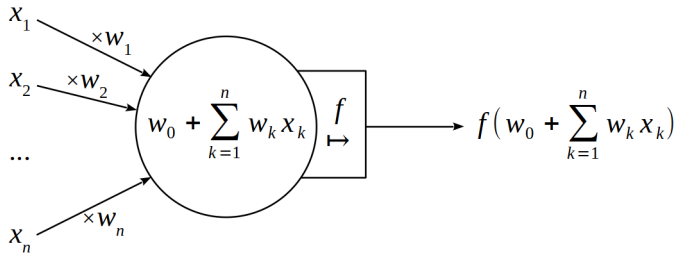
1 Deep Learning

Exposons quelques points-clés nécessaires pour appréhender ce concept, sans justifier les propriétés citées et en illustrant les concepts dans le cadre de la classification d'images numériques matricielles, en version supervisée : les catégories sont connues dès le départ, ainsi qu'un ensemble de correspondance « images $\mapsto$ catégorie » utilisées pour l'apprentissage.

1.1 Neurone formel

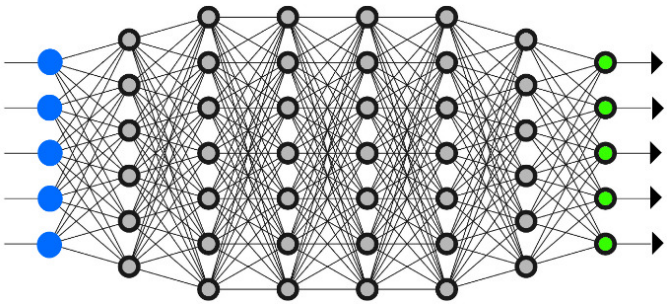
Inspiré par la biologie, ce concept a été introduit en 1 943 par Warren McCulloch et Walter Pitts. Les stimuli en entrée sont remplacés par des valeurs numériques x_k ($k = 1, \ldots, n$), affectées chacune d'un poids w_k. Le neurone en effectue la somme pondérée, en ajoutant un éventuel biais constant w_0. Enfin, il lui applique une fonction dite d'activation (ou de sortie), non linéaire, qui assure un effet de seuil (variation rapide quand la variable franchit une valeur donnée). Parmi celles qui sont couramment utilisées, citons la sigmoïde $\sigma(x) = \frac{1}{1+e^{-x}}$, la tangente hyperbolique $\tanh(x) = \frac{e^x - e^x}{e^x + e^x}$ et la fonction ReLU (*Rectified Linear Unit*) $f(x) = max(0; x)$.



1.2 Réseau de neurones

On constitue des couches successives de neurones, dont les sorties sont connectées aux entrées de ceux de la couche suivante. L'utilisation d'une fonction d'activation non linéaire est nécessaire pour dépasser la capacité d'un unique neurone et pouvoir résoudre des problèmes non linéaires.

Les données à analyser comme les pixels d'une image forment la couche d'entrée. Suivent des couches dites cachées — la profondeur dans « *deep learning* » faisant référence à leur nombre — et enfin la couche de sortie, où chaque neurone est associé à une étiquette dans les problèmes de catégorisation d'images : plus la valeur en sortie du neurone est élevée, plus la catégorie associée est supposée être probable.



1.3 Apprentissage automatique par rétropropagation

Sur le jeu de données fourni (*dataset*) , une partie importante servira à l'apprentissage, mais on en conserve une partie à part, pour pouvoir réaliser des tests sur des images nouvelles afin de détecter le sur-apprentissage (*over-fitting*. Ce phénomène traduit l'utilisation par le réseau de caractéristiques trop spécifiques aux cas utilisés en phase d'apprentissage, ce qui induit une baisse des performances sur de nouvelles entrées.

Ainsi, on initialise le réseau avec des pondérations aléatoires, puis on l'entraîne sur les images étiquetées en ajustant ses pondérations afin d'améliorer l'exactitude en sortie, tout en vérifiant régulièrement que cette dernière ne diminue pas sur les données de test non utilisées dans la phase précédente.

