

ÉCOLE
CENTRALELYON

ÉCOLE CENTRALE DE LYON

MSO 4.4
APPRENTISSAGE AUTOMATIQUE

BE1 - Image Captioning

Étudiants :

Nathan ETOURNEAU
Paul FLAGEL

Enseignant :

Emmanuel DELLANDRÉA

Introduction

Le but de ce TP est d'implémenter le modèle proposé dans l'article [1], intitulé "Show, Attend and Tell". Cet article propose une méthode permettant de générer des descriptions automatique d'images, à partir d'une image passée en entrée. Le jeu de données utilisé est le jeu de données Flickr 8k, contenant 8 000 images ayant chacune 5 descriptions candidates.

Le problème peut être perçu comme un problème de traduction automatique : la séquence d'entrée est une image, et la séquence à traduire est la description.

Par conséquent, l'architecture proposée dans le papier est une architecture inspirée de celles utilisées pour de la traduction automatique de texte : un réseau décodeur est utilisé pour apprendre une représentation de la séquence à traduire (ici l'image).

Cette représentation est ensuite passée à un réseau de neurones récurrent, en initialisant l'état caché avec cette représentation encodée de l'image. La prédiction est ensuite effectuée à partir d'un token <START> disant de commencer à prédire, jusqu'au token <STOP>.

Un mécanisme d'attention est utilisé dans le réseau récurrent : le réseau récurrent utilise pour sa prédiction la représentation courante de l'image (état caché du réseau récurrent), ainsi que la sortie précédente

Table des matières

Introduction	1
1 Présentation du modèle Encodeur-Décodeur	3
1.1 Architecture encodeur-décodeur	3
1.2 Encodeur	3
1.3 Décodeur et mécanisme d'attention	4
1.3.1 Décodeur	4
1.3.2 Mécanisme d'attention	5
2 Entraînement du modèle	6
2.1 Fonction de perte et attention bistrochastique	6
2.2 Évaluation des performances : top-5 accuracy et BLEU-score	6
2.3 Beam search	7
2.4 Hyperparamètres utilisés	8
3 Expérimentations	8
3.1 Influence de l'extracteur de features de l'encodeur	8
3.2 Influence du type de cellules récurrentes du décodeur	8
3.3 Schedule sampling	8
Conclusion	9
Bibliographie	9
A Code de l'encodeur VGG-19	9
B Code de l'encodeur ResNet-101	10

1 Présentation du modèle Encodeur-Décodeur

1.1 Architecture encodeur-décodeur

Dans le cadre de ce BE, l'architecture proposée est une architecture encodeur-décodeur. Ce type d'architecture est un classique des problèmes de NLP, car il est efficace pour les problèmes de traduction "séquence à séquence". En effet, un réseau récurrent naïf seul ne peut pas gérer des séquences de taille variable.

La solution à ce problème est proposée dans [2] est de générer à partir de la séquence d'entrée, une représentation latente de l'entrée. Ensuite, cette représentation latente est passée à un réseau décodeur, qui génère la séquence de sortie, basée sur la représentation latente de l'entrée, ainsi que sur les tokens de la séquence de sortie précédemment générés.

Cette technique a par la suite été améliorée dans [3] avec l'introduction d'un mécanisme d'attention. Le principal problème est que la représentation latente est une représentation de taille fixe et statique de la séquence d'entrée. Ce type d'architecture avait des difficultés avec les longues séquences, ne disposant que de la représentation latente, et des tokens générés précédemment. Le mécanisme d'attention permet d'ajouter un élément de contexte à cette représentation statique. Ce mécanisme apprend à identifier les sous-portions de la représentation latente de la séquence d'entrée les plus pertinentes compte tenu de la séquence ayant été générée jusqu'ici.

