation (Backpropagation) et Apprentissage

Dans les graphes computationnels utilisés pour l'apprentissage automatique, la rétropropagation est une étape cruciale. Après la propagation avant, le graphe évalue la performance (par exemple, à l'aide d'une fonction de perte) et utilise la rétropropagation pour ajuster les poids des connexions (arêtes) dans le graphe, dans le but d'améliorer les prédictions ou les sorties.

4. Mise en place du code

« Variable »

La classe « Variable » est une implémentation personnalisée d'une variable dans un graphe computationnel, similaire à ce que l'on pourrait trouver dans des bibliothèques de comme PyTorch. Cette classe est conçue pour stocker des données, gérer les gradients (dérivées) associés à ces données, et suivre les opérations effectuées sur ces données. Voici une explication détaillée de la classe et en particulier des méthodes backward et update_grad :

Méthode « update_grad »

Cette méthode est cruciale pour la mise à jour des gradients dans le graphe computationnel. Elle prend en entrée un gradient (grad), une fonction enfant (child), et un booléen (retain_graph) qui indique si le graphe doit être conservé après la mise à jour. La méthode ajuste le gradient de la variable en fonction du gradient entrant et de l'opération effectuée. Elle gère également la logique de conservation ou de suppression des références aux enfants, en fonction de la valeur de retain_graph.

Méthode « backward »

La méthode backward est au cœur de la rétropropagation dans les graphes computationnels. Elle est appelée sur une variable pour commencer la rétropropagation à partir de celle-ci. Si grad_fn est None, cela signifie que la variable est une feuille du graphe (c'est-à-dire qu'aucune opération ne l'a créée). Si grad_fn n'est pas None, la méthode déclenche la rétropropagation à travers les opérations qui ont créé cette variable.

- Si retain_graph est True, la structure du graphe est conservée après la rétropropagation, ce qui permet de faire plusieurs passages.
- Si retain_graph est False, les références aux enfants et à grad_fn sont supprimées pour libérer de la mémoire.

Functions

Le but principal de cette classe est de définir une série de fonctions mathématiques (comme l'addition, la soustraction, la multiplication, etc.) et de fournir à la fois leur implémentation directe (forward pass) et leur dérivée (backward pass) pour la rétropropagation dans la mise en place de nos graphes.

Chaque méthode de cette classe contenant les différentes fonctions est codée de la façon suivante (exemple avec la méthode « Add » permettant d'effectuer l'addition de deux éléments « Variable ») :

- **Initialisation (**__init__**)** : Cette méthode initialise l'opération d'addition. Elle prend deux variables x et y comme entrées et calcule leur somme. Le résultat est stocké dans self.result.

- **Forward Pass (**forward**)** : Lorsque la méthode forward est appelée, elle ajoute self (qui est une instance de Add) comme enfant des variables x et y. Cela signifie que x et y sont les entrées pour cette opération d'addition. Ensuite, elle crée une nouvelle Variable avec le résultat de l'addition et définit l'opération actuelle (self) comme la fonction de gradient (grad_fn) de cette nouvelle variable. Cette nouvelle variable est retournée par la méthode forward.

- **Backward Pass (**_backward**)** : La méthode _backward est appelée pendant la rétropropagation. Elle reçoit un grad (gradient) en entrée, qui est le gradient de la perte par rapport à la sortie de l'opération d'addition. Dans le cas de l'addition, le gradient de la sortie par rapport à chaque entrée est simplement 1. Par conséquent, self.dx et self.dy sont tous deux égaux à grad. Ces gradients sont ensuite utilisés pour mettre à jour les gradients des variables d'entrée x et y.

- **Propagation du Gradient (**backward**)** : Après la mise à jour des gradients des entrées, la méthode backward de chaque entrée (x et y) est appelée pour propager le gradient en arrière à travers le graphe computationnel.

Pour chacune des opérations, il sera possible de trouver sur le notebook « ComputationalGraph.ipynb » le calcul des dérivés directes par rapport aux entrées ainsi que le résultat qu'il est possible d'obtenir grâce à la ChainRule.

5. Cross Entropy Loss

Pour calculer la perte, le code utilisé dans le notebook est une version personnalisée pour calculer la Cross-Entropy Loss (CEL), et pour effectuer une rétropropagation (backpropagation) pour calculer les gradients.

$$\text{CEL} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Dans cette formule :

- « C » est le nombre total de classes.
- yi est un indicateur (0, 1) si la classe i est la classe correcte pour l'observation actuelle. (Si l'instance appartient à la 3ème classe, y sera un vecteur et aura comme valeurs : {0 , 0 , 1})
- $\hat{y}_i$ est la probabilité prédite pour la classe i, généralement obtenue en appliquant la fonction softmax aux scores logits.

Pour un problème de classification multi-classes, où chaque observation appartient à une et une seule classe, cette formule calcule la perte pour une seule observation.

Initialisation des données :

Ici, X est une variable contenant les données d'entrée de taille (3x3) et y est une variable contenant les classes correspondantes de taille (3x1). Ces données sont encapsulées dans des objets Variable, qui sont utilisés pour gérer les données et leurs gradients.

Calcul de la perte d'entropie croisée :

« cel » est une instance de la classe CrossEntropyLoss de la bibliothèque nn. Cette classe est utilisée pour calculer la perte d'entropie croisée entre les prédictions (X) et les classes (y).

Rétropropagation :

La méthode « Backward » déclenche le processus de rétropropagation. Dans ce processus, le gradient de la perte est calculé par rapport à toutes les variables impliquées dans le calcul de la perte. Cela permet de mettre à jour les poids du modèle lors de l'entraînement.

