Implémentation de la DP dans pytorch-dp

Exemple avec le MNIST

1 Ajout du bruit pour l'entraînement

1.1 Organisation générale de l'exemple du MNIST

Dans le fichier filemnist.py,

- La fonction principale `main()` commence par **parser les arguments** de la ligne de commande, notamment
  `-batch-size` (par défaut 64),
  `-epochs` (14, nombre de répétitions des cycles complets),
  `-lr` (1.0, *learning rate*),
  `-sigma` (1.0, futur écart-type σ du bruit gaussien)
  `-max-per-sample-grad_norm` (1.0, seuil du *clipping*),
  `-delta` (10^{-5}, paramètre δ de la (ε, δ)-DP)

- Puis (l. 234 *numérotation dans « ma » version commentée*) on crée les instances `train_loader` et `test_loader` (en indiquant leur taille de *batch* à chacun[1]. de `torch.utils.data.Dataloader`. Les données seront téléchargées si besoin.

- Ensuite vient la phase d'entraînement (l. 278). On crée l'instance `model` de `SampleConvNet`, définie dans le même fichier comme sous-classe de `pytorch.nn.Module`. Puis `optimizer`, de classe `torch.optim.SGD`, sans considération de DP à ce stade.

- La DP est alors implantée dans `model` :

  – l. 293, on crée `privacy_engine`, instance de `PrivacyEngine` avec en arguments
  `model`

1. Le ***batch*** est l'ensemble des données utilisées avant la mise à jour des paramètres du modèle lors de l'entraînement, alors que l'***epoch*** correspond au parcours de l'ensemble des données d'entraînement. On parle respectivement de *Stochastic / batch / mini-batch Gradient Descent* quand la taille du *batch* vaut 1 / la taille du *training set* / entre les deux. https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/

`train_loader`
`alphas` une liste des coefficients α (*cf.* Rényi-DP) valant `[1.1, 1.2, 1.3, ..., 10.9, 12, 13, ..., 63]`
`noise_multiplier` (σ, passé en argument du script)
`max_grad_norm` (seuil de *clipping*, idem).

– L'appel (l. 304) à sa méthode `.attach(optimizer)` transforme sa phase d'entraînement en version DP — détaillé plus bas.

- La fonction `train()` [2] définie dans le même fichier est appelée `epoch` fois, pour l'entraînement. Puis la valeur de retour de `test()`, fonction définie également dans ce fichier, est ajoutée à la liste `run_results` dont la moyenne indiquera la justesse (*accuracy*) du modèle...

1.2 De la version standard à la DP

La classe `PrivacyEngine` est définie dans le fichier `privacy_engine.py`.

- Les éléments locaux suivants sont importés :
  - le module `privacy_analysis as tf_privacy`, repris du code TensorFLow Privacy de Google, gérant la RDP du mécanisme gaussien — détaillé plus bas.
  - `PerSampleGradientClipper` du module `per_sample_gradient_clip` qui prend en charge le *clipping* des gradients, couche par couche.
  - `DPModelInspector` du module `dp_model_inspector` pour vérifier que le modèle est compatible avec la transformation en version DP.
- Le constructeur de `PrivacyEngine` reprend comme attributs les valeurs passées en paramètres, donc `module` pour qui recevra l'objet `model` instance de `SampleConvNet`. Il en crée aussi de nouveaux, dont :
  `sample_rate = dataloader.batch_size / len(dataloader.dataset)`,
  `validator = DPModelInspector()`,
  `clipper = PerSampleGradientClipper(self.module, self.max_grad_norm)`
- La méthode `attach(optimizer)` permet de remplacer à chaud (*monkey patching*) l'*optimizer* d'origine par sa version DP, en redéclarant sa méthode `step`.
- La méthode `step()` de `PrivacyEngine` appelle son homonyme de `clipper` pour majorer les normes des gradients de chaque couche. Cette dernière est donc définie dans `per_sample_gradient_clip.py` (l. 95) :

