Devoir 3 - Apprentissage avec des réseaux de neurones

Remise le 7 décembre sur Moodle (avant minuit) pour tous les groupes.

Consignes

- Le devoir doit être fait par groupe de 2 au maximum. Il est fortement recommandé d'être 2.
- Vous devez uniquement soumettre le fichier `models.py`
- Toutes les consignes générales du cours (interdiction de plagiat, etc.) s'appliquent pour ce devoir.
- Il est permis (et encouragé) de discuter de vos pistes de solution avec les autres groupes. Par contre, il est formellement interdit de reprendre le code d'un autre groupe ou de copier un code déjà existant (StackOverflow ou autre). Tout cas de plagiat sera sanctionné de la note minimale pour le devoir.

Le seul fichier que vous devez éditer est `models.py`.

Enoncé

Dans ce dernier devoir, nous vous proposons de concevoir l'architecture d'un réseau de neurones que vous allez par la suite utiliser afin de :

- réaliser une tâche de régression.
- réaliser une tâche de classification de chiffres représentés sous la forme d'une image.

Installation

Pour ce projet, vous aurez besoin des librairies **numpy** et **matplotlib**. Nous vous recommandons d'installer Conda pour faciliter la gestions des packages sur l'ensemble de vos différents projets python.

Afin de s'assurer que toutes ces dépendences ou été installées correctement, lancez la commande suivante :

```
1    python autograder.py --check-dependencies
```

Si les deux librairies ont été correctement installées, vous devriez voir apparaitre une fenêtre matplotlib avec une ligne tracée tourne dans un cercle.

Ressources fournies

Pour ce projet, nous vous avons fourni une mini-bibliothèque de réseaux neuronaux (`nn.py`) et une collection de jeux de données (`backend.py`).

La bibliothèque dans `nn.py` définit une collection d'objets de type `Node`. Les différents sous-types de `Node` peuvent représenter un nombre réel, une matrice de nombres réels, ou une fonction. Les opérations sur les objets `Node` sont optimisées pour fonctionner plus rapidement qu'en utilisant les types intégrés de Python (p.e., les listes).

Voici quelques-uns des types de Node fournis :

- `nn.Constant` : cette classe représente une matrice (tableau 2D) de nombres à virgule flottante. Il est généralement utilisé pour représenter les caractéristiques d'entrée ou les sorties/labels cibles. Les instances de ce type vous seront fournies par d'autres fonctions de l'API ; vous n'aurez pas besoin de les construire directement.
- `nn.Parameter` : cette classe représente un paramètre entraînable d'un réseau de neurones.
- `nn.DotProduct` : cette classe permet de calculer un produit scalaire entre les entrées données à la création de l'objet.

Il y a également plusieurs fonctions supplémentaires, comme :

- `nn.as_scalar` : cette foction permet d'extraire un nombre à virgule flottante Python d'un `Node`.

Lors de l'entraînement d'un réseau de neurones, vous utiliserez un objet de type `dataset`. Vous pouvez récupérer un ensemble d'échantillons d'entraînement en appelant `iterate_once(batch_size)` :

```
for x, y in dataset.iterate_once(batch_size):
    ...
```

A titre d'exemple, le code suivant permet d'extraire un *batch* de taille 1 (c'est-à-dire un lot d'une seule donnée d'entraînement) de l'ensemble de données d'entraînement.

```
>>> batch_size = 1
>>> for x, y in dataset.iterate_once(batch_size):
...     print(x)
...     print(y)
...     break
...
<Constant shape=1x3 at 0x11a8856a0>
<Constant shape=1x1 at 0x11a89efd0>
```

Les *features* (caractéristiques) d'entrée `x` et la valeur de sortie `y` sont données sous la forme d'un objet `Node` de sous-type `nn.constant`.