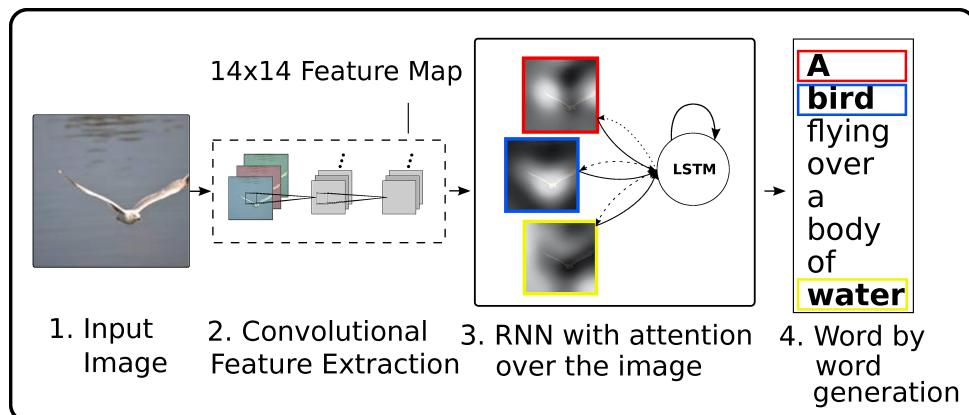


FIGURE 1 – Modèle utilisé pour la génération de descriptions automatiques

Ce BE vise à implémenter l'article [1], proposant cette architecture encodeur-décodeur avec attention, appliquée à la génération automatique de description d'images. La séquence d'entrée est ici une phrase, et la séquence de sortie est une description de cette image.

1.2 Encodage

Avant la publication de [1], les principales approches pour la description automatique d'images consistaient à détecter des objets dans une image, et à apprendre à relier ces objets dans une description. Cette approche avait le problème de voir le monde par groupe d'objets, sans réelle description.

Dans cet article, l'approche proposée a été de s'éloigner de cette description en terme d'objets, mais plutôt d'utiliser un extracteur de features déjà réputé performant pour les images. La représentation latente de l'image est obtenue en faisant passer les images à partir d'un CNN performant sur des problèmes de classification d'images : un ResNet [4], VGGNet [5], ou GoogleNet [6] par exemple. Dans le cadre de ce BE, nous nous limiterons à l'utilisation de ResNet-101 et de VGG-16 comme extracteurs de features.

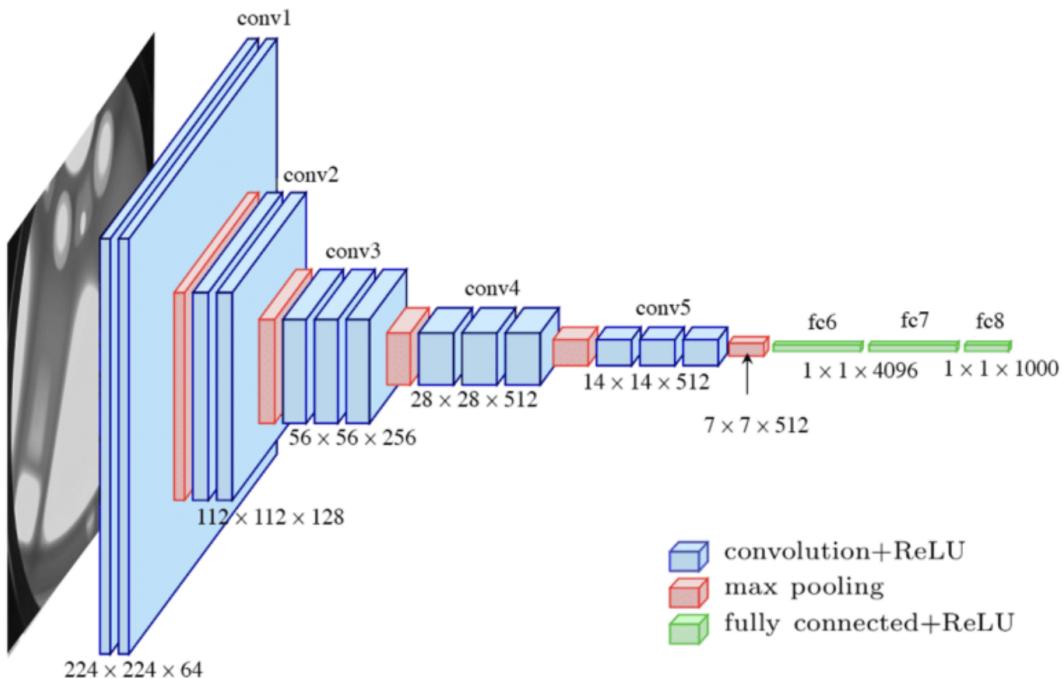


FIGURE 2 – Schéma du réseau VGG employé comme extracteur de features

L’approche développée ici est une approche de Transfer Learning : ces réseaux sont extrêmement performants en reconnaissance d’images sur la base de données ImageNet-1000. Ainsi, pour être performants sur une si grande variété de classes, notre compréhension actuelle est que le réseau est capable de détecter une très grande variété de formes, contours, et de combiner ces informations pour obtenir des représentations compactes de cette image [7].