Vérification des résultats et des gradients :

check_result_and_grads(loss, X, operation="CEL", itype="array")

« check_result_and_grads »est une fonction personnalisée, qui est utilisée pour vérifier les résultats de la perte et les gradients calculés. Les arguments passés incluent l'objet loss, l'entrée X, le type d'opération ("CEL" pour Cross-Entropy Loss), et le type de l'entrée ("array").

6. MLP Training

Un multilayer Perceptron (MLP) est un type de réseau de neurones artificiels entièrement connecté (chaque neurone d'une couche est connecté à tous les neurones de la couche suivante).

Les MLP sont généralement utilisés pour des tâches d'apprentissage supervisé telles que la classification et la régression. Ils se composent d'une couche d'entrée, d'une ou plusieurs couches cachées et d'une couche de sortie. Chaque neurone de ces couches (à l'exception des neurones d'entrée) applique une fonction d'activation non linéaire, souvent ReLU (Unité Linéaire Rectifiée) ou similaire, après avoir sommé les entrées pondérées.

Mise en place du code :

1. **Définition de la classe (MLP)** : Cette classe hérite de nn.Module, indiquant qu'il s'agit d'un modèle de réseau de neurones. Elle définit un MLP avec deux couches linéaires.

2. **Initialisation (__init__)** : Dans le constructeur, deux couches linéaires sont définies. self.linear_1 est la première couche qui prend des entrées de taille in_features et produit hidden_size caractéristiques. self.linear_2 est la seconde couche qui prend ces caractéristiques hidden_size et produit out_features, qui est typiquement le nombre de classes dans une tâche de classification.

3. **Passage en avant (forward)** : Cette méthode définit le passage en avant du réseau. Elle prend une entrée X, applique la première transformation linéaire (self.linear_1), puis applique la fonction d'activation ReLU (F.relu), et enfin applique la seconde transformation linéaire (self.linear_2). La sortie de cette méthode est les scores bruts (logits) pour chaque classe.

4. **Boucle d'entraînement** : Le script crée ensuite une instance du modèle MLP, configure un optimiseur (SGD pour la descente de gradient stochastique) et une fonction de perte (CrossEntropyLoss pour la classification).

5. **Époques et lots** : Le modèle est entraîné pour un nombre spécifié d'époques (1000 dans ce cas). À chaque époque, les données d'entraînement (X_train et y_train) sont divisées en lots de taille spécifiée (50 ici).

6. **Entraînement par lots** : Pour chaque lot, les étapes suivantes sont effectuées:

    a. **Remise à zéro du gradient** : Les gradients sont remis à zéro, ce qui est nécessaire car les gradients s'accumulent dans PyTorch.

    b. **Passage en avant** : Le modèle calcule le passage en avant sur le lot.

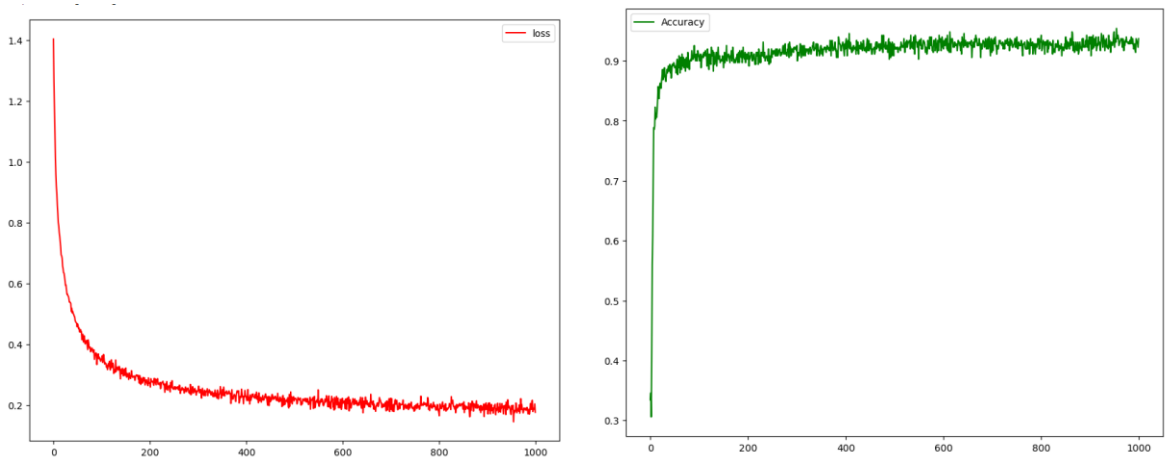
c. **Calcul de la perte** : La perte est calculée à l'aide de loss_fn, en comparant les sorties du modèl.

d. **Passage en arrière** : La backward propagation est effectuée pour calculer les gradients.

e. **Étape d'optimisation** : L'optimiseur met à jour les poids du modèle en fonction des gradients.

7. **Suivi des performances** : Après chaque époque, la perte moyenne et la précision sont calculées et stockées dans history_losses et history_acc. Ces informations sont utilisées pour surveiller le processus d'entraînement.

Le script imprime le numéro de l'époque, la perte moyenne et la précision moyenne après chaque époque, fournissant des informations sur la façon dont le modèle apprend.



Notre model obtient une précision de 90% sur le jeu d'entrainement. Sur la première image, nous pouvons la perte qui, très rapidement va diminuer et se stabiliser vers 0.2.

Avec l'implémentation des tensors à l'aide de la librairie pytorch, les résultats sont très similaires, si ce n'est que la précision est de 91%.

La précision indique que le modèle est capable de bien classer plus de 90% des exemples sur cet ensemble test, ce qui est plutôt un bon résultat.