La dimension de `x` est `(batch_size,num_features)`, et la dimension de `y` est `(batch_size,num_outputs)`. A titre d'exemple, le code suivant montre le résultat du produit scalaire de `x` par lui-même, d'abord en tant qu'un objet de type `Node`, puis en tant que nombre de type primitif.

```
>>> nn.DotProduct(x, x)
<DotProduct shape=1x1 at 0x11a89edd8>
>>> nn.as_scalar(nn.DotProduct(x, x))
1.9756581717465536
```

⚠Lors de la conception de vos algorithmes, faites bien attention à la dimension de vos matrices et tenseurs. Les problèmes de dimensions incompatibles sont très fréquents en apprentissage profond.

Critères d'évaluation

Votre code sera évalué automatiquement pour en vérifier la validité. Faites attention ne pas modifier les noms des fonctions ou des classes fournies dans le code, sinon cela risque de perturber le correcteur automatique (fonctionnement similaire au Devoir 1). Si le script ne compile pas avec vos fichiers, une note de zéro sera accordée. Vous pourriez également perdre des points si votre code manque de lisibilité. Autrement, la note que vous obtiendrez sera la même que celle obtenue par le correcteur automatique. Les points

seront attribués de la manière suivante.

Question	Points
Question 1	/6
Question 2	/6
Question 3	/6
Lisibilité du code	/2
Total	/20

Question 1 : implémentation d'un perceptron

Avant de commencer cette partie, assurez-vous que vous avez installé **numpy** et **matplotlib** !

Pour cette question, vous allez implémenter un perceptron binaire (modèle de prédiction très simple, une sorte d'*ancêtre* des réseaux de neurones) pour réaliser une tâche de classification entre 2 classes. L'équation fondamentale d'un perceptron binaire est la suivante.

$$\hat{y} = \begin{cases} 1 & \text{si } wx + b \geq 0 \\ -1 & \text{sinon.} \end{cases} \tag{1}$$

Dans cette équation, x est la donnée d'entrée, $\hat{y}$ est la valeur prédite du modèle, et b, w sont les paramètres devant être appris.

Pour le perceptron, les labels de sortie sont soit 1 ou -1, ce qui signifie que pour chaque échantillon de données (x, y) de l'ensemble de données, y est un `Node nn.Constant` qui a une valeur de soit 1 (correspondant à la classe positive) soit -1 (correspondant à la classe négative). L'apprentissage se fait de manière itérative en mettant les poids w à jour à chaque itération, en utilisant uniquement les données qui ont été mal classifiées.

$$w \leftarrow w + yx \text{ si et seulement si } \hat{y} \neq y \tag{2}$$

Notez que w et x sont des vecteurs, et que y est un scalaire. L'entraînement se finit lorsqu'il n'y a plus d'erreurs de prédiction. Vous pouvez voir ça comme une forme simplifiée de descente de gradient. Pour cette question, il vous est demandé de compléter l'implémentation de la classe `PerceptronModel` dans `models.py`.

Nous avons déjà initialisé les poids du perceptron `self.w` pour qu'ils soient un objet de type `Parameter` de dimension `(1,dimensions)`. La valeur `dimensions` indique la dimension des données qui doivent être classées. Par exemple, une valeur de 2 indique qu'on classe des données ayant 2 dimensions (par exemple, des points dans un plan 2D tels que présenté dans le cours avec le problème de l'étudiant).

Le code fourni inclue le biais (b) à l'intérieur de x si cela nécessaire. Il n'est donc pas nécessaire de définir un paramètre séparée pour le biais. Vos tâches sont les suivantes :

- Implémenter la méthode `run(self, x)`. Celle-ci doit calculer le produit scalaire entre w et x de l'Equation (1) et doit retourner un objet de type `nn.DotProduct`.
- Implémenter la méthode `get_prediction(self, x)`, qui doit retourner 1 si le produit scalaire est non-négatif (≥ 0) ou -1 sinon, comme indiqué dans l'Equation (1). Vous devez utiliser `nn.as_scalar` pour convertir un `Node` scalaire en un nombre à virgule flottante. A ce stade, vous avez un modèle de prédiction, mais qui n'est pas encore entraîné !

