RDFIA TME 4-5
Introduction aux réseaux de neurones

Arthur Douillard Yifu Chen Matthieu Cord

9 - 16 Octobre 2019

Comptes rendus à rendre

— Un compte rendu optionnel de suivi dans la soirée après la séance (par mail à `yifu.chen@lip6.fr` et `arthur.douillard@lip6.fr`). Ce compte rendu contiendra en quelques lignes : (1) les réponses aux questions précédées par ★ ; (2) votre avancement dans le sujet, ce que vous avez réussi ou non ; (3) si vous voulez, des commentaires, remarques sur ce que vous avez compris ou non, etc. Votre mail devra avoir pour object [RDFIA][TP-4-5].

— A l'issue du TP 7 (le 30 Octobre), vous devrez rendre par mail la réponse à toutes les questions. Ce rendu obligatoire sera à rendre au plus tard 10 Novembre 2019 à 23h42.

Les données et une version numérique du sujet sont accessibles
à l'adresse `https://arthurdouillard.com/rdfia/`

Objectifs

Le but du TP est de mettre en place un réseau de neurones simple afin de se familiariser avec ces modèles et la façon de les entrainer : la *backpropagation* (ou rétro-propagation) du gradient.

Pour cela, nous commencerons par étudier d'un point de vue mathématique un perceptron à une couche cachée et sa procédure d'apprentissage. On implémentera ensuite ce réseau avec la librairie PyTorch pour le tester sur un problème jouet pour vérifier son bon fonctionnement, puis sur le jeu de données MNIST.

Le site `http://playground.tensorflow.org` permet de visualiser le fonctionnement et l'apprentissage de petits réseaux de neurones. Vous pouvez vous y rendre afin de mieux appréhender ce TP.

Partie 1 – Formalisation mathématique

Pour appliquer un réseau de neurones à un problème de *machine learning* (en apprentissage supervisé), on a besoin de 4 choses :

— Un **jeu de données annoté** et un problème associé ;

— Une **architecture de réseau** (à adapter aux données) ;

— Une **fonction de coût** que l'on cherche à minimiser (à adapter au problème) ;

— Un **algorithme d'apprentissage** pour résoudre le problème d'optimisation consistant à minimiser cette fonction de coût.

1.1 Jeu de données

On s'intéresse ici à un problème de classification en apprentissage supervisé. On a donc un jeu de données constitué d'un ensemble de N couples $(x^{(i)}, y^{(i)}), i \in 1..N$, où $x^{(i)} \in \mathbb{R}^{n_x}$ est un vecteur de *features* à partir duquel on veut prédire la vérité terrain (*target, ground truth*) $y^{(i)} \in \{0,1\}^{n_y}$ vérifiant $\|y^{(i)}\| = 1$ (encodage *one-hot* : une seule composante est à 1, indiquant la classe à laquelle appartient l'exemple). Ce jeu de données est généralement découpé en plusieurs ensembles : *train*, *test* et parfois *val*.

Questions

1. ★ À quoi servent les ensembles d'apprentissage, de validation et de test ?
2. Quelle est l'influence du nombre N d'exemples ?

1.2 Architecture du réseau (phase forward)

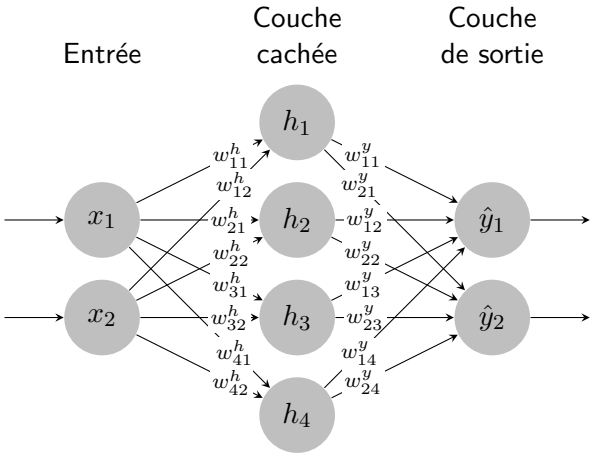

FIGURE 1 – Exemple d'architecture d'un réseau de neurones à une couche cachée.

