

ÉCOLE
CENTRALELYON

ÉCOLE CENTRALE LYON

MSO 3.4 - APPRENTISSAGE AUTOMATIQUE
BE
COMPTE RENDU

BE 1 - Image Captioning

Élèves :
Akilhoussen ONALY
Antony PARODI

Enseignant :
Mr. Thomas DUBOUDIN

Table des matières

1	Introduction	2
2	Théorie, expérimentations et résultats	2
2.1	Architecture initiale	2
2.1.1	Rappels théoriques	2
2.1.2	Expérimentations et résultats	5
2.2	Changement de l'encodeur	9
2.3	Changement du décodeur	12
2.4	Ajout du Schedule Sampling	16
3	Conclusion	23

1 Introduction

L'objectif de ce BE est d'implémenter des réseaux récurrents et créer un modèle permettant de générer une légende descriptive pour une image d'entrée. Pour ce faire, l'architecture que nous utiliserons est celle d'un Encodeur - Décodeur avec un mécanisme d'attention. Nous utiliserons également les concepts de Transfer Learning et de Beam Search afin d'optimiser les performances du modèle.

Dans une première partie, nous détaillerons brièvement l'architecture initiale du modèle. Ensuite, nous ferons quelques expérimentations sur des choix architecturaux tels que le choix de l'encodeur et du décodeur et étudierons leur impact sur la performance. Enfin, nous terminerons par l'implémentation du Schedule Sampling qui permettra d'améliorer les performances du modèle.

2 Théorie, expérimentations et résultats

2.1 Architecture initiale

2.1.1 Rappels théoriques

Comme nous l'avons précisé en introduction, le réseau utilisé est constitué tout d'abord d'un encodeur qui est un réseau ResNet 101 pré-entraîné dont nous conservons uniquement les couches convolutives tel qu'illustré sur la figure 1 et nous l'affinons sur le jeu de données du problème (Transfer Learning).

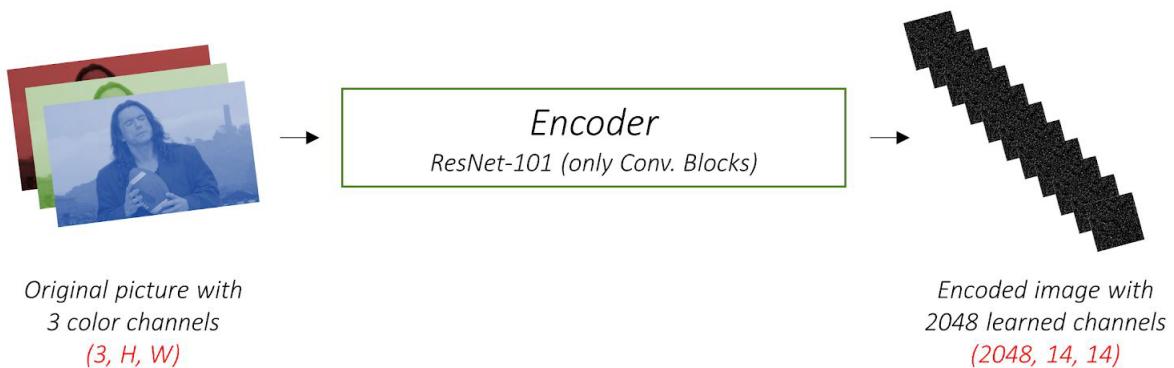


FIGURE 1 – Illustration de l'encodeur

Ensuite, le réseau contient un décodeur qui prend en entrée la sortie de l'encodeur et calcule la séquence de sortie étape par étape comme illustré sur la figure 2. Dans notre cas, il s'agit d'un réseau classique LSTM.

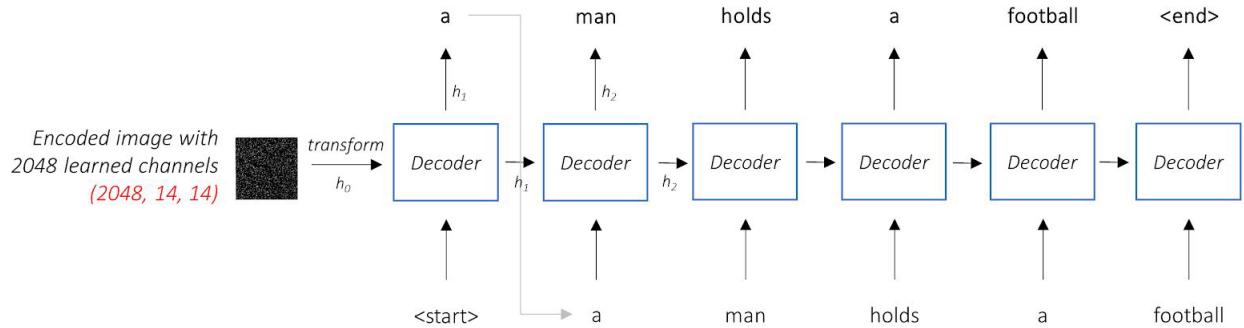


FIGURE 2 – Illustration du décodeur

Ensuite, comme évoqué en introduction, nous utilisons un mécanisme d’attention dans l’apprentissage du modèle. L’attention est un moyen donné au modèle d’utiliser uniquement les parties pertinentes de l’image à chaque étape dans le décodeur. Techniquement, cela signifie que dans le décodeur, l’entrée à l’étape suivante n’est pas seulement le mot précédent (de la vérité terrain) et l’activation à l’étape précédente. On y ajoute aussi l’image d’origine (encodée) pondérée avec certains poids (voir illustration en figure 3).

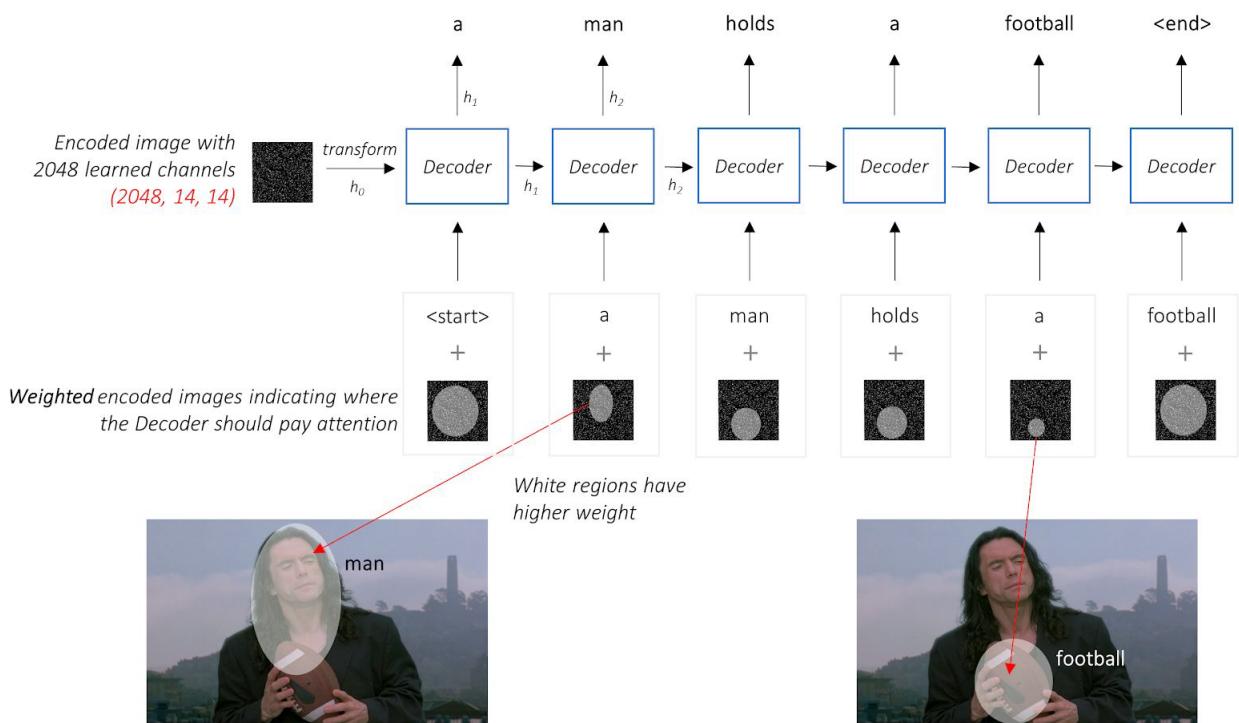


FIGURE 3 – Illustration de l’attention

Concernant les poids d’attention, ceux-ci sont appris à l’aide d’un réseau de neurones Fully-Connected qui prendra en entrée l’image d’origine (encodée) ainsi que l’activation précédente (car on souhaite que les parties pertinentes à considérer dans l’image dépendent justement des mots que le modèle a déjà prédit dans la séquence) (voir illustration en figure 4).

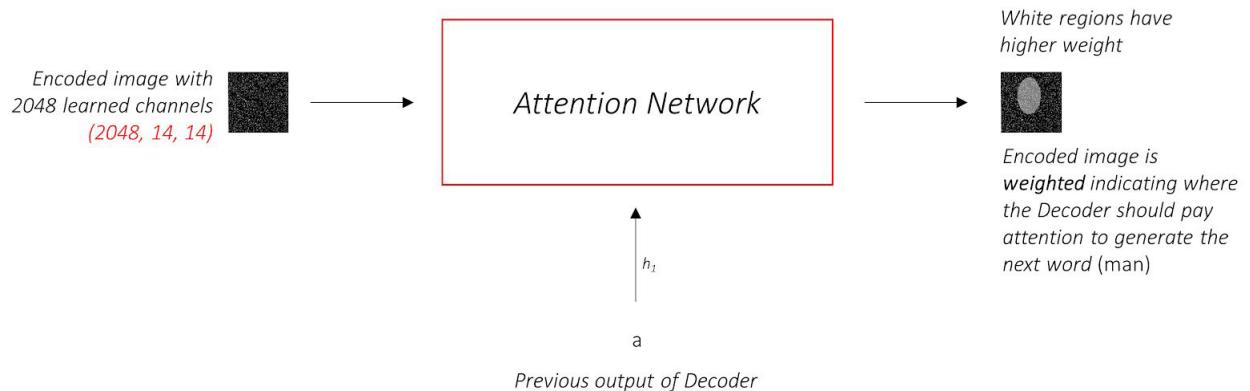


FIGURE 4 – Illustration du réseau d’attention

Ces 3 "briques" (Encodeur, Décodeur, Réseau d'Attention) sont mises ensemble dans une architecture illustrée en figure 5 et qui constitue le réseau final qui permettra de générer une description pour une image. Par ailleurs , sachant que la sortie du décodeur à chaque étape est un vecteur de probabilités (d'occurrence de chaque mot du vocabulaire), le réseau est entraîné à l'aide d'une fonction de perte d'entropie croisée.