- Implémenter la méthode `train(self)` qui vise à entraîner votre modèle. Pour cette situation assez simple où toutes les données sont linéairement séparables [1], la procédure d'entraînement que vous devez implémenter est la suivante :
  — Tant que tous les exemples ne sont pas bien classés, itérez sur tout l'ensemble d'entraînement, en prenant les instances une à une. Pour cela, utilisez la fonction `dataset.iterate_once`.
  — Pour chaque instance, repérez si elle est bien classée. Si ce n'est pas le cas, mettez à jour les poids w en suivant l'Equation (2) à l'aide de la fonction `update`.
  — Lorsqu'un passage complet sur l'ensemble de données est effectué sans erreur, cela veut dire que la précision est de 100% et l'entraînement peut prendre fin.

⚠**Note importante** : dans ce devoir, la seule façon de modifier la valeur d'un paramètre est d'appeler la fonction `parameter.update(direction, multiplier)`, qui effectuera la mise à jour des poids du modèle selon une étape de descente de gradient.

$$\text{weights} \leftarrow \text{weights} + \text{direction} \times \text{multiplier}$$

L'argument `direction` est un `Node` de même dimension que `weights`, et `multiplier` est une constante multiplicative (un scalaire), pouvant correspondre à un taux d'apprentissage.

Pour tester votre implémentation, lancez le correcteur automatique avec la commande :

```
python autograder.py -q q1
```

Remarque : l'exécution de l'autograder devrait prendre au maximum 20 secondes pour une mise en œuvre correcte. S'il prend un temps significativement long à s'exécuter, votre code contient probablement une erreur.

Question 2 : construction d'un réseau de neurones

Mise en situation

Pour cette question, vous devrez utiliser le code fourni dans `nn.py` pour concevoir un réseau de neurones permettant de résoudre un problème de régression non linéaire. Pour rappel, un réseau de neurones simple comporte des couches, où chaque couche effectue une opération linéaire (tout comme le perceptron). Les couches sont séparées par une fonction non linéaires, ce qui permet au réseau de donner une approximation de fonctions plus générales. Nous utiliserons l'opération `ReLU` comme fonction d'activation.

$$\texttt{ReLU}(x) = \max(x, 0) \tag{3}$$

A titre d'exemple, l'équation suivante présente un réseau de neurones à deux couches permettant d'obtenir des prédictions $\hat{Y}$ à partir d'un ensemble de données X.

$$\hat{Y} = W^{[2]}\texttt{ReLU}\left(W^{[1]}X + b^{[1]}\right) + b^{[2]} \tag{4}$$

Notez que la notation matricielle est utilisée. Si vous avez du mal avec cette équation, nous vous conseillons de revoir la séance théorique avant de continuer cette question. Ainsi, $\mathbf{W}_1$, $\mathbf{W}_2$, $\mathbf{b}_1$ et $\mathbf{b}_2$ sont les paramètres qui doivent être appris. En suivant ce formalisme, il est très aisé de concevoir un réseau plus profond un rajoutant une couche supplémentaire, comme présenté à l'équation suivante.

$$\hat{Y} = W^{[3]}\texttt{ReLU}\left(W^{[2]}\texttt{ReLU}\left(W^{[1]}X + b^{[1]}\right) + b^{[2]}\right) + b^{[3]} \tag{5}$$

1. Et donc peuvent être classées exactement avec notre perceptron.

On est libre de choisir n'importe quelle valeur pour la dimension de chaque couche cachée. L'utilisation d'une dimension plus grande rend généralement le réseau plus expressif (capable de s'adapter à plus de données d'apprentissage), mais peut rendre le réseau plus difficile à entraîner (puisqu'il ajoute plus de paramètres à apprendre). On augmente ainsi le risque de sur-apprentissage des données d'entraînement.