Contrairement à ce qui est fait d’habitude dans un cadre de Transfer Learning, la sortie qui est récupérée ici n’est pas celle d’un des layers fully-connected, mais celle du dernier layer de convolution (avant pooling). Le but recherché ici est d’obtenir une représentation compacte de l’image sous forme d’image, et non pas sous forme de vecteur. La sortie du réseau récurrent est passée à travers un AdaptivePooling2d, afin de forcer les images de sortie à être de taille 14×14 . Le nombre de channels dépend du réseau employé : 512 pour VGG-16, et 2048 pour ResNet-101.

En reprenant les notations de l’article, le réseau encodeur produit à partir d’une entrée x , une sortie

$$a = \{a_1, a_2, \dots, a_L\}, \quad a_i \in \mathbb{R}^{14} \times \mathbb{R}^D$$

avec $L = 14 \times 14 = 196$ et D dépendant du nombre de channels du réseau employé.

1.3 Décodeur et mécanisme d’attention

Dans cette partie, nous allons voir comment à partir de l’entrée x , et de cette représentation cachée $a = \{a_1, a_2, \dots, a_L\}$, nous sommes en mesure de générer une description de l’image $y = \{y_1, y_2, \dots, y_C\}$, avec C la taille de la phrase. Ceci se fait en deux phases : l’utilisation d’un réseau à attention, puis l’utilisation d’un réseau récurrent.

1.3.1 Décodeur

Nous allons décrire le fonctionnement du décodeur : le décodeur est un réseau récurrent, qui prend en entrée la représentation latente de l’image (à travers l’état caché), et une concaténation de la représentation de l’image, avec le token précédent. Ainsi, lors de l’entraînement du réseau, on passe au réseau les bons labels à prédire, car on suppose que pour la prédiction à l’instant $t + 1$, le label à l’instant t est connu. On n’utilise pas le label ayant été prédit, qui est potentiellement incorrect. On apprend au modèle à générer une suite de prédiction correctes. Ceci est modifié lors de l’évaluation, où l’on part du token

<START>, et où on génère récursivement les tokens. C'est ce que l'on appelle du teacher-forced learning.

La première couche de ce réseau est un layer d'embedding : la sortie du réseau décodeur (et donc son entrée aussi) est un vecteur de taille (`batch_size, vocab_size`). La dimension étant très grande, la présence de cet embedding permet de réduire la dimension des tenseurs manipulés.

La seconde couche est plus compliquée : c'est une cellule LSTM prenant en entrée la concaténation de la sortie de l'embedding précédent, et de la sortie de l'encodeur, pondérée par l'attention (nous détaillerons ce point après). Nous allons préciser les dimensions des quantités présentes ici pour y voir plus clair.

- La sortie de l'embedding est de dimension (`batch_size, embed_dim`)¹. Cette sortie correspond au token qui aurait dû être prédit, passé à travers l'embedding.
- La sortie de l'encodeur après son passage par le réseau à attention, de dimension (`batch_size, encoder_dim`).

Ainsi, l'entrée de la couche récurrente est de dimension (`batch_size, embed_dim + encoder_dim`). La sortie est de dimension (`batch_size, decoder_dim`).

De plus, avant de faire des prédictions successives, la mémoire interne de ce réseau récurrent est initialisée d'une façon très particulière : on moyenne la représentation latente de l'image, de dimension (`batch_size, encoder_dim`) sur tous les pixels. Cette moyenne est ensuite passée à travers deux réseaux feed-forward, pour obtenir la dimension (`batch_size, decoder_dim`).

$$c_0 = f_{init,c} \left(\frac{1}{L} \sum_{i=1}^L \mathbf{a}_i \right)$$

$$h_0 = f_{init,c} \left(\frac{1}{L} \sum_{i=1}^L \mathbf{a}_i \right)$$

La mémoire suit ensuite les équations classiques d'un réseau LSTM.

On passe ensuite la sortie du réseau LSTM dans un Dropout [8], puis dans une couche fully-connected afin de passer de la dimension (`batch_size, decoder_dim`) à la dimension (`batch_size, vocab_size`).

