Rapport sur le projet de programmation quantique

TensorFlow Quantum & hybrid-quantum classical neural network benchmark

(Programming Project - TensorFlow Quantum) TensorFlow Quantum (TFQ) is a new quantum machine learning library developed by Google for rapid prototyping of hybrid quantum-classical machine learning models. Write a tutorial in the form of a Jupyter notebook that explains the basic functionalities of TFQ. Implement and benchmark an hybrid-quantum classical neural network with it.

Lors de ce projet, nous allons nous intéresser à l'une des nombreuses applications de l'informatique quantique : son utilisation dans la mise en place de réseaux neuronaux.

Le réseau que nous avons mis en place est un réseau de type MLP (perceptron multicouche) est un réseau a propagation directe dans lequel l'état de chacun des neurones d'une couche affecte l'état de tous neurones de la prochaine couche, jusqu'à aboutir à la couche qui compose la sortie de notre réseau.

Afin d'arriver à obtenir des résultats cohérents, une fois créé, le réseau est entrainé et va modifier les poids synaptiques de chaque neurone après chaque essai (ou « groupe » d'essai) afin de tendre vers le résultat voulu.

Pour ce faire, on met principalement en place deux outils : la fonction Hinge Loss, et la descente du gradient.

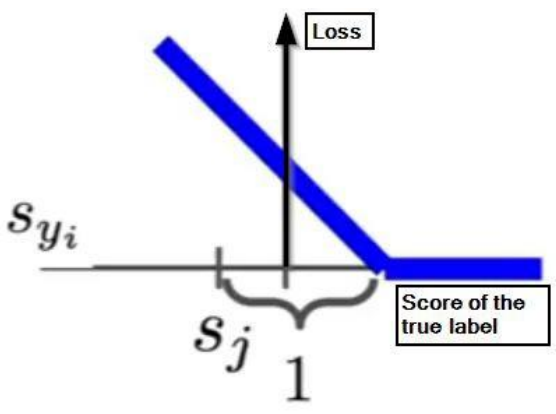
La fonction Hinge Loss

En machine learning, la fonction Hinge Loss est une « fonction de perte » utilisée pour la formation des classificateurs, et en maximisant la marge. Il existe différentes fonctions de perte (Hinge, Logistique, Exponentielle, Tangeante…).
Pour la fonction de Hinge, et dans une expérience où la sortie est : $t = \pm 1$ et le classificateur *y*, la perte de Hinge de la prédiction de *y* est défini comme : $l(y) = \max(0, 1 - t \times y)$.
On peut aussi l'écrire de la manière suivante : $L_H(x, y, f) = max\{0, 1 - y \cdot f(x)\}, \ y \in \{-1; 1\}$

Une tâche de prédiction peut être considérée comme une simple tâche d'optimisation. Le modèle tente d'optimiser ses performances en prédisant la valeur correcte. Pour ce faire, le modèle doit pouvoir mesurer ce que l'on appelle la «perte», qui dépend du problème.

Intuitivement, cette perte permet de vérifier si le score correct est une "marge" meilleure que les autres scores

Exemple :

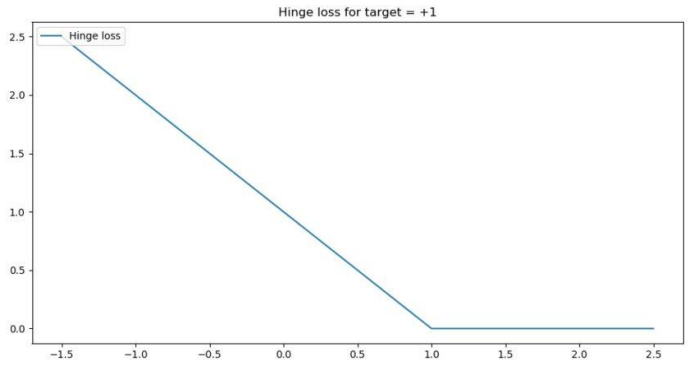


Ici, 1 est le "tampon de sécurité". Cela signifie que nous ne considérons « pas d'erreur » si :