Conseils pratiques

Utilisation du *batching*

Pour des raisons d'efficacité, on peut réaliser une itération de descente de gradient en utilisant, non pas, l'entièreté des données d'entraînement comme vu au cours, mais simplement un sous-ensemble des données. Ce sous-ensemble est communément appelé *batch* d'entraînement. On parle alors d'apprentissage par *mini-batchs*. A chaque itération sur l'ensemble des données d'entraînement, on commence par répartir aléatoirement les données en plusieurs *mini-batchs*, puis, chaque étape de descente de gradients s'effectue sur chaque *mini-batch* pris séparément. Une passe complète sur tous les *mini-batchs* est appelé une *epoch*. à L'extrême, l'exemple du perceptron n'utilisait que des *mini-batchs* n'ayant qu'une seule donnée à chaque fois. On parle dans ce cas de *descente de gradient stochastique* (SGD). Dans ce devoir, vous pouvez jouer avec la taille des *mini-batchs* à l'aide de l'argument de la fonction `dataset.iterate_once(batch_size)`.

Gestion de l'aléatoire

Les paramètres d'un réseau de neurones sont souvent initialisés aléatoirement, ce qui fait que les performances peuvent varier d'une exécution de l'apprentissage à une autre. Il en va de même avec un apprentissage par *mini-batch* où données seront utilisées à différents moments pour la descente de gradient. En conséquence, il est possible que vous échouiez occasionnellement dans certaines tâches, même avec une architecture solide - c'est le problème des *optima locaux*.

Pour ce devoir, cela ne devrait toutefois se produire que très rarement - si, lors du test de votre code, vous faites échouer le correcteur automatique deux fois de suite pour une question, c'est un signe que votre architecture doit être modifiée.

En général, une bonne pratique est d'intégrer une *seed* à vos algorithmes afin de pouvoir reproduire vos expériences.

Conseils divers

La conception d'un réseau de neurones peut nécessiter quelques essais et erreurs, surtout lorsqu'on débute dans ce domaine. Voici quelques conseils pour vous aider dans vos démarche de conception.

- **Soyez systématique** : tenez un registre de toutes les architectures, des hyperparamètres (profondeur et largeur du réseau, taux d'apprentissage, etc.) que vous avez testés. Pour chacun d'entre eux, reportez les performances obtenues. Une métrique intéressante est d'observer l'évolution de la fonction de perte lors de l'entraînement. Pour une tâche de classification, reportez également la précision de classification. Au fur et à mesure de vos expériences, vous commencerez à voir des tendances sur les valeurs qui fonctionnent le mieux. Si vous trouvez une erreur dans votre code, assurez-vous de ne pas considérer les résultats antérieurs qui ne sont pas valides à cause de l'erreur.
- **Commencez par un réseau peu profond** : concrètement, testez d'abord un réseaux à deux couches, c'est-à-dire avec une seule activation non-linéaire. Plus le réseau est grand, plus le nombre d'hyperparamètres est important. Il arrive qu'une seule mauvaise valeur soit la cause de performances

médiocres. Réduisez ainsi ce risque. Utilisez votre premier réseau pour trouver un bon taux d'apprentissage et une bonne taille de couche. Vous pourrez ensuite envisager d'ajouter d'autres couches de taille similaire.
- **Importance du taux d'apprentissage** : si votre taux d'apprentissage est mauvais, aucun des autres choix d'hyperparamètres n'a d'importance. Il s'agit d'un hyperparamètre très important qui vaut la peine d'être bien calibré.
- **Adaptez le taux d'apprentissage à la taille des *mini-batchs*** : de manière générale, les *mini-batchs* plus petits nécessitent des taux d'apprentissage plus faibles. Lorsque vous expérimentez avec différentes tailles de *mini-batchs*, tenez compte que le meilleur taux d'apprentissage peut être différent selon la taille du *mini-batchs*.
- **Évitez de rendre le réseau trop large** : il peut être tentant de rajouter plus de neurones afin d'améliorer l'expressivité du réseau. Cependant, cela vient avec un coût plus important pour l'entraînement. Si votre code prend beaucoup plus de 10 minutes lors de l'entraînement, vous devriez revoir son efficacité.
- **Analyser les valeurs de sortie** : Si votre modèle renvoie des valeurs extrêmement grandes ou `NaN`, votre taux d'apprentissage est probablement trop élevé pour votre architecture actuelle.
- **Vérifiez vos modèles** : une difficulté importante en apprentissage profond est que les erreurs ne causent pas forcément un problème de compilation. En d'autres mots, un modèle incorrect peut parfaitement être exécuté sans que vous remarquiez l'erreur. Vérifiez bien que les valeurs que vous obtenez sont cohérentes. Par exemple, vous ne devriez jamais avoir une valeur négative après l'application d'un `ReLU`. Pour cela, vous devez bien comprendre la théorie derrière les modèles.