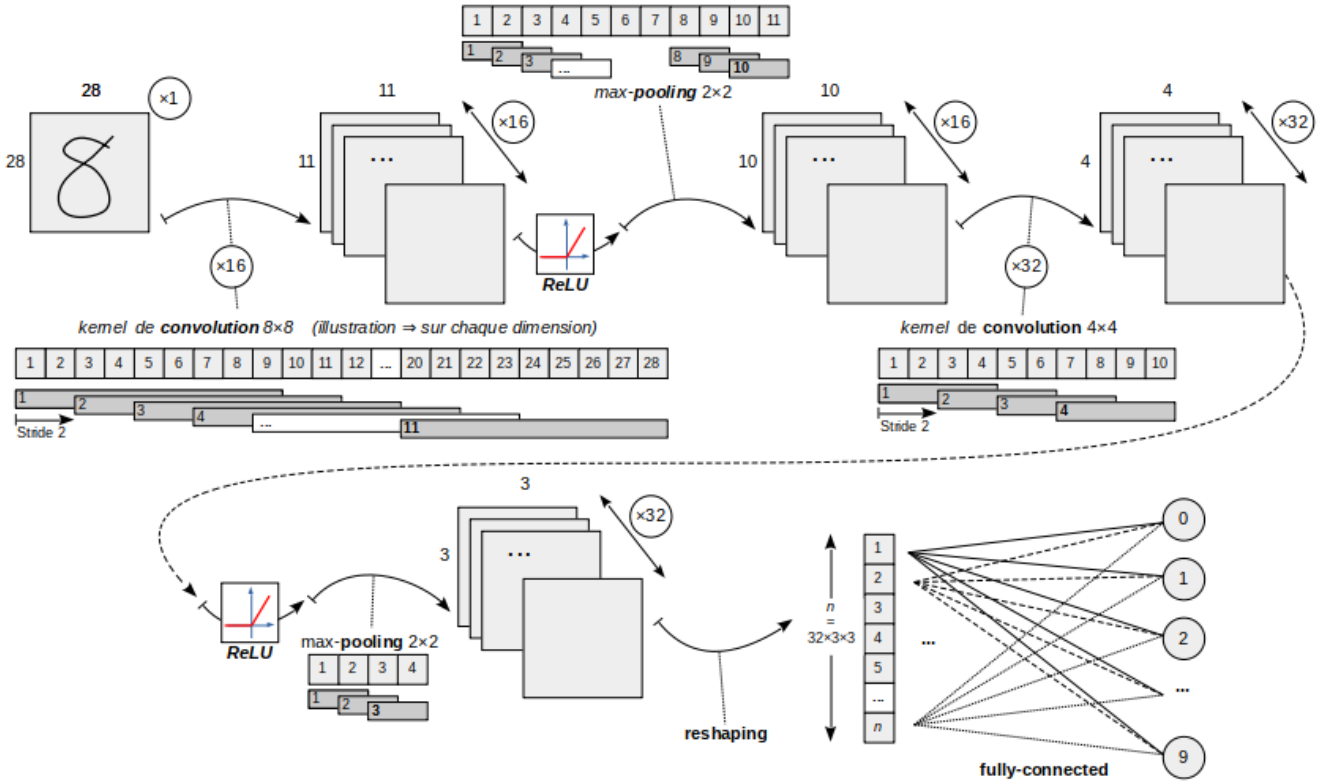
2. Cette fonction fait également intervenir la DP, au niveau calcul du budget de confidentialité, on en parle plus loin.

  - Appel de `autograd_grad_sample.compute_grad_sample(module)` qui crée un attribut `grad_sample`, tenseur contenant les gradients calculés sur les données fournies d'entraînement. Les couches de type `Linear` et `Conv2d` sont prises en compte. **Détailler ?**
  - Puis `clip_per_sample_grad_norm_(module, max_norm)` (fonction interne de `per_sample_gradient_clip.py`) assure le *clipping* des gradients, et enfin le calcul de leur moyennes par batch [3], enregistrées dans l'attribut `grad`.