Un réseau de neurones f est une succession de transformations mathématiques permettant de passer de l'espace des *features* à l'espace des prédictions :

$$\begin{array}{rccc} f: & \mathbb{R}^{n_x} & \rightarrow & \mathbb{R}^{n_y} \\ & x & \mapsto & \hat{y} \end{array} \tag{1}$$

Le calcul de la sortie $\hat{y}$ à partir de l'entrée x s'appelle la phase *forward* du réseau.

Un réseau de neurones très simple peut par exemple consister en une simple **transformation linéaire**, on aura alors $\hat{y} = f(x) = Wx$ avec W une matrice de taille $n_y \times n_x$. Cette transformation est la couche de base des réseaux de neurones classiques. On utilise d'ailleurs souvent une transformation affine $f(x) = Wx + b$.

Généralement, on fait suivre chaque transformation linéaire d'une **fonction d'activation** qui est une fonction mathématique non-linéaire dont le rôle est double. Tout d'abord, elle permet de rendre la combinaison des transformantions (linéaire + activiation) non-linéaire, permettant ainsi d'appliquer plusieurs transformations de suite. Ensuite, elle permet de choisir un intervalle d'arrivée différent de $\mathbb{R}^{n_y}$, par exemple $\tanh$ permet d'avoir un résultat dans l'intervalle $[-1,1]^{n_y}$.

Les fonctions d'activations les plus fréquentes sont :

— ReLU (Rectified Linear Unit) qui ramène à 0 les valeurs négatives ($\mathrm{ReLU}(x) = \max(0, x)$) ;
— $\tanh$ permettant de se ramener dans l'intervalle $[-1,1]$;

— σ (sigmoide) permettant de se ramener dans l'intervalle $[0, 1]$ ($\sigma(x) = 1/(1 + \exp(x))$) ;
— $\mathrm{SoftMax}$ permettant de se ramener dans l'intervalle $[0, 1]$ et tel que $\sum_i \mathrm{SoftMax}(x)_i = 1$, permettant ainsi d'avoir un vecteur assimilable à une distribution de probabilité ($\mathrm{SoftMax}(x)_i = \exp(x_i)/\sum_j \exp(x_j)$).

Comme indiqué précédemment, on aura géneralement dans un réseau de neurones **une succession de couches linéaires chacune suivie d'une fonction d'activation**. La figure 1 présente un réseau assez simple à une couche cachée (les fonctions d'activation ne sont pas représentées).

Dans le cadre de ce TP, **on va s'intéresser à l'architecture suivante** :

— Une **couche "cachée"** passant de l'entrée de taille n_x à un vecteur h de taille n_h, constituée de :
  — Une **transformation affine** utilisant une **matrice de poids** W_h de taille $n_h \times n_x$ et un vecteur de **biais** b_h de taille n_h ;
  On notera $\tilde{h}$ le vecteur résultant de cette transformation
  — La **fonction d'activation** $\tanh$;
  On notera h le vecteur résultant de cette transformation
— Une **couche de sortie** passant de la représentation cachée de taille n_h à un vecteur de sortie $\hat{y}$ de taille n_y, constituée de :
  — Une **transformation affine** utilisant une **matrice de poids** W_y de taille $n_y \times n_h$ et un vecteur de **biais** b_y de taille n_y ;
  On notera $\tilde{y}$ le vecteur résultant de cette transformation
  — La **fonction d'activation** $\mathrm{SoftMax}$.
  On notera $\hat{y}$ le vecteur résultant de cette transformation

Questions

3. Pourquoi est-il important d'ajouter des fonctions d'activation entre des transformations linéaires ?
4. ★ Quelles sont les tailles n_x, n_h, n_y sur la figure 1 ? En pratique, comment ces tailles sont-elles choisies ?
5. Que représentent les vecteurs $\hat{y}$ et y ? Quelle est la différence entre ces deux quantités ?
6. Pourquoi utiliser une fonction $\mathrm{SoftMax}$ en sortie ?
7. Écrire les équations mathématiques permettant d'effectuer la passe *forward* du réseau de neurones, c'est-à-dire permettant de produire successivement $\tilde{h}$, h, $\tilde{y}$ et $\hat{y}$ à partir de x.

