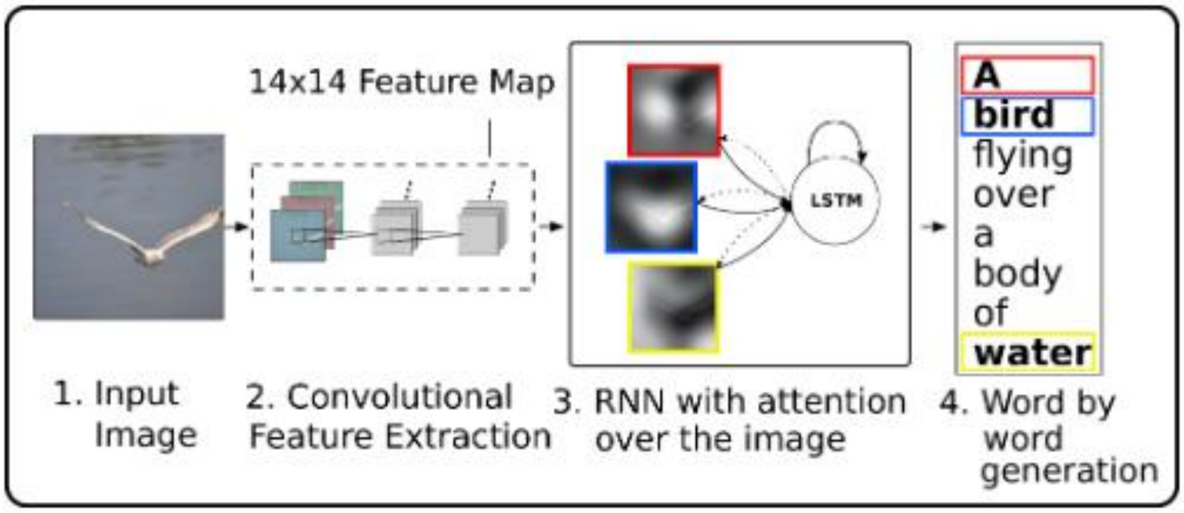
Show, Attend and Tell: Neural Image Caption Generation with Visual Attention

Group Report


14x14 Feature Map
LSTM
A
bird
flying
over
a
body
of
water
1. Input Image
2. Convolutional Feature Extraction
3. RNN with attention over the image
4. Word by word generation

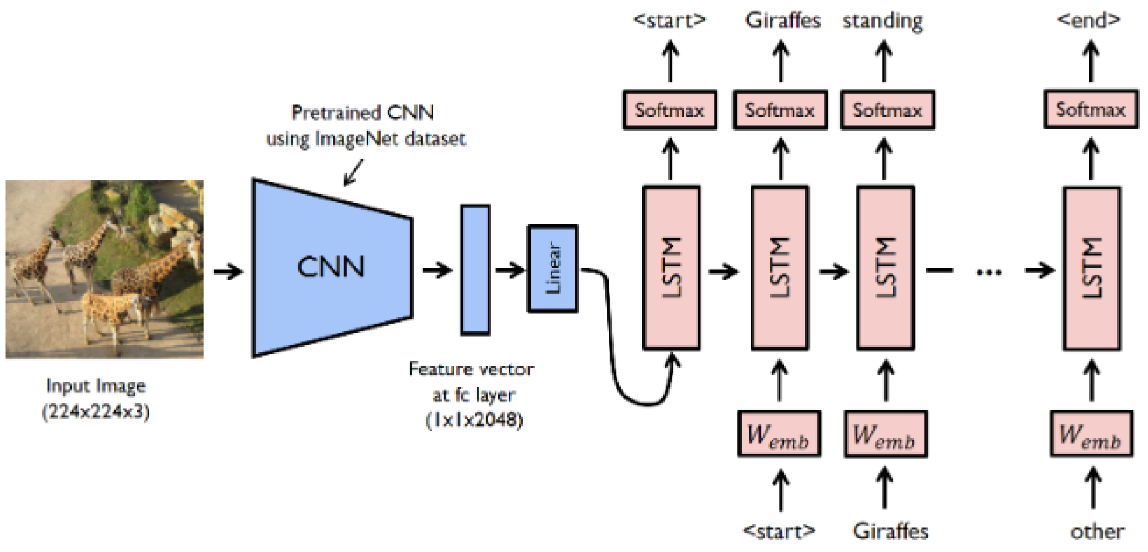
Presented by:

○ Aya Ahmed Kandil	*4699*
○ Hana Hesham Nazmy	4952
○ Monique Ehab	*4928*
○ Mohamed Abdelnaby	*5063*
○ Hania Mohamed Hani	*4697*

Introduction

Image caption generation is the problem of generating a descriptive sentence of an image. The fact that humans can do this with remarkable ease makes this a very interesting/ challenging problem for AI, combining aspects of computer vision and natural language processing.

