Description d'images

IFT780 Réseaux neuronaux - Rapport de projet

Victor Taillieu	Eliott Thomas	Luca Vaio
21 154 689	21 164 874	21 154 698

11 avril 2022

Table des matières

Introduction **2**

Prétraitement des données **3**
1.1 Transformation des images 3
1.2 Tokenisation 3

Architectures **5**
2.1 Encodeur : extracteur de caractéristiques 5
2.2 Décodeur 6
2.2.1 Réseau récurrent 6
2.2.2 Attention 8
2.3 Encodeur-Décodeur 9

Gestion de l'entraînement **10**
3.1 Checkpointing 10
3.2 Fonction de perte 10

Exécution du code **12**
4.1 Interface en ligne de commande 12
4.2 Recherche d'hyperparamètres 13

Analyse des résultats **14**
5.1 Évaluation de l'entraînement 14
5.2 Performance du modèle 16

Conclusion **18**

Appendix **19**

Introduction

Dans le cadre du cours IFT780 (Réseaux neuronaux), nous avions à choisir entre un troisième travail pratique ou un projet. Notre équipe de trois a décidé de se pencher sur un sujet de projet : la description d'images ou *Image captioning*. L'objectif de ce projet est de tester trois architectures différentes afin de comprendre quel type de structure permet de bien performer sur ce cas d'usage. Une recherche d'hyperparamètres est également attendue dans le but d'améliorer la performance du modèle obtenu.

Les données que nous avons utilisées proviennent de la base de données Flickr8k accessible sur Kaggle [1]. Les images qui la composent sont diverses et plusieurs sources [2] citent ce jeu de données comme référence pour la description d'images. Il en existe d'autres souvent plus volumineux comme Flickr30k ou MS-COCO mais considérant les ressources limitées à notre disposition, ce dataset nous a semblé suffisant pour notre projet. Les 8000 images contenues dans cette base de données sont chacune associées à 5 descriptions.

La première partie de ce rapport consistera à présenter le prétraitement des données et le chargement de celles-ci. Dans un second temps, nous détaillerons les architectures que nous avons employées pour ce problème. Puis, nous expliquerons la manière dont nous gérons l'entraînement et la recherche d'hyperparamètres. Finalement, nous exposerons et analyserons nos résultats.

1. https://www.kaggle.com/datasets/adityajn105/flickr8k
2. https://arxiv.org/pdf/1806.06422.pdf

Prétraitement des données

1.1 Transformation des images

Plusieurs étapes de transformation sont nécessaires pour les images avant de pouvoir les donner en entrée au modèle. Premièrement, les images n'ont pas la même taille. Or, la couche d'entrée du réseau requiert des dimensions fixes. De plus, certains extracteurs de caractéristiques demandent une taille d'image spécifique en entrée comme Inception v3 [1]. Nous avons donc défini un `Transforms` de *torchvision* ayant pour rôle de redimensionner les images à la taille voulue et de les convertir en tenseur tout en normalisant les valeurs des pixels entre 0 et 1.

Pour ne pas surcharger la mémoire en chargeant toutes les images d'un coup, on utilise une classe `DatasetLoader` qui implémente `Dataset` de PyTorch. Uniquement la liste des chemins des images est stockée afin de savoir où les trouver. Cette classe permet de charger les éléments au fur et à mesure des batchs avec `DataLoader` de PyTorch. Lorsqu'une image est chargée en mémoire, celle-ci est transformée selon le processus ci-dessus. Ensuite, elle est renormalisée en suivant les valeurs fournies par ImageNet. L'image est finalement prête pour être donnée en entrée au modèle.

1.2 Tokenisation

En parallèle, les descriptions ou *captions* passent également par une pipeline de prétraitement. Nous utilisons le package spaCy pour réaliser la tokenisation des phrases de description. Ce processus consiste à assigner un indice à chaque mot. Des caractères spéciaux sont à spécifier afin que le décodeur fonctionne correctement :

— <START> : balise de début de description
— <END> : balise de fin de description
— <UNK> : balise représentant un mot inconnu
— <PAD> : balise de *padding*

Ces balises sont respectivement associées aux indices 0, 1, 2 et 3. Contrairement aux images, les descriptions brutes sont toutes chargées en mémoire et stockées dans un attribut du `DatasetLoader`. Cela est nécessaire pour l'étape de la construction du vocabulaire.

1. Documentation PyTorch de Inception v3

Celle-ci a lieu juste après avoir instancié le `DatasetLoader`. Pour ce faire, toutes les descriptions sont parcourues. Chaque mot est ajouté au vocabulaire si sa fréquence totale d'apparition atteint un seuil fixé. Dans ce cas, un indice lui est attribué dans le vocabulaire.

Une fois le vocabulaire construit, la transformation des descriptions se fait au moment du chargement de l'image correspondante. Ce traitement consiste à tokeniser le contenu de la description à l'aide du vocabulaire dans un premier temps puis a y concaténer les balises de début et de fin.