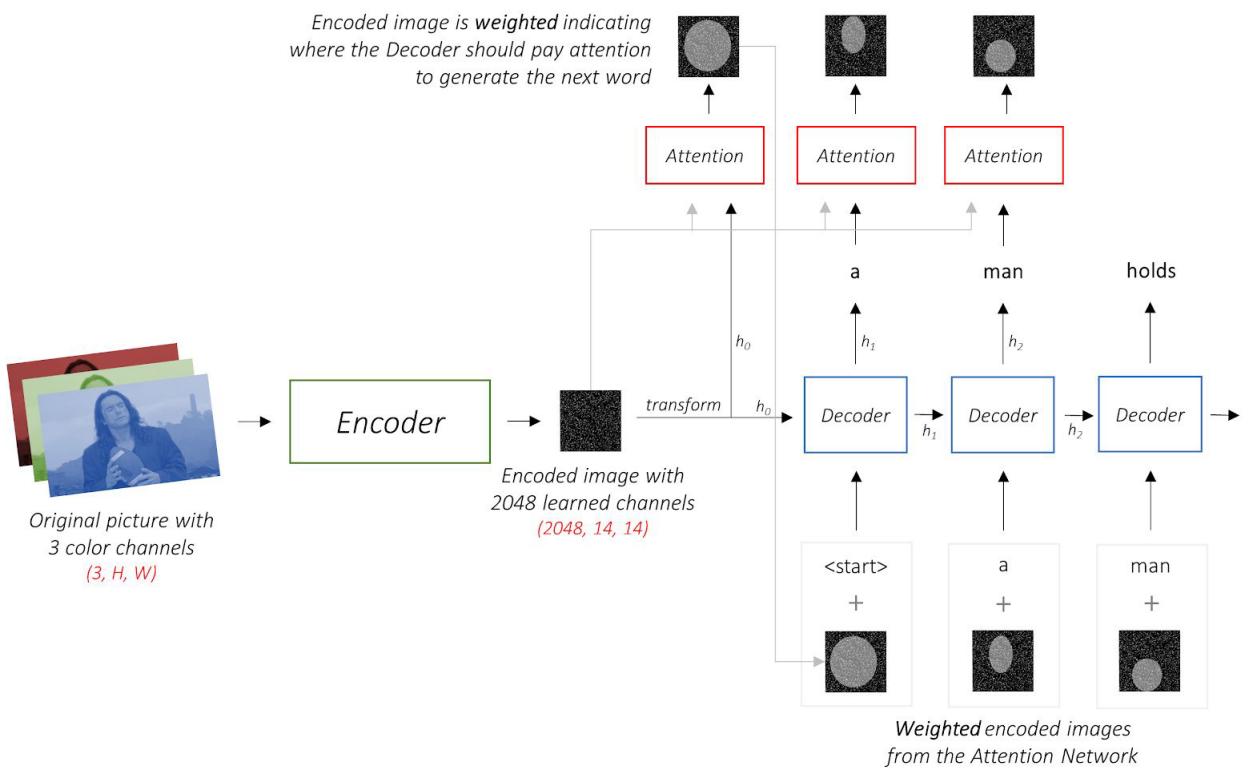


FIGURE 5 – Illustration de l’architecture complète du réseau

Détaillons à présent le Beam Search. Tout d’abord il convient de remarquer que le Beam Search n’est pas utilisé dans l’apprentissage du modèle. Il s’agit plutôt d’une stratégie utilisée pendant l’inférence. En effet, son but est de trouver la séquence optimale sans se limiter à la séquence composée du mot le plus probable de chaque étape dans

le décodeur. L'idée du Beam Search est de ne pas décider avant d'avoir fini de décoder complètement, et choisir la séquence qui a le score global le plus élevé dans un ensemble de séquences candidates. Le processus se déroule comme suit :

- Lors de la première étape de décodage, on conserve les k mots les plus probables.
- Pour chacun de ces k mots, on génère k mots suivants à l'aide du décodeur.
- On choisit les k meilleures combinaisons (premier mot + deuxième mot) parmi les k^2 possibles en ajoutant les scores (les outputs du décodeur).
- Pour chacune de ces k combinaisons (de taille 2), on génère k troisièmes mots à l'aide du décodeur et parmi toutes ces k^2 combinaisons possibles (de taille 3), on garde les k meilleures à l'aide des scores additionnés.
- On répète ce processus à chaque étape de décodage.
- À la fin, on obtient k séquences, et on choisit celle ayant le meilleur score global.

Le paramètre k décrit dans le procédé est appelé Beam Size et dans notre cas, nous fixerons sa valeur par défaut à 3 mais bien entendu, il s'agit d'un hyperparamètre qu'on peut optimiser si besoin.

Enfin, terminons en décrivant le processus d'évaluation du modèle. Comme expliqué précédemment, les sorties du décodeur à chaque étape sont un vecteur de probabilités. Une manière d'évaluer le modèle est donc la fonction d'encropie croisée qui est utilisée dans l'apprentissage (notamment parce qu'elle est différentiable). Cependant, cette fonction d'entropie croisée n'est pas optimale car deux phrases ayant un sens similaire peuvent être considérées comme différentes avec cette métrique si l'ordre des mots est différent. Par exemple, les phrases "There is a dog in the garden" et "A dog is in the garden" sont similaires mais si on les compare mot à mot, il y a un décalage. L'idée est donc d'avoir une métrique qui ne dépend pas de l'ordre des mots. Une métrique candidate est le Bleu-Score (compris entre 0 et 1) qui repose sur la fréquence d'apparition (dans la/les phrase(s) cible(s)) des mots, bigrammes, ..., N-grammes présents dans la phrase prédite. Nous utiliserons cette métrique dans le BE dans la phase d'évaluation et nous fixerons la valeur de N par défaut à 4 mais c'est un paramètre qu'on peut contrôler en fonction de l'exigence qu'on a envers l'exactitude du modèle.

2.1.2 Expérimentations et résultats

L'ensemble du code pour cette partie se situe dans le notebook **"BE1 _ Image _ Captioning-init.ipynb"**.

Le code est divisé en plusieurs parties où :

- On définit les fonctions utilitaires
- On télécharge les données d'apprentissage
- On définit une classe qui permettra d'itérer sur les données (sans les charger toutes en mémoire)
- On définit le modèle expliqué dans la partie précédente, à l'aide de Pytorch
- On entraîne le modèle
- Pour des images données, on trouve la phrase optimale avec le Beam Search (avec un Beam Size fixé à 5) et on visualise les poids d'attention à chaque étape dans le décodeur

- On évalue le modèle à l'aide du Bleu Score sur des phrases trouvées avec Beam Search (avec un Beam Size fixé à 3)

Voici un aperçu de l'entraînement du modèle (figure 6) :

```
3.0803 (3.0766) Top-5 Accuracy 73.990 (76.968)
Validation: [0/157]      Batch Time 0.281 (0.281)      Loss 3.4287 (3.4287)      Top-5 Accuracy
73.028 (73.028)
Validation: [100/157]    Batch Time 0.281 (0.279)      Loss 3.5263 (3.6282)      Top-5 Accuracy
71.392 (69.636)

* LOSS - 3.619, TOP-5 ACCURACY - 69.654, BLEU-4 - 0.16164032448536186

Epochs since last improvement: 1
```

FIGURE 6 – Architecture initiale - Aperçu de la fin de l'entraînement

Comme on peut le voir sur la figure 6, après 10 epochs, on obtient une Accuracy de 69,6 % et un Bleu-Score de 0.16 (sans Beam Search). Cela devrait nous donner des résultats décents si l'on se fie à l'Accuracy.

