BE1 - Image Captionning

Élèves :
Maxime BEAUFRETON

7 février 2022

Table des matières

Introduction

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. This is called 'Image captionning', and we will implement it based on the main ideas from the paper 'Show, Attend and Tell' by Xu et al., 2015. 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.

During this BE, we will :

— Understand and learn the architecture detailed in the problem statement

— Change the encoder from Resnet 101 to VGG, and observe the consequences on the performance

— Change the decoder from LSTM cell to GRU Cell, and observe the consequences on the performance

— Implement the schedule sampling

1 Understanding and learning

For our experiments, we will use the Flickr8k dataset. As recommended, we use the notebook and run it with Google Colaboratory which allows us to run our code on GPU. There are differents functions in the code. Let's explore them.

1.1 Pre-Processing

1.1.1 utils.py

Function create_input_files

— PretTreat the dataset. With a given dataset, it creates input files for training, validation, and test data.

— It creates a word map, with some additional caracters ('<unk>', '<start>', '<end>', '<pad>'), and save it to a JSON file.

— Sample captions for each image, save images to HDF5 file, and captions and their lengths to JSON files. (Splitting for Train, Validation and Test sets)

Function init_embedding

Fills embedding tensor with values from the uniform distribution

Function load_embeddings

Creates an embedding tensor for the specified word map, for loading into the model.

Function clip_gradient Clips gradients computed during backpropagation to avoid explosion of gradients.

Function save_checkpoint
Saves model checkpoint. If this checkpoint is the best so far, store a copy so it doesn't get overwritten by a worse checkpoint

Class AverageMeter
Keeps track of most recent, average, sum, and count of a metric.

Function adjust_learning_rate
Shrinks learning rate by a specified factor.

Function accuracy
Computes top-k accuracy, from predicted and true labels.

1.1.2 create_input_files.py

This function download a given dataset (here "Flickr8k"). The tqdm instance allows us to see the progress of the download.

1.1.3 datasets.py

In this program, we define a PyTorch Dataset class. It aims to be used in a PyTorch DataLoader to create batches. It loads the previously defined datasets (train, validation, test).

1.2 Models

We finally arrive to the core programm. Let's explore how are structured the Encoder, the Decoder, and the Attention Mechanism.

1.2.1 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 (and then VGG), 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. Since it is the role of an encoder, this seems to be a good choice.

Instead of training an encoder from scratch, we are using an existing pretrained model and fine-tune (function fine_tune) it to improve the performance. **Usually the fine**

tuning is realized on the last layers of the CNN model, freezing the over (shallower) layers. 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.

1.2.2 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 first, and then a GRU.

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. However, since the first studies on the Attention Mechanism have been published, it has shown great result in the image captionning field, and it seems more promising to implement it for our problem.

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 then 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.

The decoder use the Attention Mechanism. Hence, an instance of the class Attention is called. The decoder class has also functions to initialize its weights (LSTM or GRU), load pretrained embeddings (useful for enhance the performance in NLP), fine-tuned this embedding (a word can have different meanings, hence a fixed embedding could be a disavantage for the model). The forward function basically process these different steps :

- Flatten image
- Sort input data by decreasing lengths
- Embedding
- Initialize LSTM state
- Create tensors to hold word predicion scores and alphas
- At each time-step, decode by :
  - Attention-weighting the encoder's output based on the decoder's previous hidden state output
  - Generate a new word in the decoder with the previous word and the attention weighted encoding. Note : This step is different if we use scheduled sampling, which we will study in the last part of the report.

1.2.3 Attention

The Attention network computes these weights. The Attention mechanism 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 :

$$\sum_{p=0}^{P} \alpha_{p,t} = 1$$

We can interpret this process as computing the probability that a pixel is the place to look to generate the next word. These weights are defined in the Attention class. To clarify this notion of weights : they are learned and trained, as much as other parameters from the RNN/CNN. Note that within a channel of the encoded image, each pixel is weighted equally by the same alpha.

1.3 Training, Validating, Testing

For the training, we define the data parameters (which dataset we use), the model parameters (dimensions of embedding, attention, decoder) and the training parameters (epochs, batch size, learning rates,...). Then the training program executes the following steps :

- Read word map
- Initialize / load checkpoint (if a checkpoint if specified)
- Move to GPU, if available
- Define the loss function
- Custom dataloaders (normalize)
- Go through the epochs (train and validation steps, in validation we compute the BLEU score to follow the performance)

Then, in the program caption.py, we can visualise the ouputs of the model for different input images. It helps to grasp where are the error of the model. In particular, we can visualise the signification on the image of the attention computation, i.e to which zones the model was paying more attention. It is important to visualise it to understand the related words generated.

Finally, we compute the evaluation of the different models by the computation of the BLEU score with the BEAM search alogrithm.

