INDIAN INSTITUTE OF TECHNOLOGY, DELHI

ASSIGNMENT REPORT

Team AlphaNet : Reading Captions Embedded in Images

APAR AHUJA | ENTRY NO. 2019CS10465

ARNAV TULI | ENTRY NO. 2019CS10424

Course - COL774 | Prof. Parag Singla

Compiled on November 28, 2021

Contents

Chapter 1

Introduction to AlphaNet

1.1 What is AlphaNet?

In this assignment, we developed a machine learning model, a.k.a **AlphaNet** that aims to predict text embedded in images. This problem is very different from the *Image-Captioning* problem as the embedded text may have no relation whatsoever with the background image (random). Coming to the challenging part, the text might be rotated, translated and its font (also font size) might be different for different images. Hence, the task is to *effectively* predict the embedded text in these *dire* situations. The problem is similar to *character-recognition* once the text has been bounded in the original image, but that task is itself challenging enough as the text can be present anywhere and in any orientation in the image.
In order to solve this problem, we came up with **AlphaNet**, which is effectively a *deep-learning* model. It has an *Encoder-Decoder* architecture and uses both **convolutional** and **recurrent** *neural networks* (CNN and RNN). The goal of the model is simple: Given an image (tensor), **AlphaNet** aims to predict the text embedded in the image (word-by-word).
Before we dive deeper into our architecture of *AlphaNet*, we will first explain our data processing steps, as they allow us to effectively reduce the noise/variance in training data (raw).

1.2 Data Processing

Training Data consists of list of (image, caption) pairs, where *image* is the actual image (RGB, variable sized) and *caption* is the actual text embedded in the image. Hence, there are two types of data processing: *image processing* and *caption processing*.

1.2.1 Caption Processing

1. Split captions on spaces to generate word *tokens*
2. Prepend the list of *tokens* with **[START]** token
3. Append the list of *tokens* with **[END]** token
4. Generate a vocabulary of *tokens* and assign each token a unique *index* (starting from 1)
5. Vectorize each caption (list of tokens) by using the associated *index*
6. Use **[PAD]** token ($index = 0$) to make all the vectors of same size
7. Return the stacked tensor and list of actual caption lengths (excluding **[PAD]** token)

Since the caption data was entirely in lower case, we were not required to do that processing. Also, punctuation marks like '!', '?' were not removed and treated as word *tokens*, which had to be predicted.

1.2.2 Image Processing

1. Convert the images into Gray-Scale (from RGB format) format
2. Apply a threshold filter of $\frac{220}{255}$, and make all pixels below threshold = **0**
3. Resize images to a fixed size of (256, 256) (height × width)
4. Convert images back to RGB format

We applied a *threshold* filter to reduce the noise which is mostly due to the background images. While going through the data set, we observed that almost all the embeddings were *white* in colour, with **black** boundaries. This gave us the idea of applying the *threshold* filter and reduce the input variance. Resizing of image is done, so that *AlphaNet* can handle variable-sized images. The conversion from RGB to Gray and back to RGB was done merely out of efficiency in applying the threshold filter on grayscale-based images.
Though this is all the processing that we do offline, we also use a *neat*-trick to increase data set, reduce over-fitting and make the model more powerful. The trick involves passing all the **four** rotational variants of the image to the model during *training*. This helps in giving our model some degree of rotational invariance.

1.3 Architecture Overview

In this section, we give a brief overview of core components of **AlphaNet**:

- **Encoder:** CNN-based architecture that handles the *processed* images, and generates the corresponding *feature vector* or a *feature map*.
- **Decoder:** LSTM-based architecture which generates the captions for a given encoded image input. The caption is generated at *word*-level rather than *character*-level
- **Attention Module:** We implemented a modified version of *Bahdanau Attention Mechanism* that acts as bridge between the *Encoder* and *Decoder* and aims to provide more context to the *Decoder* while generating a caption.

Next, we mention our training strategy:

- We use *cross-entropy* as our loss function, and use *teacher-forcing* strategy to train the *Decoder*. *Adam* is used for optimization (all 3 components), with learning rate set to 10^{-3}
- We use all *four* rotational variants of the image during training. We tried doing this in both ways: *stochastically* and *batch-wise*
- We trained our model in time-slots of **eight** hours each, and observed the performance after every slot. We then chose the model that gave the highest-score on *training data*
- *Backpropagation* algorithm is used for optimizing neural network parameters, with training batch size of *16* examples. Training was accelerated using parallel workers and GPU

Now, we mention our testing strategy:

- During **Inference** time, we used *beam search* to generate a set of captions at each time step, keeping the beam size fixed (**5**). This ensures efficiency in generating captions and at the same time performs as good as the *greedy*-search method. At final time step, we simply choose the caption with highest log-likelihood.
- We use *character*-level BLEU score to evaluate our model on the training set

Chapter 2

Core Components

In all instances below, B will refer to the training batch-size.