À présent, visualisons les résultats sur des images :

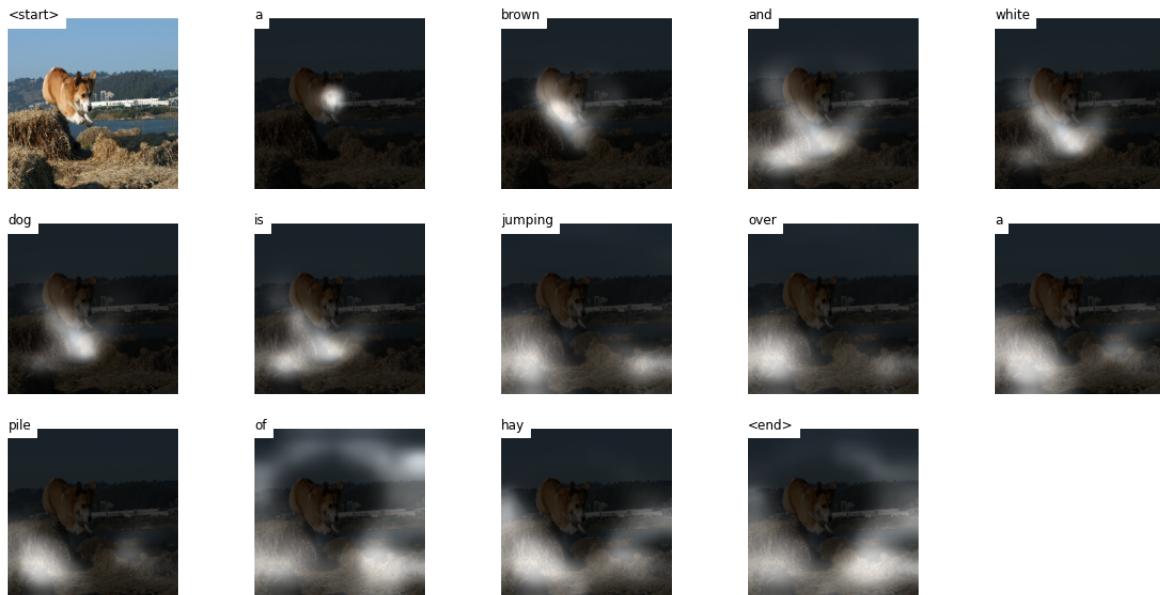


FIGURE 7 – Architecture initiale - Visualisation de l'attention (1)



FIGURE 8 – Architecture initiale - Visualisation de l'attention (2)

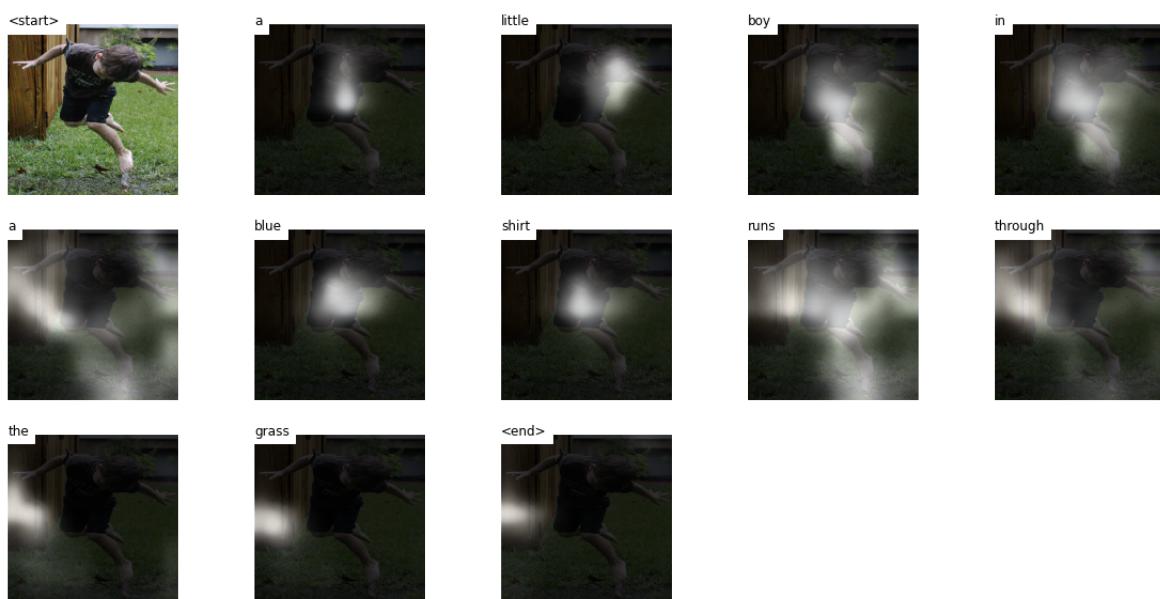


FIGURE 9 – Architecture initiale - Visualisation de l'attention (3)

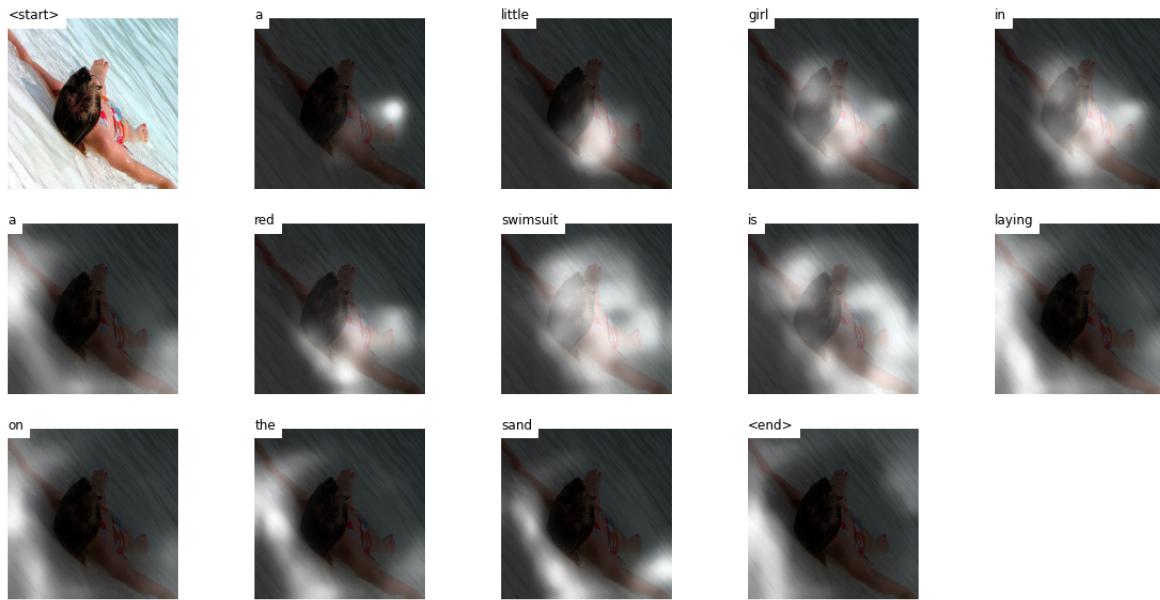


FIGURE 10 – Architecture initiale - Visualisation de l'attention (4)

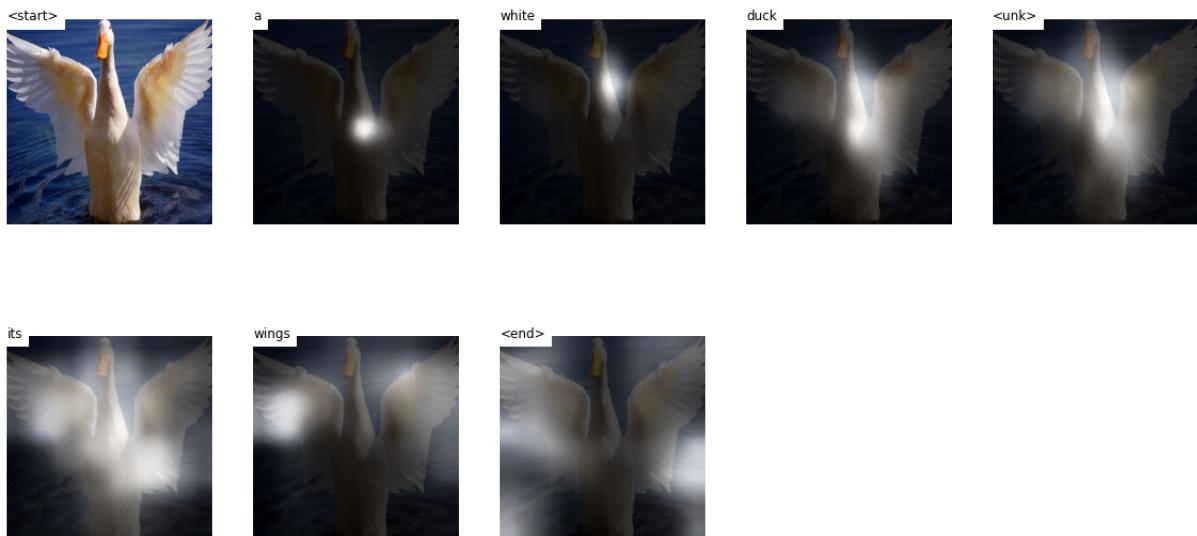


FIGURE 11 – Architecture initiale - Visualisation de l'attention (5)

En observant les figure 7 à 11, on remarque que les descriptions générées ne sont pas aberrantes (sauf peut-être pour la figure 8). Les régions mises en avant par les poids d'attention semblent aussi correspondre dans l'ensemble aux mots générés même si pour certaines images et certains mots, d'une part elles ne correspondent pas et d'autre part elles sont difficiles à interpréter. C'est le cas par exemple des mots comme "the", "in", "through", "a" pour lesquels il n'est pas évident de comprendre la pertinence des régions surlignées mais aussi les mots comme "shirt" où la région surlignée correspond plutôt au short dans l'image. Par ailleurs, pour l'image de la figure 11, le modèle génère le token "<unk>" ce qui n'est pas forcément un comportement souhaité.

```
EVALUATING AT BEAM SIZE 3: 100% |██████████| 5000/5000 [02:26<00:00, 34.21it/s]
BLEU-4 score @ beam size of 3 is 0.2216.
```

FIGURE 12 – Architecture initiale - BLEU Score

Comme visible sur la figure 12, on obtient un Bleu Score de 0.2216 avec un Beam Size fixé à 3. Il semble donc qu'il y ait encore une marge de progression pour le modèle.

2.2 Changement de l'encodeur

L'ensemble du code pour cette partie se situe dans le notebook **"BE1_Image_Captioning-vgg16.ipynb"**.

Maintenant, le but est de changer l'encodeur utilisé dans l'architecture du réseau pour essayer d'obtenir un changement de comportement intéressant. Pour ce faire, on va dans le code simplement remplacer ResNet par VGG16 et changer le paramètre **encoder_dim** par 512 au lieu de 2048.