Tout ce processus de prétraitement des images et des descriptions est ensuite pris en charge par le `DataLoader` qui permet de charger les données par batch. De plus, il a également le rôle d'ajouter le *padding* nécessaire de manière à ce que les descriptions du batch fassent toutes la même taille. Pour cela, la balise <PAD> est utilisée en tant que remplissage pour atteindre la taille de la description la plus longue.

Architectures

2.1 Encodeur : extracteur de caractéristiques

Un modèle de description d'images est composé de plusieurs éléments. Le premier d'entre eux est l'extracteur de caractéristiques. Ce dernier a pour objectif de produire un vecteur caractérisant l'image fournie en entrée. Cette opération peut être effectuée en se servant d'un classifieur d'images auquel on a ôté la partie réalisant la classification. Souvent il s'agit des couches linéaires à la fin de l'architecture.

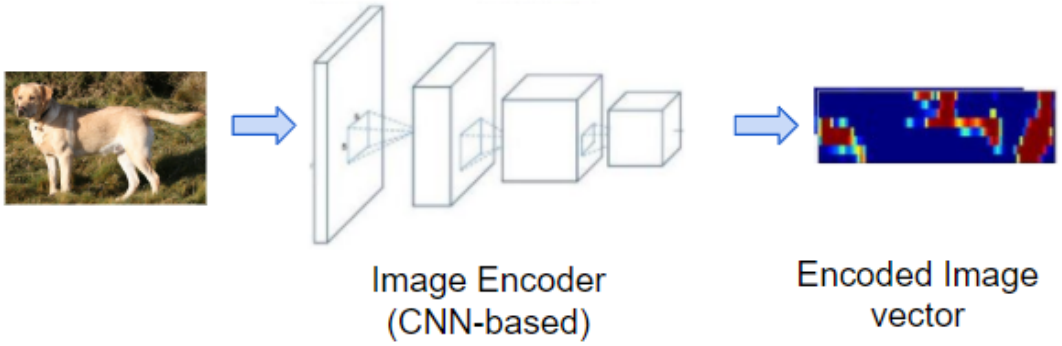

FIGURE 2.1 – Exemple d'encodeur
Source : Towards Data Science

Pour notre projet, nous avons fait le choix de nous servir de classifieurs d'images déjà entraînés tels que vgg16, resnet50 ou inception v3. Nous avons choisi ces 3 classifieurs pour leur grande popularité, ce qui nous assurait une architecture fonctionnelle. Nous sommes néanmoins conscients qu'il existe des architectures potentiellement plus performantes grâce à ce classement établi sur le site de PyTorch [1].

Cela nous permet de gagner du temps d'entraînement et probablement d'obtenir un gain de performance par rapport à ce qu'on aurait pu produire nous-mêmes. En effet, les réseaux pré-entraînés de PyTorch sont entraînés sur le jeu de données d'ImageNet qui assure un grand volume et une grande diversité d'images.

Voici ensuite les structures des 3 encodeurs que nous avons utilisé :