In this work, we introduced an "attention" based framework into the problem of image caption generation. Much in the same way human vision fixates when you perceive the visual world, the model learns to "attend" to selective regions while generating a description.



Data Sources

We have identified the three most commonly used image caption training datasets in Computer Vision research domain:
COCO dataset, Flickr8k and Flickr30k.
These datasets contain 123,000, 31,000 and 8,000 captions annotated images respectively and each image is labeled with 5 different descriptions. Currently, **MSCOCO '14** dataset is used as our primary data source over the others.

Inputs to model

1. Images

Since we are using a pretrained Encoder, we would need to process the images into the form this pretrained Encoder is accustomed to. Pretrained ImageNet models available as part of PyTorch's TorchVision module. Pixel values must be in the range [0,1] and we must then normalize the image by the mean and standard deviation of the ImageNet images' RGB channels. Therefore, images fed to the model must be a Float tensor of dimension **(**Batch Size, 3, Image Size, Image Size), and must be normalized by the aforesaid mean and standard deviation.

2. Captions

Captions are both the target and the inputs of the Decoder as each word is used to generate the next word. To generate the first word, however, we need a *zeroth* word, <start>. At the last word, we should predict <end> the Decoder must learn to predict the end of a caption. This is necessary because we need to know when to stop decoding during inference. Since we pass the captions around as fixed size Tensors, we need to pad captions (which are naturally of varying length) to the same length with <pad> tokens. Furthermore, we create a word_map which is an index mapping for each word in the corpus, including the <start>, <end>, and <pad> tokens. PyTorch, like other libraries, needs words encoded as indices to look up embeddings for them or to identify their place in the predicted word scores.

3. Captions Length

Since the captions are padded, we would need to keep track of the lengths of each caption. This is the actual length + 2 (for the <start> and <end> tokens). Caption lengths are also important because you can build dynamic graphs with PyTorch. We only process a sequence up to its length and do not waste compute on the <pad>s. Therefore, caption lengths fed to the model must be an **Int** tensor of dimension **N**.

Data pipeline

create_input_files() in utils.py reads the data downloaded and saves the following files :

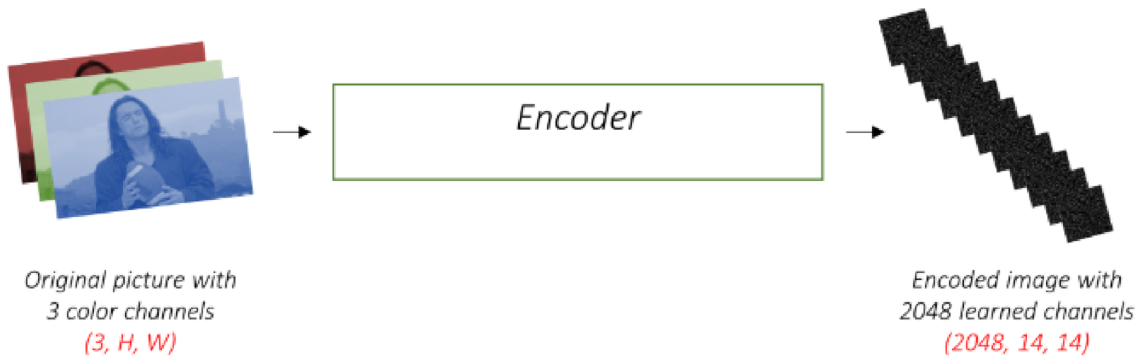
- An **HDF5 file containing images for each split in an I, 3, S, S tensor**, where I is the number of images in the split. Pixel values are still in the range [0, 255], and are stored as unsigned 8-bit Ints, and S is the image size.

- A **JSON file for each split with a list of N_c * I encoded captions**, where N_c is the number of captions sampled per image. These captions are in the same order as the images in the HDF5 file. Therefore, the ith caption will correspond to the i // N_cth image.
- A **JSON file for each split with a list of N_c * I caption lengths**. The ith value is the length of the ith caption, which corresponds to the i // N_cth image.
- A **JSON file which contains the word_map**, the word-to-index dictionary.

Before we save these files, we have the option to only use captions that are shorter than a threshold, and to bin less frequent words into an <unk> token. We use HDF5 files for the images because we will read them directly from disk during training / validation. They're simply too large to fit into RAM all at once. But we do load all captions and their lengths into memory. The Dataset will be used by a PyTorch DataLoader in train.py to create and feed batches of data to the model for training or validation.