1.3.2 Mécanisme d'attention

Cet article introduit 2 mécanismes d'attention. Un mécanisme d'attention "dur", qui empêche l'entraînement "end-to-end" du modèle, et qui nécessite du reinforcement learning pour l'entraînement. Dans ce BE, nous nous sommes intéressés uniquement à l'autre mécanisme d'attention : l'attention "douce".

On avait noté $a = \{a_1, a_2, \dots, a_L\}$ la sortie de l'encodeur. On avait vu ensuite que de décodeur est un réseau récurrent avec deux états cachés, notés h_t et c_t à l'instant t , qui prend en entrée à l'instant t le token y_{t-1} généré à l'étape précédente, et génère un nouveau token y_t . Nous avions vu que la sortie a de l'encodeur était passée à travers un mécanisme d'attention, transformant de la dimension (`batch_size, num_pixels, encoder_dim`) à la dimension (`batch_size, encoder_dim`) : le mécanisme d'attention permet de pondérer l'information contenue dans chaque pixel de l'image obtenue en sortie de l'encodeur, par importance.

À chaque pas de temps t , le mécanisme d'attention prend en entrée la représentation encodée de l'image a , de dimension (`batch_size, num_pixels, encoder_dim`), ainsi que l'état caché à l'instant $t-1$ h_{t-1} , de dimension (`batch_size, decoder_dim`). Ce mécanisme génère un vecteur de contexte \hat{z}_t , en calculant d'abord des scalaires :

$$e_{ti} = f_{att}(\mathbf{a}_i, h_{t-1})$$

puis des poids α_{ti} associés aux a_i suite à la prédiction faite à l'instant $t-1$:

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{k=1}^L \exp(e_{tk})}$$

1. On sélectionne la t -ième entrée du vecteur à la ligne 208.

où la fonction f_{att} est un réseau de neurones qui :

- transforme linéairement \mathbf{a} en un tenseur de dimension (`batch_size, num_pixels, attention_dim`)
- transforme linéairement h_{t-1} en un tenseur de dimension (`batch_size, 1, attention_dim`)
- ajoute les deux tenseurs obtenus aux deux opérations précédentes, les passe à travers une fonction d'activation ReLU, pour obtenir un tenseur de dimension (`batch_size, num_pixels, attention_dim`)
- transforme linéairement le résultat de la dimension précédente en un tenseur de dimension (`batch_size, num_pixels`)

Ensuite, le vecteur de contexte est généré :

$$\hat{z}_t = \beta_t \odot \sum_{i=1}^L \alpha_{ti} \mathbf{a}_i$$

où $\beta_t = \sigma(f_\beta(h_{t-1}))$ est un vecteur de scalaires de taille (`batch_size, encoder_dim`), soit la même dimension que chaque \mathbf{a}_i , la multiplication est réalisée terme à terme. La fonction σ est la fonction sigmoïde, et la fonction f_β est un layer linéaire ayant `decoder_dim` features en entrée, et `encoder_dim` features en sortie. Le but de ce vecteur β_t est de moduler l'information portée par le vecteur de contexte, en fonction de l'état de la mémoire h_{t-1} .

2 Entraînement du modèle

2.1 Fonction de perte et attention bistochastique

Ce modèle est certes généré à partir de 3 sous-réseaux indépendants, la prédiction des tokens se fait uniquement à l'aide de passes avant de réseau de neurones. Cette architecture permet ainsi la backpropagation et l'entraînement simultané de l'encodeur, du décodeur et du mécanisme d'attention. Dans le cadre de ce BE, l'encodeur n'est pas entraîné (fine-tuné) car il a déjà été entraîné.

La fonction de perte utilisée est particulière : la sortie du décodeur est une distribution de probabilités sur le vocabulaire. Cette sortie est comparée au mot réel qu'il aurait fallu prédire, sous forme one-hot. Cette comparaison est faite à l'aide de la `CrossEntropyLoss`.

De plus, l'article [1] introduit un mécanisme d'attention stochastique double : on a d'une part grâce au Softmax $\sum_{i=1}^L \alpha_{ti} = 1$, mais le réseau favorise également le fait d'avoir $\sum_t \alpha_{ti} \approx 1$.