Architecture recommandée

Afin de vous aider dans votre travail, voici des intervalles de valeur que nous recommandons pour vos hyperparamètres :

- *Dimension des couches cachées* : entre 10 et 400.
- *Taille des mini-batchs* : entre 1 et la taille de l'ensemble de données.
- *Taux d'apprentissage* : entre 0.001 et 1.0.
- *Nombre de couches cachées* : entre 1 et 3.

Nous demandons également que la taille totale du jeu de données soit divisible par la taille du *mini-batch*. M!s à part cette contrainte, vous êtes libres d'essayer d'autre choix.

Code Fourni

Voici une liste complète des `Node` disponibles dans `nn.py` à utiliser pour la suite du devoir.

- `nn.Parameter` : cette classe représente un paramètre entraînable d'un réseau de neurones. Tous les paramètres doivent avoir deux dimensions. Référez-vous au cours théorique si vous n'êtes pas certain de la valeur des dimensions à mettre. L'utilisation de `nn.Parameter(n, m)` construit un paramètre w de dimension $n \times m$.
- `nn.Add` : classe qui permet de réaliser la somme élement par élement (*element-wise*) de deux matrices. Ainsi, `nn.Add(x, y)` prend en entrée deux `Node` de dimension $(n \times m)$ construit un nouveau `Node` de même dimension.
- `nn.AddBias` : classe qui permet d'ajouter un vecteur de biais (b) à une matrice via une opération de *broadcasting*. Ainsi, `nn.AddBias(X, b)` prend en entrée une matrice X de dimension $(n \times m)$ et un vecteur b de dimension $(1 \times m)$, et construit un `Node` de dimension $(n \times m)$.
- `nn.Linear` : classe qui permet d'appliquer une transformation linéaire entre deux matrices (c-à-d,

une multiplication matricielle). Ainsi, `nn.Linear(X, W)` prend en entrée une matrice X de dimension $(n \times m)$ et une matrice W de dimension $(m \times p)$, et construit un `Node` de dimension $(n \times p)$.
- `nn.ReLU` : classe qui permet d' appliquer une activation `ReLU` à tous les éléments d'une matrice. Ainsi, `nn.ReLU(Z)`, retourne un `Node` de même dimension que Z.
- `nn.SquareLoss` : classe qui permet de calculer l'erreur des moindres carrés sur le *mini-batch* actuel. Ainsi, `nn.SquareLoss(a, b)`, calcule la somme des erreurs entre les valeurs contenues dans les matrices a et b. Ces deux matrices doivent avoir la même dimension $(n \times m)$
- `nn.SoftmaxLoss` : classe qui permet de calculer l'erreur d'entropie croisée (*categorical cross-entropy loss*) sur le *mini-batch* actuel. Ainsi `nn.SoftmaxLoss(a,b)` calcule la somme des erreurs entre les valeurs contenues dans les matrices a et b qui doivent être de même dimension. Attention cependant, a correspond à la sortie d'un réseau **avant l'application de la fonction softmax**. Il peut donc s'agit de nombre réels arbitraire. La matrice b reprend les labels (valeur y)de chaque donnée présente dans le *mini-batch*.