1. https://pytorch.org/vision/stable/models.html

FIGURE 2.2 – Architecture VGG-16
Source : opengenus

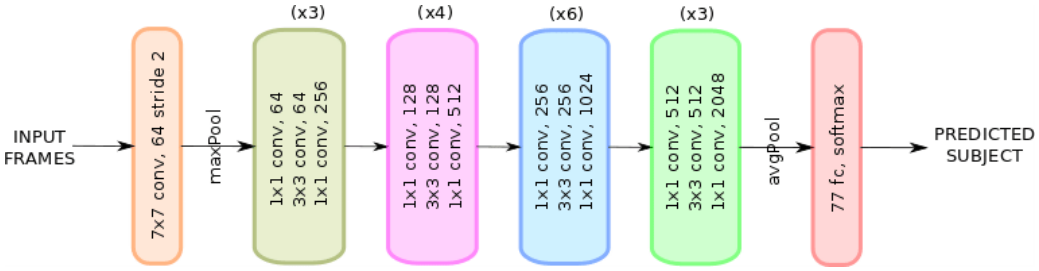

FIGURE 2.3 – Architecture ResNet-50
Source : researchgate

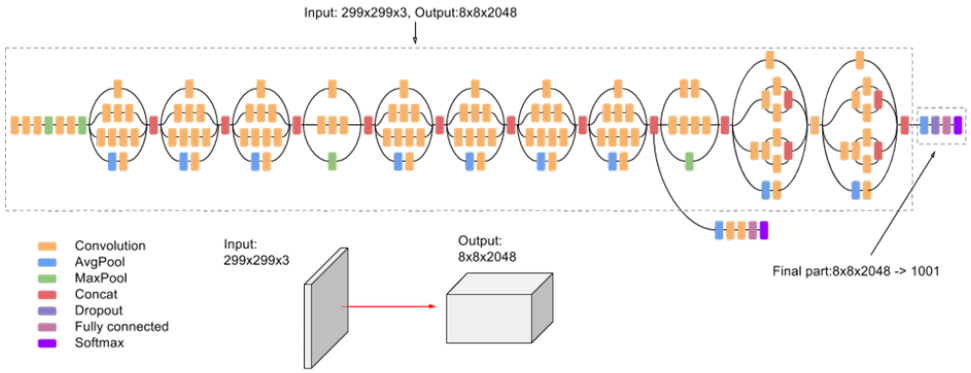

FIGURE 2.4 – Architecture Inception V3
Source : production-media

2.2 Décodeur

2.2.1 Réseau récurrent

Le décodeur a pour objectif de produire une phrase à partir des caractéristiques extraites de l'image d'entrée. On a ici nos entrées qui sont passées dans une couche d'*embedding* puis une suite de cellules LSTM. Le réseau prend en entrée l'image encodée et renvoie une séquence de *tokens* (des mots principalement). L'intérêt d'un réseau récurrent est qu'il contient un état caché faisant office de mémoire. Cette technique est particulièrement utile dans la génération de séquence car cela permet de garder une trace de ce qui a déjà été traité. L'objectif est ainsi d'obtenir une meilleure compréhension du contexte.

Le RNN va boucler pour faire sa prédiction. Il commence par prédire le premier mot, puis prédit le second à partir du premier, puis prédit le troisième à partir du deuxième et ainsi de suite.

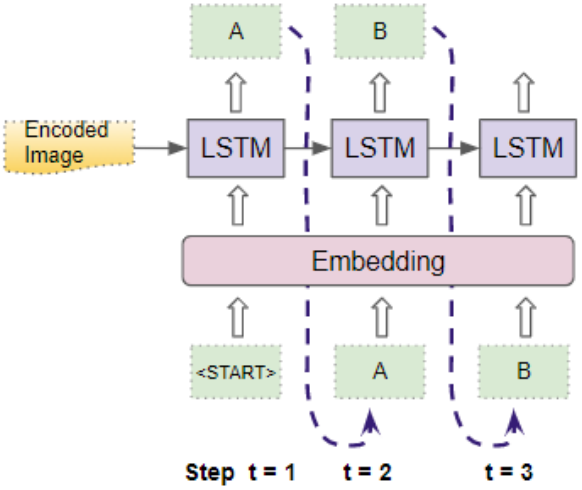

FIGURE 2.5 – Réseau récurrent
Source : Towards Data Science

Nous avons par ailleurs intégré une couche d'*Attention* dans notre décodeur pour augmenter significativement les performances de notre modèle. En effet, celle-ci a pour rôle de porter attention à certaines parties de l'entrée pour effectuer sa prédiction de manière à imiter le comportement humain.

2.2.2 Attention

La couche d'attention vient se placer entre la couche d'*embedding* et la cellule LSTM. Cette couche d'attention aide le modèle à prédire le prochain mot en se focalisant uniquement sur les parties de l'image les plus pertinentes pour le générer.

Comme indiqué dans la figure ci-dessous, on applique les étapes suivantes dans l'ordre :

— On commence par pondérer l'image et l'état caché avec des poids qui seront appris par la suite
— On fait passer ce résultat dans une fonction d'activation tangente hyperbolique
— On applique un softmax sur nos sorties
— Finalement on renvoie la somme des sorties du softmax.