2.1 Encoder

Input Specifications: $B \times 3 \times 256 \times 256$ RGB-image tensor
Linear-layer (FC) input specifications: $B \times 51200$ feature tensor ($51200 = 512 \times 10 \times 10$)
Linear-layer (FC) output specifications: Encoded tensor of size $B \times 512$
We have a total of **nine** layers in our Encoder, out of which **eight** layers are convolutional and one is a linear (FC) layer. All convolutional layers use *ReLU* as the activation function. The linear layer uses *Tanh* as activation function. We use batch normalization on layer output before applying the activation function (both convolutional and linear). We have also used Max-Pooling to reduce noise and map/parameter size). This helps in avoiding over-fitting and accelerates the training process. Refer **Figure 2.1** for block diagram of Encoder.

2.2 Decoder

Input Specifications: $B \times 512$ vector (encoded image or word embedding)
Output Specifications: For every non-*[PAD]* token in the training batch, a vector of dimension *vocabulary-size* is returned. This is later used for computing loss or caption generation
The Decoder mainly consists of an **LSTM** Cell and a *Linear*-layer (FC). At each time step, an input vector is fed into the LSTM Cell along with the hidden state vectors of previous time step, to generate the hidden state output for current time step. The hidden state output is then passed through the *Linear* layer to generate Decoder output for that time step. We also use dropout on hidden state output (before FC) to avoid over-fitting.
Image is automatically encoded into a 512-sized vector by the Encoder. The caption tokens are encoded into a 512-sized vector by means of an *embedding*-layer. Refer **Figure 3.1** and **4.1** for block diagram of Decoder in two settings.

2.3 Attention Module

We implemented a modified version of *Bahdanau Attention Mechanism*. The module acts as a bridge between the Encoder and Decoder by providing more context to Decoder while generating a caption. At every time step, the module takes the encoded feature map ($B \times 100 \times 512$) rather than the *latent* vector along with the Decoder's hidden state output of previous time step. It then passes these two maps through a linear layer each, adds them up and applies *Tanh* activation

function. The final output is passed through a linear layer and *log softmax* is applied to generate the *context*-weights. Notice that in original *Bahdanau Mechanism*, softmax was used for weight generation. However, we observed better performance by using *log softmax*, and therefore, used it in our model architecture. The final *context*-vector was generated by taking a *weighted* sum of all 100 feature vectors. Refer **Figure 4.1** for block diagram of *AlphaNet* using the *Attention Module*.

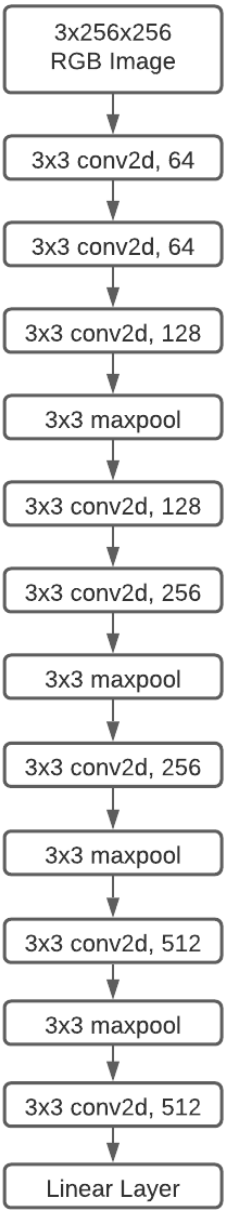

Figure 2.1: Block Diagram of Encoder

Chapter 3

Non-Competitive Part

In this part, we trained our model from scratch on the *processed* data and evaluated our model by determining the BLEU score on the train data. We did not use *Attention Module* in this part. Refer **Figure 3.1** for block diagram of our training pipeline.