Vous avez également à disposition les méthodes suivantes dans `nn.py` :

- `nn.gradients` : cette fonction calcule le gradient d'une fonction de perte vis-à-vis des paramètres W et b. Ainsi, `nn.gradients(L, [w_1, w_2, ..., w_n])` retournera une liste $[w_1, w_2, \ldots, w_n]$, où chaque élément est un `Node nn.Constant` contenant le gradient de la fonction de perte L par rapport à un paramètre.
- `nn.as_scalar` : cette fonction permet d'extraire un nombre à virgule flottante d'un `Node` d'une fonction de pertes. Cela peut être utile pour déterminer quand arrêter l'entraînement (par exemple, lorsque l'erreur résiduelle est en dessous d'un certain seuil. Ainsi, `nn.as_scalar(N)` retourne la valeur de N qui doit être de dimension (1×1).

Il y a également deux fonctions à utiliser dans le fichier`backend.py` :
- `dataset.iterate_forever(batch_size)` : cet itérateur permet de générer continuellement des *mini_batchs* d'une certaine taille.
- `dataset.get_validation_accuracy()` : cette fonction renvoie la précision de votre modèle sur l'ensemble de validation. Cela peut être utile pour déterminer quand arrêter l'entraînement

⚠**Pour la suite du devoir, n'utilisez plus la classe** `nn.DotProduct`**. Référez vous également à la documentation des fonctions proposées dans le code fourni.**

Exemple : Régression linéaire

Pour illustrer le fonctionnement du code fourni et de vous aider dans votre travail, nous allons réaliser ensemble une tâche de régression linéaire sur un ensemble de points de données. La fonction à apprendre est la suivante.

$$y = 7x_1 + 8x_2 + 3 \tag{6}$$

Supposons que nous ayons extrait 4 points $(x_1^{(i)}, x_2^{(i)} \rightarrow y^{(i)})$ de cette fonction. Sous la forme d'un *batch*, ces données peuvent être écrites de la manière suivante.

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \text{ et } Y = \begin{bmatrix} 3 \\ 11 \\ 10 \\ 18 \end{bmatrix}$$

En supposant que les données nous soient fournies sous la forme d'objets `nn.Constant`, on aura l'exécution suivante. Notez bien les dimensions des matrices.

```
>>> x
<Constant shape=4x2 at 0x10a30fe80>
>>> y
<Constant shape=4x1 at 0x10a30fef0>
```

Ensuite, construisons un modèle linéaire pour apprendre la fonction sur base des 4 points échantillonnés. Comme vu au cours, un modèle linéaire à la forme suivante.

$$\hat{y} = w_1 \times x_1 + w_2 \times w_2 + b_1 \tag{7}$$

Si cela est fait correctement, nous devrions être en mesure d'apprendre que $w_1 = 7$, $w_2 = 8$, et $b = 3$. La prochaine étape est de créer les paramètres d'apprentissage. La représentation matricielle est la suivante.

$$W = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \text{ and } B = [b_1]$$

```
w = nn.Parameter(2, 1)
b = nn.Parameter(1, 1)
>>> w
<Parameter shape=2x1 at 0x112b8b208>
>>> b
<Parameter shape=1x1 at 0x112b8beb8>
```

On peut ensuite calculer les prédictions de notre modèle pour y.

```
a = nn.Linear(x, w)
predicted_y = nn.AddBias(a, b)
```

Notez bien que l'ajout du terme du biais (b) se fait séparément avec cette librairie. L'objectif du modèle est de s'assurer que les valeurs $\hat{y}$ prédites coïncident avec les valeurs des données en entrée (y). Pour une régression linéaire, cela se fait en minimisant l'erreur quadratique.

```
loss = nn.SquareLoss(predicted_y, y)
```

La prochaine tâche est de calculer le gradient de chaque paramètre (*backpropagation*) afin de mettre à jour le réseau via une étape de descente de gradient.

```
grad_w, grad_b = nn.gradients(loss, [w, b])
>>> grad_w
<Constant shape=2x1 at 0x11a8cb160>
>>> grad_b
<Constant shape=1x1 at 0x11a8cb588>
```

Finalement, nous pouvons utiliser la méthode `update` pour actualiser nos paramètres via une descetne de gradient. Pour cela, il faut définir un taux d'apprentissage, par exemple $\alpha = 0.1$.

```
w.update(grad_w, 0.1)
b.update(grad_n, 0.1)
```

On a ainsi réaliser une itération d'apprentissage ! Pour réaliser un entraînement complet, il faudrait ré-itérer plusieurs fois jusqu'à convergence. A plus grande ampleur, les librairies modernes d'apprentissage profond (`Pytorch`, `Keras`, etc.) permettent une utilisation aisée de ces mécanismes afin que l'utilisateur puisse se focaliser sur la création de son modèle.

