

Introduction à Jobman

Dumitru Erhan
IFT 6266

Situation typique I

- On veut entraîner un MLP
- Hyperparamètres:
 - # couches cachées (1,2,3) - 3 valeurs
 - # unités par couche (200,500,1000) - 3 valeurs
 - taux d'apprentissage ($10^{-1} \dots 10^{-5}$) - 5 valeurs
 - paramètre de régularisation ($10^{-3} \dots 10^{-7}$) - 4 valeurs
 - taille de minibatch (1,20,100) - 3 valeurs
- $3 \times 3 \times 5 \times 4 \times 3 = 540$ combinaisons possibles
- Pas réaliste de faire ça à la main et en série.

Situation typique II

- On a décidé d'utiliser un script qui exécute les expériences sur le cluster
- On garde les résultats dans un fichier texte, ou chaque ligne contient:

```
nb_couches nb_unites taux L2 mb_size train_err valid_err test_err
```

- Problèmes avec cet approche:
 - Plusieurs processus écrivent en même temps
 - Pas évident comment exéc. les expériences qu'on n'a pas encore roulé.
 - Chacun a son propre façon de sauvegarder les résultats
 - Difficile de partager ou d'explorer de paramètres avec qqn d'autre.
 - Pas claire comment fusionner des fichiers.

Jobman

- Facilite l'exécution de plusieurs tâches en parallèle, en imposant un interface commune
- Avec SQL, on résout la plupart des problèmes décrites avant
- Facile a utiliser.

Petit exemple

- ```
def my_experiment(state, channel):
 error = do_AI(state.param1, state.param2)
 state.error = error
 return channel.COMPLETE
```

- **Exécution de l'expérience:**

```
jobman cmdline path.to.my_experiment param1=string1 param2=3
```

- **Jobman mets** `state.param1=string1` **et** `state.param2=3`
- `my_experiment` **mets** `state.error = error`.
- Le contenu du dictionnaire `state` **sauvegardé** (avant **et** après)
- `path.to` **doit être un module dans le** PYTHONPATH

# Jobman + SQL

- Puissance de Jobman = utilisation d'une base de données
- Idée:
  - on génère plusieurs expériences (comb. de param.)
  - on les met dans une base de données
  - avec une autre commande, on demande à la base de données de nous donner une expérience à faire et d'exécuter la commande.
  - quand l'expérience est complète, on sauvegarde les résultats dans la BD.
- **Tout ça** est fait avec
  - `jobman sqlschedule(s) et`
  - `jobman sql`

# jobman sqlschedule(s)

- `jobman sqlschedule postgres://user:pass@host/dbname/tablename path.to.my_experiment p1=s1 p2=1`
- `jobman sqlschedules postgres://user:pass@host/dbname/tablename path.to.my_experiment p1={{s1,s2}} p2={{1,3,5}}`
- produit **toutes les combinaisons** possibles des valeurs de p1 et p2
- `jobman sqlschedules postgres://user:pass@host/dbname/tablename path.to.my_experiment p1={{s1,s2}} p3={{1.0,2.0}} params.conf`
- va lire `params.conf` (qui contient un `param=value` par ligne) **et** generer les combinaisons possibles.
- permet de garder les paramètres qu'on varie *moins* dans un fichier.

# jobman sqlschedule(s)

- `jobman sqlschedule postgres://user:pass@host/dbname/  
tablename path.to.my_experiment p1="blah" p2=1.0`

- si `tablename` n'existe pas, ça va créer la table

- La commande produit:

```
ADDING {'p1': 'blah', p2=1.0, 'jobman.experiment':
'path.to.my_experiment', 'jobman.status': 0,
'jobman.sql.priority': 1.0}
```

- `status: 0` - pas commencée, `2` - complétée
- `experiment`: le nom du module.fonction qu'on appelle
- `sql.priority`: la priorité d'exécution

# jobman sql

- Une fois que les expériences “programmées” dans la base de données, c’est facile à les exécuter:

```
jobman sql 'postgres://user:pass@host/dbname/table_name' .
```

- `jobman sql` va exécuter une expérience dans `dbname/table_name` qui n’a pas encore été exécutée et qui a la plus grande priorité.
- Comme il s’agit d’une base de données, cette opération est **atomique** - donc même si on lance 540 processus en parallèle qui font appel à “`jobman sql`”, ils auront **tous** des expériences différentes à exécuter.
- Il nous faut juste un moyen d’exécuter 540 processus en parallèle (et 540 processeurs!)
- ... `dbdispatch`!

