Implémentation de la DP dans pytorch-dp

Exemple avec le MNIST

Table des matières

1 Étude de l'exemple du MNIST

1.1 Organisation générale

On omet volontairement certains détails, d'autant que le projet, très actif, est régulièrement modifié (*ce fût le cas notamment pour la structure même du modèle de cet exemple, durant son étude de notre part*).

. Rappeler ici le principe du SGM ?

Le script `examples/mnist.py` est exécutable directement en ligne de commande (sa fonction `main()` est alors appelée). Le réseau neuronal convolutif est modélisé par la classe `SampleConvNet` qui hérite de `nn.Module`.

Le répertoire `torchdp/` fournit les outils assurant le respect de la confidentialité différentielle ainsi que sa gestion. Ils interviennent ici à deux endroits : dans `main()` pour rendre le modèle conforme à cette exigence, puis dans la méthode `train()` de `SampleConvNet`, qui traite la phase d'apprentissage, en affichant au fur et à mesure notamment la consommation du « budget de confidentialité ».

1.2 Transformation du modèle en version DP

1.2.1 Utilisation classique de pytorch

Rappelons d'abord le déroulement de l'entraînement d'un modèle classique, ici d'apprentissage supervisé d'un réseau neuronal convolutif visant à catégoriser des images. **Principes généraux d'un CNN à voir avant...**

On peut instancier un objet `model` d'une classe héritant de `nn.Module` (on désigne généralement `torch.nn` par l'alias `nn`) pour décrire la structure du modèle. Le constructeur définit les différentes couches qui le constituent.

```
def __init__(self):
    super(MyNet, self).__init__()
    self.conv1 = nn.Conv2d(6, 16, 5)
    self.fc1 = nn.Linear(16*5*5, 120) # ...
```

La méthode `forward()` sera appliquée comme son nom l'indique lors de la phase de propagation directe, de calcul des sorties à partir de données d'entraînement, à comparer avec les valeurs attendues associées, toutes deux issues du *training dataset*. On indique donc dans sa définition les différentes couches du modèle, les fonctions d'activation et autres réorganisations « géométriques » des tenseurs, avant de renvoyer le résultat de cette séquence de transformations.

```
# import torch.nn.functional as F
def forward(self, x):
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x, 2, 2)
    x = F.relu(self.fc1(x)) # ...
    return x
```

On précise une fonction de coût (*loss function*) pour quantifier l'erreur entre les sorties et les valeurs attendues. Un objet `optimizer` est créé — les principaux étant disponibles « clé en main » dans le module `torch.optim` — qui gèrera l'optimisation et les calculs de gradients associés grâce à sa méthode `step()`.

```
# import torch.optim as optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(params, lr=0.1, momentum=0.0)
```

Reste alors à itérer la phase d'entraînement : remise à zéro des gradients, calcul des sorties avec les paramètres en cours puis du coût associé, rétropropagation du gradient pour mettre à jour les paramètres (et suivi des performances).

```python
# `trainloader` contient le training set
losses = []
for inputs, labels in trainloader:
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(ouputs, labels)
    loss.backward()     # Thanks to PyTorch,
    optimizer.step()    #  everthing is 'automatic' !
    losses.append(loss.item())
    # if ... : print(np.mean(losses))
```

On peut construire une phase de test de manière analogue, tout en évitant les calculs de gradients, donc sans utilisation de l'`optimizer`.

```python
with torch.no_grad():
    for inputs, labels in testloader:
        outputs = model(inputs)
        loss += criterion(ouputs, labels).item()
        # Index of the max log-probability
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(label.view_as(pred)).sum().item()
loss /= len(test_loader.dataset)
```

1.2.2 Adaptation en version DP

Dans `torchdp/privacy_engine.py` on définit une classe `PrivacyEngine`, construite notamment à partir d'une instance `model` de `SampleConvNet`, des hyper-paramètres (`batch_size`, `noise_multiplier` égal au `sigma` passé en argument, `max_grad_norm`...) et de la liste des moments `alphas` qui seront utilisés pour estimer de manière optimale, grâce à la Rényi-DP, la consommation ε du budget de confidentialité lors de la phase d'entraînement.

Cette classe se charge d'**adapter l'objet `model`, afin de le rendre** garant de la confidentialité différentielle (on le dira **DP**, pour *Differentialy Private*). Ceci est assuré en définissant une méthode `step()` spécifique, visant à remplacer son homonyme standard de l'`optimizer` choisi. Puis une autre `attach()` qui transforme à chaud ce dernier (« *monkey patch* »), en y substituant la version modifiée de `step()` qui le rendra DP.

* * *

Mais au final, pour pouvoir mettre concrètement en œuvre ces mécanismes, il faut savoir calibrer le bruit ajouté par le SGM, l'ajuster aux paramètres de DP qu'on s'est accordés. Cela dépendra également de la structure du réseau neuronal, ainsi que du nombre d'itérations en phase d'entraînement. Étudions la partie consacrée à ces calculs.

1.3 Comptabilité de la confidentialité

Parcourons les scripts du dossier `torchdp/`, pour découvrir la cascade d'appels des fonctions concernées.

1.3.1 `privacy_engine.py`

La classe `PrivacyEngine` évoquée précédemment expose deux méthodes utilisées pour gérer le budget de confidentialité.