2 Metric - BLEU Score and Analyse

2.1 BLEU Score

To evaluate the performance of each epoch, we use the BLEU Score. One of the challenges of NLP, and Image captionnong is that, given an image, there could be multiple English translations that are equally good caption of that image. The BLEU score, due to Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu helps to solve this problem by computing scores on n-grams. Without going into the details, it allows us to compare the performance between two algorithms, because it is a representation of the frequency of 'right' words proposed as a caption. (It looks at each of the words in the output and see if it appears in the references.) Here, we are considering the BLEU-4, which means 4-grams. The higher is the BLEU score (between 0 and 1), the better is the model.

2.2 Computation

At first, we tried to use Google Collab, but since their GPUs weren't always available we decided to move to Centrale's GPUs. We implemented a loop over the different models we had to analyse/implement in order to make it easier to train. It has required a few modification in the code that you can find attach to the report. The result of the BLEU score and the comparison between the different algorithms is at the end of the report.

3 Changing the encoder - VGG LSTM

Instead of ResNet 101, now we want to use VGG as an encoder. There are lots of different CNN architecture. Hence, the possibilites of Transfer Learning for the encoder part are wide. VGG is a famous architecture, with lower performance than ResNet on the ImageNet classification task. Unlike Resnet, it has no skip connections. We will compare the performance of the model with VGG Transfer Learning to the other models.

4 Changing the decoder - ResNet GRU

Now instead of using LSTM as the decoder, we want to experiment GRU. GRU are a variation of the LSTM cell, and aims to be simpler (2 gates rather than 3). Let's observe the performance of the GRU, and compare it to the LSTM's performance, for the same number of epochs. We will compare the performance of the ResNet-GRU model to the other models.

5 Implementing the schedule sampling

Scheduled sampling aims to resolve the mismatch between Train and Test phases. During training, the model has access to the 'ground-truth' previous word to generate the next one (the inputs are reference). It means that it never has to guess based on a 'wrong' word at any step of the decoding, while in testing the inputs are the outputs of the last time step (generation), so the previous word feeded to the RNN can be 'wrong'. Hence, training and testing behaviors are obviously different, and so are the training and testing distribution : This create an exposure bias. Hence, the idea is to reduce this exposure bias, making the traning phase more alike to the testing phase :

At each decoding step, we compute a random number from a U([0,1] distribution. We compare it to sheduled_sampling_param (which is 1 at the beginning and decrease to close 0 in then end of the training, the decay curve car be chosen).

— If random_number > sheduled_sampling_param : Input = output last time step.
— If random_number <= sheduled_sampling_param : Input = Reference.

By doing this, we allow the model to learn easily with the references at the beginning (always random_number < sheduled_sampling_param =1), and to have a distribution more alike to the testing phase (output of the last time step as input) in the end (always random_number > sheduled_sampling_param = 0). In the BE, we have implemented a linear decay of the sheduled_sampling_param. You can find an extract of the code below.

6 Analysis of the results

So far, we have implemented 4 different models :

— Encoder : ResNet, Decoder : LSTM, Scheduled sampling : False
— Encoder : ResNet, Decoder : GRU, Scheduled sampling : False
— Encoder : vgg, Decoder : LSTM, Scheduled sampling : False
— Encoder : ResNet, Decoder : LSTM, Scheduled sampling : True

We have tested them through 50 epochs each. On average, the training has stoped after 35 epochs only, because the model has not shown improvements since 20 epochs (epochs_since_improvement < 20 condition). For each training, we used the same hyperparameters. We have evaluated the models with a Beam-width of 3 and 7. The results are the following :

We observe that the best model for the BLEU score is the Resnet-LSTM one for both tests. Then follows ResNet-LSTM with scheduled sampling, the ResNet-GRU and finally the model VGG-LSTM.

