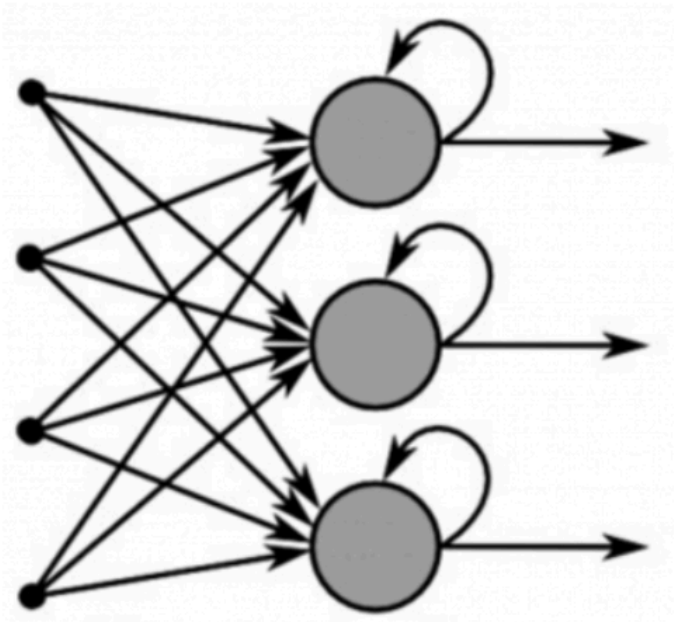
Département Télécommunications, Réseaux et Informatique

Master SDIA

Master Artificial Intelligence
&
Data Science

Rapport Attention



RNN

fait par RIM GOURRAM

Prof Youness Abouqora

année universitaire 2025/2026

Chapitre 1 : Introduction

L’évolution des techniques d’intelligence artificielle a permis le développement de modèles capables de comprendre et de générer du langage naturel à partir de données visuelles. Parmi ces applications, le “image captioning” occupe une place centrale, visant à générer automatiquement une description textuelle pertinente pour une image donnée. Cette tâche combine des compétences en traitement d’images et en traitement du langage naturel et constitue un défi majeur en vision par ordinateur et en apprentissage automatique multimodal. Dans ce travail pratique, l’objectif est de concevoir un modèle d’image captioning en combinant plusieurs techniques avancées, telles que l’extraction de caractéristiques visuelles à partir d’un ResNet50 pré-entraîné via le transfert d’apprentissage, la génération de séquences textuelles à l’aide d’un RNN (LSTM) personnalisé, l’amélioration de la qualité de génération par l’intégration d’un module d’attention et l’utilisation d’embeddings pré-entraînés (Word2Vec) pour représenter les mots des légendes. Le dataset utilisé, Flickr30k, contient plus de 30 000 images annotées avec des descriptions textuelles, ce qui permet d’entraîner et d’évaluer un modèle capable de produire des légendes cohérentes et descriptives. Ce TP permet à l’apprenant de comprendre la structure et la manipulation d’un dataset d’images annotées, d’appliquer des transformations adaptées pour préparer les données, de construire et combiner des modèles d’extraction de caractéristiques visuelles et de génération de texte, de mettre en œuvre un mécanisme d’attention pour améliorer la qualité des légendes, de gérer l’entraînement du modèle avec des embeddings pré-entraînés et un scheduler de taux d’apprentissage, et enfin d’analyser et d’interpréter les résultats obtenus, notamment en générant des légendes pour des images de test.

Chapitre 2 : Présentation des données et vectorisation des légendes

Introduction:

Dans ce chapitre, nous préparons les données pour le modèle de captioning d'images. Nous chargeons le dataset Flickr30k depuis Hugging Face, explorons quelques exemples d'images et de légendes, puis construisons le vocabulaire à partir des textes. Les légendes sont ensuite tokenisées grâce aux dictionnaires word2idx et idx2word. Enfin, nous définissons un dataset PyTorch personnalisé avec les transformations d'images et créons les DataLoader pour l'entraînement et la validation.

1. Source de donnée

Le dataset **Flickr30k** utilisé dans ce projet a été téléchargé depuis **Hugging Face**, une plateforme collaborative qui propose de nombreux jeux de données et modèles pour le

machine learning et le traitement du langage naturel. Nous avons utilisé le dataset fourni par **AnyModal/flickr30k**