1.3 Fonction de coût

Grâce à la phase *forward*, notre réseau de neurones produit pour l'entrée $x^{(i)}$ une sortie $\hat{y}^{(i)}$. On aimerait savoir à quel point cette sortie diffère de la *target* $y^{(i)}$. Pour cela, on utilise une fonction de coût (*loss*) adaptée à notre problème. La majorité du temps, la fonction de coût globale $\mathcal{L}(X, Y)$ est la moyenne d'une fonction de coût unitaire $\ell(y^{(i)}, \hat{y}^{(i)})$ entre les prédictions et les *targets* :

$$\mathcal{L}(X, Y) = \frac{1}{N} \sum_i \ell\big(y^{(i)}, f(x^{(i)})\big).$$

Il existe de nombreuses fonctions de coût, les deux plus courantes étant :

— l'**entropie croisée** (*cross-entropy*) plutôt adaptée aux problèmes de classification :

$$\ell(y, \hat{y}) = -\sum_i y_i \log \hat{y}_i;$$

— l'**erreur quadratique** (*mean squared error, MSE*) plutôt adaptée aux problèmes de regression :

$$\ell(y, \hat{y}) = ||y - \hat{y}||_2^2 = \sum_i (y_i - \hat{y}_i)^2.$$

Questions

8. Pendant l'apprentissage, on cherche à minimiser la fonction de coût. Pour l'entropie croisée et l'erreur quadratique, comment les $\hat{y}_i$ doivent-ils varier pour faire diminuer la *loss* ?
9. En quoi ces fonctions sont-elles plus adaptées aux problèmes de classification ou de régression ?

1.4 Méthode d'apprentissage

Grâce à la fonction de coût, nous disposons donc d'une mesure de l'erreur de notre reseau de neurones. Nous devons maintenant apprendre ses paramètres (les matrices et vecteurs $W.$ et $b.$ dans notre exemple) afin de minimiser cette erreur sur l'ensemble des exemples d'apprentissage.

Descente de gradient Pour ce faire, nous allons utiliser l'algorithme de descente de gradient. Il consiste à calculer la dérivée de la fonction de coût par rapport à un paramètre w, et à faire un pas dans l'opposé de la direction du gradient, c'est-à-dire à modifier la valeur de w par :

$$w \leftarrow w - \eta \frac{\partial \mathcal{L}(X, Y)}{\partial w}$$

où η est appelé taux d'apprentissage (*learning rate*) et contrôle la vitesse de l'optimisation. Cette étape est repétée un nombre pré-défini d'itérations ou jusqu'à convergence de l'apprentissage.

Le gradient $\frac{\partial \mathcal{L}(X, Y)}{\partial w}$ peut être calculé sur différents ensembles, ce qui correspond à différentes variantes de l'algorithme de descente de gradient :

— **l'ensemble d'apprentissage** (*train*) complet, c'est la **descente de gradient classique** ;
— un petit **sous-ensemble** (généralement quelques dizaines) d'exemples d'apprentissage tirés aléatoirement, c'est la **descente de gradient stochastique par mini-batch** (*stochastic gradient descent, SGD*) ;
— **une seule paire d'exemple** $(x^{(i)}, y^{(i)})$, on remplace donc $\mathcal{L}$ par le coût unitaire ℓ, c'est la **descente de gradient stochastique online**.

