Caption this, with TensorFlow!

How to build and train an image caption generator using a TensorFlow notebook.

By Raul Puri and Daniel Ricciardelli

Attention Readers: We invite you to access the corresponding Python code and iPython notebooks for this article, on GitHub.

Image caption generation models combine recent advances in computer vision and machine translation to produce realistic image captions using neural networks. Neural image caption models are trained to maximize the likelihood of producing a caption given an input image, and can be used to generate novel image descriptions. For example, the following are possible captions generated using a neural image caption generator trained on the MS COCO data set.

The man in grey swings a bat while the man in black looks on.

A big bus sitting next to a person.

Credit: Raul Puri, with images sourced from MS COCO dataset

In this article, we will walk through an intermediate-level tutorial on how to train an image caption generator on the Flickr30k data set using an adaptation of Google's Show and Tell model. We use the TensorFlow framework to construct, train, and test our model because it's relatively easy to use and has a growing online community.

Why caption generation?

Recent AI success in computer vision tasks, such as image classification and natural language processing, like machine translation, have pushed AI researchers to look to the intersection of previously separate domains for new research avenues. Caption generation models have to balance an understanding of both visual cues and natural language.

The intersection of these two traditionally unrelated fields has the possibility to effect change on a wide scale. While there are some straight-forward applications of this technology, such as generating summaries for YouTube videos, or captioning unlabeled images, more creative applications can drastically improve the quality of life for a wide cross section of the population. Similar to how traditional computer vision seeks to make the world more accessible and understandable for computers, this technology has the potential to make our world more accessible and understandable for us humans. It can serve as a tour guide, and can even serve as a visual aid for daily life, such as in the case of the Horus wearable device from the Italian AI firm Eyra.

Some assembly required

Before we begin we'll need to do some housekeeping.

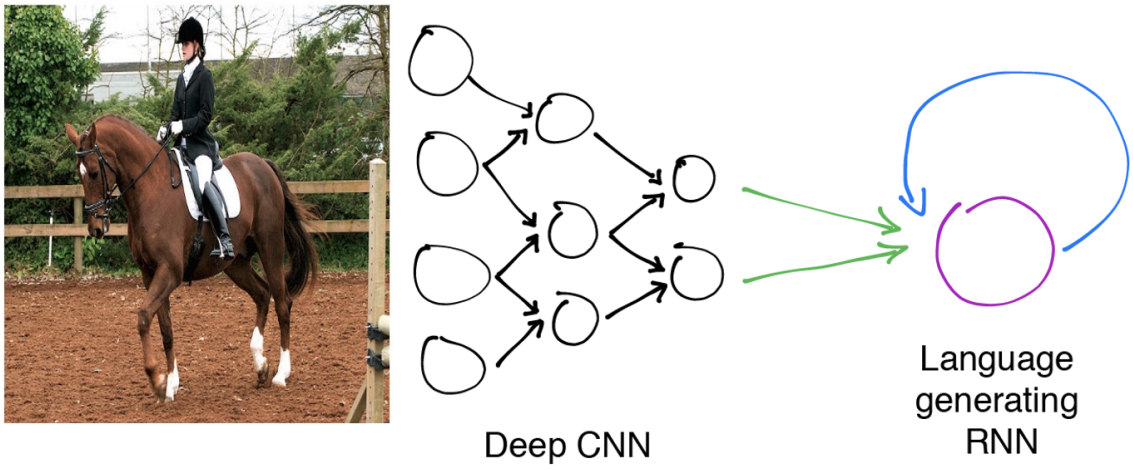
First, you will need to install Tensorflow. If this is your first time working with Tensorflow, we recommend that you first review the following article: Hello, TensorFlow! Building and training your first TensorFlow model.

You will need the pandas, opencv2, and Jupyter libraries to run the associated code. However, to simplify the install process we highly recommend that you follow the Docker install instructions on our associated **GitHub** repo.

You will also need to download the image embeddings and image captions for the Flickr30k data set. Download links are also provided on our **GitHub** repo.

Now, let's begin!