En effet, la fonction de perte employée est la suivante :

$$L(\text{image}, \mathbf{y}) = \sum_{t=1}^C \text{CrossEntropy}\left(f(\text{image}, y_{t-1}), y_t\right) + \lambda \sum_{i=1}^L \left(1 - \sum_{t=1}^C \alpha_{ti}\right)^2$$

Avec \mathbf{y} la description générée, avec $y_0 = <\text{START}>$ et $y_{C+1} = <\text{END}>$. Après la passe avant cette fonction est minimisée. La fonction f fait appel aux 3 réseaux : encodeur, attention, décodeur. Le calcul du gradient de la fonction de perte par backpropagation permet ainsi de mettre à jour les paramètres de l'ensemble des 3 réseaux. La gestion de la taille variable des séquences est gérée avec un astucieux tri des batchs par taille de séquence décroissante

2.2 Évaluation des performances : top-5 accuracy et BLEU-score

Pour l'entraînement du modèle, 2 métriques sont utilisées : la top-5 accuracy, ainsi que le BLEU-score.

La top-5 accuracy est une métrique qui mesure, à chaque prédiction, le nombre de fois où le token à prédire est dans les 5 tokens ayant le meilleur score. Sur le jeu d'entraînement, la prédiction est effectuée à partir de la vérité terrain (hors schedule sampling), alors qu'en mode évaluation, les prédictions sont effectuées récursivement. Il est ainsi probable que cette métrique soit mauvaise sur le jeu de validation, une simple divergence en début de phrase suffit à ruiner l'ensemble des prédictions suivantes.

Une autre métrique est utilisée pour évaluer la qualité des descriptions d'images, exclusivement sur le jeu de validation : c'est le BLEU-score [9]. Le BLEU-score est une métrique permettant d'évaluer la pertinence d'une traduction automatique d'un texte, étant donné un corpus de traductions de référence.

Dans le cadre de ce BE, les traductions proposées sont les descriptions de l'image générées automatiquement, et les traductions de référence sont les descriptions cibles du jeu de validation. Le BLEU-score calcule la fréquence de co-occurrence de 4-grams entre la description prédite, et celle du jeu de validation. Ce score est un score entre 0 et 1, est facile à calculer.

2.3 Beam search

Une fois notre modèle entraîné, le décodeur est capable de prédire récursivement les tokens de la phrase. Pour générer nos descriptions, plusieurs approches sont possibles : une première approche, appelée **recherche gloutonne**, consisterait à choisir, étant donné une succession de tokens, le token le plus probable (obtenu grâce au softmax en sortie de réseau). Toutefois, cette approche a un défaut majeur : si l'un des tokens est erroné en début de phrase, l'intégralité de la description va être erronée.

Dans ce BE, nous allons utiliser la technique du **beam search** (recherche par faisceau). Cette technique prend un paramètre K , et plutôt que de considérer à chaque fois la prédiction ayant le meilleur score, cette technique sélectionne K descriptions candidates ayant le meilleur score.

À la première prédiction, le modèle prend uniquement le token <START>, et donne en sortie une densité de probabilités sur le vocabulaire. À cette itération, les K tokens ayant le meilleur score sont conservés.

À l'itération suivante, pour chaque prédiction parmi ces K prédictions optimales jusqu'alors, on effectue une nouvelle passe avant. La sortie du décodeur est alors une densité de probabilités sur les mots du vocabulaire, conditionnée à la prédiction effectuée jusqu'alors. Les K prédictions précédentes ont engendré K^2 nouvelles prédictions. On ne garde que les K prédictions ayant le meilleur score.

On peut voir la beam search comme un parcours en largeur d'un arbre, où à chaque couche de l'arbre, on ne poursuit le parcours que pour les K branches les plus prometteuses. Dans notre cas, chaque noeud de l'arbre correspond à la probabilité d'occurrence de la sous-séquence générée jusqu'alors.