2. Chargement des données

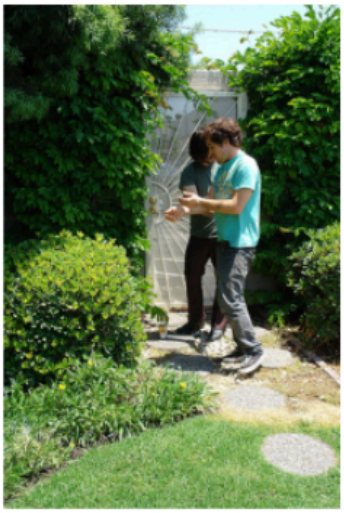
```
Téléchargement Flickr30k depuis Hugging Face...
 - Trying: AnyModal/flickr30k
README.md: 100% 762/762 [00:00<00:00, 86.8kB/s]
data/train-00000-of-00009.parquet: 100% 429M/429M [00:02<00:00, 216MB/s]
data/train-00001-of-00009.parquet: 100% 434M/434M [00:01<00:00, 514MB/s]
data/train-00002-of-00009.parquet: 100% 442M/442M [00:01<00:00, 535MB/s]
data/train-00003-of-00009.parquet: 100% 431M/431M [00:01<00:00, 587MB/s]
data/train-00004-of-00009.parquet: 100% 448M/448M [00:01<00:00, 317MB/s]
data/train-00005-of-00009.parquet: 100% 457M/457M [00:01<00:00, 415MB/s]
data/train-00006-of-00009.parquet: 100% 484M/484M [00:01<00:00, 564MB/s]
data/train-00007-of-00009.parquet: 100% 465M/465M [00:01<00:00, 329MB/s]
data/train-00008-of-00009.parquet: 100% 436M/436M [00:01<00:00, 543MB/s]
data/validation-00000-of-00001.parquet: 100% 140M/140M [00:01<00:00, 92.2MB/s]
data/test-00000-of-00001.parquet: 100% 142M/142M [00:01<00:00, 205MB/s]
Generating train split: 100% 29000/29000 [00:09<00:00, 1653.06 examples/s]
Generating validation split: 100% 1014/1014 [00:00<00:00, 1214.73 examples/s]
Generating test split: 100% 1000/1000 [00:01<00:00, 802.24 examples/s]
✅ Loaded: AnyModal/flickr30k
Splits disponibles: ['train', 'validation', 'test']
Features train: {'image': Image(mode=None, decode=True), 'alt_text': List(Value('string')), 'sentids': List(Value('string')),
```

Exemple de donnée

On définit une fonction `get_caption` qui extrait la première légende d'un exemple. On affiche ensuite les clés disponibles dans un exemple ainsi que la légende d'une image afin de vérifier que les données sont correctement chargées. Pour préparer les images en vue de l'entraînement du modèle, on **applique les transformations suivantes** : les images sont **redimensionnées** à une taille de (224, 224) et **converties en tenseurs**, ce qui permet leur traitement par PyTorch.

```
Keys: ['image', 'alt_text', 'sentids', 'split', 'img_id', 'filename', 'original_alt_text']
Caption: Two people with shaggy hair look at their hands while hanging out in the yard.
Image type: <class 'PIL.JpegImagePlugin.JpegImageFile'>
```

Two people with shaggy hair look at their hands while hangin



3. Construction du vocabulaire et des dictionnaires de tokenisation

Dans cette section, nous avons construit le vocabulaire du dataset à partir des légendes textuelles. Nous avons nettoyé les textes en supprimant la ponctuation et en mettant les mots en minuscules. Ensuite, nous avons compté la fréquence de chaque mot et conservé uniquement ceux apparaissant au moins un certain nombre de fois. Nous avons ajouté des tokens spéciaux (`<pad>`, `<bos>`, `<eos>`, `<unk>`) pour gérer le début et la fin des séquences, les mots inconnus et le padding. Enfin, nous avons créé les dictionnaires `word2idx` et `idx2word` pour convertir les mots en indices et vice-versa, étape indispensable pour la tokenisation et l'entrée dans le modèle.

```python
MAX_LEN = 20

def tokenize(sentence):
    sentence = clean_text(sentence)
    ids = [word2idx[BOS]]
    for w in sentence.split():
        ids.append(word2idx.get(w, word2idx[UNK]))
    ids.append(word2idx[EOS])

    # truncate
    if len(ids) > MAX_LEN:
        ids = ids[:MAX_LEN]
        ids[-1] = word2idx[EOS]

    # pad
    ids = ids + [word2idx[PAD]] * (MAX_LEN - len(ids))
    return ids

def untokenize(ids):
    words = []
    for t in ids:
        w = idx2word[int(t)]
        if w == EOS:
            break
        if w not in [PAD, BOS]:
            words.append(w)
    return " ".join(words)

print(tokenize("A dog runs in the grass")[:12])
```

```
[1, 19, 1027, 2814, 1690, 3452, 1480, 2, 0, 0, 0, 0]
```