En général, ResNet est considéré comme plus rapide et efficace que VGG16, car ils sont plus profonds mais contiennent moins de paramètres et arrivent assez bien à régler les problèmes habituels causés par la profondeur notamment avec les connexions résiduelles. Voici donc le résultat que l'on obtient avec un encodeur VGG16 sur 10 epochs :

```
2.9857 (2.8700) Top-5 Accuracy 79.947 (80.323)
Validation: [0/157]      Batch Time 0.245 (0.245)      Loss 3.9434 (3.9434)      Top-5 Accuracy
64.456 (64.456)
Validation: [100/157]    Batch Time 0.248 (0.250)      Loss 3.6893 (3.7058)      Top-5 Accuracy
66.205 (68.466)

* LOSS - 3.718, TOP-5 ACCURACY - 68.237, BLEU-4 - 0.14445839190140064

Epochs since last improvement: 4
```

FIGURE 13 – Encodeur VGG16 au lieu de ResNet - Aperçu de la fin de l'entraînement

On voit donc sur la figure 13 qu'on arrive cette fois à une accuracy de 68.24%, et un BLEU score de 0.14, ce qui est, comme on l'avait prévu, un score plus mauvais qu'obtenu avec l'ancien encodeur.

Observons maintenant les résultats obtenus par l'inférence avec ce nouvel encodeur :

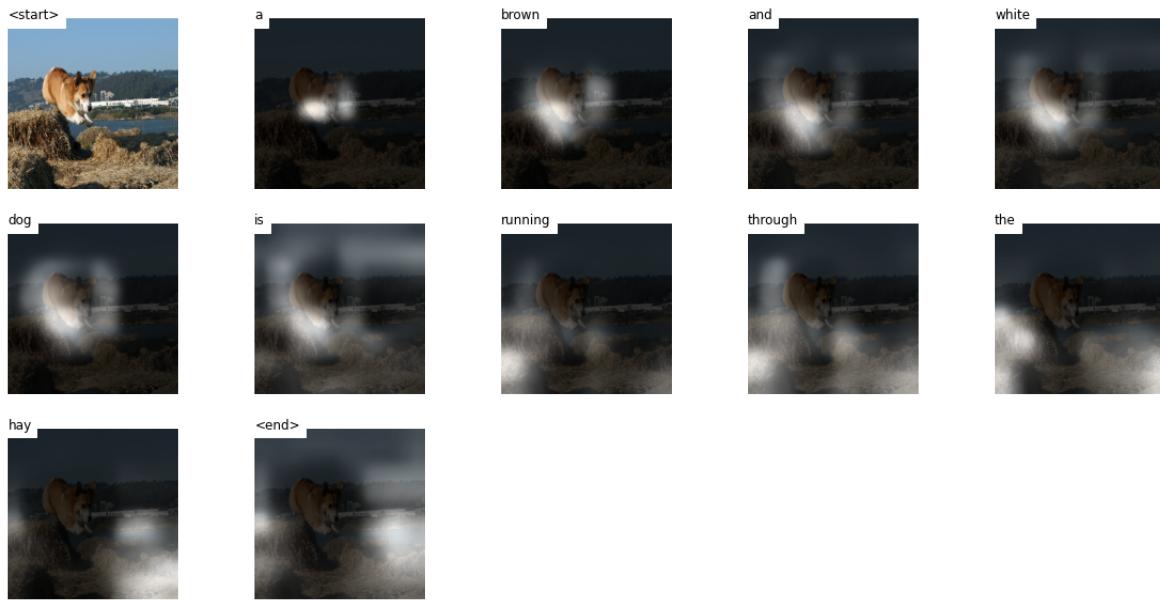


FIGURE 14 – Encodeur VGG16 au lieu de ResNet - Visualisation de l'attention (1)

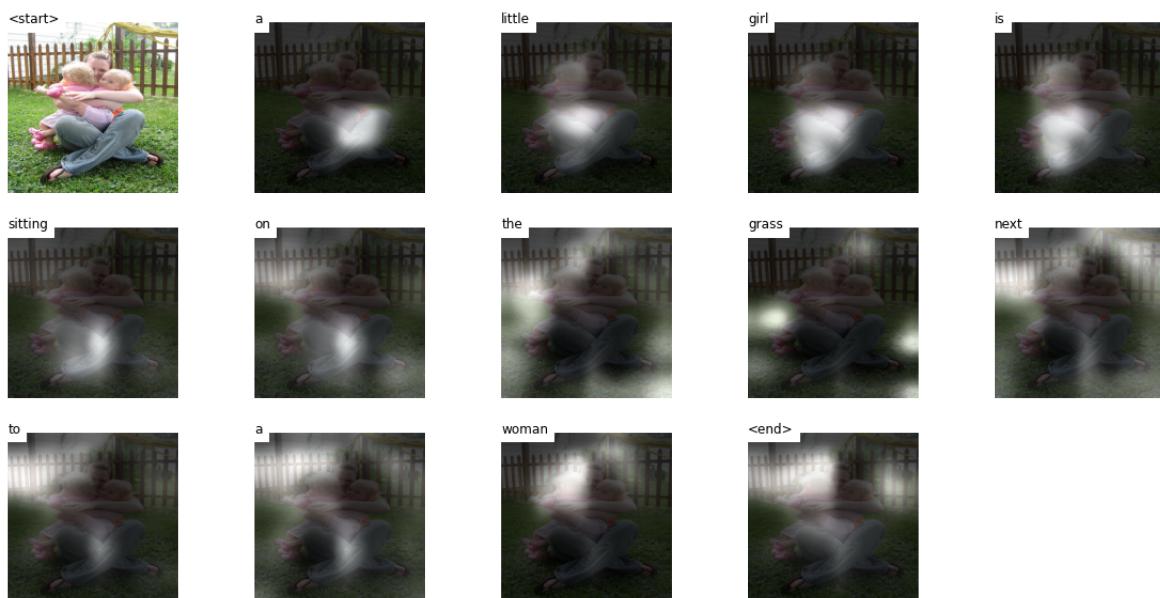


FIGURE 15 – Encodeur VGG16 au lieu de ResNet - Visualisation de l'attention (2)

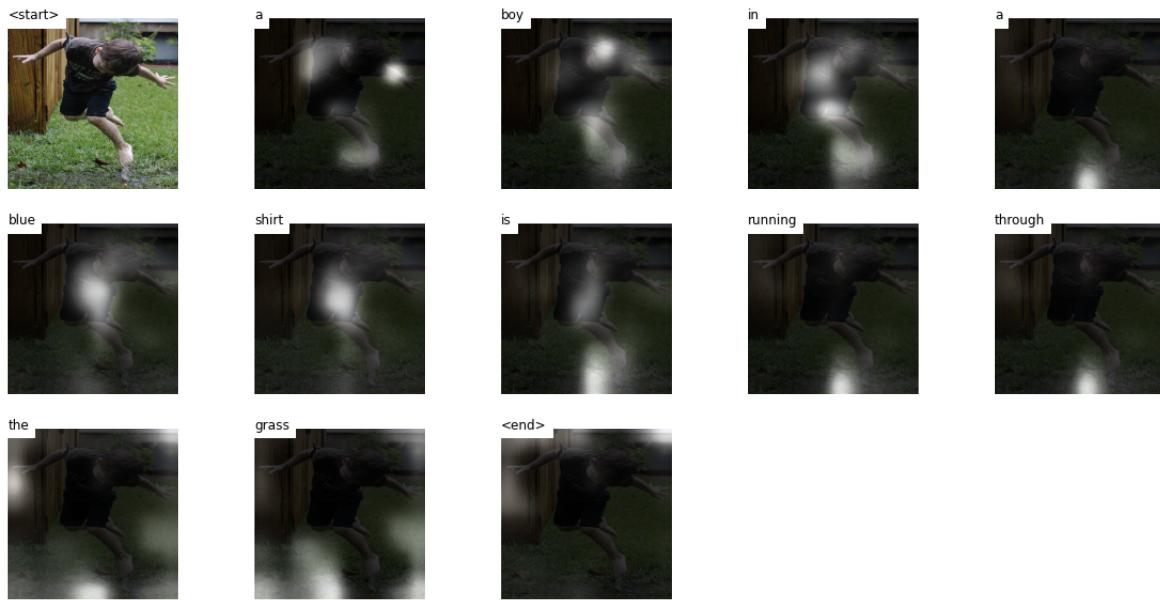


FIGURE 16 – Encodeur VGG16 au lieu de ResNet - Visualisation de l'attention (3)

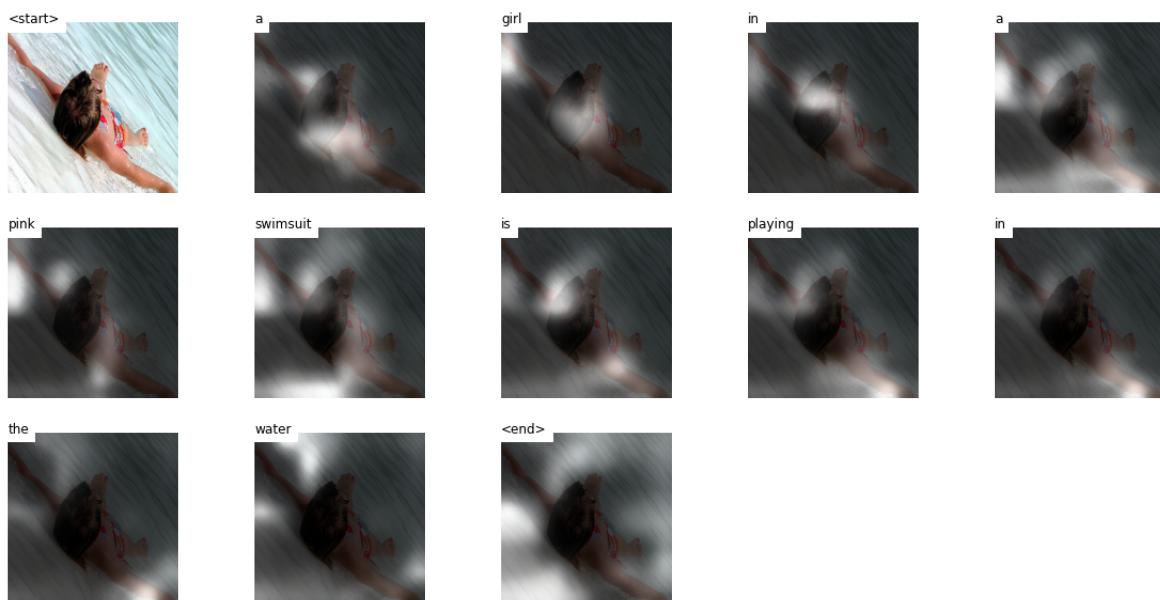


FIGURE 17 – Encodeur VGG16 au lieu de ResNet - Visualisation de l'attention (4)

