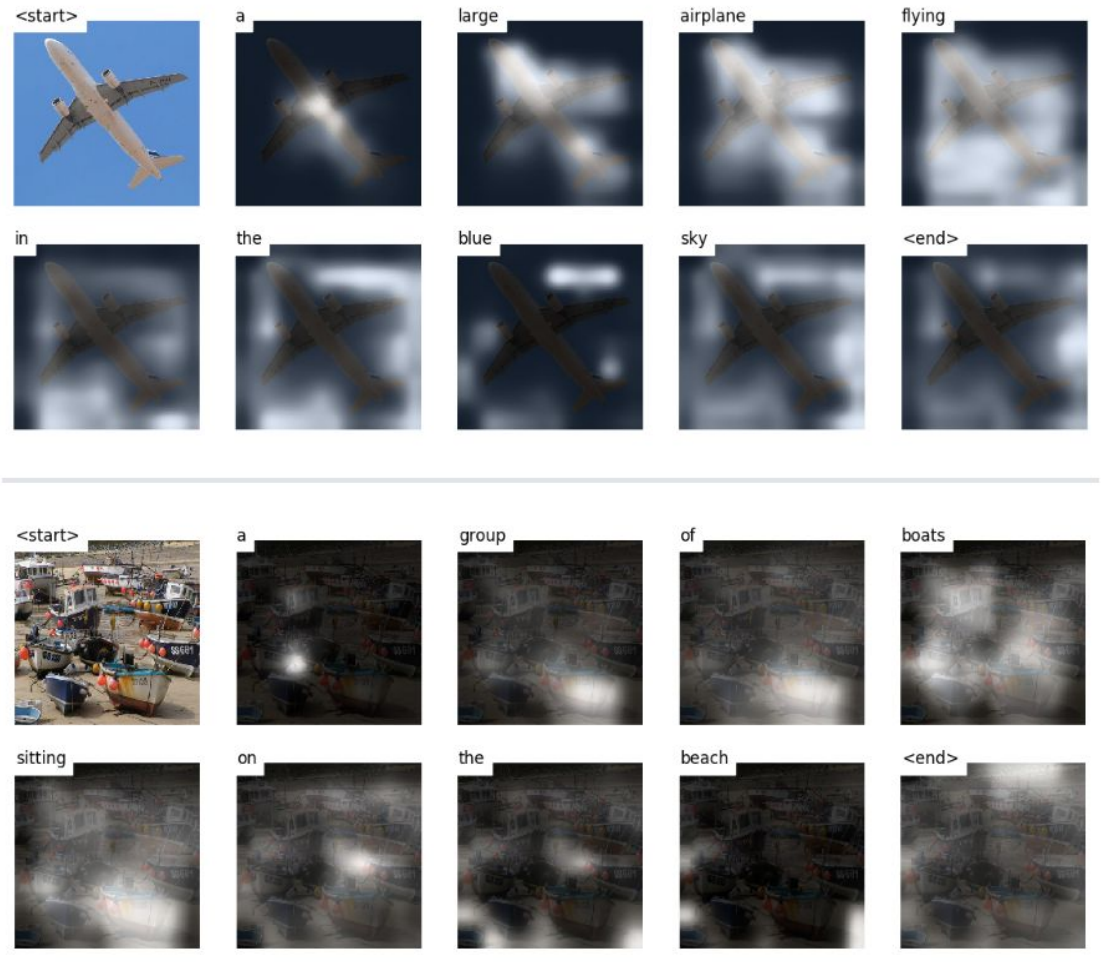
BE1- Image Captioning[1]

Show, Attend and Tell

In this BE, we aim to study RNN and the attention mechanism and learn a model that can generate a descriptive caption for an input image based on the main ideas from the paper Show, Attend and Tell by Xu *et al.*, 2015. The authors' original implementation can be found here. Specifically, the learned model generates a caption for a given input image, word by word, while shifting its gaze across the image and focusing its attention on the part of the image the most relevant to the word it is going to utter next.