Votre réalisation : régression non-linéaire

Pour cette question, vous devez entraîner un réseau de neurones pour approximer la fonction trigonométrique $sin(x)$ sur le domaine $[-2\pi, 2\pi]$. A cette fin, vous devez compléter l'implémentation de la classe `RegressionModel` dans le fichier `models.py`. Pour ce problème, une architecture relativement simple devrait suffire Utilisez une fonction de perte adaptée à un problème de régression. Concrètement, vos tâches consistent à :

- Implémenter la fonction `RegressionModel.run` pour retourner un objet `Node` de dimension $(b \times 1)$ qui représente la prédiction de votre modèle sur un *mini-batch* de taille b.
- Implémenter la fonction `RegressionModel.get_loss` pour renvoyer l'erreur pour les données choisies.
- Implémentez la fonction `RegressionModel.train` qui a pour objectif d'entraîner votre modèle via une descente de gradient.

Il n'y a qu'un seul ensemble de données pour cette tâche, c'est-à-dire qu'il n'y a que des données d'entraînement et pas de données de validation ou d'ensemble de test. Votre implémentation recevra tous les points si elle obtient une perte de 0,02 ou moins, en moyenne sur tous les exemples de l'ensemble de données. Vous pouvez utiliser la fonction de perte pour déterminer quand arrêter l'apprentissage (utilisez `nn.as_scalar` pour convertir un `Node` en valeur scalaire). Finalement, vérifiez votre implémentation via la commande suivante.

```
python autograder.py -q q2
```

Question 3 : Classification d'image de chiffres

Pour cette question, vous allez entraîner un réseau de neurones pour classer des chiffres (entre 0 et 9) écrits la main. Cette tâche est très populaire en apprentissage automatique et les données utilisées viennent de l'ensemble `MNIST`. Concrètement, chaque chiffre manuscrit est encodé comme une image de 28×28 pixels. Chaque pixel contient une valeur entière (intensité de gris) entre 0 (blanc) et 255 (noir). Dans notre cas, on stocke toutes ces valeurs dans un vecteur à 784 dimensions. Il s'agit de l'input de votre réseau. L'output est un vecteur à 10 dimensions, chacune correspond à un chiffre spécifique, qui comporte des zéros sur toutes les dimensions, à l'exception d'un 1 dans la dimension correspondant à la classe correcte du chiffre. Il s'agit d'une représentation *one-hot*.

Complétez l'implémentation de la classe `DigitClassificationModel` dans `models.py`. La valeur de retour de `DigitClassificationModel.run()` doit être un `Node` de dimension $batch\_size \times 10$ contenant des scores, où les scores plus élevés indiquent une plus grande vraisemblance qu'un chiffre appartienne à une classe particulière (0-9). Utilisez une fonction de perte adaptée à un problème de classification multi-classe.

En plus des données d'entraînement, vous avez également un ensemble de validation et un ensemble de test. Vous pouvez utiliser `dataset.get_validation_accuracy()` pour calculer la précision de validation de votre modèle, ce qui peut être utile pour décider d'arrêter la formation. L'ensemble de test sera utilisé par

le correcteur automatique pour calculer votre précision finale et ne vous est pas disponible. Pour obtenir des points pour cette question, votre modèle doit atteindre une précision d'au moins 97% sur l'ensemble de test. Finalement, vérifiez votre implémentation via la commande suivante.

```
python autograder.py -q q3
```