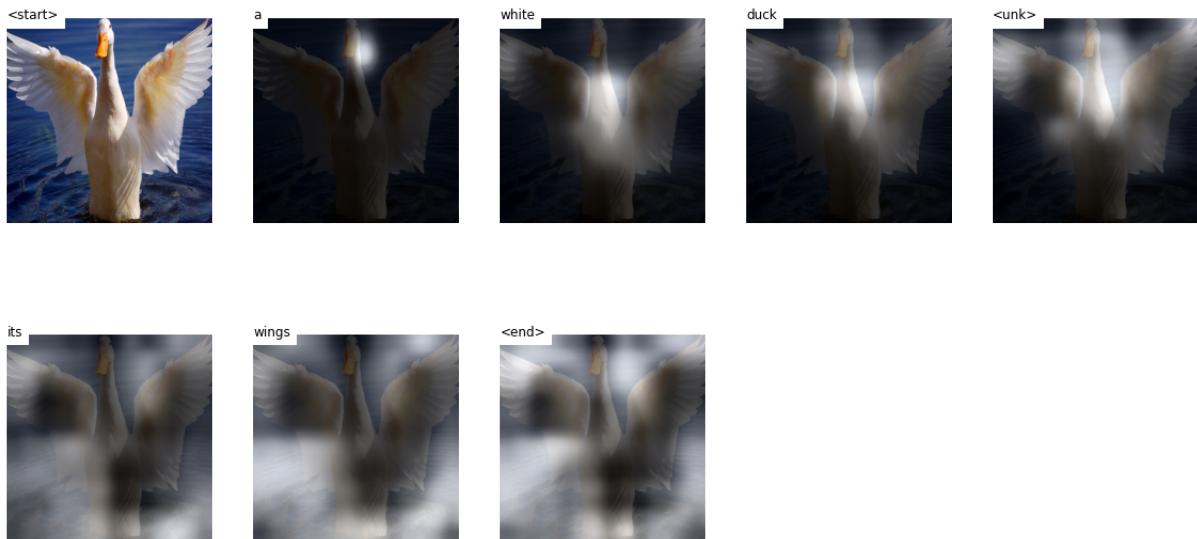


FIGURE 18 – Encodeur VGG16 au lieu de ResNet - Visualisation de l'attention (5)

Les résultats sont assez similaires à ceux obtenus avec le précédent encodeur. Sur la plupart des images, on voit que l'attention est assez bien représentée sur les objets concrets, comme pour le chien dans la figure 14, mais que les conjonctions ou autres mots de liaison n'ont pas beaucoup de sens. La figure 18 semble toujours causer problème au réseau, avec un `<unk>` qui est généré. La seule image où l'on observe une véritable différence est la figure 15, où la phrase générée est plus longue et plus complète.

```
EVALUATING AT BEAM SIZE 3: 100% |██████████| 5000/5000 [01:53<00:00, 43.92it/s]
```

```
BLEU-4 score @ beam size of 3 is 0.2082.
```

FIGURE 19 – Encodeur VGG16 au lieu de ResNet - BLEU Score

On arrive à un BLEU score de 0.2082 avec un beam size à 3. Comme on s'y attendait, il est légèrement moins bon que celui obtenu précédemment.

2.3 Changement du décodeur

L'ensemble du code pour cette partie se situe dans le notebook `"BE1_Image_Captioning-gru.ipynb"`.

On va réaliser la même expérience que précédemment, mais cette fois en changeant le décodeur. La cellule LSTM sera remplacée par une cellule GRU. La principale différence entre LSTM et GRU vient du nombre de portes. LSTM en a 3 : une d'entrée, une de sortie, et une pour oublier, alors que GRU n'en a que 2, une pour mettre à jour, et l'autre pour réinitialiser. En termes de performance, GRU est considéré comme plus rapide, mais moins précis sur de larges datasets, dû à son nombre de paramètres moins élevé. Par ailleurs, le réseau GRU n'a qu'une seul State (Hidden State) au lieu de 2 pour le LSTM (Hidden State et Cell State). Il faut donc changer cela partout dans le code (voir le notebook pour plus de détails sur les changements).

Observons les résultats, et voyons ce qui peut être dit sur cette affirmation :

```

2.8047 (2.8479) Top-5 Accuracy 80.818 (80.791)
Validation: [0/157]      Batch Time 0.252 (0.252)      Loss 3.6722 (3.6722)      Top-5 Accuracy
69.267 (69.267)
Validation: [100/157]    Batch Time 0.246 (0.251)      Loss 3.5174 (3.7326)      Top-5 Accuracy
73.316 (68.495)

* LOSS - 3.737, TOP-5 ACCURACY - 68.465, BLEU-4 - 0.1467105230830837

Epochs since last improvement: 4

```

FIGURE 20 – Décodeur GRU au lieu de LSTM - Aperçu de la fin de l’entraînement

Encore une fois, comme on s’y attendait, le réseau avait une performance légèrement meilleure avec un décodeur LSTM. La précision est maintenant de 68.47, et le score BLEU sans Beam Search est de 0.1467. Les performances ressemblent beaucoup à celles qui avaient été obtenues avec le changement de l’encodeur.

Observons maintenant les résultats obtenus via inférence :



FIGURE 21 – Décodeur GRU au lieu de LSTM - Visualisation de l’attention (1)

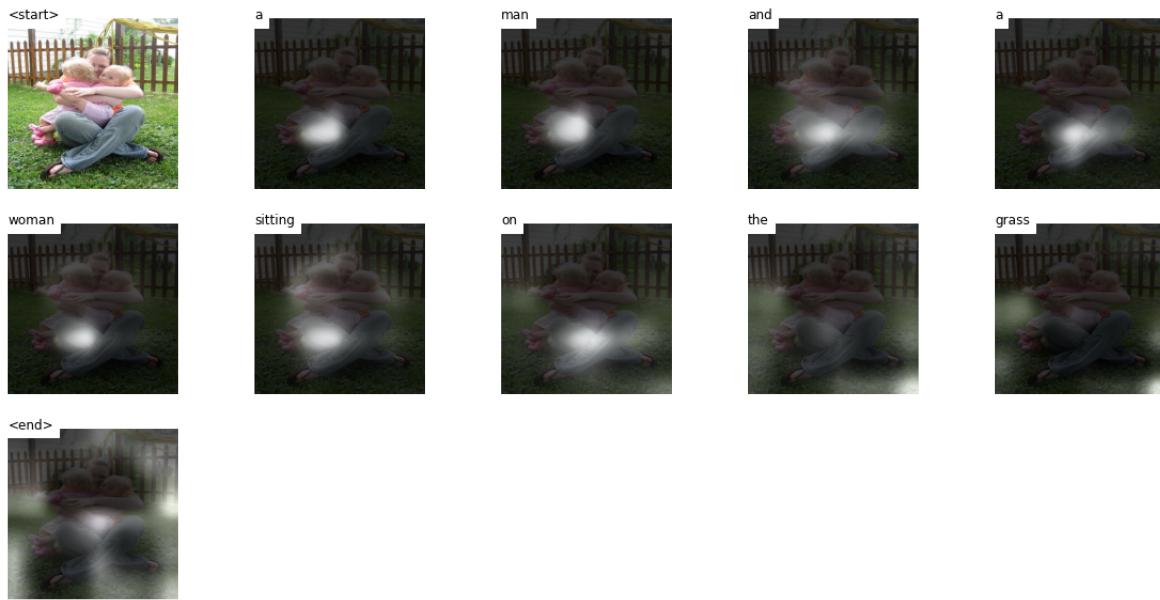


FIGURE 22 – Décodeur GRU au lieu de LSTM - Visualisation de l'attention (2)

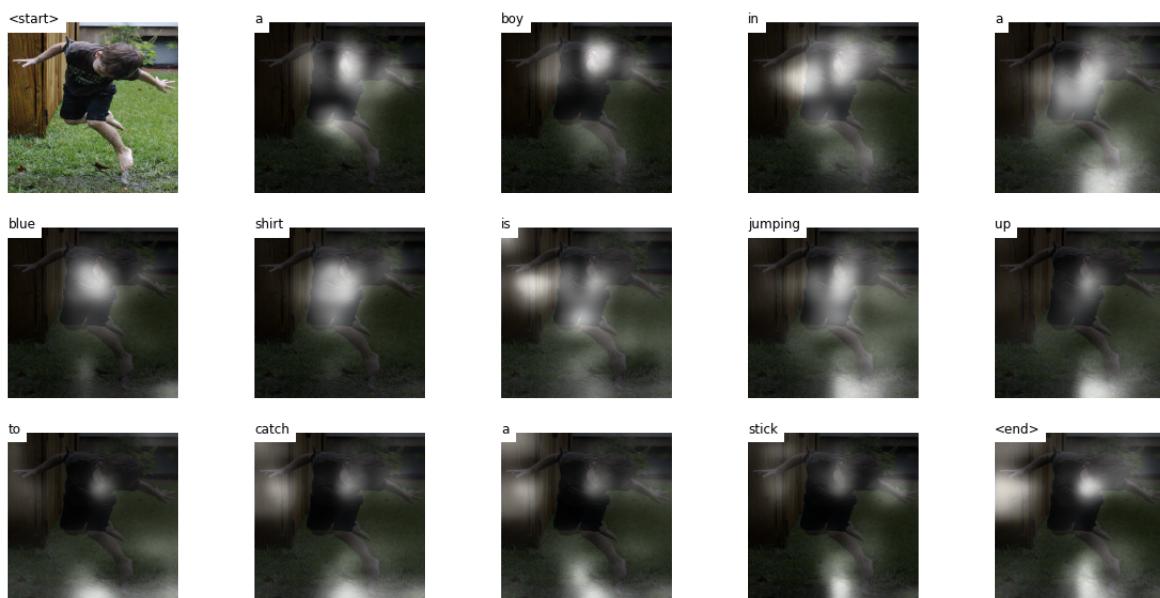


FIGURE 23 – Décodeur GRU au lieu de LSTM - Visualisation de l'attention (3)