La méthode de **rétropropagation du gradient**, introduite par Paul Werbos en 1 974 et mise au point douze ans plus tard par David Rumilhart, donne leur puissance aux réseaux multicouches [1] :

- Une image est donnée en entrée au réseau.
- Les valeurs des neurones obtenues en sortie sont comparées à celles attendues. On évalue les écarts à l'aide d'une fonction d'erreur qu'on cherche à minimiser.
- On modifie légèrement les pondérations en entrée de chaque neurone de la dernière couche en appliquant une méthode de descente de gradient, de manière à minimiser la fonction d'erreur précédente.
- L'enchaînement des calculs des différentes couches est une fonction composée, on peut donc évaluer l'impact des variations de chacune des pondérations sur les suivantes. On remonte ainsi récursivement jusqu'aux poids des neurones de la première couche cachée, en ajustant au mieux les coefficients à chaque niveau pour minimiser l'erreur en sortie. C'est la phase dite de rétropropagation du gradient.

1. La quatrième vidéo YouTube de 3blue1brown décrit progressivement les calculs en détail, sur un exemple simple, https://frama.link/backprop

- On recommence, pour affiner les réglages, tout en surveillant le taux de reconnaissance sur le jeu de test : si ce dernier tend à diminuer, on termine la phase d'apprentissage pour éviter l'*overfitting*.

En fait, on « nourit « généralement l'algorithme avec un certain nombre d'images avant de mettre à jour les paramètres du modèle. Ainsi, on dit qu'on forme un lot (*batch*), dont la taille est l'un des hyper-paramètres [2] de l'algorithme. Quand on a soumis toutes les données du *training dataset* à l'algorithme, on dit qu'on a terminé une *epoch* (une période en quelque sorte, mais généralement le terme n'est pas traduit dans les écrits francophones). Ainsi, on distingue plusieurs variantes d'algorithmes [3] :

- **SGD, *Stochastic Gradient Descent***, quand une unique image, aléatoire, est soumise avant de mettre à jour les paramètres du modèle (*batch size* de 1).
- **Batch**, quand le lot est constitué de la totalité des données d'entraînement.
- **Mini-batch** pour les cas intermédiaires, où la *batch size* est strictement comprise entre 1 et la taille du *training dataset*. Il y a donc plusieurs mises à jour des paramètres du modèle, une après l'analyse des sorties associées à chaque lot, pour une *epoch* donnée.

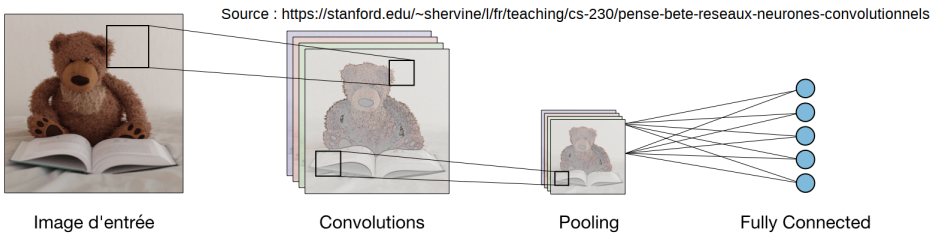
La programmation des calculs associés à la mise à jour des paramètres est vite fastidieuse. Par ailleurs, elle revient systématiquement dans ces réseaux profonds et peut se paralléliser. Plusieurs *frameworks* les ont donc implémentés, tout en tirant partie des capacités de calculs des GPU. L'outil proposé par Facebook Research, `PyTorch`, est arrivé plus tardivement que `TensorFLow` (Google) par exemple. Mais son succès semble grandissant et l'un des intérêts est d'offrir un calcul transparent du gradient : il suffit d'enregistrer au fur et à mesure les opérations effectuées à chaque étape dans la phase directe, des entrées aux sorties. Ensuite, l'ajustement des paramètres est effectué automatiquement à la demande (`autograd`).