Backpropagation Pour calculer le gradient de la sortie par rapport aux poids du réseau, on applique le théorème de dérivation des fonctions composées (*chain rule*) :

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b} \frac{\partial b}{\partial a}.$$

Dans le cas vectoriel, si a, b, c sont des vecteurs, on a pour la dérivée de la composante c_i par rapport à la composante a_k :

$$\frac{\partial c_i}{\partial a_k} = \sum_j \frac{\partial c_i}{\partial b_j} \frac{\partial b_j}{\partial a_k}. \tag{2}$$

Appliqué de façon naïve, le calcul des gradients applique à nouveau la *chain rule* depuis la *loss* pour chaque couche. Cela est très inefficace et il est possible de faire mieux en utilisant le principe de la

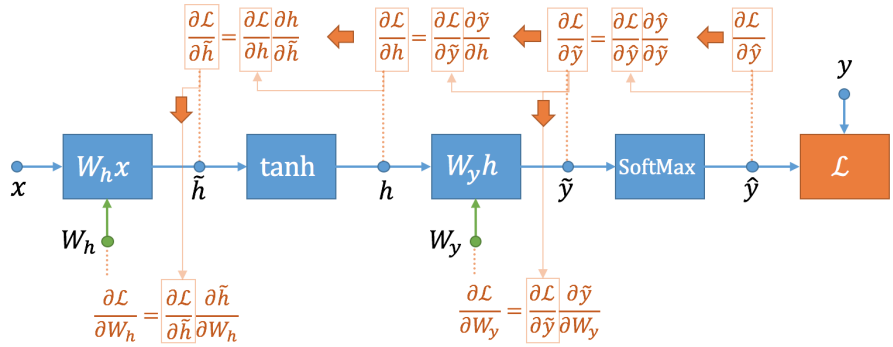

FIGURE 2 – Vue schématique de la *backpropagation* sur un réseau de neurones. On commence par calculer la dérivée du coût $\mathcal{L}$ par rapport à la sortie ($\hat{y}$ sur le schéma) puis on remonte dans le réseau en réutilisant les dérivées calculées précédemment lors du calcul des nouvelles dérivées.
Note : dans le cas vectoriel, il faut rajouter des sommes comme indiqué dans l'équation (2).

backpropagation du gradient, qui consiste à chainer les gradients pour ne parcourir qu'une seule fois le réseau. On visualise schématiquement l'application de la *backpropagation* sur un réseau de neurones sur la figure 2. Le principe est que le gradient calculé par rapport à la sortie d'une couche peut être réutilisé pour calculer les gradients par rapport à l'entrée et aux paramètres de cette même couche : il suffit de le multiplier par un simple gradient local (de la sortie par rapport à l'entrée ou aux paramètres).

Le calcul successif des gradients des différentes couches s'appelle la phase *backward* du réseau.

Questions

10. Quels semblent être les avantages et inconvénients des diverses variantes de descente de gradient entre les versions classique, stochastique sur mini-batch et stochastique online ? Laquelle semble la plus raisonnable à utiliser dans le cas général ?
11. ★ Quelle est l'influence du *learning rate* η sur l'apprentissage ?
12. ★ Comparer la complexité (en fonction du nombre de couches du réseau) du calcul des gradients de la *loss* par rapport aux paramètres, en utilisant l'approche naïve et l'algorithme de *backprop*.
13. Quel critère doit respecter l'architecture du réseau pour permettre la *backpropagation* ?
14. La fonction $\mathrm{SoftMax}$ et la *loss* de *cross-entropy* sont souvent utilisées ensemble et leur gradient est très simple. Montrez que la *loss* se simplifie en :

$$\ell = -\sum_i y_i \tilde{y}_i + \log\left(\sum_i e^{\tilde{y}_i}\right).$$

15. Écrire le gradient de la *loss* (*cross-entropy*) par rapport à la sortie intermédiaire $\tilde{y}$

$$\frac{\partial \ell}{\partial \tilde{y}_i} = \dots \qquad \Rightarrow \qquad \nabla_{\tilde{y}} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_{n_y}} \end{bmatrix} = \dots$$

16. En utilisant la *backpropagation*, écrire le gradient de la *loss* par rapport aux poids de la couche de sortie $\nabla_{W_y} \ell$. Notez que l'écriture de ce gradient utilise $\nabla_{\tilde{y}} \ell$. Faire de même pour $\nabla_{b_y} \ell$.

$$\frac{\partial \ell}{\partial W_{y,ij}} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \dots \qquad \Rightarrow \qquad \nabla_{W_y} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{W}_{y,11}} & \dots & \frac{\partial \ell}{\partial \tilde{W}_{y,1 n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \tilde{W}_{y,n_y 1}} & \dots & \frac{\partial \ell}{\partial \tilde{W}_{y,n_y n_h}} \end{bmatrix} = \dots$$

17. Calculer les autres gradients : $\nabla_{\tilde{h}} \ell$, $\nabla_{W_h} \ell$, $\nabla_{b_h} \ell$.

Partie 2 – Implémentation

On dispose désormais de toutes les équations permettant de faire des prédictions (*forward*), d'évaluer (*loss*) et apprendre (*backward* et descente de gradient) notre modèle.