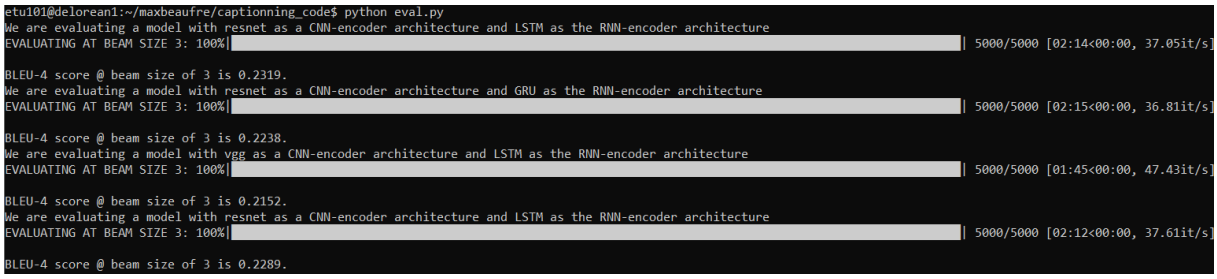

FIGURE 1 – BLEU score for a Beam width of 3

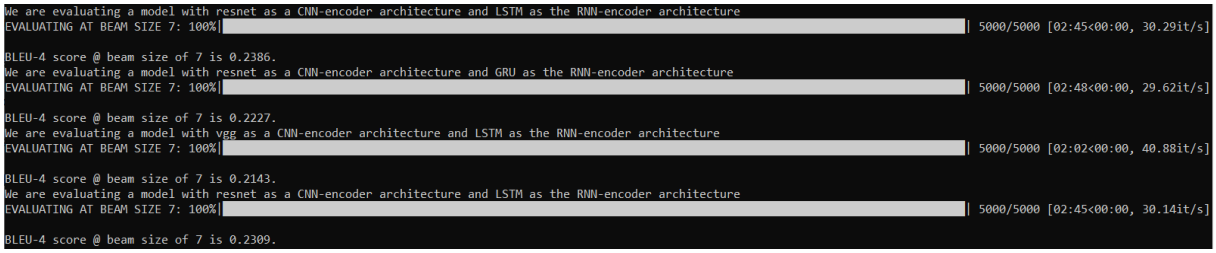

FIGURE 2 – BLEU score for a Beam width of 7

Hence, the ResNet encoder seems to perform better (at least with the default hyper-parameters) than the VGG encoder. We must note that ResNet is deeper, and has better performances than VGG for classification tasks (on ImageNet). It seems that its greater abilities to extract useful features has helped our model. The GRU is a simpler model than the LSTM. It can be faster to train, but here, at constant number of epochs, it has been less powerful than the LSTM for the image captionning task.

Finally, the ResNet-LSTM with scheduled sampling has been less competitive than the model without it. If it can be surprising at first, the reason between this result *might be* that for a computation on few epochs, the scheduled sampling can add too much noise rather than help the model. Maybe the linear decay is not the better solution for our problem, and an inverse sigmoid decay would be more appropriate here.

A potential lead to enhance the results would be to perform a stronger learning rate decay. We have seen that the algorithm stops because of too much epochs passed without improvement. Learning rate decay could help to solve this issue, e.g torch.optim.lr_scheduler.StepLR. However, a decay is already done after 8 consecutive epochs without improvement. We could try a stronger decay but it may not lead to a better performance.

The other leads are :

— Data augmentation : Find a larger dataset to learn

— Fine-tune the encode (False by default)

