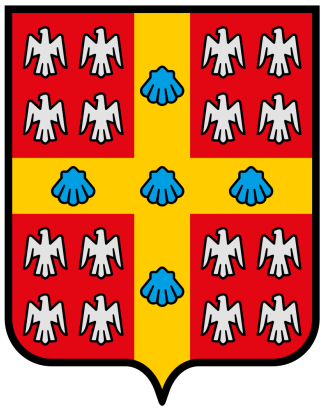

UNIVERSITÉ
LAVAL

Compte-rendu TP2

Date: 2024/04/19

par

Jossua MIGNON
NI: 537 188 246
Tanguy RELO
NI: 537 196 900

GLO-7030

Université Laval

Contents

1 Question 1

L'objectif de cette question est de découvrir, à travers la classification d'espèces d'oiseaux à partir du jeu de données CUB-200, les effets du fine-tuning et de la normalisation.

Pour pouvoir observer les différences de performance induites par le fine-tuning et la normalisation, 8 tests seront réalisés avec une même architecture et les mêmes hyper-paramètres (en excluant ceux liés au fine-tuning et à la normalisation). Les tests sont les suivants:

- Avec une normalisation à partir des données du jeu de données CUB-200 :
  - a) une initialisation aléatoire par défaut
  - b) un modèle pré-entraîné, mais en gelant (freeze) tous les paramètres de convolution
  - c) un modèle pré-entraîné, mais en gelant uniquement les paramètres dans "layer1"
  - d) un modèle pré-entraîné sans geler de paramètres
- Avec une normalisation à partir des données du jeu de données ImageNet :
  - a) une initialisation aléatoire par défaut
  - b) un modèle pré-entraîné, mais en gelant (freeze) tous les paramètres de convolution
  - c) un modèle pré-entraîné, mais en gelant uniquement les paramètres dans "layer1"
  - d) un modèle pré-entraîné sans geler de paramètres

Les moyennes et écart-types obtenus pour le jeu de données CUB-200 sont les suivants :

	R	G	B
Moyenne	0.4859	0.4996	0.4318
Écart-type	0.2250	0.2203	0.2585

Table 1: Moyennes et écart-types obtenus pour le jeu de données CUB-200

Voici les résultats obtenus :

	a	b	c	d
train	0.0211	0.7816	0.7561	0.7993
test	0.0083	0.4433	0.441	0.4413

Table 2: Résultats avec une normalisation à partir des données du jeu de données ImageNet

	a	b	c	d
train	0.0521	0.7938	0.7975	0.7543
test	0.0187	0.4487	0.4437	0.4323

Table 3: Résultats avec une normalisation à partir des données du jeu de données CUB-200

On remarque que pour un nombre limité d'epochs et de données, le fait de partir d'un modèle pré-entraîné et de geler des couches nous permet d'atteindre de meilleures performances. Cependant, on pourrait émettre l'hypothèse qu'avec un nombre plus important de données et un budget d'entraînement plus grand, les architectures non gelées seraient plus performantes car elles ont plus de degrés de liberté.

On peut également voir que l'utilisation d'un pré-entraînement aide à trouver plus rapidement de meilleures solutions.

Enfin, la normalisation avec le jeu d'entraînement a permis de gagner en performance pour les modèles a, b et c.

2 Question 2

2.1 Q2.1 / Q2.2 / Q2.3

Voir code ***q2-RNN.py***

2.2 Q2.4)

Notre architecture principale se compose d'une **couche d'embedding** reliée à une **couche de RNN causal comportant 4 sous-couches**. Cette dernière est suivie par **une couche linéaire**, une moyenne est effectuée sur la sortie.

Concernant l'entraînement, nous utilisons un **optimizer de type SGD** avec un learning rate $lr = 0.8$ et un $momentum = 0.8$. De plus, nous utilisons un horaire d'entraînement de type **ReduceLROnPlateau** monitoré sur la loss en test. Cet horaire d'entraînement applique un facteur de réduction de 0.01 au learning rate. Sachant que nous avons décidé de garder un **nombre d'epochs de 5**, nous avons réglé la patience à 0 pour que le changement se fasse dès qu'on note une stagnation.