The image caption generation model



Credit: Shannon Shih from Machine Learning @ Berkeley. Horse Image sourced from MS COCO.

At a high-level, this is the model we will be training. Each image will be encoded by a deep convolutional neural network into a 4,096 dimensional vector representation. A language generating RNN, or recurrent neural network, will then decode that representation sequentially into a natural language description.

Caption generation as an extension of image classification

Image classification is a computer vision task with a lot of history and many strong models behind it. Classification requires models that can piece together relevant visual information about the shapes and objects present in an image, to place that image into an object category. Machine learning models for other computer vision tasks such as object detection and image segmentation build on this by not only recognizing when information is present, but also by learning how to

interpret 2D space, reconcile the two understandings, and determine where an object's information is distributed in the image. For caption generation, this raises two questions:

1. How can we build upon the success of image classification models, in retrieving important information from images?
2. How can our model learn to reconcile an understanding of language, with an understanding of images?

Leveraging transfer learning

We can take advantage of pre-existing models to help caption images. Transfer learning allows us to take the data transformations learned by neural networks trained on different tasks and apply them to our data. In our case, the VGG-16 image classification model takes in 224x224 pixel images and produces a 4,096 dimensional feature vector useful for categorizing images. We can take the representation (known as the image embedding) from the VGG-16 model and use it to train the rest of our model. For the scope of this article, we have abstracted away the architecture of VGG-16 and have pre-computed the 4,096 dimensional features to speed up training.

Loading the VGG image features and image captions is relatively straightforward:

```
def get_data(annotation_path,feature_path):
    annotations = pd.read_table(annotation_path, sep='\t', header=None, names=['image', 'caption'])
    return np.load(feature_path,'r'), annotations['caption'].values
```

Understanding captions

Now that we have an image representation, we need our model to learn to decode that representation into an understandable caption. Due to the serial nature of text, we leverage recurrence in an RNN/LSTM network (to learn more, read "Understanding LSTM Networks"). These networks are trained to predict the next word in a series given previous words and the image representation.

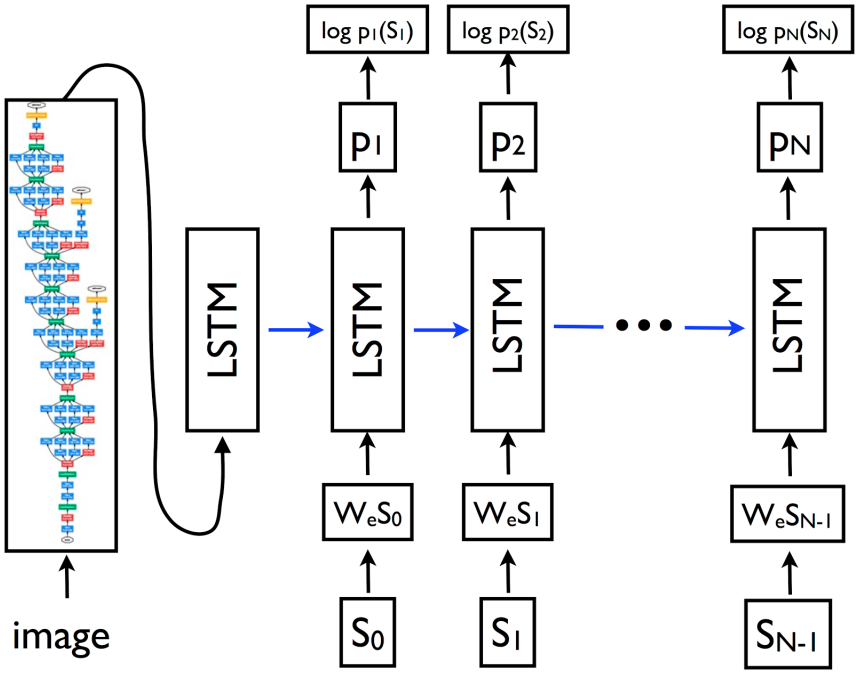
Long short-term memory (LSTM) cells allow the model to better select what information to use in the sequence of caption words, what to remember, and what information to forget. TensorFlow provides a wrapper function to generate an LSTM layer for a given input and output dimension.

