Rapport TP2 - Deep Learning

Ait Kettout Younes, Reymond Mathis, Ruiz Ocampo Jacobo, Delplace Alexis

Avril 2023

1 Introduction

Nous allons aborder la problématique de la classification d'images issu du dataset MNIST. Pour résoudre ce problème, nous allons utiliser deux modèles, le premier, le plus élémentaire, est une simple régression linéaire (sans hidden-layer). Le deuxième est un réseau de neurones profond avec un nombre arbitraire de hidden layers.

2 TP 1

Lors de ce travail pratique, nous allons définir un modèle de classification des images de bout en bout en définissant les fonctions nécessaires. Ce modèle correspond à un simple classifieur linéaire o=softmax(Wx+b), où x représente l'image en entrée, W représente les paramètres du modèle et b représente les biais du modèle. L'output représente la probabilité de chaque chiffre présent dans l'image en entrée.

2.1 Functions implementées

2.2 Backward affine transform

Cette fonction calcule le gradient de la fonction de coût par rapport aux paramètres W et b en utilisant la formule de la rétropropagation du gradient.

Implementation de cette fonction:

- **Calcul du gradient par rapport à W**: Le calcul du gradient par rapport à la matrice de projection W est effectué en utilisant le produit extérieur entre le gradient entrant g et les caractéristiques d'entrée x, donnant une matrice de la même forme que W. Le calcul est effectué en utilisant la fonction numpy 'np.outer(g, x)'.
- **Calcul du gradient par rapport à b**: Étant donné que le biais est simplement ajouté à chaque élément du produit Wx, le gradient par rapport à b est identique au gradient entrant g.

Les gradients gW et gb sont ensuite utilisés pour ajuster les paramètres W et b de la couche affine pendant l'étape de mise à jour des paramètres de l'algorithme d'optimisation.

2.3 Softmax

Dans cette section on décrit l'implémentatation d'une fonction softmax stable dès lors on décompose son calcul en deux parties:

- **Calcul des exponentielles**: La fonction softmax nécessite de calculer l'exponentielle de chaque élément du vecteur d'entrée w. Cependant, les exponentielles de grandes valeurs peuvent entraîner des problèmes de débordement numérique (c'est-à-dire des résultats trop grands pour être représentés par des nombres à virgule flottante). Afin de stabiliser le calcul, on soustrait la valeur maximale du vecteur w avant de calculer les exponentielles:

$$\exp(w[i] - \max_j w[j])$$

Cette soustraction n'affecte pas la validité des probabilités résultantes, car elle annule simplement un facteur commun dans le numérateur et le dénominateur de la formule softmax.

- **Normalisation**: Après avoir calculé les exponentielles stabilisées, la fonction divise chaque élément par la somme de toutes les exponentielles pour obtenir un vecteur de probabilités dont la somme est égale à 1. Le calcul est effectué en utilisant la fonction numpy np.sum(exp_x) pour obtenir la somme des exponentielles, puis en divisant chaque élément par cette somme:

$$o[i] = \frac{\exp(w[i] - \max_j w[j])}{\sum_j \exp(w[j] - \max_j w[j])}$$

2.4 Loss function

Dans cette section on va décrire comment on a implementé des fonctions nll (negative log-likelihood) et backward_nll stables.

2.4.1 Fonction nll

- **Calcul de la log-probabilité**: La fonction nll calcule la log-probabilité de la classe correcte ("gold") en utilisant les logits d'entrée x. Cette opération est effectuée en extrayant simplement la valeur correspondante à la classe correcte dans x : $x[gold]$.

- **Stabilité du calcul de log-sum-exp**: Pour calculer la log-probabilité normalisée, il est nécessaire de calculer le logarithme de la somme des exponentielles de x. Cette opération peut être instable numériquement en raison des problèmes de débordement numérique lors du calcul des exponentielles. Pour éviter ce problème, la fonction utilise la méthode log-sum-exp, qui permet de stabiliser le calcul en utilisant la relation suivante :

$$\log\left(\sum_i \exp(x_i)\right) = c + \log\left(\sum_i \exp(x_i - c)\right),$$

où c est la valeur maximale du vecteur x. Ainsi, la fonction nll calcule d'abord $log\_sum\_exp\_x$ en utilisant cette méthode stable.

- **Calcul de la négative log-vraisemblance**: La fonction nll calcule finalement la négative log-vraisemblance en soustrayant la log-probabilité de la classe correcte de $log\_sum\_exp\_x$. Cette opération est stable numériquement et donne une mesure de l'erreur de prédiction.

2.4.2 Fonction backward_nll

- **Calcul des probabilités**: La fonction backward_nll commence par calculer les probabilités pour chaque classe à partir des logits d'entrée x. Ceci est fait en utilisant la fonction exponentielle et en normalisant les valeurs pour que leur somme soit égale à 1, de la même manière que dans l'implémentation de la fonction softmax présentée précédemment.