Par la suite, nous ferons référence à nos trois architectures, classique, padding on batch et auto-padding, par les termes de: "Q2.1)", "Q2.2)" et "Q2.3)". Les résultats obtenus sont disponibles aux figures 1,2 et 3.

On note qu'en moyenne Q2.2) est plus rapide que Q2.1) par époch, ce qui était attendu de l'utilisation de "padding on batch". En termes de résultats, on observe que les exactitudes sur le dataset de test sont très proche: **23.49** pour Q2.1) et **23.73** pour Q2.2). Nous aurions pu nous attendre à une différence plus notable. En effet, en faisant un padding sur le batch, les différences de longueurs sont localement plus petites que sur l'ensemble complet de données. Ceci devrait diminuer l'erreur sur le batch. Du point de vue de la convergence lors de l'entraînement, on observe que les modèles n'évoluent que très peu. On notera tout de même que sur les 5 époch, la perte de Q2.1) ne cesse de diminuer, là où la perte de Q2.2) diminue sur l'époch 2 avant de "stagner".

Le modèle Q2.3) est bien plus lent en entraînement que ces prédécesseurs avec **une moyenne de temps par époch de 70,914 sec contre 51,496 sec pour Q2.1) et 38,418 sec pour Q2.2)**. Q2.3) a beau utiliser le "padding on batch" comme pour Q2.2), ce dernier est ralenti notamment par la nécessité de trier dans l'ordre croissant les données du batch. L'exactitude sur test pour Q2.3) est meilleur que pour nos deux modèles antérieures. Nous passons d'une exactitude comprise entre 23 et 24% pour Q2.1) et Q2.2) à **35,06%, soit un gain de 11,56%** avec les mêmes paramètres d'entraînement. Toutefois, on observe que la convergence est plus lente pour ce modèle. En effet, sur les deux premiers, nous atteignons rapidement un palier à partir duquel nous ne pouvons plus descendre, et cela, malgré l'horaire d'entraînement. Dans le cas de Q2.3), le modèle gagne sur les 3 premières époch une quantité significative d'exactitude, il converge donc plus lentement.

```
Epoch: 1/5 Train steps: 1696 54.81s loss: 1.954410 acc: 23.138352
Epoch: 2/5 Train steps: 1696 49.20s loss: 1.876694 acc: 23.232817
Epoch 00002: reducing learning rate of group 0 to 8.0000e-03.
Epoch: 3/5 Train steps: 1696 49.81s loss: 1.849764 acc: 24.485282
Epoch 00003: reducing learning rate of group 0 to 8.0000e-05.
Epoch: 4/5 Train steps: 1696 51.31s loss: 1.847334 acc: 24.493576
Epoch 00004: reducing learning rate of group 0 to 8.0000e-07.
Epoch: 5/5 Train steps: 1696 52.35s loss: 1.845280 acc: 24.629514
Epoch 00005: reducing learning rate of group 0 to 8.0000e-09.
Test steps: 424 3.97s test_loss: 1.784580 test_acc: 23.491355
INFO:root:1 - Loss: 1.7845797055407535  Acc:23.491355473984573
```

Figure 1: Q2.1 résultats

```
Epoch: 1/5 Train steps: 1696 40.43s loss: 2.020679 acc: 20.637108
Epoch: 2/5 Train steps: 1696 39.68s loss: 2.169240 acc: 17.523916
Epoch 00002: reducing learning rate of group 0 to 8.0000e-03.
Epoch: 3/5 Train steps: 1696 36.37s loss: 2.037425 acc: 20.898844
Epoch 00003: reducing learning rate of group 0 to 8.0000e-05.
Epoch: 4/5 Train steps: 1696 37.50s loss: 2.037470 acc: 20.910364
Epoch 00004: reducing learning rate of group 0 to 8.0000e-07.
Epoch: 5/5 Train steps: 1696 38.11s loss: 2.037207 acc: 20.934326
Epoch 00005: reducing learning rate of group 0 to 8.0000e-09.
Test steps: 424 3.66s test_loss: 1.881100 test_acc: 23.729126
INFO:root:2 - Loss: 1.881099626168696   Acc:23.729125964468867
```