2 Réseaux de neurones convolutifs

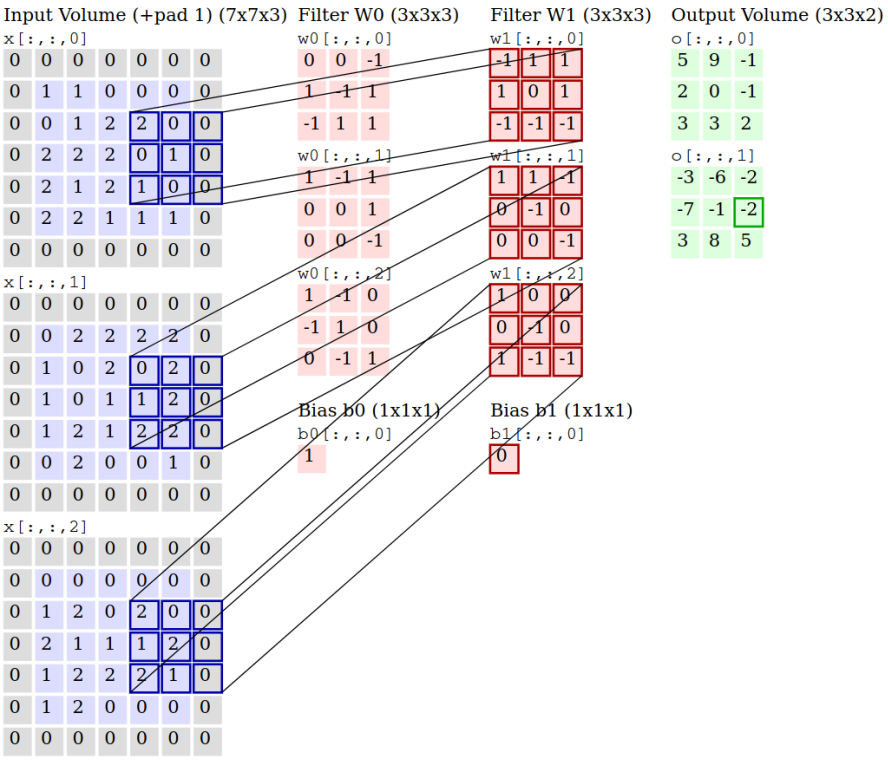
On a occulté jusqu'ici la préparation des données en entrée, spécifique à chaque format et importante pour obtenir de bons résultats. Les CNN,

2. On distingue par ce terme les réglages généralement fixes de l'algorithme, de ceux du modèle qu'il construire et qui évolueront au fur du déroulement.
3. *Cf.* https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size

Convolutional Neural Network ou réseau de neurones convolutifs, constituent une évolution du réseau neuronal multicouche basique, particulièrement adaptée au traitement des images, car il inclut une forme de prétraitement. Le français Yann Le Cun en a été le pionnier.



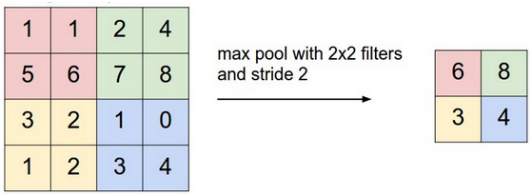
- Les premières couches dites convolutionnelles (*convolutional layers*) donnent leur nom aux CNN. Elles ont pour but de « faire apparaître » des motifs exploitables dans les données de départ. On parle également de *feature map* ou *activation map*.
  En entrée on trouve la grille des pixels d'une image (la « profondeur », nombre de canaux ou *channels* est de 1 pour ces images monochromes, de 3 pour du RGB, etc.) Un filtre (ou *kernel*) constitué d'un tenseur de petite taille balaye l'image (ou les données précédentes) en entrée, en leur appliquant un produit terme à terme, pour produire une couche suivante en sortie. Si la profondeur est supérieure à 1 en entrée, on additionne les valeurs obtenues pour chaque canal. On peut ajouter une éventuelle valeur constante qualifiée de biais. En fait, on peut même traiter ce nombre comme un poids parmi les autres, associé à un pseudo-neurone de valeur constante, ce qui est fait pour se ramener à des produits matriciels, en interne.
  Le pas du décalage du point d'application du filtre sur l'image en entrée, d'une étape à la suivante (*stride*, à 1 par défaut, suivant chaque dimension), ainsi que la taille des marges remplies de 0 ajoutées autour de la matrice d'entrée (*padding*, par défaut à 0) peuvent être précisées.