To transform words into a fixed-length representation suitable for LSTM input, we use an embedding layer that learns to map words to 256 dimensional features (or word-embeddings). Word-embeddings help us represent our words as vectors, where similar word-vectors are semantically similar. To learn more about how word-embeddings capture the relationships between different words, check out "Capturing semantic meaning using deep learning."

In the VGG-16 image classifier, the convolutional layers extract a 4,096 dimensional representation to pass through a final softmax layer for classification. Because the LSTM cells expect 256 dimensional textual features as input, we need to translate the image representation into the representation used for the target captions. To do this, we utilize another embedding layer that learns to map the 4,096 dimensional image features into the space of 256 dimensional textual features.

Building and training the model

All together, this is what the Show and Tell Model looks like:



Source: "Show and Tell: Lessons learned from the 2015 MSCOCO Image Captioning Challenge", https://arxiv.org/abs/1609.06647

In this diagram, $\{s_0, s_1, ..., s_N\}$ represent the words of the caption we are trying to predict and $\{w_e s_0, w_e s_1, ..., w_e s_{N-1}\}$ are the word embedding vectors for each word. The outputs $\{p1, p2, ..., pN\}$ of the LSTM are probability distributions generated by the model for the next word in the sentence. The model is trained to minimize the negative sum of the log probabilities of each word.

```
def build_model(self):
    # declaring the placeholders for our extracted image feature vectors, our caption, and our mask
    # (describes how long our caption is with an array of 0/1 values of length `maxlen`
    img = tf.placeholder(tf.float32,[self.batch_size, self.dim_in])
    caption_placeholder = tf.placeholder(tf.int32,[self.batch_size, self.n_lstm_steps])
    mask = tf.placeholder(tf.float32,[self.batch_size, self.n_lstm_steps])

    # getting an initial LSTM embedding from our image_imbedding
    image_embedding = tf.matmul(img, self.img_embedding) + self.img_embedding_bias

    # setting initial state of our LSTM
    state = self.lstm.zero_state(self.batch_size, dtype=tf.float32)

   total_ loss = 0.0
    with tf.variable_scope("RNN"):
      for i in range(self.n_lstm_steps):
        if i > 0:
          #if this isn't the first iteration of our LSTM we need to get the word_embedding corresponding
          # to the (i-1)th word in our caption
           with tf.device("/cpu:0"):
             current_embedding = tf.nn.embedding_lookup(self.word_embedding, caption_placeholder[:,i-1])
+ self.embedding_bias
        else:
           #if this is the first iteration of our LSTM we utilize the embedded image as our input
           current_embedding = image_embedding
        if i > 0:
          # allows us to reuse the LSTM tensor variable on each iteration
          tf.get_variable_scope().reuse_variables()

        out, state = self.lstm(current_embedding, state)

        print (out,self.word_encoding,self.word_encoding_bias)

        if i > 0:
          #get the one-hot representation of the next word in our caption
          labels = tf.expand_dims(caption_placeholder[:,i], 1)
```

```
            ix_range=tf.range(0, self.batch_size, 1)
            ixs = tf.expand_dims(ix_range, 1)
            concat = tf.concat([ixs, labels],1)
            onehot = tf.sparse_to_dense(
                concat, tf.stack([self.batch_size, self.n_words]), 1.0, 0.0)

            #perform a softmax classification to generate the next word in the caption
            logit = tf.matmul(out, self.word_encoding) + self.word_encoding_bias
            xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=logit,labels=onehot)
            xentropy = xentropy * mask[:,i]

            loss = tf.reduce_sum(xentropy)
            total_loss += loss

        total_loss = total_loss / tf.reduce_sum(mask[:,1:])
        return total_loss, img,  caption_placeholder, mask
```

Generating captions using inference

After training, we have a model that gives the probability of a word appearing next in a caption, given the image and all previous words. How can we use this to generate new captions?