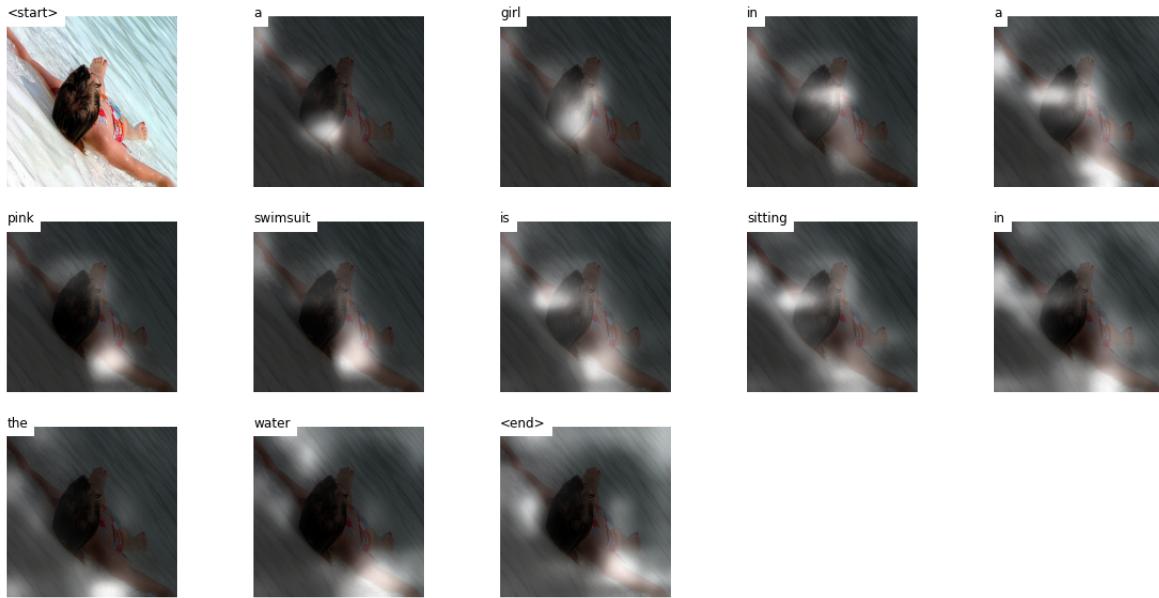


FIGURE 24 – Décodeur GRU au lieu de LSTM - Visualisation de l’attention (4)

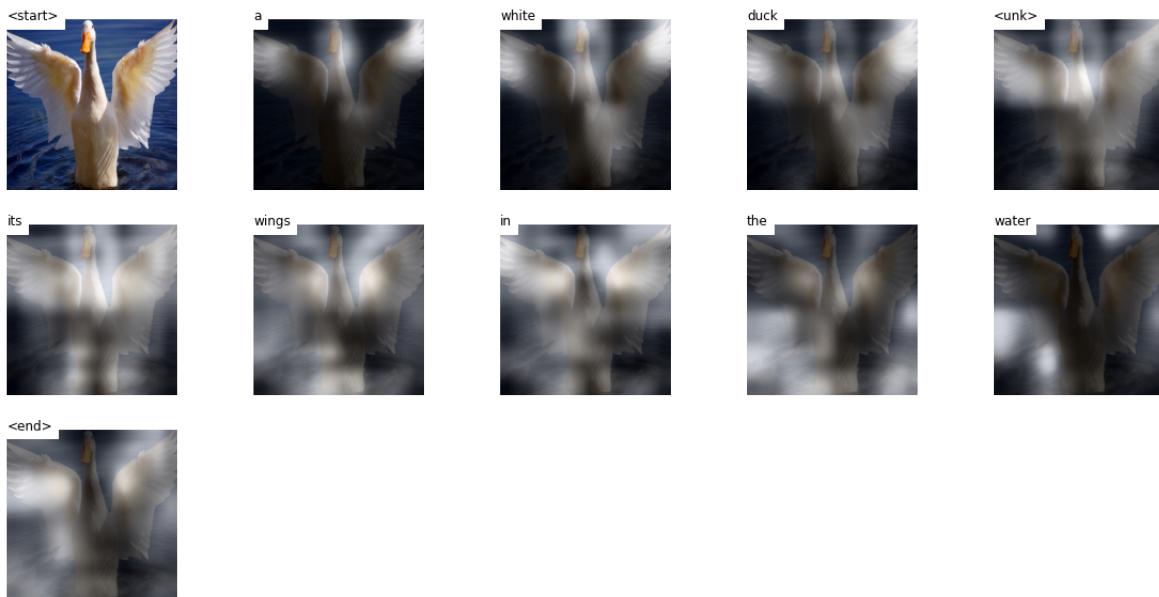


FIGURE 25 – Décodeur GRU au lieu de LSTM - Visualisation de l’attention (5)

Le changement de décodeur n'a pas non plus eu beaucoup d'effet sur les problèmes que l'on avait pu observer dans l'inférence. Les résultats sont toujours très similaires aussi, voire quasiment identiques à ceux obtenus par le premier réseau. On peut voir que sur la figure 21, la notion de couleur a totalement disparu de la description de l'image. Cependant, dans la figure 25, le réseau a su reconnaître l'eau dans l'arrière-plan qui n'avait jusque là pas été mis en avant.

```
EVALUATING AT BEAM SIZE 3: 100%|██████████| 5000/5000 [01:43<00:00, 48.36it/s]

BLEU-4 score @ beam size of 3 is 0.2208.
```

FIGURE 26 – Décodeur GRU au lieu de LSTM - BLEU Score

Avec un Beam Size de 3, le BLEU score obtenu est de 0.2208, ce qui est meilleur que lorsque du changement de l'encodeur, mais toujours moins bon que le réseau initial.

2.4 Ajout du Schedule Sampling

Dans la partie 2.1.1, nous avions évoqué le fait qu'à chaque étape dans le décodeur, les entrées étaient le mot précédent de la vérité terrain, l'activation de l'étape précédente et l'image encodée pondérée par les poids d'attention. L'utilisation du mot précédent de la vérité terrain (appelé aussi Teacher Forcing) est un bon moyen d'accélérer la convergence du modèle, cependant en pratique cela peut constituer une source de biais et d'instabilité car on donne au modèle pratiquement une partie de la réponse alors que sur des images complètement nouvelles, nous n'avons pas de vérité terrain et le modèle devra être quand même capable de prédire une description pertinente. Pour remédier à cela, une bonne stratégie est de reproduire au moment de l'apprentissage la situation où le modèle n'a pas accès à la vérité terrain. L'idéal serait que le modèle ait accès à la vérité terrain de temps en temps sur la base d'une probabilité et au fur et à mesure que le modèle est entraîné, on diminue l'accès aux mots de la vérité terrain. Cette stratégie s'appelle le Schedule Sampling.

Afin d'implémenter le Schedule Sampling, nous allons modifier la fonction de propagation en avant du décodeur. Voici comment elle était définie initialement :

```

1 def forward(self, encoder_out, encoded_captions, caption_lengths):
2     """
3         Forward propagation.
4
5         :param encoder_out: encoded images, a tensor of dimension
6             (batch_size, enc_image_size, enc_image_size, encoder_dim)
7         :param encoded_captions: encoded captions, a tensor of dimension
8             (batch_size, max_caption_length)
9         :param caption_lengths: caption lengths, a tensor of dimension
10            (batch_size, 1)
11         :return: scores for vocabulary, sorted encoded captions, decode
12             lengths, weights, sort indices
13         """
14
15         batch_size = encoder_out.size(0)
16         encoder_dim = encoder_out.size(-1)
17         vocab_size = self.vocab_size
18
19         # Flatten image
20         encoder_out = encoder_out.view(batch_size, -1, encoder_dim)  #
21             (batch_size, num_pixels, encoder_dim)
```

```

17     num_pixels = encoder_out.size(1)

18
19     # Sort input data by decreasing lengths; why? apparent below
20     caption_lengths, sort_ind = caption_lengths.squeeze(1).sort(dim=0,
21         ↓ descending=True)
22     encoder_out = encoder_out[sort_ind]
23     encoded_captions = encoded_captions[sort_ind]

24     # Embedding
25     embeddings = self.embedding(encoded_captions)  # (batch_size,
26         ↓ max_caption_length, embed_dim)

27     # Initialize LSTM state
28     h, c = self.init_hidden_state(encoder_out)  # (batch_size,
29         ↓ decoder_dim)

30     # We won't decode at the <end> position, since we've finished
31         ↓ generating as soon as we generate <end>
32     # So, decoding lengths are actual lengths - 1
33     decode_lengths = (caption_lengths - 1).tolist()

34     # Create tensors to hold word prediction scores and alphas
35     predictions = torch.zeros(batch_size, max(decode_lengths),
36         ↓ vocab_size).to(device)
36     alphas = torch.zeros(batch_size, max(decode_lengths),
37         ↓ num_pixels).to(device)

38     # At each time-step, decode by
39     # attention-weighting the encoder's output based on the decoder's
40         ↓ previous hidden state output
41     # then generate a new word in the decoder with the previous word
42         ↓ and the attention weighted encoding
43     for t in range(max(decode_lengths)):
44         batch_size_t = sum([l > t for l in decode_lengths])
45         attention_weighted_encoding, alpha =
46             ↓ self.attention(encoder_out[:batch_size_t], h[:batch_size_t])
47         gate = self.sigmoid(self.f_beta(h[:batch_size_t]))  # gating
48             ↓ scalar, (batch_size_t, encoder_dim)
49         attention_weighted_encoding = gate *
50             ↓ attention_weighted_encoding
51         h, c = self.decode_step(
52             torch.cat([embeddings[:batch_size_t, t, :],
53                 ↓ attention_weighted_encoding], dim=1),
54             (h[:batch_size_t], c[:batch_size_t]))  # (batch_size_t,
55                 ↓ decoder_dim)
56         preds = self.fc(self.dropout(h))  # (batch_size_t, vocab_size)