Encoder (CNN)

The encoder needs to extract image features of various sizes and encodes them into vector space which can be fed to RNN in a later.
We use pretrained models already available in PyTorch's torchvision module. Discard the last two layers (pooling and linear layers), since we only need to encode the image, and not classify it. We do add an AdaptiveAvgPool2d() layer to **resize the encoding to a fixed size**. This makes it possible to feed images of variable size to the Encoder.
Since we may want to fine-tune the Encoder, we add a fine_tune() method which enables or disables the calculation of gradients for the Encoder's parameters.

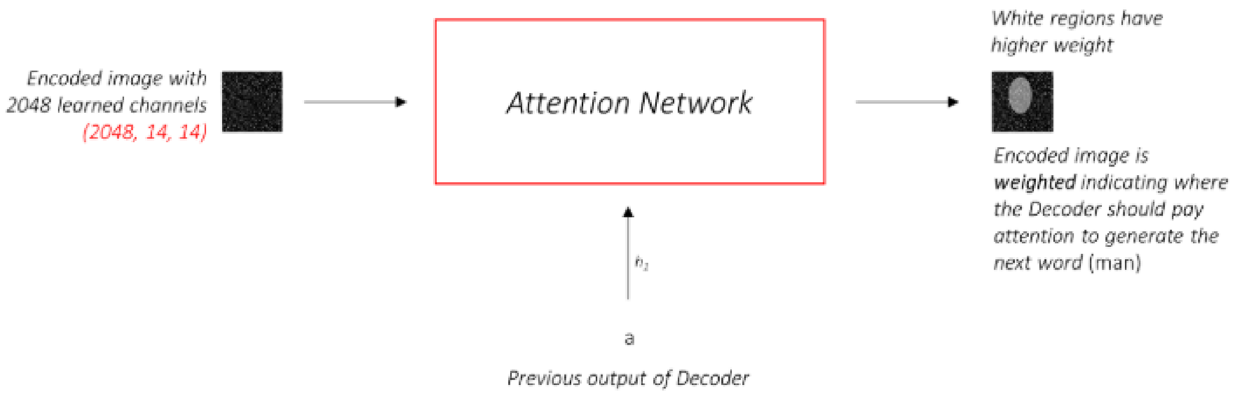


Soft Attention Mechanism

Following the CNN, we built the soft trainable Attention mechanism introduced in Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. Attention mechanism tells the network which part of the image should be focused on for generating the next word in the description. We calculated the attention area through adding the encoder output, which is updated in each iteration.

In implementing the attention layer, we performed linear transformation on both the encoder output *(flattened to N, 14 * 14, 2048)* and the hidden state (output) from the Decoder to the same dimension. The linear activated outputs are summed up and the activation function is ReLU. A third linear layer transforms this result to a dimension of 1**,** whereupon we apply the softmax to generate the weights *Alpha*.

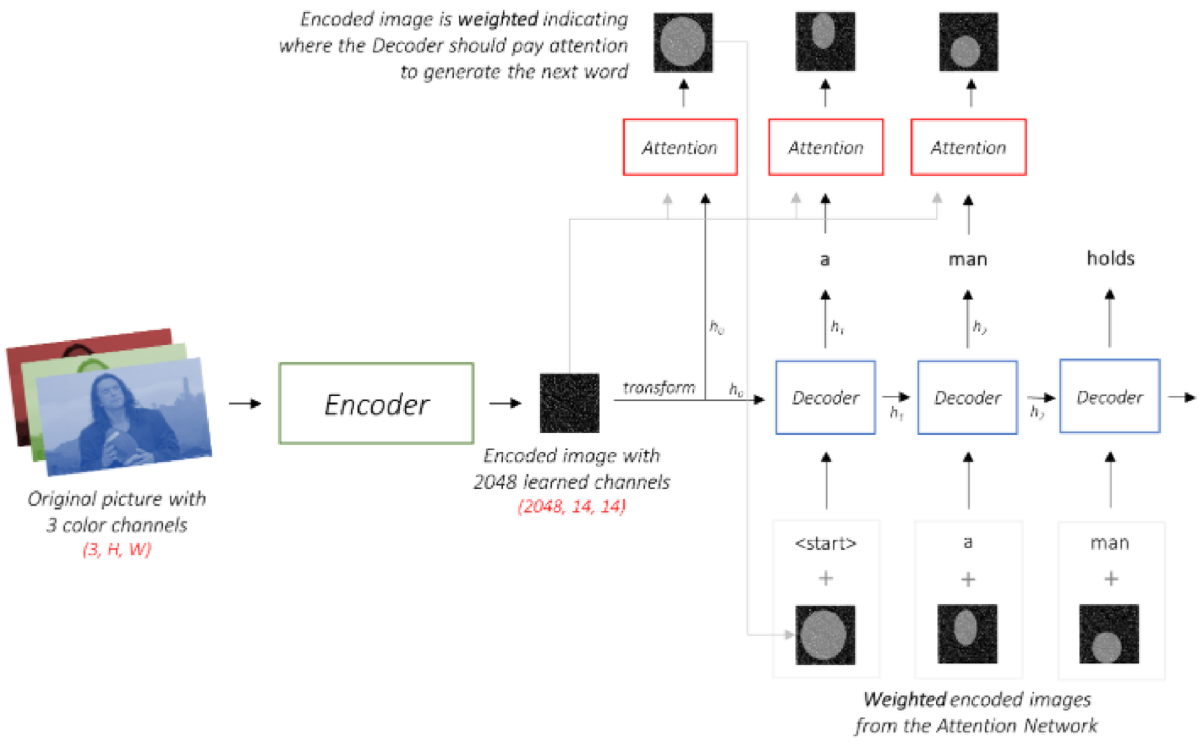
Different from CNN without fine-tuning, the attention mechanism is trainable with back-propagation. The returned attention area is used later in decoder involving RNN.