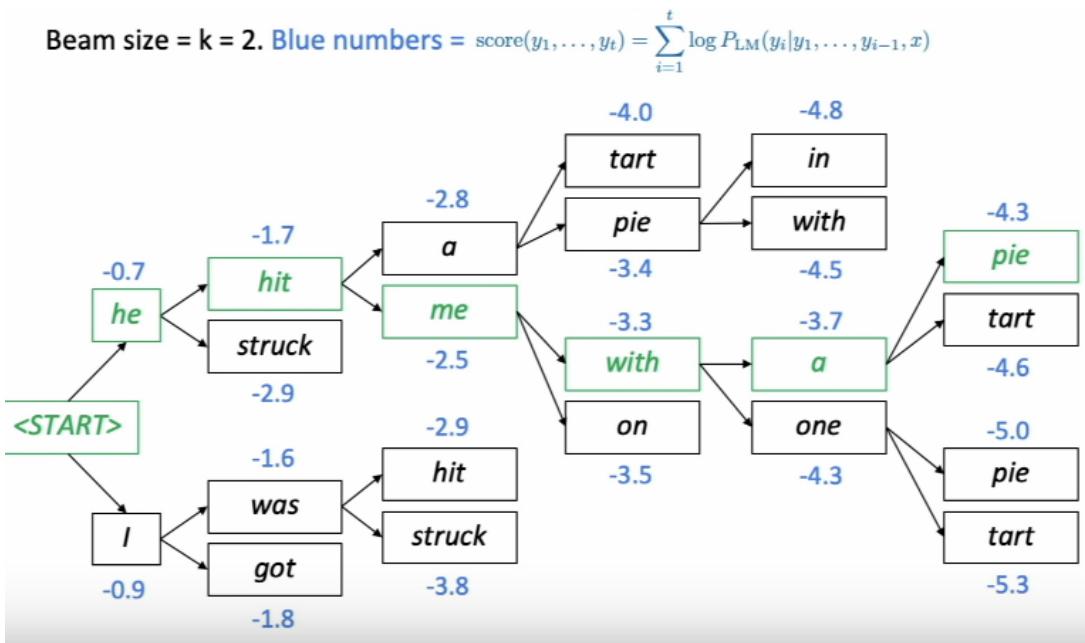


FIGURE 3 – Représentation du fonctionnement de la recherche par faisceau

2.4 Hyperparamètres utilisés

Parameter	Value
batch size	32
dropout	0.5
decoder_lr	4e-4
gradient clipping	5
λ (attention bistrochastique)	1
number of epochs	120
early stopping	20
reduce lr on plateau	0.8
optimizer	Adam

TABLE 1 – Hyperparamètres utilisés dans le BE

3 Expérimentations

3.1 Influence de l'extracteur de features de l'encodeur

Nous avons modifié l'extracteur de features : nous avons essayé le ResNet-101 ainsi que le VGG16. Les résultats sont les suivants :

Model	BLEU4-score
ResNet-101	0,2011
VGG16	0,1844

TABLE 2 – Résultats de la variation de l'extracteur de features

3.2 Influence du type de cellules récurrentes du décodeur

Nous avons modifié le type de cellules récurrentes employées. Nous avons essayé des cellules de type LSTM, ainsi que des cellules GRU et RNN classiques.

Les principales modifications dans le code concernent la gestion de la mémoire : le LSTM possède deux états cachés : un état caché h_t , qui varie à court terme, et un état de la cellule c_t , qui varie à plus long terme.

La cellule GRU fusionne ces deux états. Le code est ainsi modifié en conséquence. De même, la cellule RNN ne contient qu'un seul état.

Model	Reccurent units	BLEU4-score
ResNet-101	LSTM	0,2011
ResNet-101	GRU	0.1903
ResNet-101	RNN	0.1528

TABLE 3 – Résultats de la variation de la cellule récurrente

3.3 Schedule sampling

Dans cette partie, nous aurions dû coder le Schedule Sampling, qui permet de générer des prédictions de meilleure qualité lorsque correctement codé : en effet, les prédictions sont ici générées à partir du label "ground truth", et non à partir du token prédit à l'instant précédent.

Nous ne l'avons pas fait car nous avons eu quelques difficultés à prendre en main le code : le code pour l'entraînement ainsi que la validation étaient effectués en mode teacher-forcing, et comme le code pour l'évaluation sur le jeu de test incluait la recherche par faisceau il a été difficile de s'inspirer.

Conclusion

Dans ce BE, nous avons pu nous familiariser avec le mécanisme d'attention ainsi qu'avec les architectures séquences-à-séquence. Ces architectures sont celles qui réalisent l'état de l'art de la plupart des applications du NLP, et ce TP a ainsi été une précieuse introduction.

Ce BE est un très bon exemple d'application d'une architecture encodeur-décodeur, et de révisions sur les réseaux récurrents et convolutifs. Nous avons bien mieux compris comment fonctionnaient ces architectures, et nous imaginons très bien comment nous pouvons adapter l'architecture du BE à des problèmes de traduction de texte.