Here are some captions generated on test images not seen during training or validation:



[1] This BE is adapted from the tutorial on image captioning by Sagar Vinodababu

1. Concepts

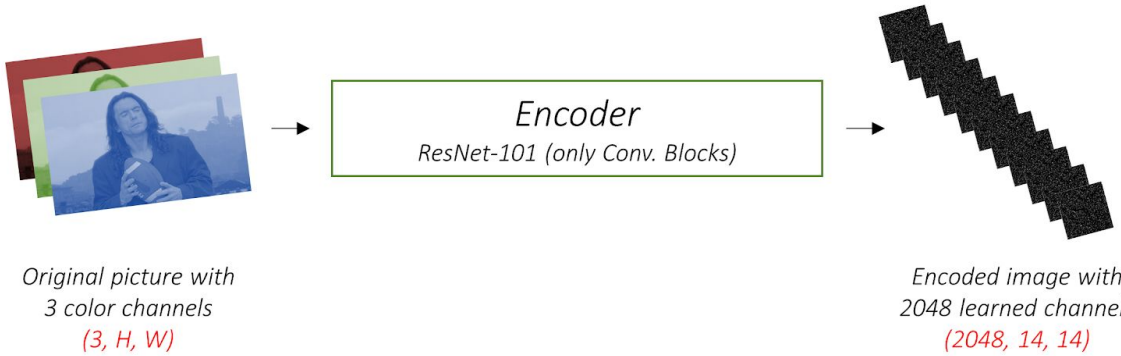
- **Image captioning**.
- **Encoder-Decoder architecture**. Typically, a model that generates sequences will use an Encoder to encode the input into a fixed form and a Decoder to decode it, word by word, into a sequence.
- **Attention**. The use of Attention networks is widespread in deep learning, and with good reason. This is a way for a model to choose only those parts of the encoding that it thinks is relevant to the task at hand. The same mechanism you see employed here can be used in any model where the Encoder's output has multiple points in space or time. In image captioning, you consider some pixels more important than others. In sequence to sequence tasks like machine translation, you consider some words more important than others.
- **Transfer Learning**. This is when you borrow from an existing model by using parts of it in a new model. This is almost always better than training a new model from scratch (i.e., knowing nothing). As you will see, you can always fine-tune this second-hand knowledge to a specific task at hand. Using pretrained word embeddings is a dumb but valid example. For our image captioning problem, we will use a pretrained Encoder, and then fine-tune it as needed.
- **Beam Search**. This is where you don't let your Decoder be lazy and simply choose the words with the *best* score at each decode-step. Beam Search is useful for any language modeling problem because it finds the most optimal sequence.

2. Overview

In this section, we present the overview of the target model.

Encoder

The encoder aims to encode an input image with 3 color channels into a smaller image which is a summary representation of the visual content of the input image. For this purpose, we are using an existing CNN, e.g., ResNet 101, already pretrained using ImageNet classification task. Because of its outstanding performance, it is expected that such a model can capture the essence of the visual content of images.



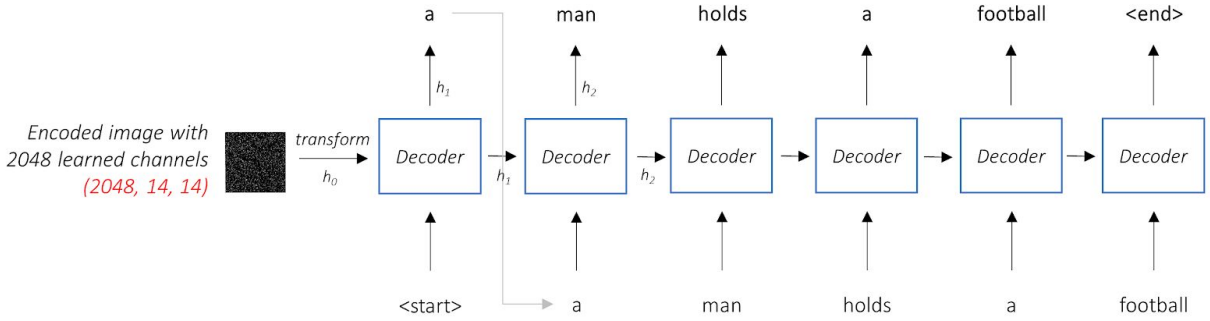
Here comes in the concept of transfer learning. Instead of training an encoder from scratch, we are using an existing pretrained model and fine-tune it to improve the performance.