Nous allons désormais implémenter ce réseau avec PyTorch. L'objectif de cette partie est de se familiariser progressivement avec ce framework.

Cette partie est inspirée du tutoriel de prise en main de PyTorch disponible à l'adresse `http://pytorch.org/tutorials/beginner/pytorch_with_examples.html`. N'hésitez pas a le regarder en parallèle de ce TP pour vous aider à écrire les fonctions demandées.

2.1 Code fourni

Mettre les fichiers et données fournies dans un dossier. Le fichier `tme5.py` fourni la classe `CirclesData` dont voici un exemple d'utilisation :

```
# Chargement de la classe

from tme5 import CirclesData # import de la classe
data = CirclesData() # instancie la classe fournie

# Accès aux données

Xtrain = data.Xtrain # torch.Tensor contenant les entrées du réseau pour
↪ l'apprentissage
print(Xtrain.shape) # affiche la taille des données : torch.Size([200, 2])
N = Xtrain.shape[0] # nombre d'exemples
nx = Xtrain.shape[1] # dimensionalité d'entrée
# données disponibles : data.Xtrain, data.Ytrain, data.Xtest, data.Ytest,
↪ data.Xgrid

# Fonctions d'affichage

data.plot_data() # affiche les points de train et test
Ygrid = forward(params, data.Xgrid) # calcul des predictions Y pour tous les points
↪ de la grille (forward et params non fournis, à coder)
data.plot_data_with_grid(Ygrid) # affichage des points et de la frontière de
↪ décision grâce à la grille
data.plot_loss(loss_train, loss_train, acc_train, acc_test) # affiche les courbes
↪ de loss et accuracy en train et test. Les valeurs à fournir sont des scalaires,
↪ elles sont stockées pour vous, il suffit de passer les nouvelles valeurs à
↪ chaque itération
```

2.2 Forward et backward manuels

On va commencer par coder le réseau de neurones en utilisant simplement les opérations mathématiques de base et en transcrivant donc directement nos équations mathématiques. On utilisera les fonctions que l'on trouve dans le package `torch` : `https://pytorch.org/docs/0.4.1/index.html`.

En torch, les données sont stockées dans des objets de type `torch.Tensor`, équivalent de `numpy.array`. Les fonctions de base de PyTorch retournent des objets de ce type.

On partira du fichier `circlesV1.py` qui contient la structure des fonctions à écrire. Testez votre code progressivement.

1. Écrire la fonction `init_params(nx, nh, ny)` qui initialise les poids d'un réseau à partir des tailles n_x, n_h et n_y et les stocke dans un dictionnaire. Tous les poids seront initialisés selon une loi normale de moyenne 0 et d'écart-type 0.3.
2. Écrire la fonction `forward(params, X)` qui calcule les étapes intermédiaires et la sortie du réseau à partir d'un batch d'entrée `X` de taille $n_{batch} \times n_x$ et des poids stockés dans `params` et les stocke dans un dictionnaire. On retourne le dictionnaire des étapes intermédiaires et la sortie $\hat{Y}$ du réseau.
3. Écrire la fonction `loss_accuracy(Yhat, Y)` qui calcule la fonction de coût et la précision (taux de bonnes prédictions) à partir d'une matrice de sortie $\hat{Y}$ (sortie de `forward`) vis-à-vis d'une matrice de vérité terrain Y de même taille, et retourne la *loss* `L` et la précision `acc`.
   Note : On utilisera la fonction `_, indsY = torch.max(Y, 1)` qui retourne l'indice de la classe prédite (ou à prédire) pour chaque exemple.
4. Écrire la fonction `backward(params, outputs, Y)` qui calcule les gradients de la *loss* par rapport aux paramètres et les stocke dans un dictionnaire.
5. Écrire la fonction `sgd(params, grads, eta)` qui applique une descente de gradient stochastique par mini-batch et met à jour les paramètres du réseau à partir de leurs gradients et du pas d'apprentissage.
6. Écrire l'algorithme global d'apprentissage utilisant ces fonctions.