Decoder (RNN)

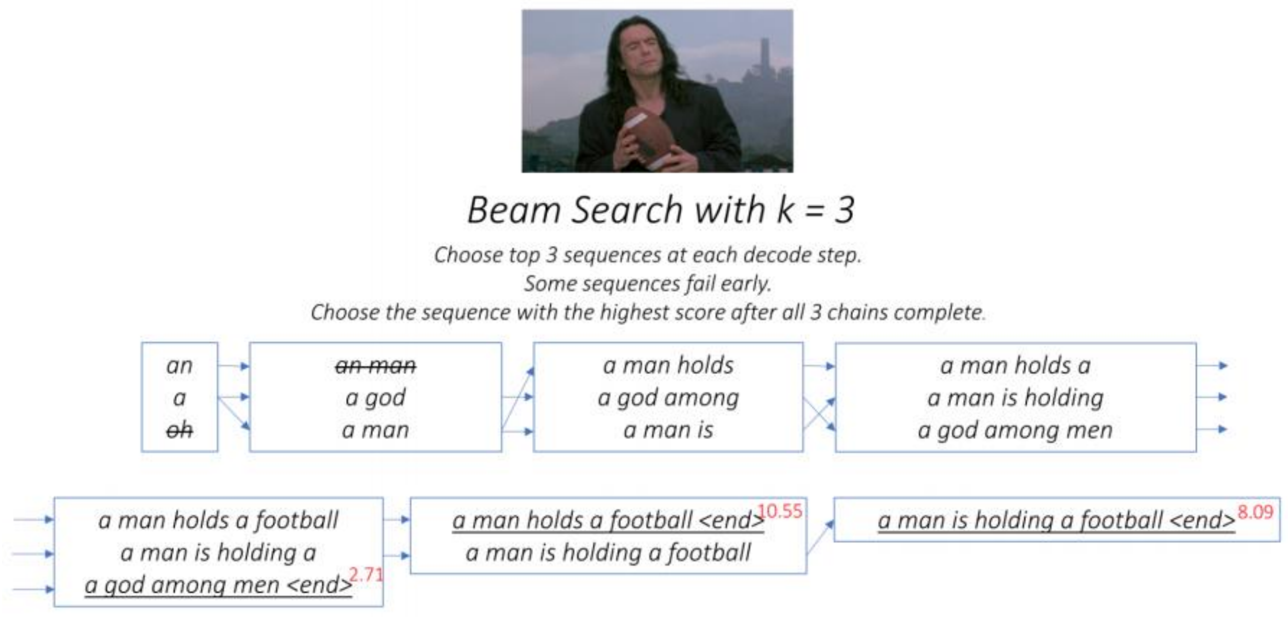
The decoder needs to generate image captions word by word using a Recurrent Neural Network - LSTMs which is able to sequentially generate words. The input for the decoder is the encoded image feature vectors from CNN and the encoded image captions produced in data preprocessing stage flattened to dimensions N, 14 * 14, 2048. The decoder consists of an attention module, an LSTM cell module and two separate linear layers provided by PyTorch library for the initialization of the states of LSTM cell and word dictionary. When receiving the encoded images and captions, we first sort the encoded images and captions by encoded key length of images in descending order. We can iterate over each timestep, processing only the colored regions, which are the effective batch size N_t at that timestep. The sorting allows the top N_t at any timestep to align with