- **Calcul du gradient**: La fonction backward_nll calcule ensuite le gradient par rapport aux logits d'entrée x. Pour ce faire, elle initialise un vecteur de gradient g_x de la même forme que x avec des zéros. Le gradient de la classe correcte ("gold") est mis à jour en soustrayant le gradient entrant g : $g_x[gold] = -g$. Enfin, la fonction ajoute le produit du gradient entrant g et les probabilités calculées précédemment à g_x. Cette opération est stable numériquement et donne le gradient correct par rapport aux logits d'entrée pour la négative log-vraisemblance.

2.5 Modèle

Notre modèle utilise toutes les fonctions énumérées précédemment. Tout d'abord, nous fixons le nombre d'epochs à 5 et le learning rate à 0,01. Ensuite, nous entrons dans une boucle qui itère le nombre d'epochs. À chaque itération, nous nous assurons de mélanger les données pour éviter toute dépendance créée avec des données spécifiques. Pour chaque image dans le jeu de données d'entraînement, nous calculons son score, enregistrons sa perte grâce à la fonction NLL et accumulons cette perte dans une variable "totalloss". À la fin du programme, cette variable nous permettra de savoir si notre modèle n'a pas trop appris (overfitting). Ensuite, nous mettons à jour les paramètres W et b en utilisant la fonction "backwardaffinetransform", qui calcule le gradient de la fonction de perte et met à jour les paramètres W et b en conséquence.

Le gradient est obtenu à partir de la dérivée partielle de la fonction de perte par rapport à la sortie de la dernière couche, utilisant la fonction "backward" de la loss function.On soustrait les paramètres précedents avec les nouveau multiplié avec le facteur étape. À la fin du parcours du parcours du jeu de données nous actualisons nos paramètre si jamais le résultat de la fonction evaluate est meilleur que le précedent.

2.6 Evaluate

Cette fonction consiste à parcourir le jeu de données correspondant à l'évaluation. Pour chaque image, nous calculons le score à partir de la fonction Affine-Transform, puis nous utilisons la fonction Softmax pour obtenir la probabilité de chaque classe. Nous prenons la valeur maximale de ce tableau de probabilités, qui correspond à la classe prédite, et nous la comparons au vrai label de l'image. Si ces derniers sont égaux, nous incrémentons le nombre de réponses correctes. À la fin de la boucle, nous renvoyons la moyenne des réponses correctes.

2.7 Résultats

la meilleure moynne qu'on a eu est de 0.78 sous un nombre d'epoch de 12 et étape égale à 0.1, Nous remarquons que lors des premières itérations d'epoch le mean loss est énorme puis commence à baisser petit à petit , or que notre parametre converge vers le bon parametre .

3 TP 2

Après avoir implémenté toutes les fonctions nécessaires pour le réseau de neurones et les avoir testées sur un modèle linéaire, nous allons implémenter un réseau de neurones profonds.

3.1 Classes prédéfinies

Nous allons procéder à l'explication des classes prédéfinies présentées dans le notebook.

3.1.1 Tensor

La classe Tensor représente des données numériques ainsi qu'un attribut indiquant si le calcul du gradient de ce tensor est nécessaire. La classe Tensor contient une fonction qui initialise son gradient, une autre qui accumule les valeurs de son gradient, ainsi qu'une fonction de propagation. Si le tensor est un noeud feuille, on initialise sa valeur de gradient avec g. Sinon, on parcourt récursivement tous les tensors qui précèdent le noeud actuel en passant son gradient comme paramètre.

3.1.2 Paramètres

Comme son nom l'indique, la classe Parameter définit un objet paramètre en ayant Tensor comme attribut. La classe Parameter hérite de la classe Tensor, ce qui lui permet de stocker des données numériques ainsi que de déterminer si le calcul de gradient est nécessaire. En outre, la classe Parameter ajoute un attribut name qui permet de nommer le paramètre.

3.1.3 Module

Cette classe permet de récupérer la liste de tous les paramètres présents dans le réseau de neurones.

3.1.4 SGD

La classe SGD permet de mettre à jour les paramètres du réseau de neurones à chaque itération d'une donnée en entrée pendant la phase d'entraînement.

3.1.5 Linear Network

Comme défini dans le TP précédent, cette classe sert à initialiser les paramètres du réseau de neurones et à calculer le score correspondant en utilisant les paramètres et l'entrée correspondants.