CNN models progressively create smaller and smaller representations of the original image, and each subsequent representation is more “learned” with a greater number of channels. The final encoding produced by the ResNet-101 encoder has a size of 14x14 with 2048 channels.

Decoder

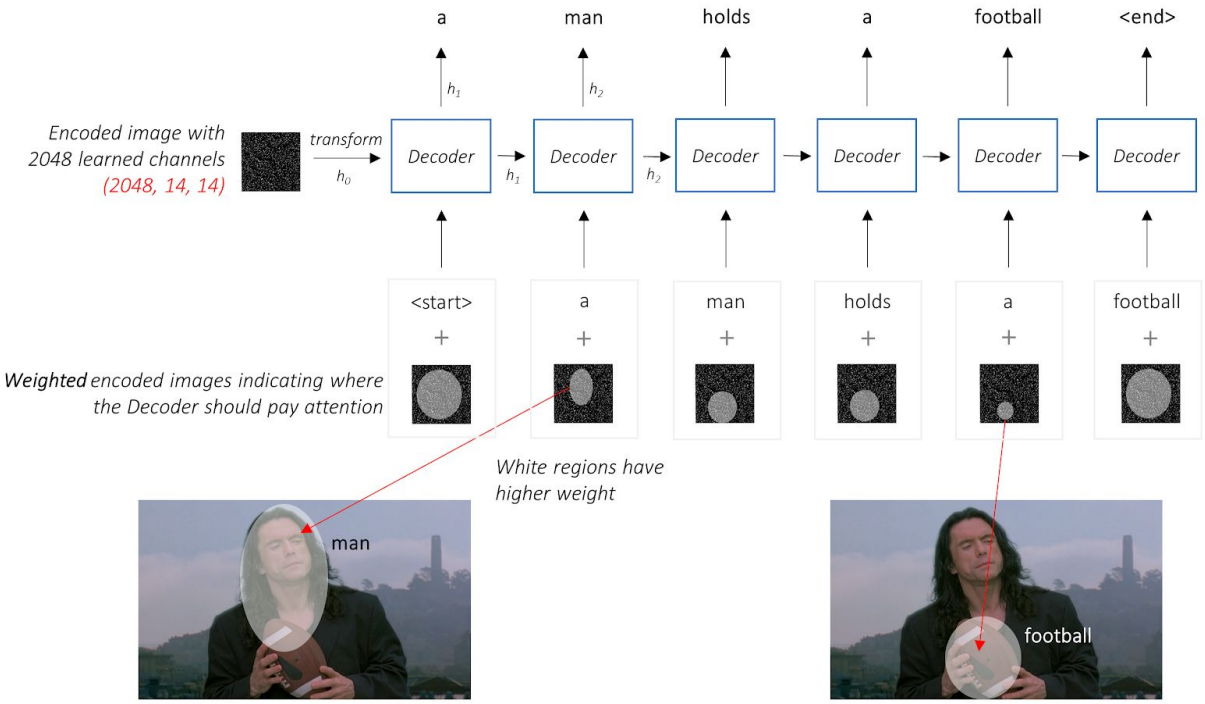
The decoder’s role is to look at the encoded image and generate a caption word by word. Because we are generating a sequence of words, we are using a Recurrent Neural Network (RNN), specifically a LSTM.