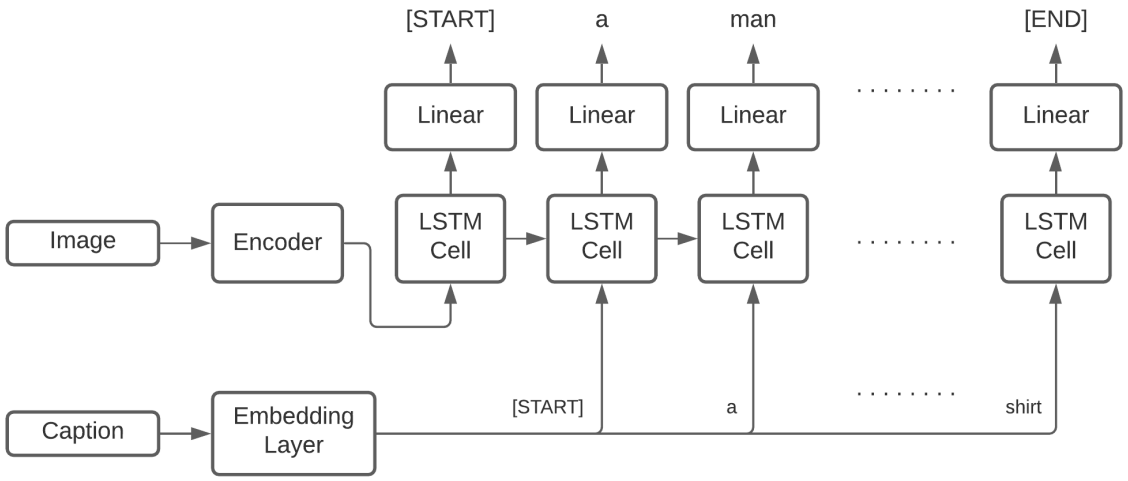

Figure 3.1: Training Pipeline

Hyperparameter Specifications:

- Batch Size: 16
- Optimizer: Adam (LR = 0.001)
- Dropout Probability: 0.5
- Hidden State and Embedding Dimension: 512
- Beam Size (for inference): 5

Results:

- Character-Level **BLEU** score on training set: 0.49216606013771247
- Character-level **BLEU** score on test set: 0.3111187378186736 (from EvalAI)

Chapter 4

Competitive Part

Since, we were allowed to use any pretrained image encoders, pretrained word embeddings etc., we tried using pretrained image encoders like - *ResNet-152* and *EfficientNet_b7*. We also tried using pretrained word embeddings from *Glove's* data set. However, the performance of the learnt model in each case was poor (even poorer than non-competitive part). Hence, we decided to use our Encoder-Decoder architecture that we developed in Non-Competitive part. In order to improve its performance, we decided to add the Attention-Module (*Bahdanau*). The final training pipeline can be seen in **Figure 4.1**. In diagram, $\bigoplus$ represents concatenation of the two vectors.

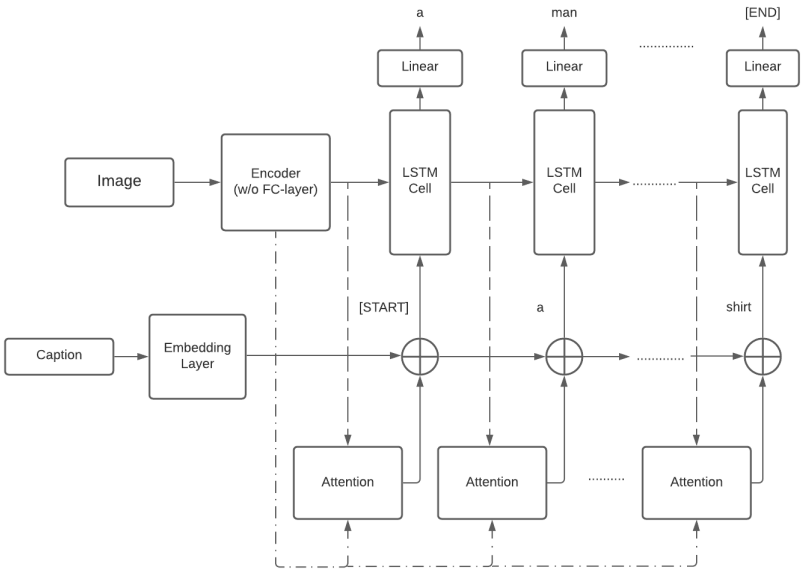

Figure 4.1: Training Pipeline (with Attention Module)

Hyperparameter Specifications:

- Dropout Probability: 0.1

All other hyperparameter specifications are same as that in *Non-Competitive* part.
Results:

- Character-Level **BLEU** score on test set: 0.5959289925481539 (from EvalAI)