- `get_privacy_spent()` est appelée par `train()` de notre exemple de `mnist.py` . Elle renvoie les valeurs (ε, α) du budget ε consommé pour chaque *epoch* et l'ordre du moment optimal associé (ce α étant celui de la Rényi-DP utilisée pour les calculs), à partir de la valeur de δ qu'on s'est accordée en termes de (ε, δ)-DP.

- La méthode `get_renyi_divergence(self)` est utilisée par la précédente, à qui elle renvoie le coût en termes de Rényi-DP-ε pour une occurrence du *Sampled Gaussian Mechanism*. Toutes deux présentent des interfaces pratiques, mais le cœur des calculs est déporté dans des fonctions du module `tf_privacy`, qui est un alias pour le fichier suivant.

1.3.2 `privacy_analysis.py`

C'est en effet le moteur des calculs de DP, qui exploite les résultats théoriques récents. C'est une reprise, quasiment à l'identique, des outils de calculs du projet **TensorFlow Privacy** de Google. Il expose deux fonctions :

- `get_privacy_spent(orders, rdp, delta)` renvoie le couple de valeurs Rényi-DP (ε, α) optimal : elle détermine celui pour lequel le ε' qu'on peut en déduire en termes de (ε', δ)-DP est minimal, parmi les valeurs de α envisagées et passées par la liste `orders` et celles associées de ε données dans `rdp`.
  La **justification** du calcul associé à chaque couple (ε', α) est donnée par la propriété suivante[1] qui permet de déterminer la « courbe de budget », à savoir les couples (ε, α) pour la Rényi-DP :

Pour tout $0 < \delta < 1$,
si un mécanisme aléatoire est (ε, α)-Rényi DP,
alors il est également (ε', δ)-DP pour $\varepsilon' = \varepsilon + \frac{\ln 1/\delta}{\alpha - 1}$.

---

1. Proposition 3 dans « Rényi Differential Privacy », Ilya Mironov, août 2017 [arXiv :1702.07476] qu'on notera `[MIR17]` dans la suite

- `compute_rdp(q, noise_multiplier, steps, orders)` fournit justement les valeurs de ε (ici `rdp`), en fonction des ordres des moments α envisagés indiqués par `orders`, du taux d'échantillonnage `q` (*détailler, lot / batch, ici ? en note de bas de page ?*), de l'écart-type `noise_multiplier` du bruit gaussien ajouté et du nombre `steps` de répétitions. Le calcul est en réalité sous-traité par la fonction « privée » `_compute_rdp()`.

`_compute_rdp(q, sigma, alpha)` renvoie le coût ε de la RDP à l'ordre `alpha` du mécanisme gaussien (SGM, *Sampled Gaussian Mechanism*) de taux d'échantillonnage `q` et d'écart-type `sigma`.

Le calcul est direct dans le cas où `q` vaut `1.0`, grâce à la propriété suivante[2] — qui montre une « courbe de budget RDP » qui est une droite passant par l'origine puisque $\varepsilon = \frac{1}{2\sigma^2}\alpha$:

Si une fonction f a une sensibilité de 1,
alors le SGM d'écart-type σ appliqué à f est
$(\alpha, \alpha/(2\sigma^2))$-Rényi DP pour tout $\alpha > 1$.

Quand `0 < q < 1`, `_compute_log_a(q, sigma, alpha)` renvoie une valeur $\ln(A_\alpha)$ calculée en fonction de $\alpha > 1$, telle que[3] :

Si f a une sensibilité de 1 et si $\varepsilon \geqslant \frac{\ln(A_\alpha)}{\alpha-1}$,
alors le SGM appliqué à f (avec les paramètres `q`, `sigma`) est (α, ε)-RDP.

En pratique `_compute_rdp()` choisit bien sûr la valeur $\varepsilon = \frac{\ln(A_\alpha)}{\alpha-1}$, pour minimiser le budget consommé.

Le calcul de $\ln(A_\alpha)$ est traité différemment[4] selon que α est entier (calcul direct assuré par `_compute_log_a_int()`) ou non (approximation par une série convergente via `_compute_log_a_frac()`).

Pour l'instant, j'ai un souci avec la sensibilité 1 !
En effet, cela ne me semble pas garanti dans l'algo. 1 du « Abadi 2016 » utilisé dans pytorch-dp, si le seuil de clipping du gradient est supérieur à 1...

- pour $q = 1$: pour une fonction f de sensibilité S_f, le SGM sur f d'écart-type $S_f\sigma$ est (ε, δ)-DP si $\varepsilon < 1$ et $\delta > \frac{4}{5}\exp(-\sigma^2\varepsilon^2/2)$, *cf.* Dwork Roth « Algorithmic Fundation of DP » en 2014 ou 2013), mais

2. Proposition 7 & Corollary 3, dans `[MIR17]`.
3. ***ref. erreur inégalité def Aalpha*****
4. *cf.* `[MIR19]`, partie 3.3.

on parle alors de DP classique, pas de RDP. Or c'est bien la RDP qui justifie nos calculs « serrés » ici...

- Pour l'autre cas avec A_α : je n'ai pas totalement saisi comment Mironov déduit (dans [MIR17]) coroll. 3 de de la prop. 7. Si l'idée que je devine vaguement est correcte (je me fie à ce qu'il fait juste avant au coroll. 2 pour Laplace, « since the Laplace mecanism is additive... »), on devrait pouvoir passer de la Prop. 7 avec un μ égal à la sensibilité de f à la RDP pour $\varepsilon = \alpha/(2\mu^2\sigma^2)$, dans une sorte de Coroll. 3bis... À approfondir !

Fin de la parenthèse...