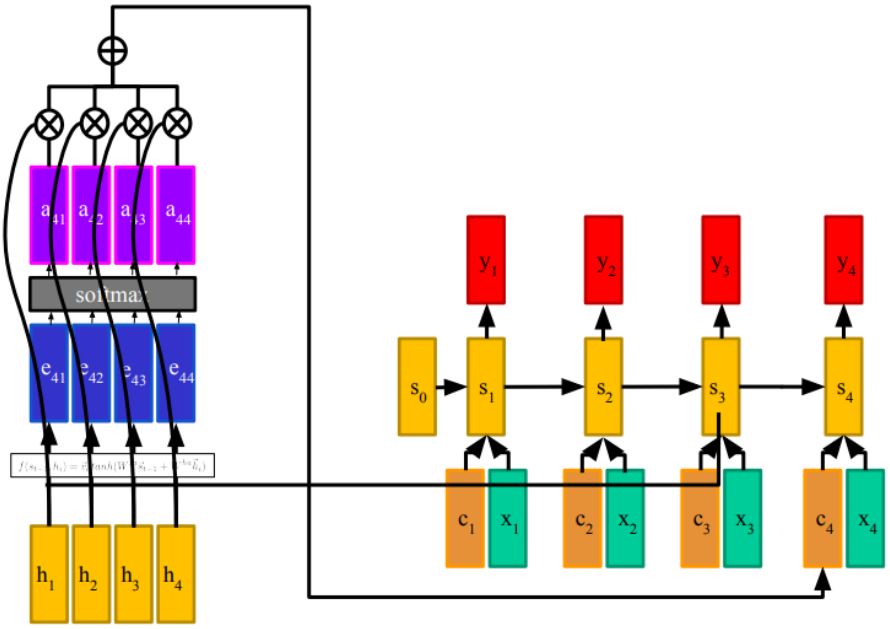

FIGURE 2.6 – Structure de l'attention
Source : Cours IFT780 de l'UdS

2.3 Encodeur-Décodeur

L'encodeur-décodeur consiste à combiner les deux modèles précédents en un unique. La propagation des images dans le modèle s'effectue en deux étapes :

— Les images sont données en entrée dans l'encodeur pour obtenir les caractéristiques de l'image
— Les caractéristiques sont ensuite fournies au décodeur qui retourne les sorties

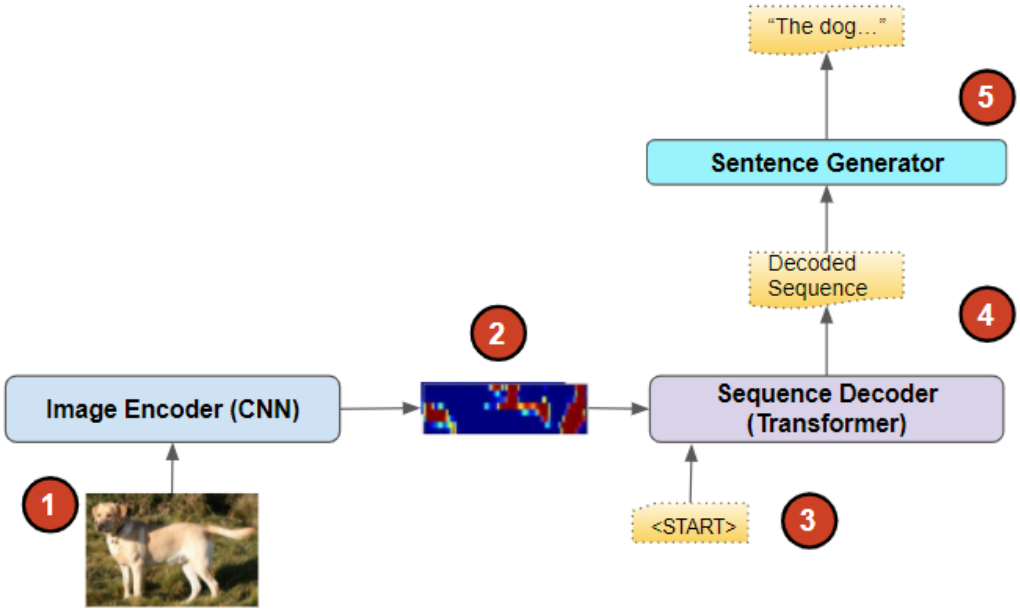

FIGURE 2.7 – Architecture globale
Source : Towards Data Science

Gestion de l'entraînement

3.1 Checkpointing

L'exécution de la méthode fit sur les données d'entraînement peut être longue du fait de la grande quantité de données ainsi que la complexité de la tâche. Nous avons donc mis en place des checkpoints pour pouvoir :

— Interrompre l'entrainement et le reprendre plus tard
— Reprendre au même point après un arrêt dû à un problème de connexion ou un arrêt de la machine.

Nous effectuons un checkpoint à la fin de chaque époque du fit. Ainsi la reprise de l'entraînement se fera après le nombre d'époques terminées. Par exemple si nous avons effectué 4 époques et que l'interruption a lieu lors de la cinquième, l'entraînement reprendra au début de la cinquième époque.

Nous sauvegardons les informations suivantes à la fin de chaque époque grâce à la méthode *save* de la classe *EncoderDecoder* :

— num_epochs : Le nombre d'époques achevées jusqu'ici
— embed_size : La taille de l'embeding
— vocab_size : La taille du vocabulaire
— attention_dim : La taille de l'attention
— encoder_dim : La taille de l'encoder
— decoder_dim : La taille du decoder
— state_dict : La structure du modèle ainsi que les poids obtenus jusqu'ici
— loss_history : La liste des *loss* pour pouvoir afficher la progression de l'apprentissage du modèle

3.2 Fonction de perte

Lors de la phase d'entraînement, nous avons besoin d'une fonction de perte ou *loss*, pour pouvoir appliquer la rétro-propagation. La fonction de perte que nous avons choisi est l'entropie croisée.

Nous avons considéré d'autres fonctions de perte présentes sur le site de la documen-

tation de PyTorch [1]. La seule alternative pertinente à l'entropie croisée semble être la NLLLOSS pour *negative log likelihood loss*. La NLLLOSS avec une couche `LogSoftmax` [2] est équivalente à une entropie croisée.