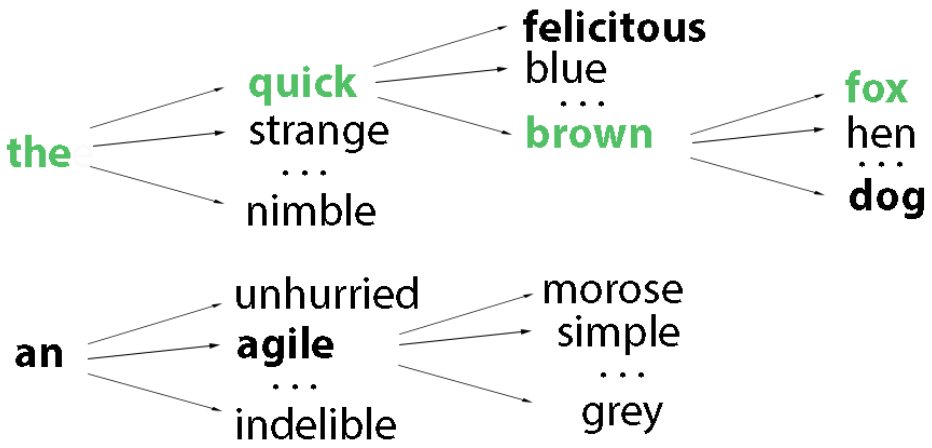
The simplest approach is to take an input image and iteratively output the next most probable word, building up a single caption.

```
def build_generator(self, maxlen, batchsize=1):
    #same setup as `build_model` function
    img = tf.placeholder(tf.float32,[self.batch_size, self.dim_in])
    image_embedding = tf.matmul(img, self.img_embedding) + self.img_embedding_bias
    state = self.lstm.zero_state(batchsize,dtype=tf.float32)

    #declare list to hold the words of our generated captions
    all_words = []
    print (state,image_embedding,img)
    with tf.variable_scope("RNN"):
        # in the first iteration we have no previous word, so we directly pass in the image embedding
        # and set the `previous_word` to the embedding of the start token ([0]) for the future iterations
        output, state = self.lstm(image_embedding,state)
        previous_word = tf.nn.embedding_lookup(self.word_embedding, [0]) + self.embedding_bias

        for i in range(maxlen):
```

```
        tf.get_variable_scope().reuse_variables()

        out, state = self.lstm(previous_word, state)

       # get a one-hot word encoding from the output of the LSTM
        logit = tf.matmul(out, self.word_encoding) + self.word_encoding_bias
        best_word = tf.argmax(logit, 1)

        with tf.device("/cpu:0"):
          # get the embedding of the best_word to use as input to the next iteration of our LSTM
           previous_word = tf.nn.embedding_lookup(self.word_embedding, best_word)

        previous_word += self.embedding_bias

        all_words.append(best_word)

    return img, all_words
```

In many cases this works, but by "greedily" taking the most probable words, we may not end up with the most probable caption overall.

One possible way to circumvent this, is by using a method called "Beam Search." The algorithm iteratively considers the set of the **k** best sentences up to length **t** as candidates to generate sentences of size **t + 1**, and keep only the resulting best **k** of them. This allows one to explore a larger space of good captions while keeping inference computationally tractable. In the example below, the algorithm maintains a list of **k = 2** candidate sentences shown by the path to each bold word for each vertical time step.