In a typical setting without Attention, we could simply average the encoded image across all pixels and feed it, with or without a linear transformation, into the Decoder as its first hidden state and generate the caption. Each predicted word is used to generate the next one.



In a setting with Attention, we want the Decoder to be able to look at different parts of the image at different points in the word sequence. For example, while generating the word *football* in the

caption "*a man holds a football*", the Decoder would know to focus on the football within the image.



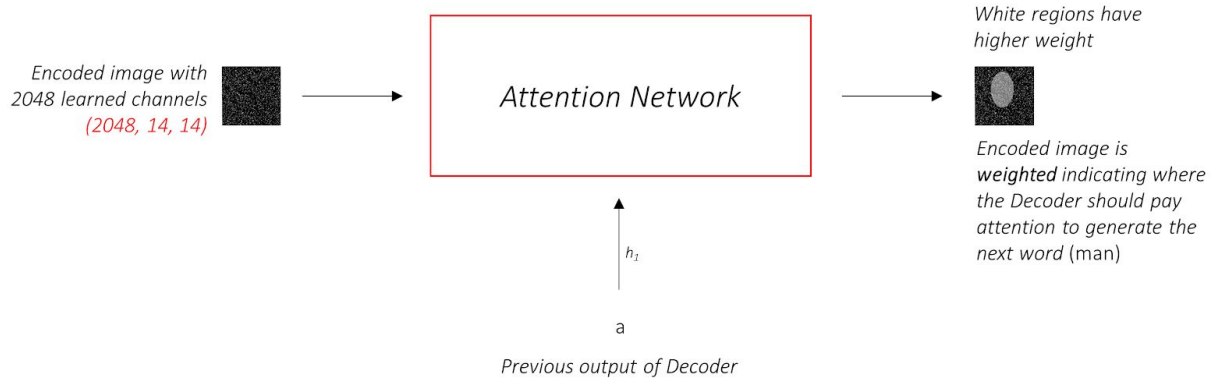
Instead of the simple average, we use the *weighted* average across all pixels, with the weights of the important pixels being greater. This weighted representation of the image can be concatenated with the previously generated word at each step to generate the next word.

Attention

The Attention network computes these weights.

Intuitively, how would you estimate the importance of a certain part of an image? You would need to be aware of the sequence you have generated *so far*, so you can look at the image and decide what needs describing next. For example, after you mention `a man`, it is logical to declare that he is `holding a football`.

This is exactly what the Attention mechanism does – it considers the sequence generated thus far, and *attends* to the part of the image that needs describing next.