4. Préparation du dataset et création des DataLoaders

Pour préparer les données, nous avons créé un dataset PyTorch personnalisé `CaptionDS` qui prend en entrée le dataset Flickr30k. Pour chaque exemple, l'image est transformée (redimensionnée et convertie en tenseur) et la légende est **tokenisée** en identifiants numériques correspondant au vocabulaire construit précédemment. Le dataset complet a été divisé en ensembles d'entraînement et de validation selon un ratio de 90/10 à l'aide de `random_split`. Des `DataLoader` ont été créés pour chaque split, permettant de générer des batches et de mélanger les données lors de l'entraînement. Une vérification rapide a confirmé les tailles des ensembles, les dimensions des images et des légendes, ainsi que la cohérence de la tokenisation et de la détokenisation.

```
Train size: 26100 Val size: 2900
Images: torch.Size([32, 3, 224, 224]) Captions: torch.Size([32, 20])
Ex caption: a person wearing a gray shirt puts on a white apron
```

Chapitre 3 : Modélisation et entraînement du réseau de légendes d'images

Introduction:

Dans ce chapitre, nous présentons la construction et l'entraînement du modèle de génération de légendes d'images. Nous commençons par extraire les caractéristiques des images à l'aide d'un réseau convolutionnel pré-entraîné (ResNet50), puis nous intégrons un mécanisme d'attention permettant au modèle de se concentrer sur les parties pertinentes de chaque image lors de la génération de mots. Enfin, nous combinons ces éléments avec un LSTM pour produire des légendes séquentielles et détaillons la procédure d'entraînement du modèle.

Étape 1 : Chargement d'un modèle ResNet50 pré-entraîné

Dans cette étape, nous utilisons un réseau ResNet50 pré-entraîné pour extraire des représentations riches des images. Nous retirons les couches finales de pooling et de classification afin de conserver uniquement les caractéristiques convolutionnelles. Les paramètres du réseau sont figés pour ne pas être mis à jour pendant l'entraînement du modèle de légende. Enfin, nous passons un batch d'images à travers le réseau pour vérifier la forme des tenseurs de

caractéristiques obtenus.

Étape 2 : L'architecture du modèle Restnet

Nous avons affiché l'architecture complète du modèle ResNet50 pré-entraîné afin de visualiser ses différentes couches. On observe que le modèle est composé d'une couche initiale de convolution et de normalisation, suivie de quatre blocs de couches résiduelles (`layer1` à `layer4`) qui extraient progressivement des caractéristiques de plus en plus complexes des images. Enfin, le modèle original se termine par une couche de pooling adaptatif (`avgpool`) et une couche entièrement connectée (`fc`) pour la classification.

Cette étape nous permet de comprendre la structure du réseau avant de le modifier pour notre tâche de génération de légendes, où seules les couches convolutionnelles sont conservées et les couches finales de classification sont supprimées.

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
```

Étape 3 : L'architecture du modèle gelé

À cette étape, nous avons modifié le modèle ResNet50 pré-entraîné pour notre tâche spécifique. Plus précisément, nous avons conservé toutes les couches convolutionnelles du réseau, responsables de l'extraction hiérarchique des caractéristiques, et supprimé les deux dernières couches entièrement connectées (`avgpool` et `fc`) qui sont spécifiques à la classification originale.

Ensuite, nous avons gelé les poids de toutes les couches conservées afin de préserver les représentations déjà apprises par le modèle sur le dataset ImageNet. Cette approche nous permet d'utiliser ResNet comme extracteur de caractéristiques robuste, tout en évitant de recalculer les gradients pour ces couches et en se concentrant uniquement sur l'entraînement des couches supplémentaires qui seront ajoutées pour notre tâche.

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
```

Pourquoi est-il important de geler les poids du ResNet dans ce TP ?

Dans ce TP, nous avons choisi de geler les poids du modèle ResNet50 pré-entraîné. Cette approche permet de réutiliser les connaissances acquises par le réseau lors de son entraînement sur un large dataset, en conservant sa capacité à détecter des caractéristiques générales comme les bords, textures et formes. En gelant les poids, l'entraînement devient plus rapide et nécessite moins de ressources, car seules les couches ajoutées pour notre tâche spécifique sont optimisées. De plus, cette méthode réduit le risque de surapprentissage sur notre dataset, souvent plus petit, et permet d'exploiter efficacement un modèle puissant sans modifier ses paramètres préalablement appris.

Étape 4 : Attention

Dans cette étape, nous implémentons un module d'attention qui permet au modèle de se concentrer sur les parties les plus pertinentes des caractéristiques extraites par le CNN. L'attention calcule un vecteur de contexte en pondérant les caractéristiques spatiales selon leur importance relative pour

la tâche de prédiction. Plus précisément, pour chaque pas de temps, le module combine les caractéristiques extraites du CNN (`features`) et l'état caché du LSTM (`hidden`) pour générer des scores d'attention, normalisés via un softmax. Le vecteur de contexte obtenu met en avant les informations les plus significatives, améliorant ainsi la capacité du modèle à se focaliser sur les zones importantes de l'image pour la tâche de classification ou de génération.