# dbdispatch

- dbdispatch est l'outil pour lancer des expériences sur la grappe de LISA/DIRO
- `dbdispatch --condor --repeat_jobs=3 jobman sql 'postgres://user:pass@host/dbname/tablename' .`
- va lancer “`jobman sql 'postgres://user:pass@host/dbname/tablename' .`” sur le back-end condor
- `--repeat_jobs=3` va lancer cette expérience **3 fois**
- donc a la fin, 3 combinaisons d'hyperparametres seront choisis, selon la priorité, de `dbname/tablename`, et passée au 3 copies du module.experience définie dans la même table.
- dbdispatch a (beaucoup, beaucoup) plus d'options :)

# jobman sqlview

- Une fois les expériences finis, c'est utile d'accéder au résultats.
- La base de données créée par Jobman est un peu compliqué a lire

```
jobman sqlview postgres://user:pass@host/dbname/
mlp_dumi dumi_mlp_view
```

- va créer un “view” SQL qui contient seulement les affaires qui nous intéressent.

# View the view

- `psql -h gershwin -U ift6266h10 -d ift6266h10_sandbox_db`

`ift6266h10_sandbox_db=> select`

`batchsize,bestvalidationloss,learningrate,minutestrained,nhidden,niter,testscore from  
dumi_mlp_view where jobman_status=2 order by bestvalidationloss;`

| batchsize | bestvalidationloss | learningrate | minutestrained | nhidden | niter | testscore |
|-----------|--------------------|--------------|----------------|---------|-------|-----------|
| 20        | 0.0202             | 0.05         | 22.0346666667  | 100     | 50    | 0.0216    |
| 20        | 0.0204             | 0.05         | 28.1765        | 150     | 50    | 0.019     |
| 20        | 0.0208             | 0.1          | 25.8041666667  | 150     | 50    | 0.0214    |
| 20        | 0.0226             | 0.1          | 26.4415        | 100     | 50    | 0.0227    |
| 20        | 0.0255             | 0.02         | 9.53816666667  | 50      | 50    | 0.0276    |
| 20        | 0.0265             | 0.05         | 19.237         | 50      | 50    | 0.0284    |
| 20        | 0.0267             | 0.1          | 10.2448333333  | 50      | 50    | 0.028     |
| 20        | 0.0267             | 0.01         | 23.9046666667  | 100     | 50    | 0.0272    |
| 20        | 0.0288             | 0.01         | 19.6711666667  | 50      | 50    | 0.0303    |
| 20        | 0.1173             | 0.01         | 0.135          | 30      | 1     | 0.1187    |
| 20        | 0.1266             | 0.01         | 0.101833333333 | 20      | 1     | 0.1352    |
| 20        | 0.1491             | 0.01         | 0.0655         | 10      | 1     | 0.1563    |

(12 rows)

# Jobman pour IFT6266

- On a crée deux bases pour le cours:
  - db1: ift6266h10\_sandbox\_db
  - db2: ift6266h10\_db
- pour le tester:

```
psql -h gershwin -U ift6266h10 -d
ift6266h10_sandbox_db
```
- ift6266h10\_sandbox\_db : un "sandbox"
- ift6266h10\_db pour les "vrais" expériences.

# Choses pas couvertes

- La structure de la base de données Jobman
- L'interface Python
- L'utilisation sans SQL
- Methodes avancées pour passer des paramètres.
- `channel.switch()`,  
`channel.save_and_switch()`

# Liens

- Jobman: <http://deeplearning.net/software/jobman> (incluant comment installer)
- Contient un “all-in-one” pour explorer les hyper-parametres d’un MLP a une couche
- Page wiki sur Assembla (“Jobman au DIRO”)

# Résumé

- Jobman - très bon outil pour mieux organiser les expériences qu'on fait en Machine Learning
- Pour que le partage marche bien, il faudra s'entendre (sur les noms de states etc)