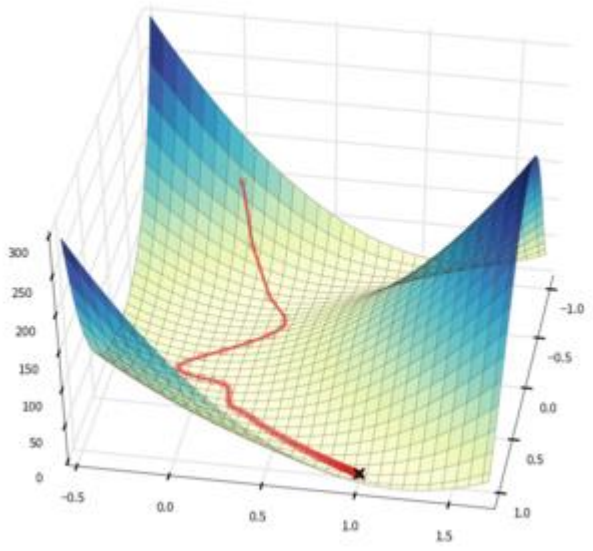
$S_{yi} \geq S_j + 1$ *(le score de la vraie étiquette est plus grand que le score de la mauvaise étiquette - avec un tampon).*

De manière plus simple, imaginons que la variable visée est $t = \pm 1$ (la valeur visée pour la prédiction) et y (la prédiction)

- Si $t = y = 1$, $max(0\,, 1 - 1) = max\,(0{,}0) = 0$, donc parfait. Score du vrai label
- Si *t* est très différent de *y* , disons *t=1* mais *y=-1,* alors la perte est maximale : $max(0\,, 1 - (-1)) = max(0{,}2) = 2$
- Si y n'est pas correct mais assez proche de t, disons t=1 mais y=0.9, alors la perte vaut $max(0{,}0.1) = 0.1$.



La descente de gradient



Ce qui fait que notre circuit quantique apprend de ses erreurs, c'est la descente de gradient. Pour notre algorithme, nous utilisons une fonction coût de type Hinge Loss (mais personnalisée). Le but de la descente de gradient est de minimiser la valeur de cette fonction coût en faisant varier les paramètres du circuit. Dans notre exemple, nous avons 32 paramètres qui peuvent varier. On ajuste donc à chaque fois ces paramètres pour diminuer la valeur de la fonction coût.

Pour minimiser la fonction coût, nous avons besoin du gradient (vecteur contenant les dérivées partielles) et du taux d'apprentissage. Si la dérivée partielle d'un paramètre est positive, alors on diminue la valeur en lui soustrayant le taux d'apprentissage. A l'inverse, si la dérivée partielle est négative, on ajoute à notre paramètre le taux d'apprentissage. En appliquant ces étapes un grand nombre de fois, on s'approche normalement des paramètres optimaux pour la détection de chiffres par le circuit.

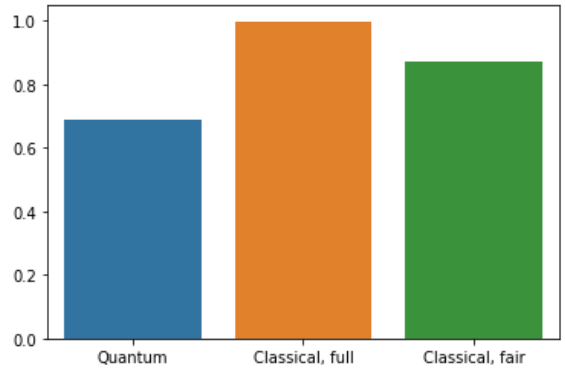
Résultats du circuit

Pour un entrainement du circuit avec 500 données du set, on obtient la précision suivante :

```
qnn_accuracy = qnn_results[1]
cnn_accuracy = cnn_results[1]
fair_nn_accuracy = fair_nn_results[1]

sns.barplot(["Quantum", "Classical, full", "Classical, fair"],
            [qnn_accuracy, cnn_accuracy, fair_nn_accuracy])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc0954bcda0>
```