Si l'on souhaite une profondeur multiple en sortie, on utilise un *kernel* à plusieurs canaux, chacun agissant comme un filtre indépendant qui fournit un niveau de profondeur en sortie. Il y en a deux notés W0 et W1 sur l'illustration précédente [4].

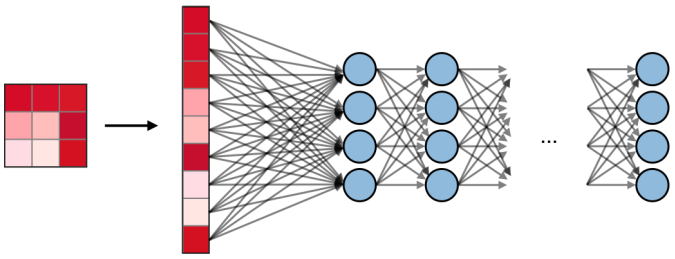
- On prolonge souvent les convolutions par un *pooling*, qui permet de regrouper les valeurs de pixels voisins pour sous-échantillonner et réduire la taille des données à traiter. Traditionnellement, on procède en remplaçant des groupes carrés (ou rectangulaires) de pixels de petites dimensions par leur maximum ou leur moyenne, en balayant là aussi les images ou données en entrée comme pour les convolutions.
  Là encore, on peut choisir des *stride* et *padding* spécifiques ou conserver les valeurs par défaut. [5]

4. Source : image animée sur https://cs231n.github.io/convolutional-networks/
5. Les deux illustrations suivantes proviennent respectivement de https://pathmind.com/wiki/convolutional-network et de http://stanford.edu/~shervine/l/fr/teaching/cs-230/pense-bete-reseaux-neurones-convolutionnels



Notons qu'on peut enchaîner plusieurs étapes de convolutions/*poolings* avant de passer à l'étape suivante.

- On termine par les couches complètement connectées (*fully connected layers*) d'un perceptron classique, avec au final les sorties associées à chacune des catégories à pronostiquer, les 10 chiffres dans le cas qui nous concerne.



2.1 Exemple du MNIST

Le MNIST [6] est une base d'images de chiffres manuscrits associés à leur valeur numérique (60 000 d'entraînement et 10 000 de test), devenue une référence en matière de reconnaissance d'images.

2.1.1 Projet étudié

Dans la suite, on considère `github.com/facebookresearch/pytorch-dp`, sans se préoccuper de préservation de la confidentialité dans un premier temps.

Le projet s'installe classiquement via
`git clone https://github.com/facebookresearch/pytorch-dp.git`,
puis depuis le répertoire `pytorch-dp` créé : `pip3 install -e .` [7]

Le réseau est une instance de la classe `SampleConvNet` (fichier `mnist.py` du dossier `examples`) qui hérite de `nn.Module`, où `nn` est l'alias habituel pour `torch.nn` qui sert de base aux divers réseaux neuronaux dans PyTorch.

6. http://yann.lecun.com/exdb/mnist/
7. Testé sous Debian 10.3 vierge — environnement de bureau et utilitaires systèmes inclus lors de l'installation —, après installation des paquets `git` et `python3-pip` ; utilisation de `pip3` au lieu de `pip` indiqué sur le dépôt GitHub, pour travailler avec Python3. Il a fallu allouer au moins 2,5Go de RAM à la machine virtuelle utilisée, le temps de l'installation

2.1.2 Structure du réseau de neurones

Chacune des entrées sera une image en niveaux de gris de 28×28 pixels (normalisée au préalable). Le constructeur crée deux couches convolutives, suivies d'une couche classique *fully connected* qui aboutit en sortie aux 10 neurones associés à chacun des chiffres à identifier. La méthode `forward()` qui s'exécutera à chaque *epoch*, autrement dit à chaque itération du processus d'apprentissage, définit les transitions d'une couche à la suivante.