Bibliographie

- [1] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell : Neural image caption generation with visual attention, 2016.
- [2] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.
- [3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv :1409.0473*, 2014.
- [4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [5] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv :1409.1556*, 2014.
- [6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
- [7] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.
- [8] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1) :1929–1958, 2014.
- [9] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu : a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

A Code de l'encodeur VGG-19

Pour le réseau encodeur VGG-19, le code d'initialisation du réseau est le suivant :

```

class Encoder(nn.Module):
    """
    Encoder.

    """

    def __init__(self, encoded_image_size=14):
        super(Encoder, self).__init__()
        self.enc_image_size = encoded_image_size

        resnet = torchvision.models.vgg16(
            pretrained=True
        ) # pretrained ImageNet ResNet-101 or vgg-16

        # Remove linear and pool layers (since we're not doing classification)
        modules = list(resnet.children())[:-1]
        self.resnet = nn.Sequential(*modules)

        # Resize image to fixed size to allow input images of variable size
        self.adaptive_pool = nn.AdaptiveAvgPool2d(
            (encoded_image_size, encoded_image_size)
        )

        self.fine_tune()

    def forward(self, images):
        """
        Forward propagation.

        :param images: images, a tensor of dimensions (batch_size, 3, image_size, image_size)
        :return: encoded images
        """
        out = self.resnet(images) # (batch_size, 2048, image_size/32, image_size/32)
        out = self.adaptive_pool(
            out
        ) # (batch_size, 2048, encoded_image_size, encoded_image_size)
        out = out.permute(
            0, 2, 3, 1
        ) # (batch_size, encoded_image_size, encoded_image_size, 2048)
        return out

    def fine_tune(self, fine_tune=True):
        """
        Allow or prevent the computation of gradients for convolutional blocks 2 through 4 of the encoder.

        :param fine_tune: Allow?
        """
        for p in self.resnet.parameters():
            p.requires_grad = False
        # If fine-tuning, only fine-tune convolutional blocks 2 through 4
        for c in list(self.resnet.children())[5:]:
            for p in c.parameters():
                p.requires_grad = fine_tune

```

Listing 1 – Code Python de l'encodeur VGG-19

B Code de l'encodeur ResNet-101

Pour le réseau encodeur ResNet-101, le code d'initialisation de l'encodeur est le suivant :

```

class Encoder(nn.Module):
    """
    Encoder.
    """

    def __init__(self, encoded_image_size=14):
        super(Encoder, self).__init__()
        self.enc_image_size = encoded_image_size

        resnet = torchvision.models.resnet101(
            pretrained=True
        ) # pretrained ImageNet ResNet-101 or vgg-16

        # Remove linear and pool layers (since we're not doing classification)
        modules = list(resnet.children())[
            :-2
        ] # -1 pour le vgg (on ne garde que le feature extractor), -2 pour le resnet (on enlève les deux dernières couches)
        self.resnet = nn.Sequential(*modules)

        # Resize image to fixed size to allow input images of variable size
        self.adaptive_pool = nn.AdaptiveAvgPool2d(
            (encoded_image_size, encoded_image_size)
        )

        self.fine_tune()

        # print(self.resnet)

    def forward(self, images):
        """
        Forward propagation.

        :param images: images, a tensor of dimensions (batch_size, 3, image_size, image_size)
        :return: encoded images
        """
        out = self.resnet(images) # (batch_size, 2048, image_size/32, image_size/32)
        out = self.adaptive_pool(
            out
        ) # (batch_size, 2048, encoded_image_size, encoded_image_size)
        out = out.permute(
            0, 2, 3, 1
        ) # (batch_size, encoded_image_size, encoded_image_size, 2048)
        return out

    def fine_tune(self, fine_tune=True):
        """
        Allow or prevent the computation of gradients for convolutional blocks 2 through 4 of the encoder.

        :param fine_tune: Allow?
        """
        for p in self.resnet.parameters():
            p.requires_grad = False
        # If fine-tuning, only fine-tune convolutional blocks 2 through 4
        for c in list(self.resnet.children())[5:]:
            for p in c.parameters():
                p.requires_grad = fine_tune

```

Listing 2 – Code Python de l'encodeur ResNet-101