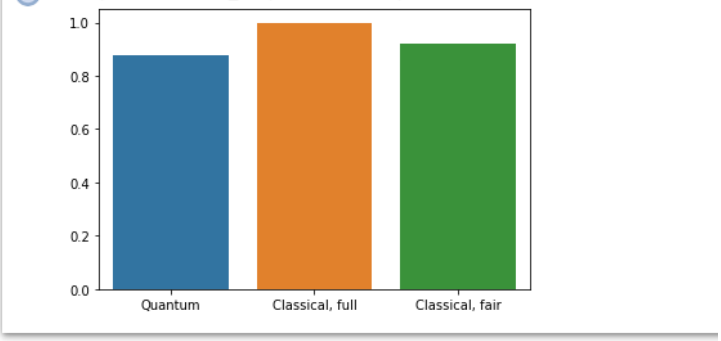


La performance du circuit quantique est moins bonne que celle du réseau de neurone classique. Ensuite, nous avons testé avec un entrainement sur 12000 images. Nous obtenons les résultats suivants :

```
qnn_accuracy = qnn_results[1]
cnn_accuracy = cnn_results[1]
fair_nn_accuracy = fair_nn_results[1]

sns.barplot(["Quantum", "Classical, full", "Classical, fair"],
            [qnn_accuracy, cnn_accuracy, fair_nn_accuracy])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2319248240>
```



Nous constatons que même sur un grand nombre de données d'entrainement, l'algorithme classique est plus précis que le quantique. Cela confirme le fait que pour traiter des données classiques, les algorithmes classiques restent les meilleurs.

Pour finir, nous pouvons nous intéresser à d'autres utilisations, plus actuelles, des réseaux neuronaux hybrides.

De la même manière que les réseaux neuronaux classiques, les réseaux neuronaux hybrides ont vocation à évoluer afin de pouvoir traiter des données plus complexes, en plus grand nombre, plus vite que les réseaux dits à perceptron multicouche. Une évolution possible des réseaux quantique est, comme pour les réseaux classiques : les réseaux de neurones quantiques convolutifs.

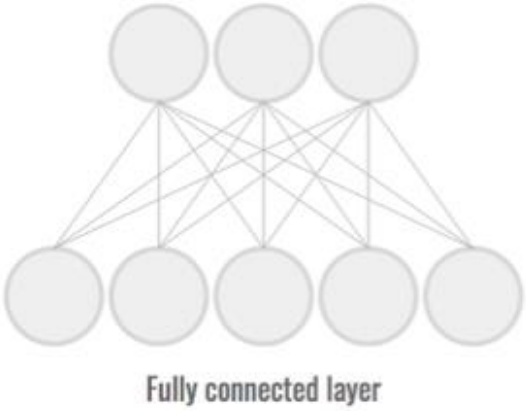
Le besoin d'évoluer et de changer la structure des réseaux neuronaux vient du fait que les MLP (réseaux à perceptron multicouche) peuvent prendre beaucoup de temps à être entrainés, surtout lorsque les données en entrée deviennent de plus en plus complexe, c'est un problème de scalabilité.

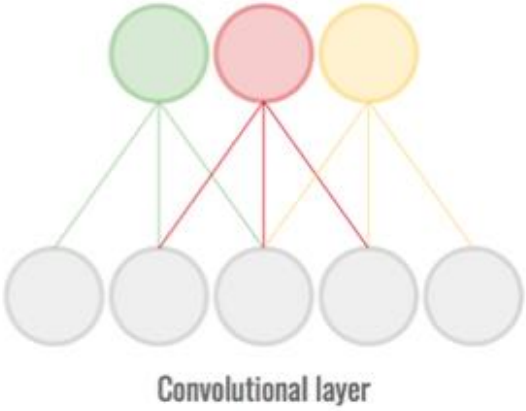
Les réseaux convolutifs sont plus adaptés que les MLP pour l'analyse d'image notamment car :

- ils permettent de reconnaitre des objets/paternes même s'ils sont translatés dans l'image

- ils nécessitent moins de paramètres (poids) pour fonctionner correctement que les MLP (à volume de données en entrée égal).

Afin d'être plus efficaces que les MLP, les QCCNN (Quantum-Classical Convolutional Neural Networks) utilisent la cohérence spatiale locale de l'entrée afin de diminuer les poids requis pour traiter les données. Les neurones de la couche de convolution ne sont pas reliés à tous les neurones de la couche suivante contrairement dans les réseaux MLP :





Ces réseaux convolutifs commencent déjà à être mis en place de manière hybride et commencent à surpasser des réseaux convolutifs classiques sur certaines tâches notamment en analyse et classification d'image (voir : Hybrid Quantum-Classical Convolutional Neural Networks).

Vous pouvez retrouver le code de réseaux neuronaux ainsi que le notebook Jupyter expliquant les fonctionnalités de TensorFlow Quantum sur :
https://github.com/Gyrok/Rendu_QC_CPE_SKAF_OVIGNE_VERDIER_KLAAS