- Ensuite, pour chaque élément du réseau, le **bruit gaussien** est ajouté [4]. Il est centré et d'écart-type `noise_multiplier * max_grad_norm` (soit σC en notation de l'algorithme 1 de l'article arXiv :1607.00133 *Deep Learning with Differential Privacy*). Plus précisément, on crée un tenseur aléatoire `noise` de même *shape* que celui qui représente le gradient associé à la couche traitée, puis on ajoute à ce dernier `noise` divisé par la taille du batch. En effet, la moyenne de chaque gradient a déjà été effectuée, on distribue donc dans la formule indiquée dans l'algorithme le facteur $\frac{1}{L}$ sur chacun des termes avant de les additionner. Les gradients bruités sont à nouveau stockés dans l'attribut `grad`.

Les *shapes* des éléments « paramètres » itérés permettent de comprendre leur nature quand on les met en parallèle avec l'architecture du réseau neuronal : [16, 1, 8, 8] (kernel de convolution pour 16 couches en sortie, 1 niveau en entrée, en dimensions 8×8), puis [16] (ReLU sur chaque couche ; rien concernant le *max-pooling*), [32, 16, 4, 4] (convolution, idem), [32] (ReLU), puis [10, 288] (couche complètetement connectée, des $3 \times 3 \times 32$ valeurs réalignées vers les 10 catégories en sortie).

3. Elle utilise l'itérateur `torch.nn.Module.parameters()` pour traiter chacun des éléments du réseau neuronal.
4. *Idem.*



2 Optimisation du budget de DP

La fonction `train()` du fichier `mnist.py` qui affiche les performances appelle la méthode `get_privacy_spent(targeted_delta)` de `privacy_analysis.py` qui renvoie les valeurs optimales de ε et du α associé [en parcourant les points (α, ε) sur la « courbe de budget » de confidentialité *cf.* article « Rényi-DP », pour la liste des ordres α qu'on se donne].

Cette méthode utilise la fonction homonyme de `privacy_analysis.py` à qui l'on passe la listes des α utilisables, le tenseur `rdp` [qui semble présenter les ε courant de la (α, ε)-RDP cumulée **pas sûr ! vérifier…**] et le δ attendu. Elle ajoute (l. 209 ; terme à terme, pour chacune des données incluses dans ce tenseur) à `rdp` la valeur $\frac{\ln(1/\delta)}{\alpha-1}$, conformément à la propriété 3 du IV. de l'article « Rényi DP » : du ε de la (α, ε)-RDP on déduit à partir du δ attendu le $\varepsilon' = \varepsilon + \frac{\ln(1/\delta)}{\alpha-1}$ pour la (ε', δ)-DP alors garantie. Elle détermine enfin la valeur minimale de ε, avant de renvoyer le couple $(\varepsilon; \alpha)$ associé.

+ lien avec ce qui est affiché ; + comment cumuler /budget ?

La valeur de `rdp` dans ce contexte de SGD est calculée par la méthode `get_renyi_divergence()` de `PrivacyEngine` qui utilise directement la fonction `compute_rdp()` (l. 169) de `privacy_analysis.py`. Celle-ci multiplie

par le nombre d'étapes (*de batches ?* **à vérifier**) les valeurs renvoyées par la fonction « locale » `_compute_rdp(q, sigma, alpha)` où `q` représente le taux d'échantillonnage.

Calculs à justifier (vérifier la sensibilité 1) Grossièrement :

- Cas q=1 / corollaire 3 VI de l'article « RDP »
- Sinon, dans arXiv :1908.10530 "Rényi-DP of Sampled Gaussian Mecanism", conséquence en section 3 du théorème 4 et corollaire 7. pour justifier le $\varepsilon \leqslant \frac{\ln(A_\alpha)}{\alpha-1}$ suffisant pour avoir la (ε, α)-RDP.
  Puis pour le calcul de A_α soit entier soit en fraction, même article 3.3 page 11.