```

```

50         predictions[:batch_size_t, t, :] = preds
51         alphas[:batch_size_t, t, :] = alpha
52
53     return predictions, encoded_captions, decode_lengths, alphas,
54     ↵ sort_ind

```

Nous allons modifier les lignes 41 à 51 du code précédent comme suit (voir les commentaires dans le code pour plus de détails) :

```

1 def forward(self, encoder_out, encoded_captions,
2             ↵ caption_lengths, threshold):
3     # We have added a threshold parameter to control when we want to
4     ↵ use Teacher Forcing. If threshold is negative, we use Teacher
5     ↵ Forcing, else we sample a random number and use teacher
6     ↵ forcing based on the comparison between the random number and
7     ↵ the threshold.
8
9     # Same as lines 11 to 40 of the previous code chunk
10
11    for t in range(max(decode_lengths)):
12        batch_size_t = sum([l > t for l in decode_lengths])
13        attention_weighted_encoding, alpha =
14            ↵ self.attention(encoder_out[:batch_size_t], h[:batch_size_t])
15        gate = self.sigmoid(self.f_beta(h[:batch_size_t])) # gating
16            ↵ scalar, (batch_size_t, encoder_dim)
17        attention_weighted_encoding = gate *
18            ↵ attention_weighted_encoding
19        if threshold < 0 or t < 1:
20            #if threshold is negative or starting of the sequence :
21            h, c = self.decode_step(
22                torch.cat([embeddings[:batch_size_t, t, :],
23                          ↵ attention_weighted_encoding], dim=1),
24                (h[:batch_size_t], c[:batch_size_t])) #
25                    ↵ (batch_size_t, decoder_dim)
26        else:
27            sample = random.random()
28            if sample < threshold: #if random number is below the
29                ↵ threshold
30            h, c = self.decode_step(
31                torch.cat([embeddings[:batch_size_t, t, :],
32                          ↵ attention_weighted_encoding], dim=1),
33                (h[:batch_size_t], c[:batch_size_t])) #
34                    ↵ (batch_size_t, decoder_dim)
35        else:
36            #Previous pred logits :
37            previous_pred_logits = predictions[:batch_size_t, t-1,
38                ↵ :]

```

```

25             #Previous pred tokens (indices):
26     previous_pred_probs =
27         ↳ F.log_softmax(previous_pred_logits, dim=1)
28         ↳ #compute softmax
29     previous_pred =
30         ↳ torch.argmax(previous_pred_probs, dim=1) #take
31         ↳ argmax to find index
32     #Compute embeddings for sampled tokens
33     sampled_previous_pred_embedding =
34         ↳ self.embedding(previous_pred)
35     # Compute the next step outputs
36     h, c = self.decode_step(
37         torch.cat([sampled_previous_pred_embedding,
38             ↳ attention_weighted_encoding], dim=1),
39         (h[:batch_size_t], c[:batch_size_t])) #
40             ↳ (batch_size_t, decoder_dim)
41     preds = self.fc(self.dropout(h)) # (batch_size_t, vocab_size)
42     predictions[:batch_size_t, t, :] = preds
43     alphas[:batch_size_t, t, :] = alpha

```

Comme on peut le voir dans le code précédent, de temps en temps, au lieu de donner le mot précédent de la vérité terrain à l'étape suivante, nous donnons l'Embedding du mot le plus probable parmi les sorties du modèle. L'idéal serait même d'échantillonner un mot au hasard parmi tous les mots du vocabulaire en se basant sur la distribution de probabilités sorties par le modèle. Cependant, étant donné que nous entraînons le modèle pour 10 epochs seulement, les distributions de probabilité ne sont pas forcément fiables et donner directement le mot le plus probable serait en quelque sorte équivalent à choisir un mot au hasard.

Voyons à présent comment la fonction **forward** du décodeur sera utilisée à l'intérieur de la fonction **train** du modèle global. Tout d'abord, nous définissons une valeur de seuil de départ, une valeur de seuil de fin et le taux de décroissance exponentielle du seuil (il s'agit de variables qui sont globales) :

```

1  eps_start = 1 #Starting threshold
2  eps_end = 0.5 #Ending threshold
3  eps_decay = (math.log(eps_start) - math.log(eps_end))/epochs
    ↳ #Threshold exponential decay rate

```

Comme on peut le voir, nous commençons par un seuil de 1, c'est-à-dire qu'au début de l'apprentissage, nous utilisons uniquement le Teacher Forcing et nous diminuons cette valeur avec une décroissance exponentielle à chaque epoch pour atteindre une valeur de 0.5 à la fin des 10 epochs. Nous avons choisi 0.5 comme valeur finale car étant donné que nous entraînons le modèle seulement pour 10 epochs, se fier aux sorties du modèle plus souvent que la vérité terrain n'est pas forcément idéal. Un scénario 50-50 paraît donc comme un bon compromis. La fonction **train** est modifiée comme suit pour tenir compte

du seuil :

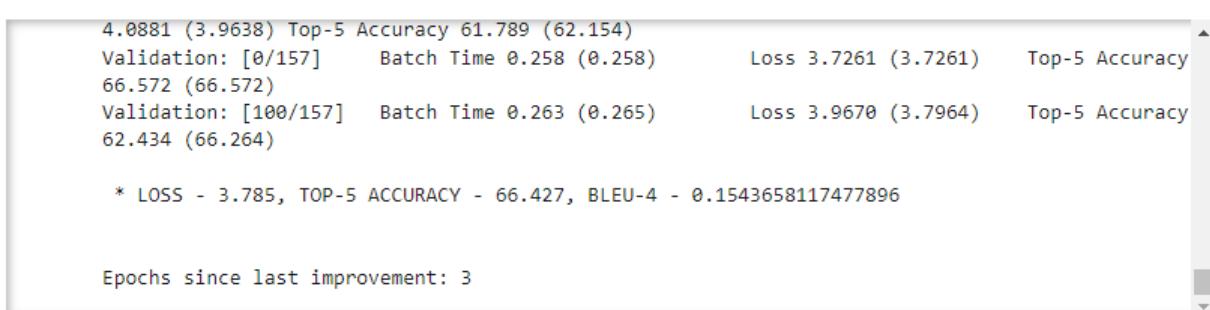
```

1 # .....
2 # .....
3 threshold = eps_start*math.exp(-eps_decay*epoch)
4 print(f'Threshold for epoch {epoch}/{epochs} : {threshold:.2f}')
5
6 # Batches
7 for i, (imgs, caps, caplens) in enumerate(train_loader):
8     data_time.update(time.time() - start)
9
10    # Move to GPU, if available
11    imgs = imgs.to(device)
12    caps = caps.to(device)
13    caplens = caplens.to(device)
14
15    # Forward prop.
16    imgs = encoder(imgs)
17    scores, caps_sorted, decode_lengths, alphas, sort_ind =
18        ↪ decoder(imgs, caps, caplens, threshold)
19
20    # ... the rest is the same

```

L'ensemble du code pour cette partie se situe dans le notebook
"BE1_Image_Captioning-schedule-sampling.ipynb".

Voyons à présent l'impact du Schedule Sampling sur la performance.



	Validation: [0/157]	Batch Time	Loss	Top-5 Accuracy
4.0881 (3.9638)	Top-5 Accuracy	61.789 (62.154)	3.7261 (3.7261)	66.572 (66.572)
66.434 (66.264)	Validation: [100/157]	Batch Time 0.263 (0.265)	Loss 3.9670 (3.7964)	Top-5 Accuracy
* LOSS - 3.785, TOP-5 ACCURACY - 66.427, BLEU-4 - 0.1543658117477896				
Epochs since last improvement: 3				

FIGURE 27 – Schedule sampling - Aperçu de la fin de l'entraînement

Comme on peut le voir sur la figure 27, l'Accuracy et le Bleu Score (sans Beam Search) après 10 epochs ont diminué par rapport aux résultats à la fin de l'entraînement sans Schedule Sampling (figure 6). Cela est normal car le modèle n'a pas accès tout le temps à la vérité terrain au moment de l'apprentissage. La convergence est donc plus lente.