Nous avons donc décidé de ne pas implémenter de fonction de perte alternative car il ne nous a pas semblé pertinent de changer la structure du modèle pour obtenir une fonction de perte équivalente, le résultat aurait été exactement le même.

1. https://pytorch.org/docs/stable/nn.html#loss-functions
2. PyTorch CrossEntropyLoss vs. NLLLoss

Exécution du code

4.1 Interface en ligne de commande

Afin de rendre le code facilement exécutable sans devoir le modifier, il nous a été demandé de ne rien coder en dur. Les aspects suivants devraient pouvoir être spécifiés :

— le modèle utilisé
— les hyperparamètres
— la recherche d'hyperparamètres

Ces options doivent être paramétrables depuis la ligne de commande en changeant les paramètres associés. Nous avons donc réalisé une interface en ligne de commande. Pour ce faire, nous avons utilisé la librairie click [1] qui offre la possibilité de définir facilement et de façon concise les arguments et options de l'interface. Dans notre cas, le fichier principal `main.py`, responsable de l'entraînement, peut être exécuté en spécifiant :

— l'extracteur de caractéristiques (vgg16, resnet50 ou inception v3)
— la taille des batchs
— la taille de *l'embedding*
— la dimension de l'attention
— la dimension du décodeur
— le taux d'apprentissage
— le taux de *dropout*
— le nombre d'époques
— le choix de faire une recherche d'hyperparamètres
— le choix de charger le checkpoint d'un modèle pré-entraîné
— le choix de prédire la description d'une image spécifique
— le choix d'afficher les zones d'attention sur une image

La librairie click permet de fixer une liste de choix fermée pour les arguments et une valeur par défaut pour les options. Tout cela est configurable en tant que décorateurs de la fonction à exécuter avec les paramètres fournis dans la ligne de commande. Le programme est ensuite lancé avec les valeurs fournies. De plus, la librairie s'occupe de générer l'option d'aide pour la commande et de vérifier que les contraintes de l'interface sont respectées.

1. Article de blog Medium présentant click

4.2 Recherche d'hyperparamètres

Dans l'optique d'effectuer une recherche d'hyperparamètres précise tout en restant efficace, nous nous sommes aidés du framework Optuna, qui est spécifiquement conçu pour l'apprentissage automatique. La modularité d'Optuna nous permet facilement de définir les espaces de recherche pour nos hyperparamètres.

Le but va être de minimiser (ou de maximiser) une fonction objectif. Une étude (ou *study*) est une optimisation basée sur cette fonction objectif. Le but d'une étude est de trouver l'ensemble optimal de valeurs d'hyperparamètres avec un nombre défini d'essais. Optuna est conçu pour accélérer la recherche en sélectionnant les valeurs à tester selon des estimateurs probabilistes.

Dans notre cas, les hyperparamètres pertinents à ajuster sont les suivants :

— Le taux d'apprentissage
— La taille de *l'embedding*
— La dimension de l'attention
— La dimension du décodeur
— Le taux de *dropout*

Nous avons donc un grand nombre d'hyperparamètres. Cela implique un nombre important d'essais au sein de l'étude, et donc un temps de calcul conséquent. Nous avons ainsi simplement effectué la recherche d'hyperparamètres sur un ensemble très réduit de données, de l'ordre de quelque centaines d'images. Les résultats obtenus sont donc peu représentatifs, mais selon nos essais les hyperparamètres suivants sont pertinents et fonctionnent bien :

— Taux d'apprentissage : 10^{-3}
— Taille de *l'embedding* : 300
— Dimension de l'attention : 256
— Dimension du décodeur : 512
— Taux de *dropout* : 0.2

Analyse des résultats

5.1 Évaluation de l'entraînement

Nous pouvons à présent analyser les résultats liés aux courbes d'apprentissage sur nos 3 modèles.

On peut voir que la structure la plus stable est celle avec l'extracteur *ResNet 50*. En effet, pour les 2 autres structures on peut distinctement voir les séparations entre les époques. Ceci est en accord avec nos prédictions obtenues au cours de la phase d'entraînement : la structure avec *ResNet 50* avait tendance à donner des résultats plus cohérents tout le long de l'entraînement.