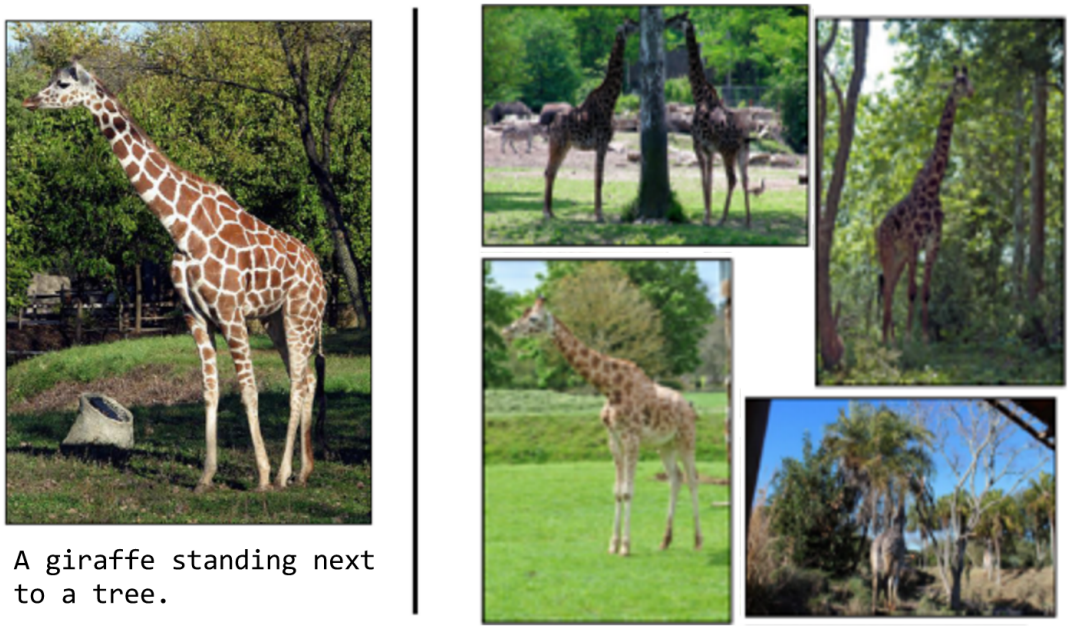
Credit: Daniel Ricciardelli

Limitations and discussion

The neural image caption generator gives a useful framework for learning to map from images to human-level image captions. By training on large numbers of image-caption pairs, the model learns to capture relevant semantic information from visual features.

However, with a static image, embedding our caption generator will focus on features of our images useful for image classification and not necessarily features useful for caption generation. To improve the amount of task-relevant information contained in each feature, we can train the image embedding model (the VGG-16 network used to encode features) as a piece of the caption generation model, allowing us to fine-tune the image encoder to better fit the role of generating captions.

Also, if we actually look closely at the captions generated, we notice that they are rather mundane and commonplace. Take this possible image-caption pair for instance:



Credit: Raul Puri, with images sourced from MS COCO data set

This is most certainly a “giraffe standing next to a tree.” However, if we look at other pictures, we will likely notice that it generates a caption of “a giraffe next to a tree” for any picture with a giraffe because giraffes in the training set often appear near trees.

Next steps

First, if you want to improve on the model explained here, take a look at Google's open source Show and Tell network, trainable with the MS COCO data set and an Inception-v3 image embedding.

Current state-of-the-art image captioning models include a visual attention mechanism, which allows the model to identify areas of interest in the image to selectively focus on when generating captions.

If you are interested in this state-of-the-art implementation of caption generation, check out the following paper: *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*.

Note: Don't forget to access the corresponding Python code and iPython notebooks for this article, on GitHub!

Related:

Getting Started with TensorFlow -- a tutorial at Strata + Hadoop World San Jose on March 14, 9 a.m. to 12:30 p.m.

Machine Learning with TensorFlow -- in-person training at Strata + Hadoop World San Jose on March 13-14, 9 a.m. to 5 p.m.

Raul Puri

Raul Puri is a graduating undergraduate researcher at UC Berkeley CO 2017. Raul has contributed to research projects in several fields including but not limited to: Robotics and Automation, Computer Vision, Medical Imaging, BioMems devices, etc. However, the bulk of his research work is focused on Machine Learning and Machine Learning Systems with applications to security, anomaly detection, NLP, and computer vision and robotics. Raul is also passionate about giving back to the community by teaching applied ML concepts and is a teaching assistant or instructor in several Berkeley ML class offerings.

Daniel Ricciardelli

Dan Ricciardelli is an undergraduate researcher at the University of California, Berkeley. His research interests include Natural Language Processing for finance and industrial applications, Computer Vision, Deep Active Learning, and Automated Knowledge Discovery. Dan is excited about making machine learning

more accessible to technical and non-technical students and professionals alongside Machine Learning at Berkeley.