3.2 Functions ajoutées

3.2.1 TanH

La fonction tanh est également une fonction d'activation non linéaire utilisée dans les réseaux de neurones pour introduire de la non-linéarité. La

différence avec la fonction ReLU est que la plage de sortie de valeurs est différente, celle-ci est comprise entre -1 et 1. Son calcul consiste en l'appel de la fonction np.tanh. Quant à sa dérivée, elle est représentée par l'expression 1 - np.tanh(x.data) ** 2. Contrairement à ReLU, elle ne peut pas souffrir de neurones morts, car dans ReLU, certains neurones peuvent être bloqués et ne pas être activés.

3.2.2 AffineTransform

L'implémentation est presque identique à celle de ReLU, à l'exception que l'on spécifie si le tenseur a besoin d'un gradient en fonction de ses ancêtres (backptrs). Pour cela, on utilise la fonction any_require_grad.

3.2.3 Fonction glorot_init

Cette méthode d'initialisation est conçue pour améliorer la convergence lors de l'apprentissage de réseaux de neurones profonds en assurant une variance uniforme des activations des neurones. La fonction prend en compte les dimensions d'entrée et de sortie de la matrice de poids W et calcule la limite de l'intervalle d'initialisation en utilisant la formule suivante:

$$limit = \sqrt{\frac{6.0}{in\_dim + out\_dim}}$$

Ensuite, la fonction génère une matrice de poids avec des valeurs uniformément réparties dans l'intervalle $[-limit, limit]$.

3.2.4 Fonction kaiming_init

La fonction prend en compte les dimensions d'entrée et de sortie de la matrice de poids W et calcule la limite de l'intervalle d'initialisation en utilisant la formule suivante:

$$limit = \sqrt{\frac{6.0}{out\_dim}}$$

Ensuite, la fonction génère une matrice de poids avec des valeurs uniformément réparties dans l'intervalle $[-limit, limit]$.

3.2.5 Training Loop

La fonction training_loop est utilisée pour entraîner un réseau de neurones linéaire. La fonction effectue les étapes suivantes pour chaque époque :

(A) Initialiser la perte totale à 0.

(B) Pour chaque donnée et étiquette d'entraînement (x, gold) :

  (a) Convertir x en un tenseur.

  (b) Effectuer la propagation avant pour calculer la sortie du réseau v.

  (c) Calculer la perte en utilisant la fonction de coût nll (Negative Log Likelihood).

  (d) Réinitialiser les gradients à zéro avec optimizer.zero_grad().

  (e) Effectuer la rétropropagation en appelant loss.backward(1.0).

  (f) Mettre à jour les poids du réseau en utilisant l'optimiseur.

  (g) Accumuler la perte totale.

  (h) Calculer la perte moyenne, la précision d'entraînement et la précision de validation.

  (i) Afficher les résultats pour l'époque en cours.

3.3 Description de la classe DeepNetwork

La classe DeepNetwork effectue les opérations suivantes :

1. Crée des listes de modules pour les poids W et les biais b des couches cachées.

2. Initialise la fonction d'activation et la fonction de rétropropagation correspondante en fonction du paramètre activation_is_tanh.

3. Ajoute la première couche cachée et les couches cachées restantes à la liste des modules de poids et de biais.

4. Ajoute la couche de sortie.

5. Initialise les paramètres du réseau.

Fonctionnement de la classe DeepNetwork :

Les listes de modules self.W et self.b sont initialisées pour stocker les poids et les biais des couches cachées, respectivement.

La fonction d'activation et la fonction de rétropropagation sont définies en fonction de la valeur de activation_is_tanh.

Les couches cachées sont ajoutées à la liste des modules avec leurs poids et biais respectifs. La première couche cachée est ajoutée séparément, puis les couches cachées restantes sont ajoutées dans une boucle.

La couche de sortie est définie avec ses poids et biais respectifs (self.output_proj et self.output_bias).

La méthode forward effectue les opérations suivantes pour chaque entrée x :

Pour chaque couche cachée, elle applique une transformation affine en utilisant les poids et les biais de la couche, puis applique la fonction d'activation.

Les gradients et les pointeurs de rétropropagation sont mis à jour lors de la propagation avant.

Pour la couche de sortie, elle applique une transformation affine en utilisant les poids et les biais de la couche de sortie.

La sortie x de la dernière couche est renvoyée.

4 Résultats

Finalement, l'entrainement du modèle s'est effectué sur un seul coeur d'un seul processeur (nous n'avons pas investi des bibliothèques comme joblib qui permettent de distribuer les calculs, cenqui serait utile si l'on utilisait des mini-batch). Pour cette raison, nous avons limité la taille du modèle à un hidden-layer de 100 neurones. La fonction d'activation choisie est la tangente hyperbolique. Nous avons fixé le learning rate à 0.01 et entrainé le modèle sur 10 epochs.

On obtient des performances raisonnables, proches de celles attendues. On observe un léger over-fitting sur lequel on aurait pu travailler (c.f. dernière section).