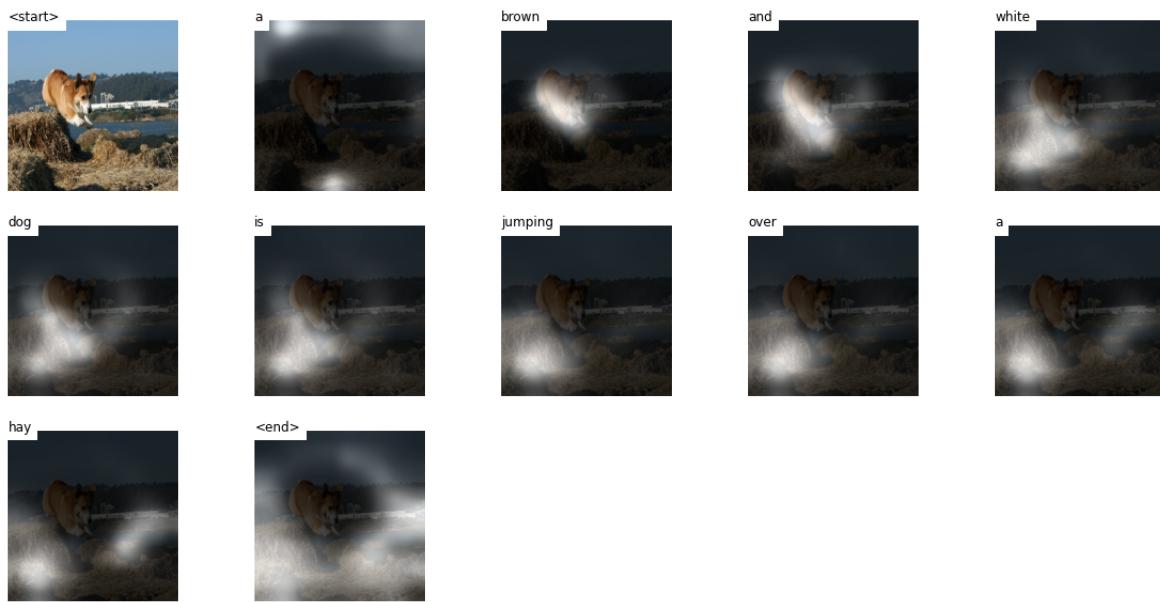


FIGURE 28 – Schedule sampling - Visualisation de l'attention (1)

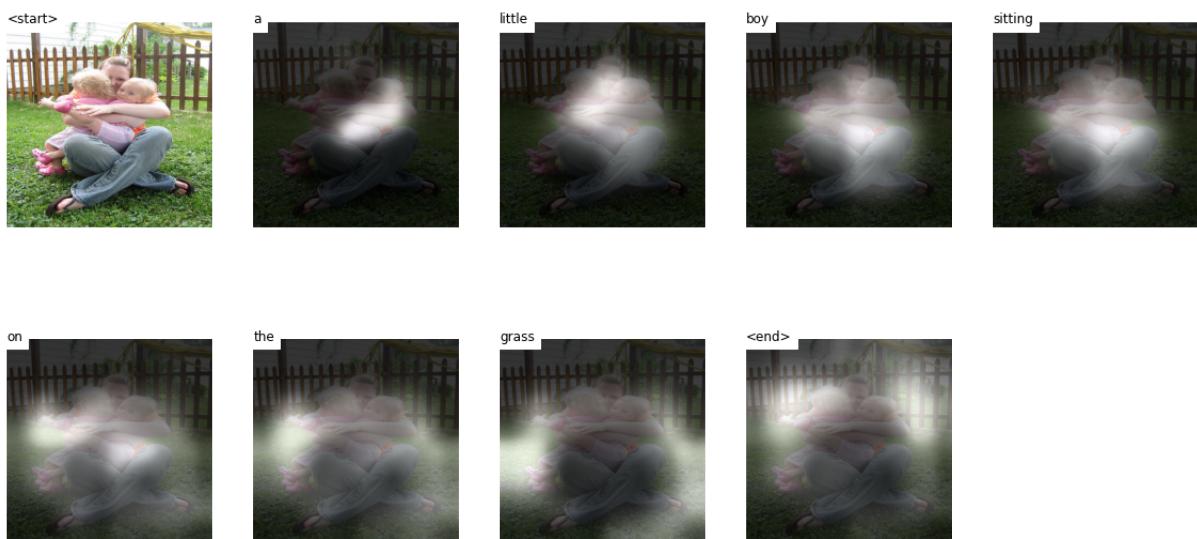


FIGURE 29 – Schedule sampling - Visualisation de l'attention (2)



FIGURE 30 – Schedule sampling - Visualisation de l'attention (3)

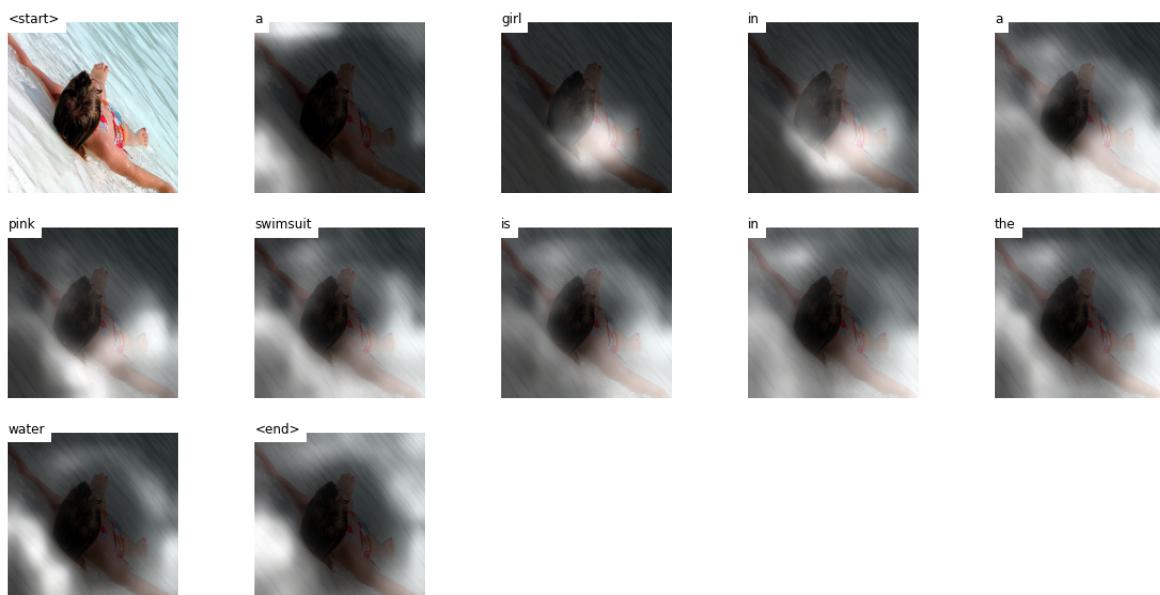


FIGURE 31 – Schedule sampling - Visualisation de l'attention (4)

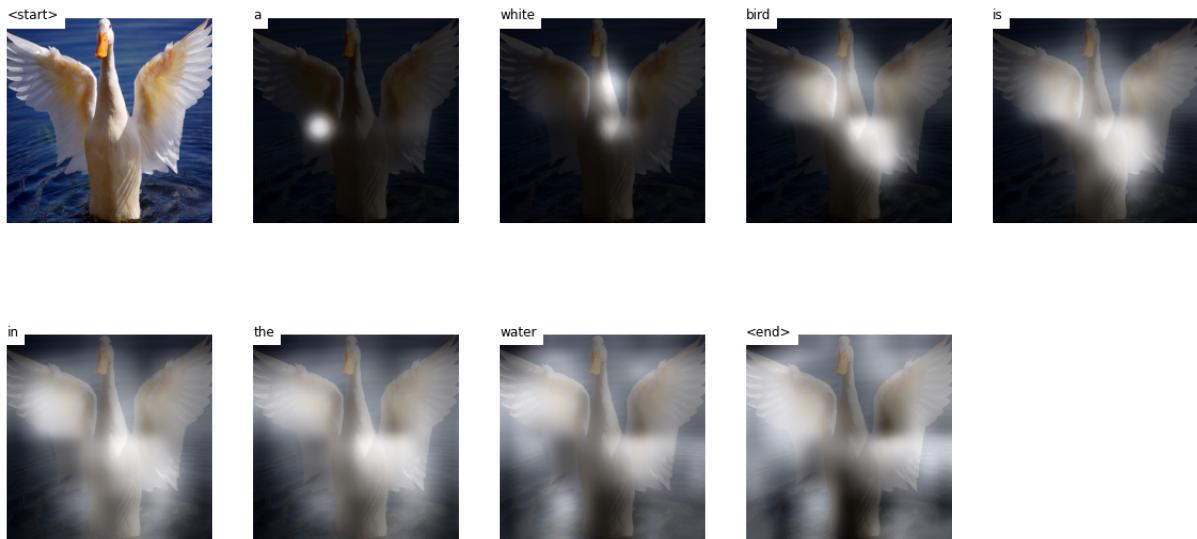


FIGURE 32 – Schedule sampling - Visualisation de l'attention (5)

En ce qui concerne les poids d'attention (figures 28 à 32), il n'y a pas de changement majeur à part le fait que les descriptions générées semblent être plus courtes et moins verbeuses et cela peut diminuer le risque d'erreurs même si le modèle devient plus "conservateur".

```
EVALUATING AT BEAM SIZE 3: 100% |██████████| 5000/5000 [02:18<00:00, 36.14it/s]
```

```
BLEU-4 score @ beam size of 3 is 0.2360.
```

FIGURE 33 – Schedule sampling - BLEU Score

Enfin, la figure 33 montre que le Schedule Sampling a bien eu son effet. Le Bleu Score (avec Beam Search) a augmenté de 1.4 % (par rapport aux résultats de la partie 2.1.2), ce qui signifie que le modèle est plus stable sur de nouvelles données (non utilisées pour l'entraînement) comme nous l'attendions.

3 Conclusion

À travers ce BE, il aura été possible d'étudier un réseau, son architecture, et ses méthodes d'inférence pour le problème de l'Image Captioning. Il a aussi été possible d'expérimenter sur son architecture en effectuant divers changements, et de voir leurs impacts en analysant les performances du réseau avec divers indicateurs. Au final, le réseau initial (constitué d'un Encodeur ResNet101, d'un décodeur LSTM et mécanisme d'attention) est resté le plus performant et la stratégie du Schedule Sampling a permis d'améliorer les résultats.