Étape 5: CaptionModel

```
CaptionModel(
  (embedding): Embedding(3881, 256)
  (attn): Attention(
    (Wf): Linear(in_features=2048, out_features=256, bias=True)
    (Wh): Linear(in_features=512, out_features=256, bias=True)
    (v): Linear(in_features=256, out_features=1, bias=True)
  )
  (lstm): LSTMCell(2304, 512)
  (fc): Linear(in_features=512, out_features=3881, bias=True)
)
```

Dans cette étape, nous définissons le modèle de génération de légendes (`CaptionModel`) qui combine un module d'attention avec un LSTM pour produire des légendes à partir des caractéristiques extraites par le CNN. Le modèle

commence par encoder chaque mot d'entrée via une couche d'embedding, puis utilise le module d'attention pour calculer un vecteur de contexte pondéré sur les caractéristiques visuelles. Ce vecteur de contexte est concaténé avec l'embedding du mot courant et transmis au LSTM pour générer l'état caché suivant. Enfin, une couche linéaire prédit la probabilité de chaque mot du vocabulaire à chaque pas de temps. Cette architecture permet au modèle de se concentrer dynamiquement sur différentes parties de l'image lors de la génération de chaque mot de la légende.

Étape 6: Entraînement du modèle et early stopping

```
Epoch 01 | train_loss=3.7176 | val_loss=3.1838
Saved best: /kaggle/working/best_caption_model.pt
Epoch 02 | train_loss=2.9293 | val_loss=2.9227
Saved best: /kaggle/working/best_caption_model.pt
Epoch 03 | train_loss=2.6199 | val_loss=2.8227
Saved best: /kaggle/working/best_caption_model.pt
Epoch 04 | train_loss=2.3812 | val_loss=2.7756
Saved best: /kaggle/working/best_caption_model.pt
Epoch 05 | train_loss=2.1660 | val_loss=2.7778
Epoch 06 | train_loss=1.9592 | val_loss=2.7915
Epoch 07 | train_loss=1.7542 | val_loss=2.8420
Early stopping.
Best val loss: 2.7755557337960046
```

Dans cette étape, le modèle de génération de légendes est entraîné sur le dataset d'images et de légendes en utilisant l'optimiseur Adam et la fonction de perte `CrossEntropyLoss`. La boucle d'entraînement suit le schéma standard : extraction des caractéristiques via le CNN gelé, passage des caractéristiques dans le modèle LSTM avec attention, calcul de la perte et mise à jour des poids.

L'apprentissage est surveillé sur un ensemble de validation pour éviter le surapprentissage. Le mécanisme d'early stopping a été utilisé avec une patience de 3 époques : l'entraînement s'arrête si la perte de validation ne s'améliore pas pendant 3 époques consécutives. Dans notre cas, la perte de validation a diminué jusqu'à la 5e époque, puis a commencé à stagner et augmenter légèrement, déclenchant l'arrêt anticipé à la 7e époque. Le meilleur modèle, correspondant à une perte de validation minimale de 2.776, a été sauvegardé pour une utilisation ultérieure dans la génération de légendes.

Étape 7: Génération de légendes à partir d'images test

Dans cette étape, le modèle entraîné est utilisé pour générer automatiquement des légendes à partir d'images. Chaque image est prétraitée et transformée en tenseur avant d'être passée au modèle. Les caractéristiques extraites par le CNN gelé servent d'entrée au LSTM avec attention, qui génère les

mots séquentiellement jusqu'au token de fin (EOS) ou jusqu'à la longueur maximale.

```
GT  : a group of kids and a person walk on the boardwalk eating ice cream cones
PRED: a group of kids including a person with a wheelchair are walking down a sidewalk
============================================================
```

```
GT  : two people standing outside on the sidewalk one wearing a yellow jacket and the other wearing a blue
PRED: a person is walking down a street with a taxi in the background and a taxi taxi in the
```

Conclusion

Dans ce TP, nous avons implémenté un modèle de captioning d'images combinant un ResNet50 pré-entraîné, un LSTM et un module d'attention. Le ResNet a été gelé pour exploiter le transfert d'apprentissage, tandis que l'attention permettait au modèle de se concentrer sur les régions pertinentes de l'image lors de la génération des légendes. L'utilisation d'embeddings pré-entraînés a amélioré la représentation des mots. Les résultats montrent que le modèle produit des descriptions cohérentes et pertinentes, illustrant l'efficacité de l'association vision et séquence pour le captioning d'images.