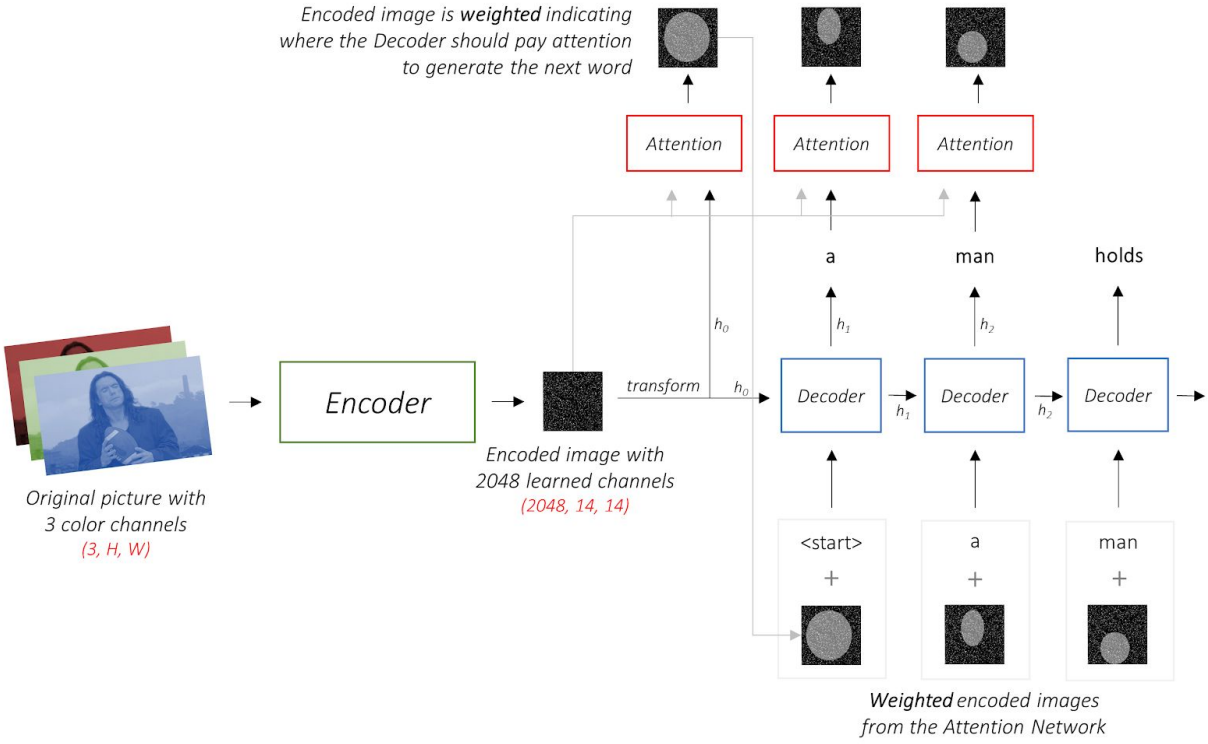
We will use *soft* Attention, where the weights of the pixels add up to 1. If there are `P` pixels in our encoded image, then at each timestep `t` –

$$\sum_{p}^{P} \alpha_{p,t} = 1$$

You could interpret this entire process as computing the probability that a pixel is *the* place to look to generate the next word.

Putting it all together

It might be clear by now what our combined network looks like.



- Once the Encoder generates the encoded image, we transform the encoding to create the initial hidden state `h` (and cell state `c`) for the LSTM Decoder.
- At each decode step,
  - the encoded image and the previous hidden state is used to generate weights for each pixel in the Attention network.
  - the previously generated word and the weighted average of the encoding are fed to the LSTM Decoder to generate the next word.

Beam Search

We use a linear layer to transform the Decoder's output into a score for each word in the vocabulary.

The straightforward – and greedy – option would be to choose the word with the highest score and use it to predict the next word. But this is not optimal because the rest of the sequence hinges on that first word you choose. If that choice isn't the best, everything

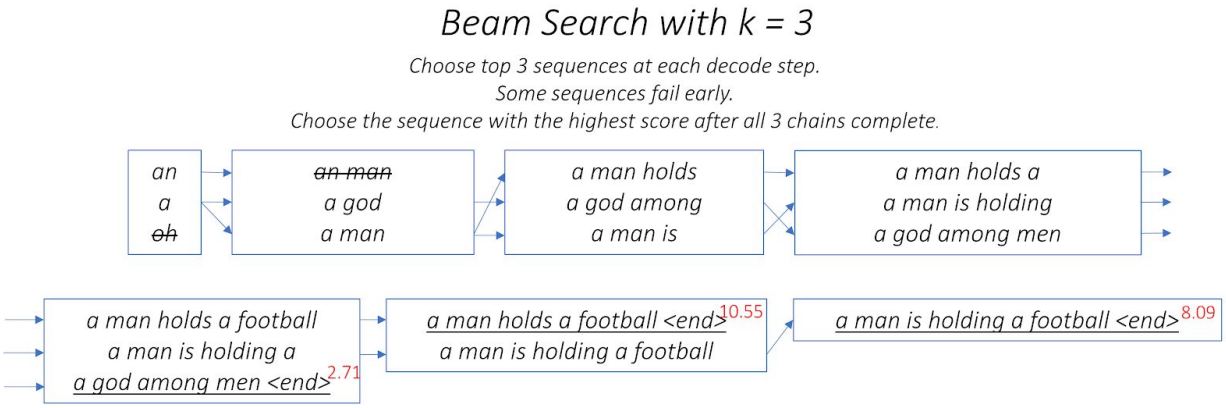
that follows is sub-optimal. And it's not just the first word – each word in the sequence has consequences for the ones that succeed it.