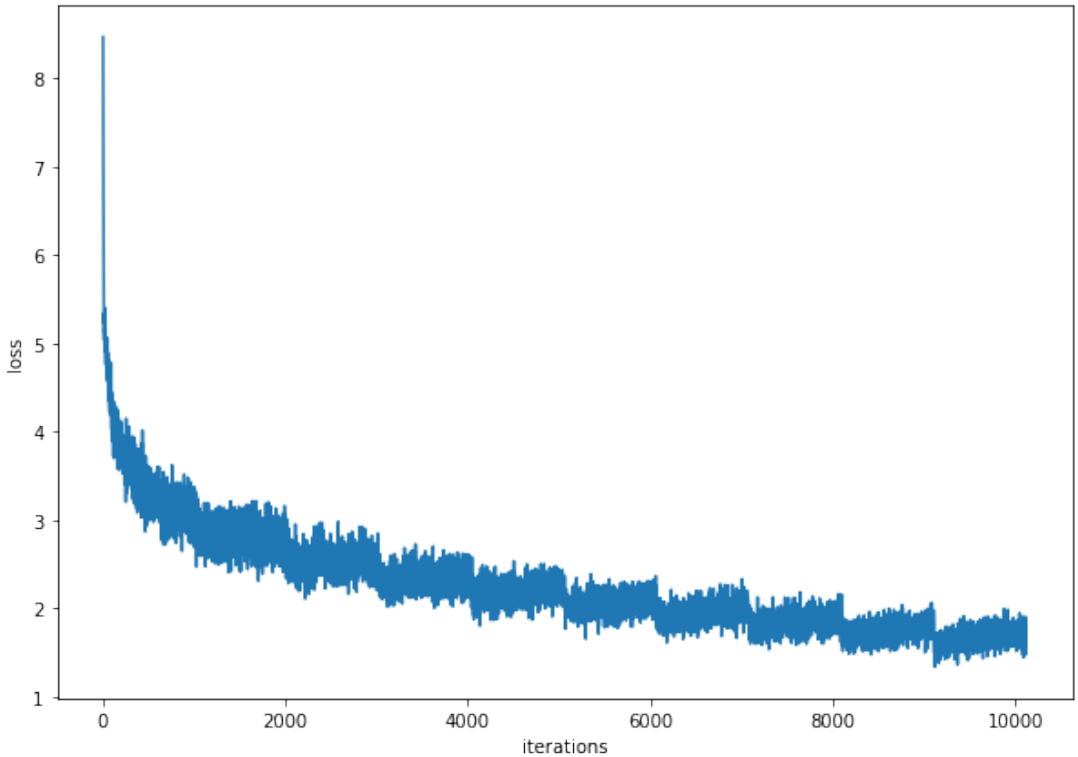

FIGURE 5.1 – Évolution de la perte avec l'extracteur Inception V3

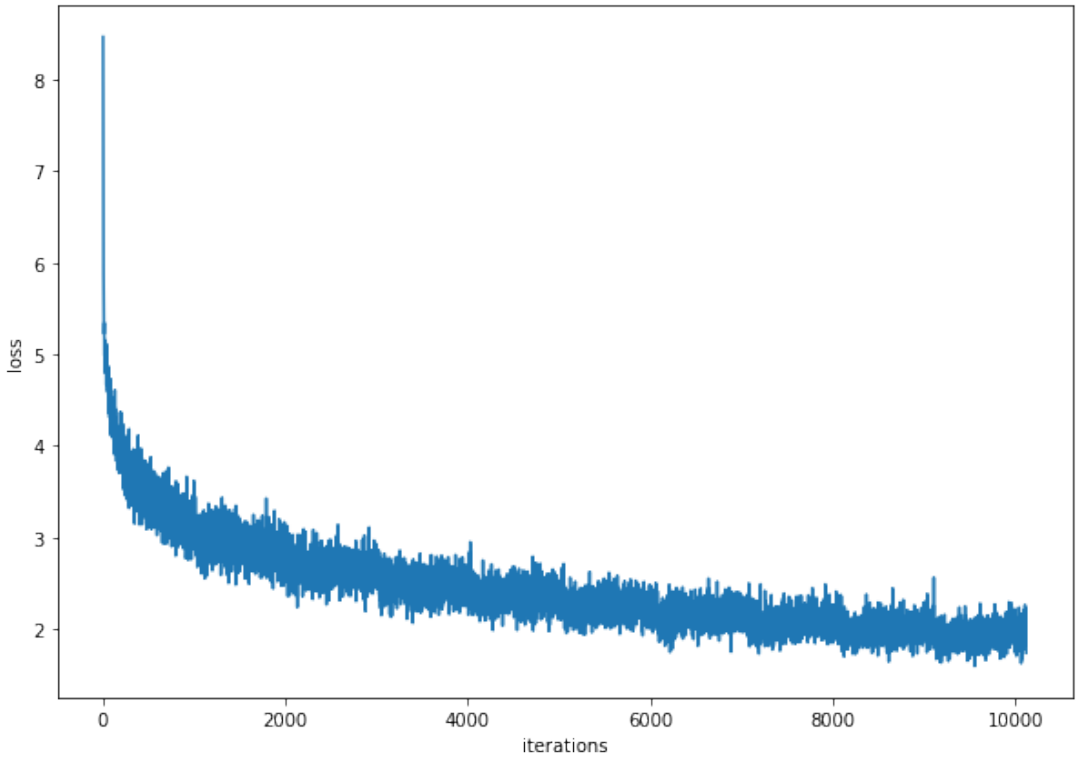

FIGURE 5.2 – Évolution de la perte avec l'extracteur ResNet 50

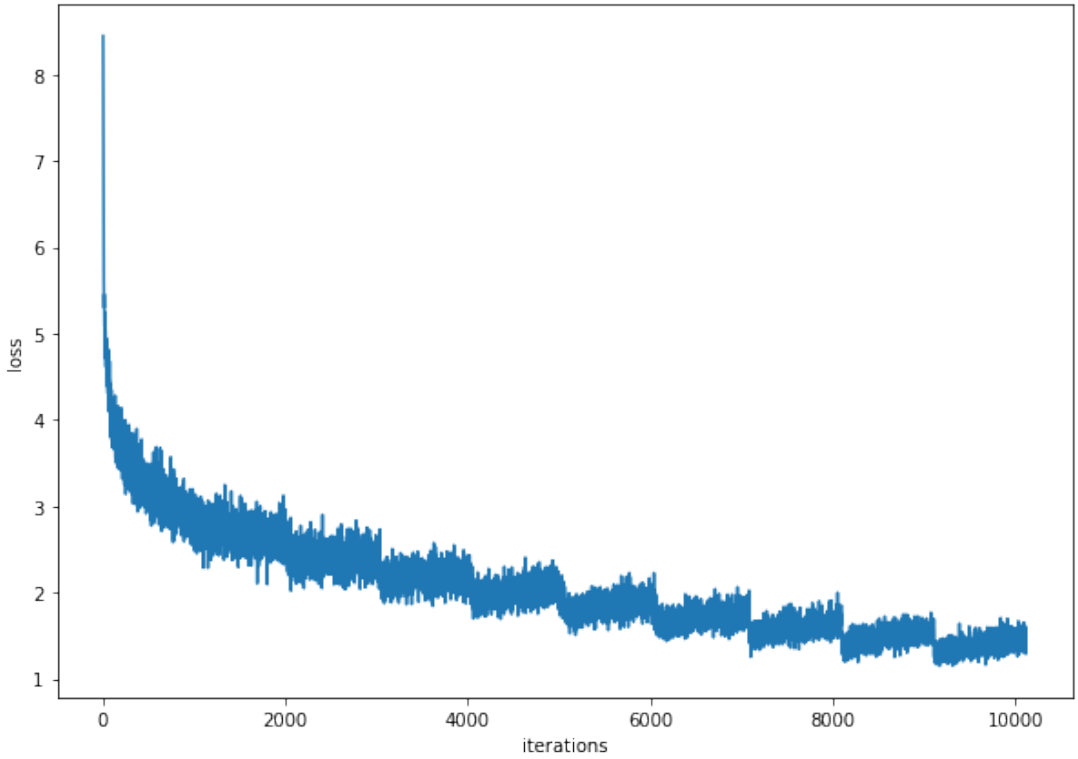

FIGURE 5.3 – Évolution de la perte avec l'extracteur VGG 16

5.2 Performance du modèle

Bien que cela ne semble être fait sur aucun projet de description d'images que nous ayons vu, nous avons décidé de séparer le jeu de données en test/entraînement (code disonible dans le notebook *train_ test_ split*), nous avons ajouté dans notre programme la possibilité de calculer un score de test en passant en entrée le chemin d'un ensemble de test. Par manque de temps nous n'avons pas pu faire l'ensemble de nos tests entièrement, il ne nous a donc pas paru pertinent d'analyser cela davantage.

Dans les images suivantes, on présente des exemples d'attention produites par le modèle associée à la prédiction du modèle. Dans le premier exemple, on voit que l'attention est bien portée sur le chien sur la deuxième image. De même, l'attention est bien portée sur l'objet attrapé, mais il s'agit plutôt d'une branche que d'un frisbee.