Annexes

```
for t in range(max(decode_lengths)):
        batch_size_t = sum([l > t for l in decode_lengths])
        attention_weighted_encoding, alpha = self.attention(
            encoder_out[:batch_size_t], h[:batch_size_t])
        gate = self.sigmoid(self.f_beta(h[:batch_size_t]))  # gating
            scalar, (batch_size_t, encoder_dim)
        attention_weighted_encoding = gate *
            attention_weighted_encoding  # why - Is it a variation of
            the classis attention mechanism (Kind of update gate ?)

        if self.scheduled_sampling and t>0 and random.random() >
            scheduled_sampling_param:
                h, c = self.decode_step(torch.cat([
                    previous_word_embedding[:batch_size_t, :],
                    attention_weighted_encoding], dim=1),
                        (h[:batch_size_t], c[:batch_size_t]))  # (
                            batch_size_t, decoder_dim)

        else:
          h, c = self.decode_step(
                    torch.cat([embeddings[:batch_size_t, t, :],
                        attention_weighted_encoding], dim=1),
                    (h[:batch_size_t], c[:batch_size_t]))  # (
                        batch_size_t, decoder_dim)

        preds = self.fc(self.dropout(h))  # (batch_size_t, vocab_size
            )
        predictions[:batch_size_t, t, :] = preds
        if self.scheduled_sampling:
          scores = self.softmax(preds) # (batch_size_t, vocab_size) #
              We compute the score to choose the best word
          words = torch.argmax(scores, dim=1).squeeze() # (
              batch_size_t,)
          previous_word_embedding = self.embedding(words) # (
              batch_size_t, embed_dim)
          alphas[:batch_size_t, t, :] = alpha

return predictions, encoded_captions, decode_lengths, alphas,
    sort_ind
```

```
 * LOSS - 3.771, TOP-5 ACCURACY - 69.345, BLEU-4 - 0.16228286937921116

Epoch: [12][0/938]      Batch Time 0.445 (0.445)        Data Load Time 0.051 (0.051)    Loss 2.8945 (2.8945)    Top-5 Accuracy 80.645 (80.645)
Epoch: [12][100/938]    Batch Time 0.403 (0.406)        Data Load Time 0.046 (0.048)    Loss 2.8872 (2.9116)    Top-5 Accuracy 81.330 (81.229)
Epoch: [12][200/938]    Batch Time 0.399 (0.404)        Data Load Time 0.046 (0.047)    Loss 3.1446 (2.9297)    Top-5 Accuracy 78.571 (81.059)
Epoch: [12][300/938]    Batch Time 0.395 (0.404)        Data Load Time 0.045 (0.047)    Loss 3.1326 (2.9283)    Top-5 Accuracy 76.966 (81.013)
Epoch: [12][400/938]    Batch Time 0.399 (0.404)        Data Load Time 0.049 (0.047)    Loss 2.9087 (2.9348)    Top-5 Accuracy 81.818 (80.914)
Epoch: [12][500/938]    Batch Time 0.433 (0.403)        Data Load Time 0.047 (0.047)    Loss 2.8965 (2.9424)    Top-5 Accuracy 82.009 (80.818)
Epoch: [12][600/938]    Batch Time 0.410 (0.403)        Data Load Time 0.046 (0.047)    Loss 3.0540 (2.9508)    Top-5 Accuracy 78.765 (80.672)
Epoch: [12][700/938]    Batch Time 0.425 (0.403)        Data Load Time 0.046 (0.047)    Loss 3.1402 (2.9583)    Top-5 Accuracy 78.133 (80.557)
Epoch: [12][800/938]    Batch Time 0.423 (0.403)        Data Load Time 0.057 (0.047)    Loss 3.0178 (2.9632)    Top-5 Accuracy 81.796 (80.486)
Epoch: [12][900/938]    Batch Time 0.414 (0.403)        Data Load Time 0.047 (0.047)    Loss 3.0831 (2.9653)    Top-5 Accuracy 77.094 (80.449)
Validation: [0/157]     Batch Time 0.323 (0.323)        Loss 3.7511 (3.7511)    Top-5 Accuracy 69.853 (69.853)
Validation: [100/157]   Batch Time 0.316 (0.320)        Loss 3.9077 (3.7715)    Top-5 Accuracy 66.048 (69.392)

 * LOSS - 3.780, TOP-5 ACCURACY - 69.196, BLEU-4 - 0.15552608084592975


Epochs since last improvement: 1

Epoch: [13][0/938]      Batch Time 0.479 (0.479)        Data Load Time 0.051 (0.051)    Loss 3.0545 (3.0545)    Top-5 Accuracy 80.430 (80.430)
Epoch: [13][100/938]    Batch Time 0.393 (0.406)        Data Load Time 0.048 (0.048)    Loss 2.9728 (2.8643)    Top-5 Accuracy 80.286 (82.174)
Epoch: [13][200/938]    Batch Time 0.395 (0.404)        Data Load Time 0.048 (0.047)    Loss 2.7982 (2.8577)    Top-5 Accuracy 83.844 (82.165)
Epoch: [13][300/938]    Batch Time 0.411 (0.403)        Data Load Time 0.046 (0.047)    Loss 2.8986 (2.8668)    Top-5 Accuracy 83.292 (82.030)
Epoch: [13][400/938]    Batch Time 0.398 (0.403)        Data Load Time 0.050 (0.047)    Loss 2.9935 (2.8780)    Top-5 Accuracy 80.415 (81.895)
Epoch: [13][500/938]    Batch Time 0.397 (0.403)        Data Load Time 0.047 (0.047)    Loss 2.8987 (2.8796)    Top-5 Accuracy 80.874 (81.838)
Epoch: [13][600/938]    Batch Time 0.397 (0.403)        Data Load Time 0.045 (0.047)    Loss 3.0683 (2.8853)    Top-5 Accuracy 78.100 (81.761)
Epoch: [13][700/938]    Batch Time 0.396 (0.403)        Data Load Time 0.047 (0.047)    Loss 2.8044 (2.8934)    Top-5 Accuracy 85.095 (81.632)
Epoch: [13][800/938]    Batch Time 0.404 (0.403)        Data Load Time 0.046 (0.047)    Loss 3.0486 (2.8983)    Top-5 Accuracy 80.556 (81.538)
Epoch: [13][900/938]    Batch Time 0.412 (0.403)        Data Load Time 0.046 (0.047)    Loss 2.8206 (2.9020)    Top-5 Accuracy 80.440 (81.457)
Validation: [0/157]     Batch Time 0.315 (0.315)        Loss 3.7252 (3.7252)    Top-5 Accuracy 66.578 (66.578)
Validation: [100/157]   Batch Time 0.318 (0.319)        Loss 3.7905 (3.8184)    Top-5 Accuracy 67.116 (68.984)
```

FIGURE 3 – Example of computation of ResNet-LSTM model on Google Collab