- Chaque image du *batch* est transformée par convolution avec un *kernel* (un filtre) de dimension 8×8 et qui parcourt l'image par pas de 2 (*stride*) suivant chacune de ses dimensions, sans *padding*. Il a une profondeur de 16 et produit donc en sortie une couche de 11×11 sur 16 *channels*[8] (*cf.* `self.conv1 = nn.Conv2d(1, 16, 8, 2)` dans le constructeur).

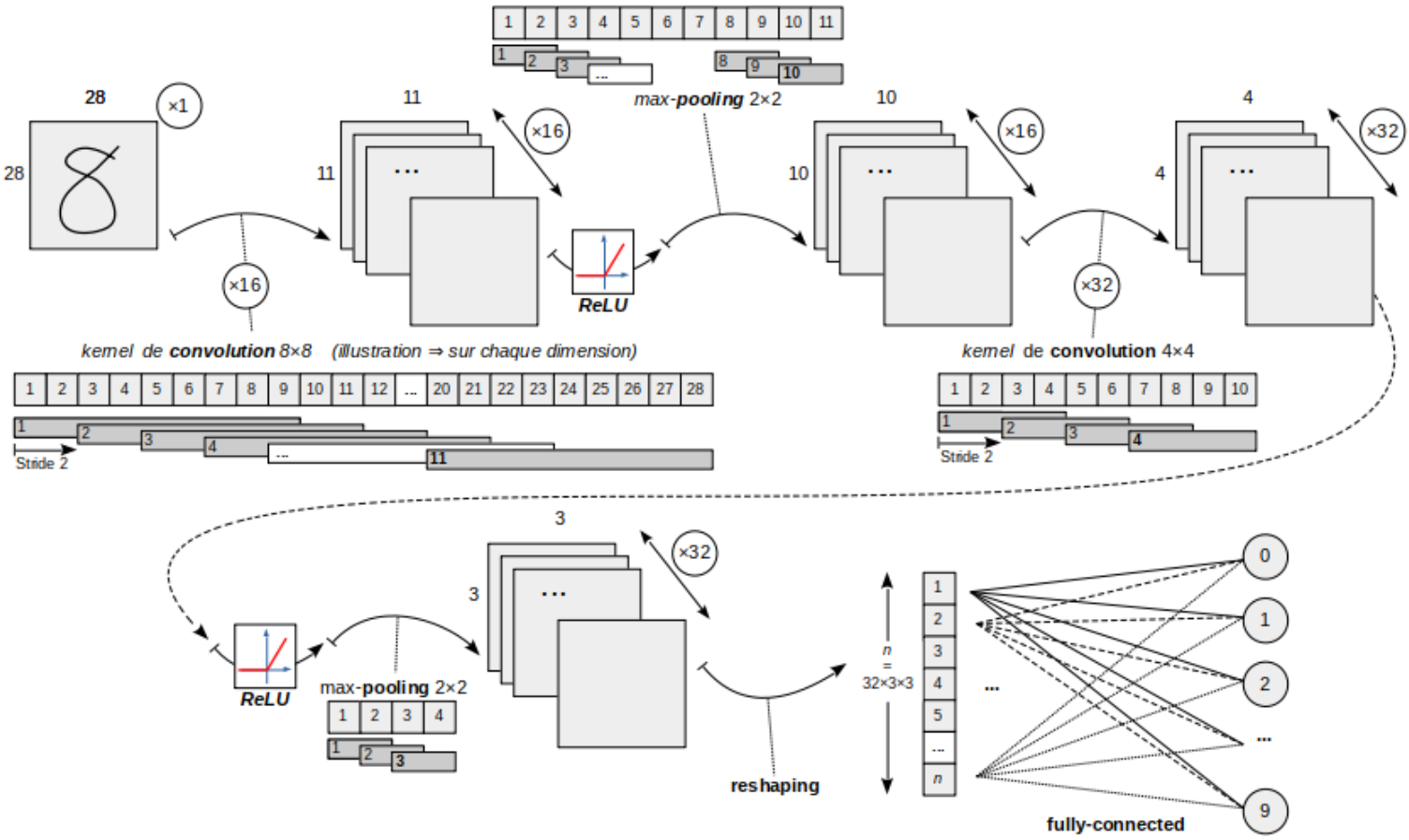
- En sortie, la non-linéarité est assurée par la fonction ReLU (*Rectified Linear Unit*) qui se réduit à l'identité sur l'ensemble des positifs et à la fonction nulle pour les négatifs. Les valeurs sont alors sous-échantillonnées par un « max-*pooling* » de 2×2, d'où les dimensions 10×10 en sortie (`x = F.relu(self.conv1(x))` puis `x = F.max_pool2d(x, 2, 1)` dans la méthode `forward()`).