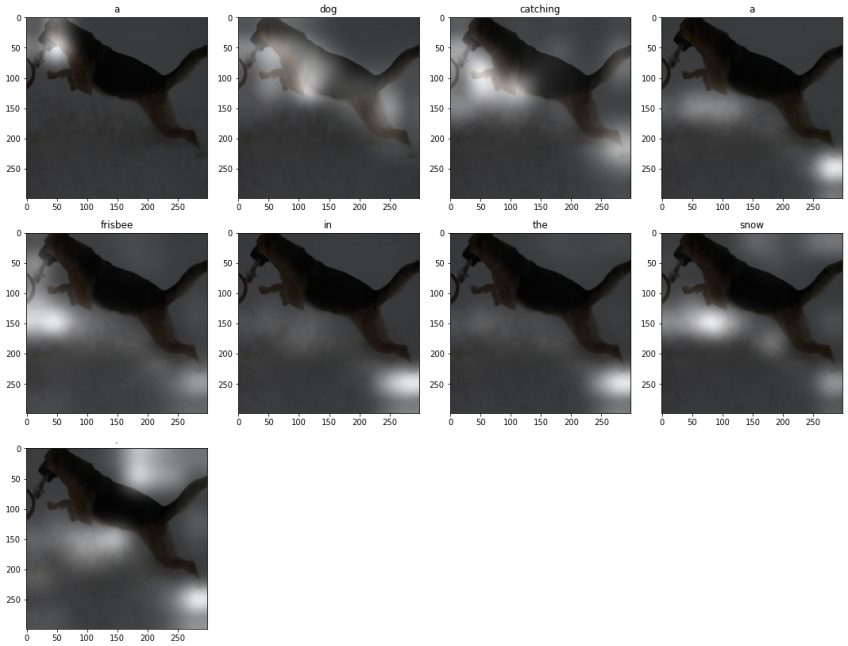

FIGURE 5.4 – Exemple d'attention sur une image de chien

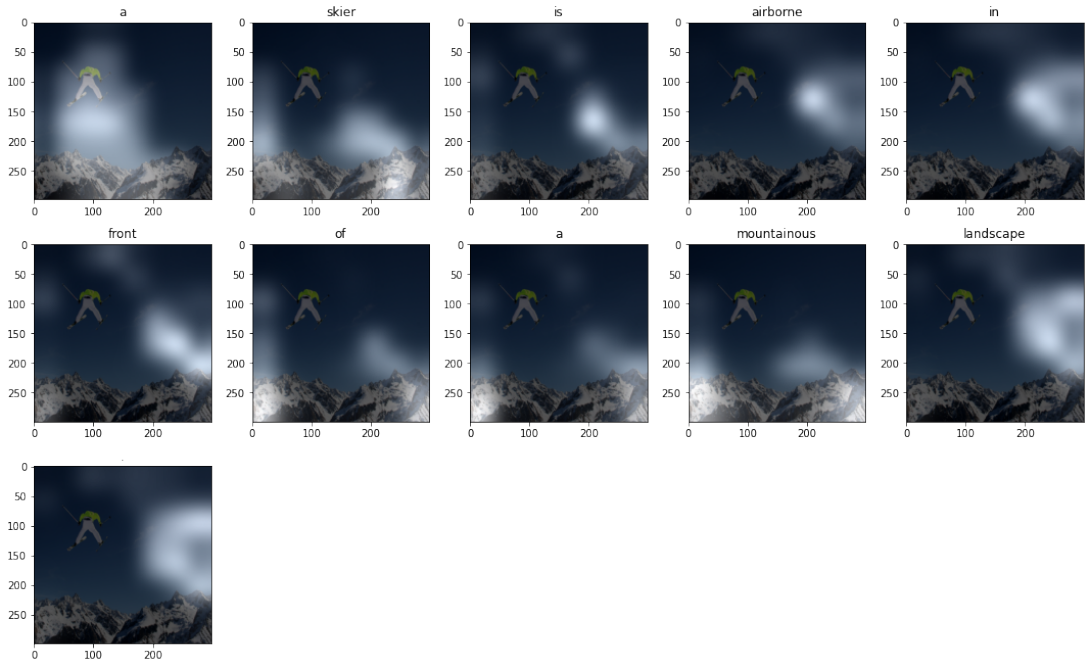

FIGURE 5.5 – Exemple d'attention sur une image de skieur

Conclusion

Pour conclure, nous étions très intéressés par ce projet et enthousiastes à l'idée de le réaliser. La description d'images est une tâche nécessitant à la fois du traitement d'images par extraction de caractéristiques et du traitement de texte en générant des phrases. Ces types de données sont aujourd'hui beaucoup étudiés et traités en utilisant respectivement des réseaux convolutifs et des réseaux récurrents. Nous sommes satisfaits des résultats que nous avons obtenus et agréablement surpris des prédictions que notre modèle peut faire. Le principe d'attention est particulièrement efficace pour détecter les points d'intérêts dans une image afin d'en produire une description.

Plusieurs prolongements sont envisageables pour notre projet. Premièrement, nous aurions pu tester un autre jeu de données possédant une diversité d'images différente. En effet, nous avons remarqué beaucoup d'images similaires dans celui que nous avons utilisé. Cela a pour conséquence de produire un certain biais lors de l'entraînement. D'autre part, nous aurions pu utiliser un autre système d'*embedding* plutôt que l'*embedding* naïf proposé par PyTorch. Par exemple, il aurait été intéressant de comparer les performances de notre modèle avec un autre utilisant un *embedding* basé sur Word2vec ou GloVe. Finalement, une autre alternative aurait été de voir si l'utilisation de GRU au lieu de cellules LSTM aurait eu un impact significatif sur les scores.

Appendix

Prédiction sur des images du jeu de test, puis sur des photos personnelles.