Nous avons aussi implémenter une fonction pour calculer et afficher la matrice de confusion. Cependant, comme les permances sont très bonnes, elle est

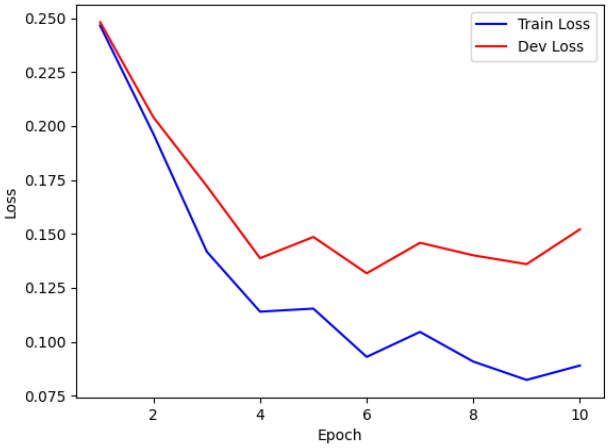

Figure 1: Loss moyenne en fonction du nombre d'epochs

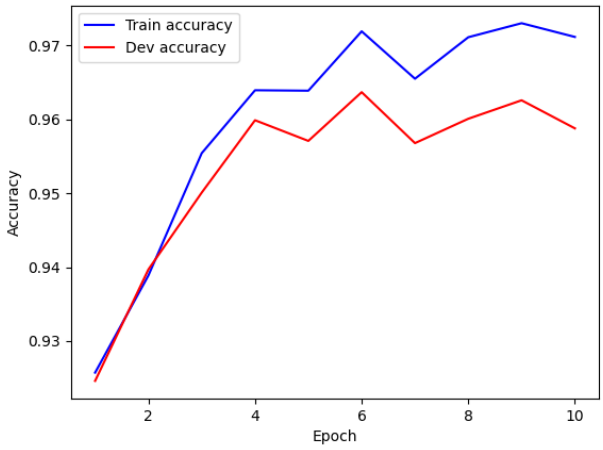

Figure 2: Accuracy moyenne en fonction du nombre d'epochs

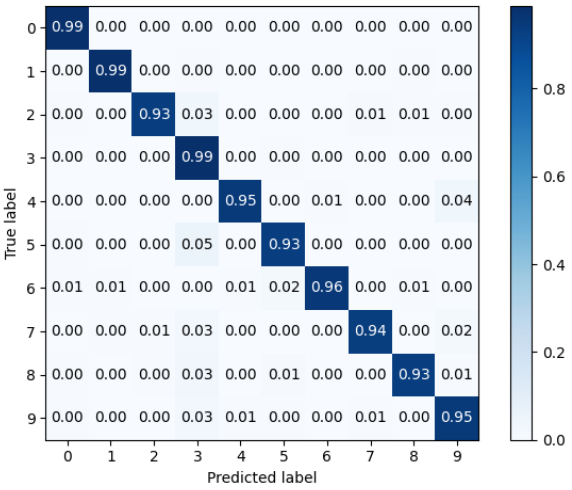

Figure 3: Matrice de confusion

assez peu informative. On peut simplement remarquer que le modèle a une légère tendance à prendre les 4 pour des 9 et qu'il performe moins bien sur les chiffres 2,5 et 8.

5 Difficultés

Ce cours est riche en informations, chacune étant pertinente pour l'implémentation du réseau de neurones. C'est pourquoi il est primordial de comprendre chaque aspect. Le plus difficile a été de faire le lien entre chaque classe et d'effectuer les bons appels de fonctions. En raison d'un manque de familiarité avec la programmation utilisant des classes, nous avons eu beaucoup de mal à comprendre pourquoi notre modèle à un seul layer ne fonctionnait pas. Cette ignorance a conduit à une perte de temps considérable, pour finalement réaliser que notre implémentation de glorot-init n'était pas en place, alors que l'appel à la méthode 'init-parameters' de la classe 'LinearNetwork' pour l'initialisation des paramètres se faisait en place. De plus, l'utilisation de nombreuses classes rend l'organisation du notebook non linéaire, ce qui rendait difficile de se déplacer dans le notebook pour corriger les erreurs.

6 Further work

Si nous avions disposé de plus de temps, nous aurions aimé pouvoir implémenter et tester quelques stratégies supplémentaires comme le learning-decay, qui consiste à diminuer la valeur du learning rate à mesure que la loss converge ; ou une stochastic gradient descent avec mini-batch, pour sortir plus aisément des minimas locaux et limiter l'over-fitting. Enfin, nous aurions aussi aimé tenter d'implémenter la méthode du drop-out, pour voir si elle se serait avérée efficace même sur un layer peu dense pour limiter davantage l'over-fitting.