charger / préparer les données
initialiser le réseau
// Itérer N_{epoch} fois sur le dataset
pour $i = 1..N_{epoch}$
    // Itérer sur les divers batchs du dataset
    pour $j = 1..(N/N_{batch})$
        // Batch d'apprentissage, selon le type de descente de gradient
        $(X_{batch}, Y_{batch}) \leftarrow$ sous-ensemble d'exemples de *train*
        passe *forward* sur le batch
        calcul de la *loss* sur le batch
        passe *backward* sur le batch
        descente de gradient
    calcul et affichage de la *loss* et de la précision sur *train* et *test*
    afficher la frontière de décision avec `plot_data_with_grid`
affichage des courbes de *loss*

2.3 Simplification du backward avec `torch.autograd`

PyTorch fournit un mécanisme de différentiation automatique (*autograd*) disponible sur tous les tenseurs. Par défaut, sur un tenseur, cette fonctionnalité est désactivée. Pour l'activer, il faut mettre `requires_grad` à `True`, soit lors de la création du tenseur (`torch.tensor(data, requires_grad=True)`) soit définir le *flag* ensuite (`x.requires_grad = True`).

Pour utiliser autograd, on fait `loss.backward()`, ici sur le tenseur nommé `loss`, et autograd calcule alors la dérivée de `loss` par rapport à tous les tenseurs qui ont produit le tenseur `loss`. Les gradients de tous les tenseur feuille (ceux qui ne sont pas le résultat de calculs d'autres tenseurs) sont stockés dans l'attribut `grad` de ces tenseurs, par exemple `W.grad` si `W` est un tenseur feuille.

Concrètement, dans votre code, il faut :

1. Faire une copie de votre travail dans un nouveau fichier `circlesV2.py` et travailler dans ce fichier.
2. Activez autograd sur les poids du reseau :

```
# Par exemple si on avait :
params["Wh"] = torch.randn(nh, nx)
# on aura :
params["Wh"] = torch.randn(nh, nx, requires_grad=True)

# Attention, si vous faites :
params["Wh"] = torch.randn(nh, nx) / 10
# Vous devez activer autograd comme cela (pour que ce soit une feuille) :
params["Wh"] = torch.randn(nh, nx) / 10
params["Wh"].requires_grad = True
```

3. Supprimez votre fonction `backward` et, à la place, pour effecteur le calcul des gradients, appelez la méthode `backward` sur l'élément par rapport auquel on veut calculer le gradient, c'est-à-dire la *loss* :

```
L, _ = loss_accuracy(Yhat, Y)
# grads = backward(params, outputs, Y) # supprimé
L.backward() # calcule les gradients
```

4. Modifier votre fonction `sgd` pour utiliser le gradient calculé dans les variables. La signature devient `sgd(params, eta)` :

```
# Par exemple quand on avait :
params['Wy'] -= eta * grads['Wy']
# on aura :
with torch.no_grad(): # Attention a bien utiliser torch.no_grad()
    params['Wy'] -= eta * params['Wy'].grad
    params['Wy'].grad.zero_() # remet l'accumulateur de gradient à zéro
```

5. Testez et faites quelques modifications de votre code pour corriger les problèmes qui pourraient résulter de cette modification. En particulier, pour `plot_data_with_grid` vous devez détacher les prédictions du graphe de calcul autograd en faisant `Ygrid.detach()`.

2.4 Simplification du forward avec les couches `torch.nn`

PyTorch fournit les couches standard des réseaux de neurones sous la forme de modules, permettant de simplifier et rendre plus lisible l'écriture de l'architecture du réseau.