- La seconde convolution est associée à un *kernel* de 4×4 et de profondeur 32, appliqué par pas de 2 sans *padding*, d'où la dimension 4×4 de la couche suivante (`self.conv2 = nn.Conv2d(16, 32, 4, 2)` dans le constructeur).

- Cette dernière subit à nouveau un max-*pooling* 2×2 sans *padding*, pour arriver à une couche en 3×3 de profondeur 32 (*commandes similaires*).

- Ces données sont remises « à plat » sous forme d'un vecteur de taille $32 \times 3 \times 3$. Ce dernier est enfin relié aux 10 neurones de la couche visible de sortie, en mode *fully connected* comme dans un réseau de neurone basique (`self.fc1 = nn.Linear(32 * 3 * 3, 10)` dans le constructeur, et `x = x.view(-1, 32 * 3 * 3)`, puis `x = self.fc1(x)` dans `forward()`).

8. Voir les explications sur le schéma général. Idem pour les dimensions suivantes



2.1.3 Déroulement du script

Après avoir pris en compte les éventuels paramètres passés lors de l'appel, la fonction `main()` crée deux instances de `torch.utils.data.Dataloader`, des itérables sur les *datasets* d'entraînement et de test, respectivement.

Ensuite, le réseau de neurones est instancié, l'optimisation basée sur une SGD (c'est à ce stade que la méthode est adaptée pour y injecter la garantie de confidentialité différentielle, nous y reviendrons), puis la phase d'entraînement assuré par la fonction `train()` répétée `epochs` fois. Enfin, la fonction `test()` évalue l'état du réseau obtenu. Ce bloc d'instructions est répété éventuellement `n_runs` fois et c'est la moyenne des résultats qui est prise en compte au final et affichée (voire sérialisée sur disque suivant les paramètres).

Plus précisément, `train()`

- Passe le réseau de neurone en « mode entraînement »

- Choisit la fonction d'évaluation des pertes (`CrossEntropyLoss()` ici, soit un `LogSolftmax()` — où le logarithme assure un meilleur étagement des valeurs et évite d'atteindre les limites de la représentation des flottants quand on est trop proche de zéro d'où une meilleure compatibilité avec la SDG — suivi d'une *Negative Log Likehood* `NLLLoss()` qui donne en sortie l'indice d'une des classes).

- Pour chaque *batch* des données d'entraînement : réinitialise les gradients, évalue les pertes sur les exemples, et la applique la rétropropagation du gradient pour mettre à jour les paramètres — la méthode `backward` assure automatiquement ce calcul, c'est l'un des principaux atouts d'un *framework* comme PyTorch.