Figure 2: Q2.2 résultats

```
Epoch: 1/5 Train steps: 1696 53.45s loss: 1.686277 acc: 29.328793
Epoch: 2/5 Train steps: 1696 58.67s loss: 1.612123 acc: 32.602805
Epoch 00002: reducing learning rate of group 0 to 8.0000e-03.
Epoch: 3/5 Train steps: 1696 1m15.89s loss: 1.587118 acc: 35.132159
Epoch 00003: reducing learning rate of group 0 to 8.0000e-05.
Epoch: 4/5 Train steps: 1696 1m23.76s loss: 1.586671 acc: 35.221555
Epoch 00004: reducing learning rate of group 0 to 8.0000e-07.
Epoch: 5/5 Train steps: 1696 1m22.80s loss: 1.586264 acc: 35.257036
Epoch 00005: reducing learning rate of group 0 to 8.0000e-09.
Test steps: 424 10.26s test_loss: 1.586110 test_acc: 35.059166
INFO:root:3 - Loss: 1.5861100607002359  Acc:35.059166145837985
```

Figure 3: Q2.3 résultats

3 Question 3

Nom d'équipe : Equipe44_Mignon_Relo

L'objectif de cette question est de réaliser un classificateur d'images de ville par réseau CNN.

3.1 Architecture

Nous avons décidé de nous servir de l'architecture EfficientNetV2 qui est une amélioration des architectures EfficientNet classiques. Cette dernière est plus performante et présente moins de paramètres à entraînée ce qui est un avantage sachant que nous avons peu de données. EfficientNetV2 se décompose en 3 modèles : s, m et l. Nous nous sommes intéressés ici au modèle EfficientNetV2-L.

3.2 Paramètres d'entraînements

Pour l'entraînement, nous avons fait le choix de partir du modèle pré entraîné sur ImageNet1k et de geler les paramètres de toute les couches sauf de *feature 7.5*, *feature 7.6*, *feature 8* et *classifier* afin de réaliser un fine-tuning.

Après recherche, nous avons fait le choix d'entraîner notre modèle à l'aide de **l'optimiseur ADAM** car ce dernier semblait le plus optimal pour cette architecture. Nous avons initialisé le **learning rate à 0.01**, valeur petite, mais qui s'explique par le fait que nous ne faisons qu'un fine-tuning.

Afin d'assurer un entraînement efficace, nous utilisons l'horaire d'entraînement **ReduceLROnPlateau** qui permet de diminuer la valeur du learning rate lorsque la perte stagne. Le paramètre de **patience est égal à 3**. Afin de tirer le meilleur de l'entraînement de notre modèle, nous avons mis en place un **early stopping avec une patience de 6**.

L'entraînement s'effectue avec un maximum de **100 epochs**, avec **80%** du jeu dédiés à **l'entraînement** du modèle et **20%** dédiés à la partie **test**.

Enfin, la quantité de donnée limitée nous a poussé à effectuer de la **data-augmentation** sue ces dernières en plus des méthodes de redimensionnement et de la normalisation utilisée sur ImageNet. Nous avons fait le choix d'utiliser les méthodes suivantes: **v2.RandomHorizontalFlip(), v2.ColorJitter() et v2.GaussianBlur()**.

3.3 Résultats

La métrique d'évaluation utilisée est l'exactidute (**Accuracy**), soit le ratio de ce qui est prédit vrai (True positive et False positive) sur le tout.

Lors de nos expérimentations nous avons pu obtenir:

- Accuracy en train: **90.39 %**
- Accuracy en test: **73.33 %**
- Accuracy sur le jeu de la compétion: **76.33 %**

Ces résultats sont encourageant et démontre les performances de notre modèle. Toutefois, nous pensons pouvoir obtenir de meilleurs résultats en ajoutant du **weight decay** pour régularitser, ou encore en pré-entraînant notre modèle sur le jeu de données suivant:
https://www.kaggle.com/datasets/stelath/city-street-view-dataset, qui est composé de 50k images de villes.