the outputs from the previous step. This iteration is performed manually in a for loop with a PyTorch LSTMCell instead of iterating automatically without a loop with a PyTorch LSTM. This is because we need to execute the Attention mechanism between each decode step. An LSTMCell is a single timestep operation, whereas an LSTM would iterate over multiple timesteps continuously and provide all outputs at once. We compute the weights and attention-weighted encoding at each timestep with the Attention network.
We concatenate this filtered attention-weighted encoding with the embedding of the previous word and run the LSTMCell to generate the new hidden state (or output). A linear layer transforms this new hidden state into scores for each word in the vocabulary, which is stored. We also store the weights returned by the Attention network at each timestep.


Encoded image is weighted indicating where the Decoder should pay attention to generate the next word
Attention
Attention
Attention
a
man
holds
h_0
h_1
h_2
Original picture with 3 color channels
(3, H, W)
Encoder
transform
h_0
Encoded image with 2048 learned channels
(2048, 14, 14)
Decoder
h_1
Decoder
h_2
Decoder
<start>
+
a
+
man
+
Weighted encoded images from the Attention Network

Beam Search

The final step to generate a sentence with the highest likelihood of occurrence given the input image. caption_image_beam_search() reads an image, encodes it, and applies the layers in the Decoder in the correct order, while using the previously generated word as the input to the LSTM at each timestep.


Beam Search with k = 3
Choose top 3 sequences at each decode step.
Some sequences fail early.
Choose the sequence with the highest score after all 3 chains complete.
an
a
~~oh~~
~~an man~~
a god
a man
a man holds
a god among
a man is
a man holds a
a man is holding
a god among men
a man holds a football
a man is holding a
a god among men <end> 2.71
a man holds a football <end> 10.55
a man is holding a football
a man is holding a football <end> 8.09

Training Phase

For the encoder part, the pretrained CNN extracts the feature vector from a given input image. The feature vector is linearly transformed to have the same dimension as the input dimension of the LSTM network. For the decoder part, source and target texts are predefined. For example, if the image description is "Giraffes standing next to each other", the source sequence is a list containing ['<start>', 'Giraffes', 'standing', 'next', 'to', 'each', 'other'] and the target sequence is a list containing ['Giraffes', 'standing', 'next', 'to', 'each', 'other', '<end>']. Using these source and target sequences and the feature vector, the LSTM decoder is trained as a language model conditioned on the feature vector.

Training Parameters:

- Number of epochs to train for: 5
- Batch size: 128
- Workers for data loading: 4
- Learning rate for encoder: 1e-4
- Learning rate for decoder: 4e-4
- Regularization parameter for 'doubly stochastic attention': 5
- Fine tune encoder = False

Loss Function

Since we're generating a sequence of words, we use **CrossEntropyLoss**. We only need to submit the raw scores from the final layer in the Decoder, and the loss function will perform the softmax and log operations.
The authors of the paper recommend using a second loss – a "**doubly stochastic regularization**". We know the weights sum to 1 at a given timestep. But we also encourage the weights at a single pixel p to sum to 1 across *all* timesteps T –

$$\sum_{t}^{T} \alpha_{p,t} \approx 1$$

This means we want the model to attend to every pixel over the course of generating the entire sequence. Therefore, we try to **minimize the difference between 1 and the sum of a pixel's weights across all timesteps**. **We do not compute losses over the padded regions**. An easy way to do get rid of the pads is to use PyTorch's pack_padded_sequence(), which flattens the tensor by timestep while ignoring the padded regions.

Validation Phase

To evaluate the model's performance on the validation set, we will use the automated Bilingual Evaluation Understudy (BLEU) evaluation metric. This evaluates a generated caption against reference captions. For each generated caption, we will use all N_c captions available for that image as the reference captions.
The authors of the *Show, Attend and Tell* paper observe that correlation between the loss and the BLEU score breaks down after a point, so they recommend to stop training early on when the BLEU score begins to degrade, even if the loss continues to decrease.

Test Phase

In the test phase, the encoder part is almost same as the training phase. The only difference is that batchnorm layer uses moving average and variance instead of mini-batch statistics. For the decoder part, there is a significant difference between the training phase and the test phase. In the test phase, the LSTM decoder cannot see the image description. To deal with this problem, the LSTM decoder feeds back the previously generated word to the next input.