It might very well happen that if you'd chosen the *third* best word at that first step, and the *second* best word at the second step, and so on... *that* would be the best sequence you could generate.

It would be best if we could somehow *not* decide until we've finished decoding completely, and choose the sequence that has the highest *overall* score from a basket of candidate sequences.

Beam Search does exactly this.

- At the first decode step, consider the top k candidates.
- Generate k second words for each of these k first words.
- Choose the top k [first word, second word] combinations considering additive scores.
- For each of these k second words, choose k third words, choose the top k [first word, second word, third word] combinations.
- Repeat at each decode step.
- After k sequences terminate, choose the sequence with the best overall score.



As you can see, some sequences (striked out) may fail early, as they don't make it to the top `k` at the next step. Once `k` sequences (underlined) generate the `<end>` token, we choose the one with the highest score.

3. Work to do

Understanding and learning

In this part, you are asked to understand the architecture detailed in section 2 and the corresponding Python code. This code is provided under the form of Python files ".py" and a Python notebook.

We recommend that you use the notebook and run it with Google Colaboratory (https://colab.research.google.com) which allows you to run your code on GPU once you are connected with your Google ids. Be sure, to select a GPU session before running your code (Menu Exécution->Modifier le type d'exécution->Accélérateur matériel: GPU).

But if you prefer, it is possible to use the GPU servers at ECL. This solution requires that you use a VPN to connect to the server. Information to install and use a VPN can be found at: https://dsi.ec-lyon.fr/reseau-et-serveurs/service-vpn/comment-acceder-au-service-vpn

Two servers are available :

1) 156.18.90.97 - login: etu101 , password : ohthae
2) 156.18.90.98 - login: etu201 , password : rielei

Once connected by ssh, please create your working directory using your name.

For your experiments, you will use the Flickr8k dataset.

How to use the provided code:

1) Run create_input_files.py to download and prepare the dataset
   ```
   python create_input_files.py
   ```
2) Run train.py to train the models
   ```
   python train.py
   ```
3) Run caption.py to perform inference
   ```
   python caption.py --img='path/to/image.jpeg'
   --model='path/to/BEST_checkpoint_flickr8k_5_cap_per_img_5_min_word_freq.pth.tar'
   --word_map='path/to/WORDMAP_flickr8k_5_cap_per_img_5_min_word_freq.json' --beam_size=5
   ```

Changing the encoder

Instead of ResNet 101, we want to use VGG as an encoder. Please modify the code accordingly, re-train the model and explain possibly the performance difference.

Changing the decoder

Now instead of using LSTM as the decoder, we want to experiment GRU. Please modify the code accordingly, re-train the model and analyze the results.

Implementing the schedule sampling

When training a sequence model, it is important that training data match testing data. However, the “exposure bias” between training and testing typically occurs when only ground truth token sentences are used in the training process. The schedule sampling strategy avoids such an exposure bias and randomly chooses the next token to feed to the recurrent network during its training based on tossing a coin. During the beginning of the training process, much ground truth tokens are used but when the model gets more and more trained, tokens fed to the network for training at every step are more and more sampled from the tokens generated by the model.

How to submit your work ?

Your work should be uploaded within 3 weeks into the Moodle section “Devoir 1 - Image Captioning”. It should be one “.zip” file containing your report in a “pdf” format describing your work and experiments as well as your code (Python files “.py” or notebook).