La fonction `test()`, quant à elle choisit le mode « évalutation », puis procède de manière analogue sur le jeu de test, mais sans « suivi » par PyTorch du gradient, puisqu'il n'y a pas de rétropropagation de gradient à effectuer. En sortie du réseau de neurones, on obtient pour chacune des images testées les probabilités estimées pour chaque chiffre (dans `output=model(data)`) puis lequel est prédit (via `argmax`, résultat dans `pred`). Les statistiques sont alors affichées.

3 Confidentialité différentielle dans `mnist.py`

La prise en compte des opérations garantissant la confidentialité différentielle (DP, *differential privacy*) n'intervient qu'en deux endroits dans `mnist.py`, à savoir dans les blocs des `if not args.disable_dp:`

Le premier est dans la fonction `train()`. C'est un ajout facultatif, pour récupérer puis présenter, en plus de ce qui est affiché en version standard, les paramètres ϵ, δ et α. L'autre constitue la partie incontournable, qui sera à reprendre pour adapter `pytorch-dp` à d'autres *datasets*. Elle se situe dans `main`. Il s'agit de remplacer l'*optimizer* `SGD` standard par sa version DP-compatible.

Repousser plus loin le lien avec la théorie / Rényi-DP et ses coeff. ?
Ou au contraire, commencer au moins par qualifier les paramètres ?

Un objet de la classe `PrivacyEngine` (*cf.* `privacy_engine.py`) est créé, ce qui permet de préciser les paramètres concernant la DP, notamment la liste `alpha` des coefficients α, ici 1,1 ; 1,2 ; 1,3 ; ... ; 10,9 puis 12 ; 13 ; ... ; 63, ou le seuil `max_grad_norm` de *clipping* des normes des gradients. Puis, grâce à la méthode `attach()` d'un tel objet, de remplacer l'*optimizer* du réseau de neurones par sa version DP. En particulier, sa méthode `step()` est modifiée, et il expose `get_privacy_spendt()` qui permet de suivre le budget de confidentialité (*cf. partie incluse dans* ***`train()`***).

détailler le rôle des argu. ? Et donc intérêt / explication théorique avant...

Détailler les autres méthodes : get_etc. et step() surtout.

4 pytorch-dp

Le dossier `torchdp` contient ce qui assure la DP : ajout de bruit, décompte du budget de confidentialité.

- `privacy_engine.py` définit la classe `PrivacyEngine`, au cœur du dispositif de DP. Comme on l'a vu, sa méthode `attach()` permet d'insérer la gestion de la DP dans l'*optimizer* standart passé en paramètre.// Ses méthodes `get_renyi_divergence()` et `get_privacy_spent(targetted_delta)` se chargent des calculs associés en se basant sur les outils de `privacy_analysis.py`.
  détailler leur but // théorie
- `privacy_analysis.py`, qui reprend le travail de Google TensorFlow Privacy, fournit essentiellement les deux fonctions appelées comme on vient de le voir par `PrivacyEngine` (les autres n'étant utiles qu'en interne). **Détailler le rôle des param. ?**
- `dp_model_inspector.py`, qui s'appuie sur la classe `ModelInspector` de `utils.py` et `per_sample_gradient_clip.py` contiennent deux classes utilisées par `privacy_engine.py`. + **Détailler leurs rôles...** +
- `autograd_grad_sample.py`
  À explorer, semble non essentiel pour le fonctionnement

5 Notes hors plan du document pour l'instant

Pour mémoire, liens entres différentes garanties de DP :

ε-DP $\Rightarrow$ (ε,δ)-DP

ε-DP $\Rightarrow$ (∞,ε)-RDP $\Rightarrow$ (α,ε)-RDP $\overset{\forall 0<\delta<1}{\Longrightarrow}$ $\left(\varepsilon + \frac{\ln 1/\delta}{\alpha - 1}, \delta\right)$-DP

et

ε-DP / Laplace

(ε,δ)-DP / Gauss, faible probabilité δ de perte de confidentialité

(α,ε)-RDP (Rényi) moment accountant, Gauss / random subsets