1. Faire une copie de votre travail dans un nouveau fichier `circlesV3.py` et travailler dans ce fichier.
2. Supprimer `init_params` et `forward` et les remplacer par une fonction `init_model(nx, nh, ny)` qui va déclarer l'architecture du model et la *loss* et les retourner.
   Exemple (à adapter au problème) :

```
def init_model(nx, nh, ny):
    model = torch.nn.Sequential(
        torch.nn.Linear(nx, nh),
        torch.nn.ReLU(),
        torch.nn.Linear(nh, ny),
        torch.nn.Sigmoid()
    )
    loss = torch.nn.MSELoss()
    return model, loss
```

3. Remplacer l'appel à la fonction `forward` par un appel au modèle, et le calcul de la *loss* dans `loss_accuracy` par un appel à la fonction de *loss* (nouveau paramètre d'entrée de la fonction `loss_accuracy`) :

```
# Quand on avait
Yhat, outs = forward(params, X)
# on aura
Yhat = model(X)

# Dans loss_accuracy, on a :
L = loss(Yhat, Y)
```

4. La fonction `sgd` prend désormais le modèle en entrée. Pour faire une descente de gradient, on itère sur ses paramètres et on applique notre descente de gradient :

```
def sgd(model, eta):
    with torch.no_grad():
        for param in model.parameters():
            param -= eta * param.grad
        model.zero_grad()
```

Attention, on peut voir que la fonction `torch.nn.CrossEntropyLoss` s'applique sur le vecteur $\tilde{y}$ (donc sans $\mathrm{SoftMax}$). Il faut penser à appliquer le $\mathrm{SoftMax}$ sur la sortie du modèle là où c'est nécéssaire, par exemple sur `Ygrid` en faisant `torch.nn.Softmax()(Ygrid)` pour que l'affichage soit correct.

2.5 Simplification de SGD avec `torch.optim`

Le package `torch.optim` contient de nombreux optimiseurs courants (différentes variantes de l'algorithme de SGD simple que nous utilisons ici).

1. Faire une copie de votre travail dans un nouveau fichier `circlesV4.py` et travailler dans ce fichier.
2. Modifier `init_model` en y ajoutant le *learning rate* en paramètre d'entrée, et faire en sorte qu'il retourne également l'optimiseur voulu :

```
def init_model(nx, nh, ny, eta):
        # contenu précédent
        optim = torch.optim.SGD(model.parameters(), lr=eta)
        return model, loss, optim
```

3. Supprimer votre fonction `sgd`. Faire un appel à `optim.zero_grad()` avant de faire le *backward*, et faire un appel à `optim.step()` après le *backward*.

```
# Si on avait
L.backward()
model = sgd(model, 0.03)
# on aura
optim.zero_grad()
L.backward()
optim.step()
```

2.6 Application à MNIST

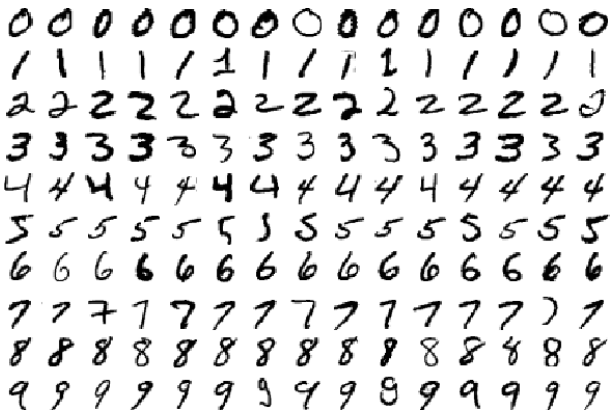

FIGURE 3 – Jeu de données MNIST.

Pour finir, vous pouvez appliquer votre code au jeu de données MNIST. MNIST est un jeu d'images de chiffres manuscrits (voir figure 3), il y a donc 10 classes ($n_y = 10$). Les images font 28×28 pixels, mais elles seront représentées comme un vecteur de 784 valeurs. Pour utiliser ce jeu de données, utilisez la classe `MNISTData` qui fonctionne comme la classe `CirclesData` sans la grille de points et les fonctions d'affichage.